Javier Garrido

Componente de acceso a la tarjeta SIM en Windows Mobile

El acceso a la tarjeta SIM en Windows Mobile es poco habitual, principalmente porque cuando se introduce en la PDA, Windows Mobile agrega automáticamente el contenido de la misma a Pocket Outlook. Pero pensemos que queremos hacer lo contrario, y además desde C#: copiar en una SIM limpia los contactos y mensajes, administrarla, y capturar a través de callbacks los mensajes que nos envía el sistema.

En este artículo vamos a desarrollar un componente que nos permita acceder a toda la información de la tarjeta SIM desde una aplicación Windows Mobile[1]. Tristemente, y por desgracia para muchos, .NET Compact Framework 2.0 ó 3.5 no tiene una librería administrada que lo haga. Sin embargo, Windows Mobile posee una API llamada **SIM Manager**, que a través de llamadas a código nativo mediante **PInvoke**, y traduciendo los tipos de datos a enviar y recibir por esas llamadas, nos va permitir realizar las operaciones.

Para ello, vamos a necesitar Visual Studio 2008, C#, el SDK de Windows Mobile 5 ó 6 para conocer a fondo el funcionamiento de SIM Manager, y ver los ficheros de cabecera de las funciones que contiene la API, en MSDN. Para realizar las pruebas, hemos utilizado una PDA HTC P3300 con Windows Mobile 5.

Pinvokando

Para todos aquellos a los que le suene raro esto de PInvoke, no es más que realizar llamadas desde un ensamblado .NET a funciones contenidas en una DLL que no ha sido desarrollada para .NET (si fuera un ensamblado de .NET, bastaría con agregarlo como referencia en nuestro proyecto). En muchos sitios, lo vamos a encontrar como "hacer llamadas desde código administrado a código no administrado". Esa DLL, en la mayoría de los casos, habrá sido desarrollada en C++.

Cuando se realizan llamadas a código nativo, el CLR localiza la DLL, la carga en memoria, calcula automáticamente las referencias a los argumentos y pasa el control a la función no administrada.

Sin irnos tan lejos, cuando nosotros necesitemos hacer llamadas a código nativo a través de PInvoke, lo primero que necesitaremos es conocer el nombre de la DLL, así como la lista de funciones y los parámetros de las mismas. Posteriormente, deberemos crear prototipos de funciones administradas que representarán a las funciones no administradas, y entonces ya podremos llamar a las funciones como si se tratara de funciones administradas corrientes. Resumámoslo en una lista de pasos, para que quede más claro:

1. Conocer el nombre de la DLL que contiene la función nativa que deseamos llamar.
2. Conocer el nombre de la función contenida en la DLL que vamos a llamar desde código administrado, así como sus parámetros.
3. Crear el prototipo de función administrada correspondiente.
4. Llamar a la función como si de una función administrada se tratara.

Los dos primeros pasos, así como el último, son evidentes. No lo es tanto el punto 3. Cuando se crea

Javier Garrido es Ingeniero Técnico en Informática de Gestión y Microsoft Certified Professional. Actualmente es profesor de la Universidad Europea Miguel de Cervantes de Valladolid.

1 Tendremos en cuenta que ya se ha introducido el PIN en la SIM por motivos de espacio, pues aunque existe una función de SIM Manager, nos extenderíamos innecesariamente.

un prototipo de función administrada, nos estamos refiriendo a mapear la función contenida en la DLL externa en una declaración de código manejado. Para ello, debemos declarar un método con ciertos modificadores y asociarle el atributo `DllImport`. En el caso de la API SIM Manager, tenemos una función llamada `SimInitialize` (ver tabla 1). La definición de mapeo de esta función sería:

```
[DllImport("cellcore.dll",
           SetLastError = true)]
 internal static extern int
  SimInitialize(
              uint dwFlags,
              IntPtr lpfnCallBack,
              uint dwParam,
              out IntPtr lphSim);
```

Listado 1

El atributo `DllImport` aplicado sobre la función nos permite aportar la información necesaria para llamar a una función localizada en un archivo DLL no administrado. Como mínimo, debemos suministrar el nombre de la DLL. En nuestro caso, las funciones de SIM Manager se encuentran en `cellcore.dll`. Se pueden establecer otros parámetros en el constructor de la clase `DllImportAttribute`, como por ejemplo el punto de entrada de la DLL (`EntryPoint`), que nos permite indicar un nombre diferente para función administrada, y que ésta "apunte" al nombre real de la función contenida en la DLL, o por ejemplo el parámetro `SetLastError`, que nos permite establecer que la API llamará al método `SetLastError`. Para que nos entendamos: si se produce algún error interno durante la ejecución de la función, podremos recuperar el código generado a través de otra llamada nativa a `GetLastError`.

