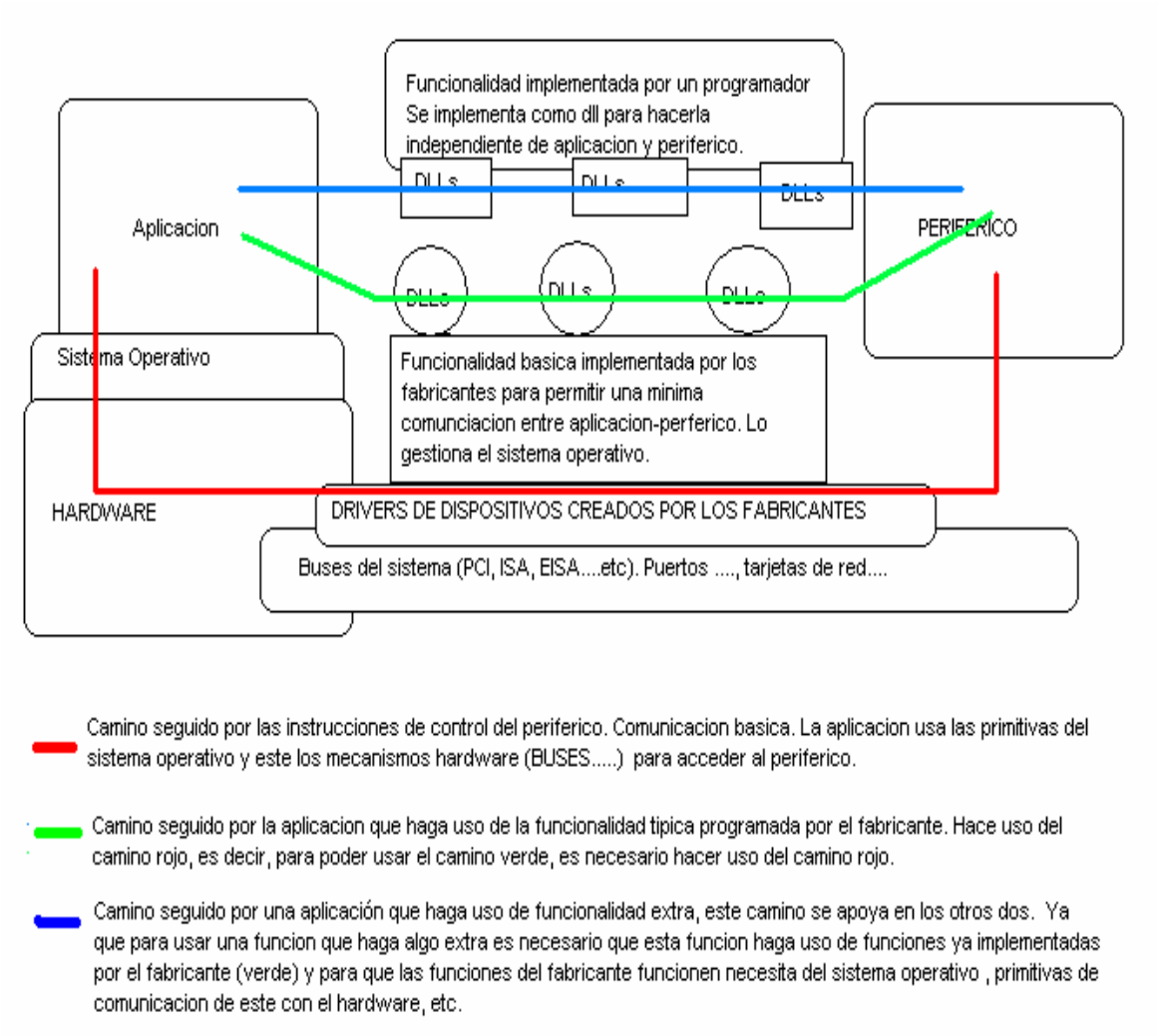
Práctica 5: Creación de DLLs de 32 bits.
Periféricos Avanzados

Motivación

La motivación de esta práctica es la de que capacitar al alumno para el desarrollo de manejadores de dispositivos dotandole de los conocimientos y herramientas necesarias para tal fin. El que se introduzca al alumno en la creación de librerías dinámicas no persigue otro fin que este que hemos citado. Ejemplo :



Camino seguido por las instrucciones de control del periferico. Comunicacion basica. La aplicacion usa las primitivas del sistema operativo y este los mecanismos hardware (BUSES.....) para acceder al periferico.

Camino seguido por la aplicacion que haga uso de la funcionalidad tipica programada por el fabricante. Hace uso del camino rojo, es decir, para poder usar el camino verde, es necesario hacer uso del camino rojo.

Camino seguido por una aplicación que haga uso de funcionalidad extra, este camino se apoya en los otros dos. Ya que para usar una funcion que haga algo extra es necesario que esta funcion haga uso de funciones ya implementadas por el fabricante (verde) y para que las funciones del fabricante funcionen necesita del sistema operativo , primitivas de comunicacion de este con el hardware, etc.

Introducción

Para entender qué es una DLL lo primero que debe tenerse claro es la diferencia entre vínculo dinámico (dll) y vínculo estático, y ésta es que en vínculo estático las librerias se asocian con el ejecutable en tiempo de compilación, es decir, que a la hora de crear el ejecutable en este se incluyen las funciones que va a usar, esto tiene dos puntos en contra del vinculo dinamico y son:

1.- No podemos distinguir codigo real del programa de codigo de la librería, ademas de la imposibilidad de poder modificar funciones erroneas en las librerias.

2.- Cada vez que se usa el programa se carga toda la librería en memoria, si hay varias ejecuciones del programa, la librería se carga completamente dentro del espacio de direcciones de cada uno de los procesos de ese programa.

Visto esto, las dlls deben verse como una alternativa eficaz a los dos problemas planteados anteriormente. Ya que no hay ninguna restriccion en cuanto a programas que puedan linkar un numero arbitrario de funciones de cualquier dll.

Windows reconoce solo un tipo de funcion DLL y dos formas de acceso de programas a dlls: durante el proceso de carga con el shell o durante la ejecución, llamando uno mismo a las correspondientes funciones de la API.

Las funciones API Win32 más importantes y su relacion con las DLLs se pueden sintetizar en la siguiente tabla (en los ejemplos de creacion de DLLs se puede ver para qué y cómo se usan):

Función	Descripción
LoadLibrary	Carga una DLL en el espacio de direcciones de una aplicación.
GetProcAddress	Suministra la direccion de una funcion dentro de una DLL.
FreeLibrary	Libera una DLL previamente cargada en el espacio de direcciones de una aplicación.
FreeLibraryAndExitThread	Libera la DLL finalizando el subproceso actual.
DLLEntryPoint	El punto de entrada que se va a ejecutar de la DLL una vez que esta se cargue.
DisableThreadLibraryCalls	Suprime el aviso de una funcion DLL tanto al conectar como al separar un subproceso
GetModuleFileName	Suministra el nombre del archivo DLL o EXE cargados.
GetModuleHandle	Suministra el manejador adecuado al modulo correspondiente.
LoadLibraryEx	Ampliacion de LoadLibrary()

Una aproximacion sobre cómo trabajar con DLLs

Según la estructura interna de un archivo DLL, y siempre que las circunstancias lo permitan, uno pretende ponerse al corriente de las conexiones, tanto de nuevos procesos, como de subprocesos a los archivos DLL y poder así preparar concienzudamente los recursos necesarios o ejecutar los trabajos de inicialización. La API Win32 ofrece esta posibilidad a partir de una funcion llamada "Entry". Cada DLL puede definir si así lo desea , una función de estas características. De hacerlo, el sistema reclamará la función DLL-Entry cada vez que se asigne o separe un nuevo proceso de la misma. Lo mismo sucederá cada vez que uno de los procesos conectados, cree o defina un nuevo subproceso. En estos casos el sistema no nos facilitará el nombre de la función DLL-Entry pero si su sintaxis :

```
BOOL WINAPI DllEntryPoint (HINSTANCE instanciaDLL, DWORD fwdCodigo, LPVOID lpReserved);
```

La orden de preparación (esto se verá casi al final del guión) de una funcion Entry con un determinado nombre, hay que pasarsela al ensamblador cuando se elabora el archivo DLL a partir de los diferentes archivos objeto. Para ello dispone de un atributo especial con el nombre "–entry:" que irá seguido del nombre de la función DLL-Entry.

```
link testdll.obj –out:testdll.dll testdll.exp –entry:NomreFuncionDLLEntry
```

En los diferentes entornos de desarrollo, como Borland C++ existen opciones para habilitar esto desde menus y no desde la linea de comandos.