Respecto a los modificadores asociados al prototipo de la función, destacamos el modificador `internal`, que hace que este prototipo sea visible a todo el ensamblado en el que se encapsula nuestro componente; el modificador `extern`, que indica que el método se implementa externamente; y por último, el modificador `static`: definir un método estático cuando se utilizan implementaciones externas es una restricción. Las funciones externas se mapean como métodos de clase y no de instancia.

SIM Manager

Si nos dirigimos directamente al contenido de MSDN, podremos encontrar toda la información sobre esta API, incluyendo las funciones, notificaciones, estructuras y constantes de error que nos proporciona. En la tabla 1 hemos detallado una descripción de algunas funciones. En nuestro caso, estas funciones son las que vamos a necesitar para "pinvokar" al código nativo y de esta forma acceder a la tarjeta SIM.

Respecto a las estructuras de datos, códigos de error y notificaciones, las iremos viendo poco a poco según vayamos desarrollando el artículo.

Creando el componente

Como nuestra idea principal es crear un componente de tal forma que podamos arrastrarlo a nuestros formularios de Windows Mobile, vamos a crear una clase en Visual Studio que herede directamente de `Component`. A mayores, se crearán las enumeraciones, delegados y estructuras de datos que se necesiten para la consecución del componente.

Para los que nunca hayan creado un componente en Visual Studio 2008 para .NET Compact Framework, vamos a detallar los pasos iniciales.

Lo primero, abrir Visual Studio, y lo siguiente elegir el tipo de proyecto, tal y como se muestra en la figura 1.

Función	Descripción
`SIMCALLBACK`	Prototipo de función de *callback* que SIM Manager utiliza para enviar notificaciones. En otras palabras: un delegado para capturar los mensajes que envía el sistema.
`SimDeinitialize`	Deinicializa la SIM, o lo que es lo mismo, libera el manejador asociado a la misma.
`SimDeleteMessage`	Función para eliminar un SMS almacenado en la SIM.
`SimDeletePhonebookEntry`	Función para eliminar un registro de la agenda de la SIM.
`SimGetDevCaps`	Función para recuperar las capacidades de la SIM.
`SimGetPhonebookStatus`	Función para recuperar el estado de la agenda de la SIM (posiciones utilizadas, total de posiciones…).
`SimGetPhoneLockedState`	Función para saber si la SIM está esperando algún PIN o PUK.
`SimGetRecordInfo`	Función para recuperar la información de un registro particular de la SIM.
`SimInitialize`	Función para inicializar la API. Como en muchas otras API, es necesario primero llamar a esta función para poder utilizar el resto de funciones.
`SimReadMessage`	Función para leer un SMS desde una posición en concreto de la SIM.
`SimReadPhonebookEntry`	Función para leer una entrada de contacto de la SIM.
`SimWriteMessage`	Función para escribir un SMS en la SIM.
`SimWritePhonebookEntry`	Función para escribir un contacto en la SIM.