Cuando se llama a la funcion DLL-Entry, aparece como primer parámetro el "Module Handle" que identifica su archivo DLL. En las siguientes llamadas, podrá incorporarse alguna de las funciones *module* de API win32, por ejemplo, al llamar a *GetModuleFileName()* . La segunda información que le llega a la función DLL-Entry es el inicio de su activación con el parámetro fwdCodigo. Este parametro sirve para activar una de las cuatro opciones de que se dispone (DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH, DLL_THREAD_DETACH), la práctica de todo esto se puede ver en el ejemplo que hay casi al final del guión bajo el epigrafe "Diseño de DllEntryPoint".

PRACTICANDO LA CREACION DE DLLs

Creación de la dll minimalista (la mas simple):

Versión 32 bits

1.- Pulsa en el boton de la barra de herramientas “New Project”, dentro del IDE de Borland C++. A continuación aparecerá el dialogo de “New Target”

2.- Introduce la ruta del proyecto y nómbralo Ejemplo1.

3.- Selecciona “Dynamic Library [.dll] como el tipo del proyecto.

4.- OK

5.- En la ventana de proyecto elimina los nodos .RC y .DEF.

6.- En el archivo .CPP teclea lo siguiente :

```
#include <windows.h>
extern "C"
void __declspec(dllexport) WINAPI TuFuncion ()
{
  MessageBox (NULL,
            "Estamos dentro de la function de tu dll",
            "Perifericos",
            NULL );
}
```

7.- Click el boton de Build Project (Compilar) y tu dll estará completa.

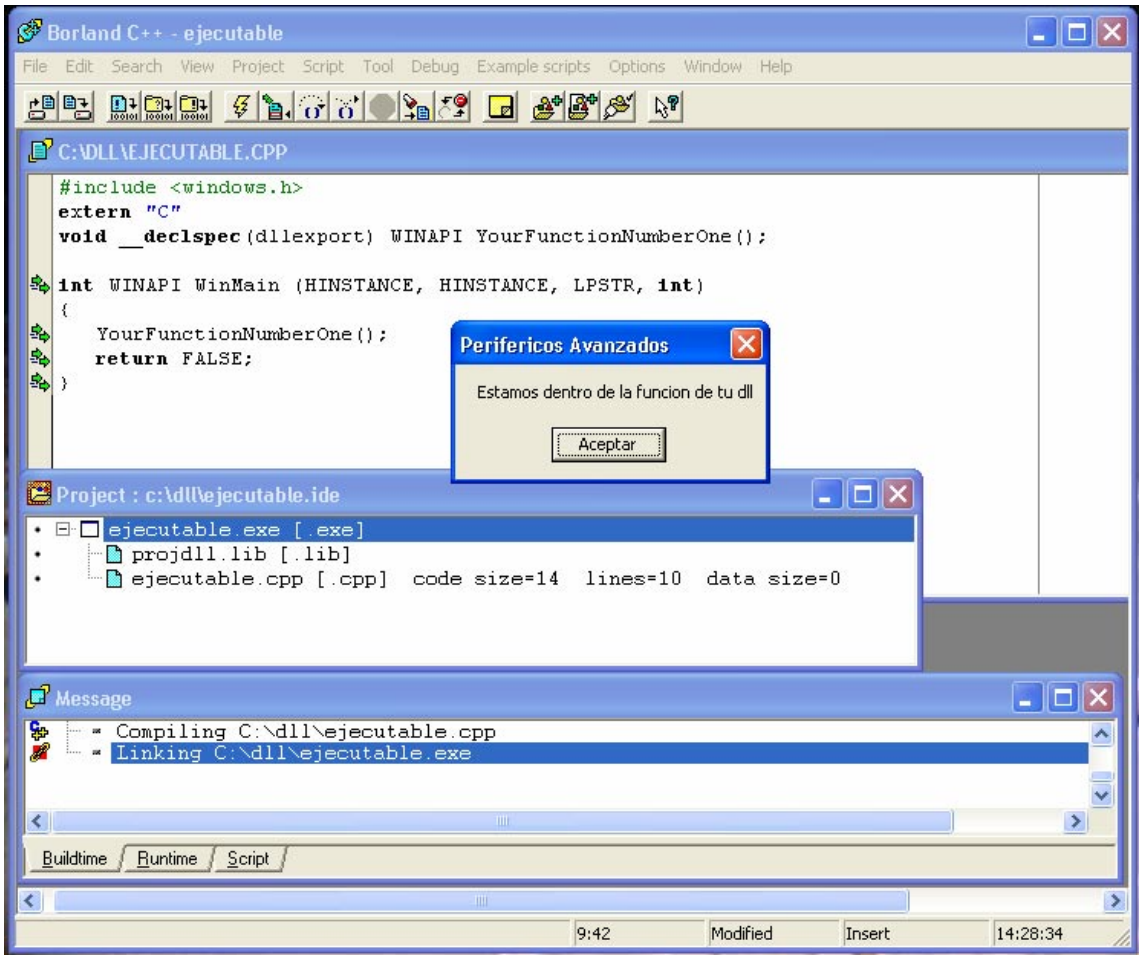
Para usar la dll desde un ejecutable …. :