Tabla 1. Funciones de SIM Manager

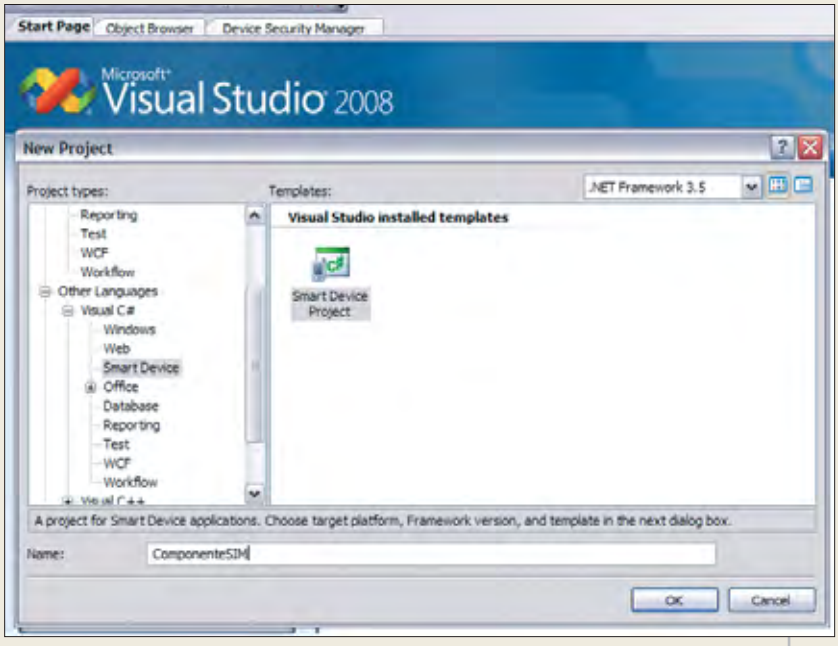


Figura 1. Iniciar proyecto Visual Studio

Ponemos un nombre a nuestro proyecto, en este caso `ComponenteSIM`, y pulsamos el botón "OK". A continuación, debemos elegir el tipo de proyecto que queremos desarrollar (.NET Compact Framework Forms, Class Library…). En nuestro caso, crearemos un proyecto de tipo Class Library, tal y como se muestra en la figura 2.

Una vez que tengamos iniciado nuestro proyecto, comenzaremos a introducir el código necesario para crear nuestro componente de acceso a SIM.

Como norma general, en mi caso, cuando necesito desarrollar algo que accede a código nativo, siempre creo una clase que va a contener todas las especificaciones de llamada a las funciones no administradas. El inconveniente puede aparecer ahora. ¿Cómo creamos esas funciones que acceden a código no administrado y cómo conocemos la firma de cada una de las funciones? Pues la respuesta es relativamente sencilla. En el SDK de Windows Mobile tenemos todos los ficheros de cabecera de las funciones. En el caso de la SIM, debemos buscar un fichero llamado `simmgr.h`, que encontraremos en el directorio donde tengamos instalado el SDK. Si editamos ese fichero, podemos ver los tipos de datos que reciben las funciones, lo que devuelven, las constantes con las que trabaja la API y, por supuesto, los nombres de las funciones. El problema es que esos tipos de datos son tipos usados por la API Win32, tal y como aparece en el listado 2 referente a la función `SimInitialize`, cuya descripción se puede ver en la tabla 1. Y ésa va a ser una de nuestras labores: traducir esos tipos de datos de la API Win32 a los tipos de .NET Compact Framework.

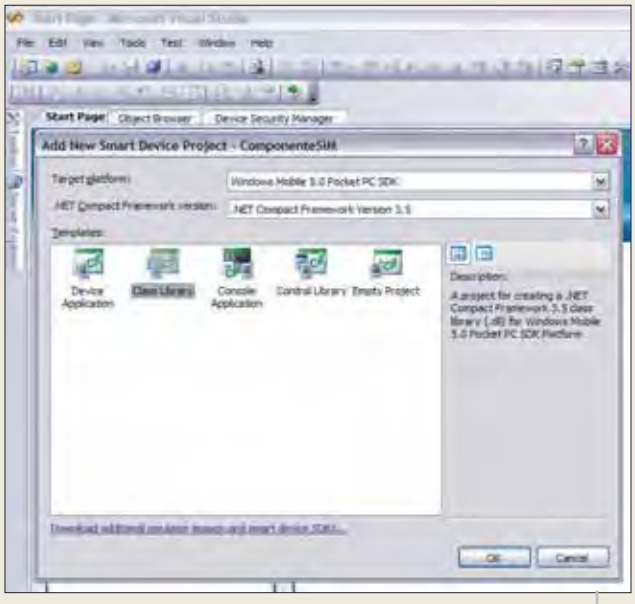


Figura 2. Seleccionar proyecto Class Library

```
HRESULT SimInitialize(
    DWORD dwFlags,
    SIMCALLBACK lpfnCallBack,
    DWORD dwParam,
    LPHSIM lphSim
);
```

Listado 2

La traducción sería tal y como aparece en el listado 3.

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace ComponenteSIM
{
  internal class FuncionesNativas
  {
    [DllImport("cellcore.dll",
               SetLastError = true)]
    internal static extern int
         SimInitialize(
            uint dwFlags,
            IntPtr lpfnCallBack,
            uint dwParam,
            out IntPtr lphSim);
  }
}
```

Listado 3

Seguro que muchos de ustedes estarán pensando ¿de dónde ha salido eso? ¿Por qué, si la función `SimInitialize` nece-

Tipo Nativo	Clase Administrada
HANDLE	System.IntPtr
BYTE	System.Byte
WORD	System.UInt16
LONG	System.Int32
DWORD	System.UInt32
LPSTR o LPCSTR o LPWSTR o LPCWSTR	System.String
FLOAT	System.Single
DOUBLE	System.Double

Tabla 2. Algunas equivalencias de tipos

sita un parámetro `DWORD`, en C# es un `uint`? Pues de MSDN, así de fácil. En la tabla 2 puede ver una referencia a los tipos más comunes usados en la API SIM Manager y su equivalente en .NET, que nos servirá de ayuda para entender más a fondo el desarrollo. Según vayamos desarrollando el artículo, iremos viendo otras equivalencias.

En realidad, no es tan sencillo, no les voy a engañar. No es solo utilizar la equivalencia de tipos de datos, sino saber cómo tenemos que llamar a esa función. Me explico: es muy común que en las llamadas a código nativo, lo que se necesite sea un puntero a un tipo de dato específico o recuperar un puntero a `void (void*)`, o lo que es lo mismo: un puntero a cualquier cosa. En función de lo que se necesite, debemos escribir la definición del método. Volvamos a la definición de `SimInitialize`. Si nos vamos a MSDN o al fichero de cabecera, y estudiamos la firma de la función, vemos que el primer parámetro de la función se utiliza para establecer los tipos de notificaciones que va a recibir nuestra clase, y que el segundo parámetro es un prototipo de función `SIMCALLBACK`. Este parámetro es un puntero a una función, que será el que utilizará el sistema operativo para enviarnos las notificaciones. Y un puntero a una función no es más que un delegado con una definición concreta. Esa definición no es más que otra traducción de los tipos de datos de la API Win32 a .NET. Estas notificaciones —algunas de ellas se pueden ver en la tabla 3— son mensajes que va a enviar el sistema operativo a nuestra clase.

El componente encapsulador de SIM Manager

Inicialización

Una vez "traducida" la definición de `SimInitialize` tal y como hemos mostrado en el listado 3, lo que tenemos que hacer es crear un prototipo de método que encapsule la llamada a la función y deposite el manejador obtenido en un campo privado, para que esté disponible para el resto de los métodos de la clase. El método `Inicializar` resultante se presenta en el listado 4.

La figura 3 muestra el diagrama de clase correspondiente al componente que vamos a desarrollar. A continuación, detallaremos la implementación de algunos de los demás métodos de la clase. El lector interesado puede descargar todo el

```
public void Inicializar()
{
  try
  {
    handleSIM   = IntPtr.Zero;
    int hresult = 0;
    hresult = FuncionesNativas.SimInitialize(
         SIM_INIT_NOTIFICATIONS.NONE,
         IntPtr.Zero, 0, out handleSIM);
    if (hresult != 0)
      throw new Exception(
     "No se ha podido inicializar la SIM");
  }
  catch (Exception ex)
  {
    throw ex;
  }
}
```

Listado 4

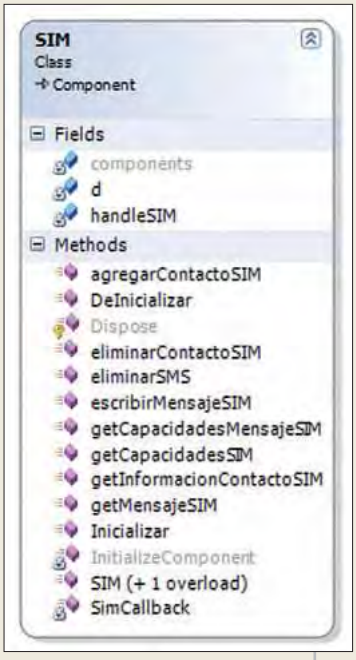


Figura 3. Diseño inicial del componente

Valor Notificación	Descripción
SIM_NOTIFY_CARD_REMOVED	La SIM se ha quitado del dispositivo.
SIM_NOTIFY_FILE_REFRESH	Los ficheros de la SIM han sido actualizados.
SIM_NOTIFY_MSG_STORED	Se ha almacenado un mensaje en la SIM.
SIM_NOTIFY_MSG_DELETED	Se ha eliminado un mensaje de la SIM.
SIM_NOTIFY_PBE_STORED	Un nuevo registro de contacto se ha agregado en la SIM.
SIM_NOTIFY_PBE_DELETED	Un registro de contacto se ha eliminado de la SIM.

Tabla 3. Algunos tipos de notificación enviados por Windows Mobile a SIM Manager

código fuente del componente del sitio Web de la revista, www.dotnetmania.com.

Con relación a los parámetros de la función, el primero de ellos se utiliza en este caso para determinar si la clase capturará mensajes enviados por el sistema operativo. Deberemos pasar el valor 1 para que se capturen, y 0 para indicar lo contrario. Para que quede más estructurado y más claro, he creado una enumeración con dos valores, como se muestra en el listado 5.