1.- Cread un proyecto nuevo y seleccionad un NewTarget , pero ahora es un ejecutable.
2.- Introduce la ruta donde guardaras el proyecto y denominalo Ejemplo2
3.- Como la dll la creamos para 32 bits, tenemos que seleccionar en plataforma : win32.
4.- OK
5.- Borramos de nuevo los ficheros .RC y .DEF
6.- En el archivo CPP tecleamos lo siguiente :

```
#include <windows.h>
extern "C"
void __declspec(dllexport) WINAPI YourFunctionNumberOne ();

int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
  YourFunctionNumberOne ();
   return FALSE;
}
```

Veamos un ejemplo gráfico del resultado :



¿Extern C?

Omitir este modificador en la definición del prototipo de una función puede causar efectos frustantes. Si deseas que todo vaya sobre ruedas y no quieres tener mas complicaciones que las mínimas, entonces delate de cada definición de función (prototipo) deberemos teclear "extern c" esto hace que se desactive el "**name mangling**". Cuando se compila, se genera la codificacion de los metodos y funciones y junto a estos se genera información sobre el tipo de los argumentos, esto es lo que se ha dado en llamar " name mangling" , la sintaxis de mangling para los nombres de funcion es totalmente dependiente de la casa que desarrolla el compilador. Usando "extern C" prevenimos el name mangling y por lo tanto facilitamos la exportacion de funciones en las dlls, mas conciso, suprimiendo el name mangling otras aplicaciones pueden llamar a las funciones usando la misma sintaxis que aparece en el codigo de la dll.

En el ejemplo anterior, esta claro, que podemos suprimir extern C por que ambos programas (tanto la dll, como el exe) están desarrollados bajo Borland C++ 5.0.

__declspec(dllexport) y funciones de exportacion_

La forma mas sencilla de hacer una función dentro de una dll visible a otras aplicaciones es usar en los prototipos de las mismas los siguientes modificadores:

__declspec(dllexport) en el modo de 32 bits
__export en el modo de 16 bits

Por eficiencia, es aconsejable establecer las opciones de "Entry/Exit code" a "Windows DLL All functions Exportable" .

Datos

32 Bits :

La manera de exportar "datos" más sencilla es

```
// LaDLLenCuestion.cpp
int __declspec(dllexport) VariableGlobal;

//ElEjecutableQueLaUsa.cpp
extern int __declspec(dllimport) VariableGlobal;
```

Cuando se vaya compiler ElEjecutableQueLaUsa.exe hay que añadirle como se hizo en el ejemplo el fichero LaDLLenCuestion.LIB.

Clases

'__declspec(dllexport)' y '__export' pueden usarse tambien para exporter clases de una dll (entendiendo por clase a la definicion de un objeto). Veamos un ejemplo en 32 bits :

```
#include <windows.h>
class __declspec(dllexport) TuClase
{
 private:
         int AtributoUno;
 public:
         static int AtributoDos;
          int FuncionUno ();
};

int TuClase::AtributoDos;
int TuClase::FuncionUno ()
{
   AtributoUno = 55;
   return AtributoUno;
}
```

Ahora en el exe :

```
#include <windows.h>
class __declspec(dllimport) TuClase
{
 private:
         int AtributoUno;
 public:
         static int AtributoDos;
       int FuncionUno ();
};


int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
 TuClase A;
 char szMensaje[40];

 int i = A.FuncionUno ();
 wvsprintf (szMensaje, "El numero es %i", &i);
 MessageBox (NULL, szMensaje, "Perifericos", NULL);

 A.AtributoDos= 59;
 wvsprintf (szMensaje, "El numero es %i", &A.AtributoDos);
 MessageBox (NULL, szMensaje, "Perifericos", NULL);

 return FALSE;
}
```

Advertencia :

Las clases son una caracteristica de C++ que lleva implicita el “name mangling” por tanto para invocar funciones es necesario nombrarlas teniendo en cuenta esto, para este codigo no es necesario por que tanto la dll como el exe se han desarrollado con el mismo compilador.

WINAPI

En el archivo de cabecera windows.h para la versión de 32 bits, WINAPI se define como sigue :

#define WINAPI __stdcall

donde __stdcall es la “convencion” para llamar a funciones de la API de Windows 32, en ella está estipulado que los parámetros se insertan en el stack de derecha a izquierda, la funcion que ha sido invocada, limpia el stack al terminar su ejecución y el nombre de la funcion no se modifica en tiempo de compilación.

CARGA y LINKADO de DLLS

Vamos a ver tres metodos de linkar funciones que residen dentro de dlls (vease __declspec(dllexport) y __export para ver como linkar variables y clases exportadas de una dll:

1.- Librerias Import / Import Libraries :

Es el metodo seguido en los ejemplos y se trata de incluir en el proyecto del exe al fichero .LIB correspondiente a la DLL que contiene las funciones que nos interesan. Este metodo es el metodo más sencillo para linkar a una dll y a estas librerias .LIB se las llaman librerias de importación . Una librería de importación se puede crear usando la utilidad IMPLIB.EXE . Los .LIB de una DLL se generan directamente al compilar una DLL desde Borland C++ 5.0

2.- LoadLibrary y GetProcAddress

Las DLLs tambien las podemos cargar explícitamente en una aplicación usando la funcion de la API LoadLibrary, pero una vez que hemos cargado la DLL en el contexto de la aplicación que va a usarla tendremos que usar GetProcAddress para localizar la direccion de la funcion que queremos usar y que reside dentro de la DLL que hemos cargado .

Para aquellos que se aventuren a usar objetos en C++ y dlls deben saber antes de intentarlo que este metodo no se puede realizar con metodos que no sean estáticos o clases exportadas de DLLs la razon de esto es el puntero oculto “this” (no profundizaremos mas aquí por que no es motivo de perifericos instruir a nadie en la metodología orientada a objetos).

El ejemplo de dll del principio lo desarrollamos usando el primer metodo que hemos explicado, vamos a ver ese mismo ejemplo pero usando este metodo:

* Debeis eliminar el .LIB del proyecto del ejecutable.

```
#include <windows.h>

int PASCAL WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
  HINSTANCE instanciaDLL = LoadLibrary ("projdll.DLL");
  FARPROC lpYFNO = GetProcAddress (instanciaDLL,"nombreFuncion");
  if (lpYFNO != NULL) (*lpYFNO)();
  else
         MessageBox (NULL, "no funciona", NULL, NULL);
  FreeLibrary (hinstanciaDLL);
  return FALSE;
}
```

Si el modificador extern C no se ha usado en el prototipo de la función de la librería que estamos usando, entonces la llamada a GetProcess para esta funcion (tal y como se ha advertido a lo largo del guión) habrá de usar el nombre “mangled” :

```
FARPROC lpYFNO = GetProcAddress (hinstanciaDLL, "@nombreFuncion$qv");
```

3.- Ficheros de definicion de módulos

Cualquier programa windows necesita de un fichero .DEF (definicion de modulos), si no se lo dais vosotros al linkar windows toma uno por defecto. En el caso de Borland C++ (aprovecho la ocasión para aclarar que el guión referencia en todo momento a Borland por motivos didacticos ya que desde un primer momento se están usando sus herramientas) el fichero DEF por defecto es :

c:\ rutaDeVuestroBorland\LIBS\DEFAULT.DEF

Las funciones pueden ser exportadas/importadas en las aplicaciones sin mas que linkando vuestro propio fichero DEF y editando sus secciones de EXPORT e IMPORT (que son secciones usadas para saber que funciones se exportan y que funciones se importan), Exportar e importar usando esta técnica no es tan sencillo como en las anteriores , una de las utilidades o ventajas de usar esta tecnica es que se le pueden asignar “alias” a las funciones que exportas / importas.

Para importar funciones en vuestra aplicación :

a) Usa IMPDEF para encontrar el nombre con el que se está exportando la función que te interesa. Cuando una función está dentro de una dll y esta se declara exportable, entonces las formas de importarla pueden ser dos , la primera usando su nombre la segunda mediante un cardinal que la identifica dentro de la dll.

Ejemplo :

IMPDEF projdll.def projdll.dll

Esto generaría algo parecido a :

```
LIBRARY     PROJDLL
    DESCRIPTION 'PROJDLL.DLL'
    EXPORTS
        WEP                     @1
        _FuncionA         @2   ; FuncionA(char)
        FUNCIONB      @3   ; FuncionB(char)
       @FuncionC$qc    @4   ; FuncionC(char)
```

Por tanto usando la informacion de IMPDEF podemos modificar el fichero .DEF de la siguiente forma :

```
IMPORTS
  PROJDLL._FunctionA    ; FuncionA(char)  cdecl extern "C"
  PROJDLL.FUNCTIONB     ; FuncionB(char) PASCAL extern "C"
  PROJDLL.@FunctionC$qc ; FuncionC(char) cdecl mangled C++
```

O si se prefiere usar los numeros ordinales :

```
IMPORTS
  PROJDLL.2
  PROJDLL.3
  PROJDLL.4
```

Este método es bastante más eficaz.

En la seccion de Imports podemos establecer alias para evitar el mal trago de usar nombres « mangled ».

```
IMPORTS
        FuncionImportada1=PROJDLL._FuncionA
        FuncionImportada2=PROJDLL.FUNCIONB
        FuncionImportada3=PROJDLL.@FuncionC$qc
```

b) Finalmente añade el fichero .DEF a tu proyecto.

Para exportar funciones de la dll :

1.- Solo hay que usar simbolo __export para todas las funciones que deseas exportar y las opciones –WDE/-WCDE opciones de entrada / salida -> “Windows DLL Explicit Functions Exported”.

2.- Usad la seccion EXPORTS del fichero .DEF para asignarle alias a vuestras funciones si lo considerais necesario.

```
EXPORTS
  Leer1     =@__smleer$qv          @1
  Leer2     =@__smleer$qcc         @2
  Escribir1 =@__smescribir$qv      @3
```

Como antes, los alias aquí nos permiten una alternativa muy cómoda a los nombres "mangled".

NOTA : Exportar las funciones usando el ordinal reduce la memoria que necesita la DLL para cargarse.

LibMain, WEP, DLLEntryPoint

Borland C++ rellena estas funciones con funciones suyas por defecto, si vosotros no le proporcionais estas funciones.

LibMain y WEP son dos funciones muy apropiadas apara reservar y/o liberar memoria global para la DLL. LibMain tambien es muy buena para inicializar variables globales en tu DLL. Es necesario usar GlobalAlloc con el flag GMEM_SHARE cuando se reserve memoria para la DLL. Si se usa el operador de reserva de memoria de Borland "new" entonces es necesario retocar la variable "_WinAllocFlag" (para hacer esto os recomiendo que busqueis informacion en el documento "Understanding Memory Allocation Under Windows" de Borland C++) .

DLLEntryPoint reemplaza a ambas funciones WEP y LibMain en una DLL de 32 bits y consiste en un switch de cuatro opciones para manejar procesos y para adherir o separar al proceso de hebras de ejecucion :

Diseño de DllEntryPoint

```
BOOL WINAPI DLLEntryPoint (HINSTANCE instanciaDLL, DWORD fdwCodigo,  LPVOID  lpvReserved)
{
 switch (fdwCodigo) {
   case DLL_PROCESS_ATTACH:
     // La DLL se "mapea dentro del espacio de direcciones del proceso
     break;
    case DLL_THREAD_ATTACH:
     // Se crea una hebra de ejecucion.
     break;

   case DLL_THREAD_DETACH:
     // Finalizacion de una hebra
     break;

   case DLL_PROCESS_DETACH:
     // La DLL se retira del espacio de direcciones del proceso.
     break;
 }
 return(TRUE);
}
```

Win32 invoca a esta funcion siempre que una DLL se asigna a un proceso y siempre que se separa del mismo, como comentario de los argumentos cabe destacar que instanciaDLL es el manejador de la DLL (una referencia a la misma), instanciaDLL es la direccion virtual de memoria en la que se mapea la DLL en el espacio de direcciones del proceso.

Realización práctica :

1.- Implementar y probar la dll del ejemplo.

2.- Implementad una dll en la que se implemente , mediante el acceso a puertos , una función que lea el codigo scan pulsado por una tecla.

Ayudas: funciones inportb y outportb. La funcion es similar a la implementada en la dll pero en el texto ha de indicar el codigo scan pulsado.

Tiempo : 1 sesión.

Documentos interesantes para leer relacionados con el tema …….

Understanding Memory Allocation Under Windows

This document is a brief summary of how functions such as
malloc, calloc, and new allocate memory in a Windows application
created with Borland C++ 3.x

In all cases, new calls malloc to allocate the memory. What
happens next depends on the memory model your in and whether your
building a Dynamic Link Library (.DLL) or an Exectuable (.EXE).

If you are in the small or medium memory model building an .EXE
file, where data pointers default to near...

new calls malloc which calls LocalAlloc(LMEM_FIXED);
calloc calls LocalAlloc(LMEM_FIXED | LMEM_ZEROINIT);

If you are in the compact or large model building an .EXE file,
where data pointers default to far, malloc calls farmalloc...

new calls malloc which calls farmalloc which calls
GlobalLock(GlobalAlloc(GMEM_MOVEABLE | _WinAllocFlag));

calloc calls farcalloc which ultimately make a call to
GlobalLock(GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT |
_WinAllocFlag);

If you are in any memory model (Small, Medium, Compact, or Large)
building a .DLL file, all data pointers, by default, are far and
the procedure used by the Runtime Library to allocate memory is
exactly the same as a large model .EXE. Large model .EXE files
are discussed in the previous paragraph.

_WinAllocFlag is a WORD in the startup code which is always 0.
One could set it to GMEM_SHARE or whatever other flags are needed
before doing a new or malloc. To use it, first make an extern
for it in your program...

extern WORD _WinAllocFlag;

Some additional Notes, in Borland C++ 3.x, and in the Object
Windows Library for Borland C++ 2.0, Borland has implemented a
suballocation scheme for use with global allocations. This means
memory is allocated as discussed before, but if a memory request
is small, a pointer will be returned into a larger chunk. This
reduces the amount of selectors Windows has to allocate.
In Borland C++ 3.x, the size of the chunks the runtime library
allocates are always 4K of RAM. If you ask for more than 4K, the
request for memory goes straight to GlobalAlloc, and you can
expect an offset of 0 for your pointer.

In Borland C++ 3.x, if you link to the static libraries and you
set _WinAllocFlag to GMEM_SHARE before doing an allocation, we do
not use the sub allocator, your memory request maps directly to
GlobalAlloc. This is particular important when allocating memory
in a DLL which provides services to two or more applications.
By default memory allocated in a DLL belongs to the task (.EXE)
which called the DLL. The memory is automatically freed if the
task terminates without releasing the memory. If a DLL used by
more than one tasks allocates memory, the memory should not be
suballocated since depending on the order the tasks terminates,
memory still in use by the DLL and another task could be freed
when the task owning the memory terminates. Using the
GMEM_SHARE flag results in the memory being owned by the DLL.

Otro documento interesante, editado por microsoft sobre el uso del operador new está en:

Allocating Memory the Newfangled Way: The *new* Operator
http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/msdn_fangle.asp

(en esta URL hay muchos articulos más relacionados con el tema)

Libros

"Windows Programmer's Guide to DLLs and Memory Management"
Mike Klein, Sams Publishing, Carmel, IN, 1992

Borland C++ 4.5 "Programmer's Guide",
"Writing Dynamic-Link Libraries", p. 201-206
"Module Definition Files" .ff, p. 185-190.

Borland C++ 4.5 "User's Guide", p. 138-139, p. 143-146