```
internal enum SIM_INIT_NOTIFICATIONS : int
{
  NONE  = 0,
  SIM_INIT_SIMCARD_NOTIFICATIONS = 0x00000001,
}
```

Listado 5

El segundo parámetro de SimInitialize se utiliza para especificar el método al que se deberá llamar cuando el sistema operativo envíe alguna notificación. Si en el primer parámetro hemos pasado un 0, este parámetro debe ser nulo. En caso contrario, deberá ser una instancia de delegado que apunte a un método. De momento, vamos a fijar que no nos envíe ninguna notificación, pasando IntPtr.Zero. Más adelante haremos los cambios necesarios para recibir las llamadas.

El tercer parámetro nos permite especificar un valor que se va a pasar en cada llamada de notificación. Aquí no lo utilizaremos, y por tanto hemos indicado un 0 en la llamada. Y por último, a través del cuarto parámetro nos llegará (observe el modificador out) el *handle* o manejador que identificará a la tarjeta SIM ante el sistema y que necesitaremos en las llamadas siguientes que hagamos.

Cabe destacar además que normalmente, todas las funciones de la API Win32 nos devuelven un valor de retorno que indica si la llamada ha sido satisfactoria. Como casi siempre, en el caso de SIM Manager, si la llamada a la función ha sido satisfactoria, se nos devuelve un valor 0. En caso contrario, lo que se nos devuelve es un código de error; los posibles valores están disponibles también en MSDN.

Deinicialización

Como es lógico, cuando finalicemos la utilización de la SIM, deberemos liberar los recursos utilizados por el sistema. Para ello disponemos de la función SimDeinitialize. La definición de esta función se presenta en el listado 6, y nuestro prototipo de función administrada en el listado 7.

```
HRESULT SimDeinitialize(
    HSIM hSim
);
```

Listado 6

```
[DllImport("CellCore.dll", SetLastError = true)]
  internal static extern int
        SimDeinitialize(IntPtr hSim);
```

Listado 7

Cuando se necesite deinicializar la SIM, llamaremos a esta función desde otra que encapsule el código correspondiente, como se muestra en el listado 8.

Recuperar las capacidades de la tarjeta SIM

Para saber cuántos números puede almacenar nuestra SIM, o la longitud máxima de caracteres para un nombre de un registro en la SIM, disponemos de la función SimGetDevCaps. Veamos su definición en el fichero simmgr.h.

```
HRESULT SimGetDevCaps(
    HSIM hSim,
    DWORD dwCapsType,
    LPSIMCAPS lpSimCaps
);
```

El primer parámetro a enviar a la función es el manejador asociado a nuestra SIM, mientras que el segundo parámetro se utiliza para decirle a la función qué tipo de capacidades queremos obtener. Sus posibles valores los encontramos en MSDN. Como en el resto de los casos, los he encapsulado en una enumeración, como se muestra en el listado 9.

```
public void DeInicializar()
{
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      int hresult = FuncionesNativas.
                    SimDeinitialize(handleSIM);
      if (hresult != 0)
        throw new Exception(
            "No se ha podido Deinicializar SIM");
    }
    else
    {
      throw new Exception(
          "La SIM no está inicializada");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
}
```

Listado 8

```
public enum CapsType
  {

    SIM_CAPSTYPE_PBENTRYLENGTH      = (0x00000001),
    SIM_CAPSTYPE_PBSTORELOCATIONS   = (0x00000002),
    SIM_CAPSTYPE_LOCKFACILITIES     = (0x00000004),
    SIM_CAPSTYPE_PBINDEXRANGE       = (0x00000008),
    SIM_CAPSTYPE_LOCKINGPWDLENGTHS  = (0x00000010),
    SIM_CAPSTYPE_MSGMEMORYLOCATIONS = (0x00000020),
    SIM_CAPSTYPE_ALL                = (0x0000003F),
  }
```

Listado 9

El último parámetro es un puntero a una estructura donde SIM Manager nos colocará la información que le estamos pidiendo. En este caso, podemos enviar un valor de tipo IntPtr que apunte a donde se encuentra la estructura en memoria, o enviar la referencia de la estructura. La definición original de dicha estructura, sacada del fichero de cabecera, se muestra en el listado 10.

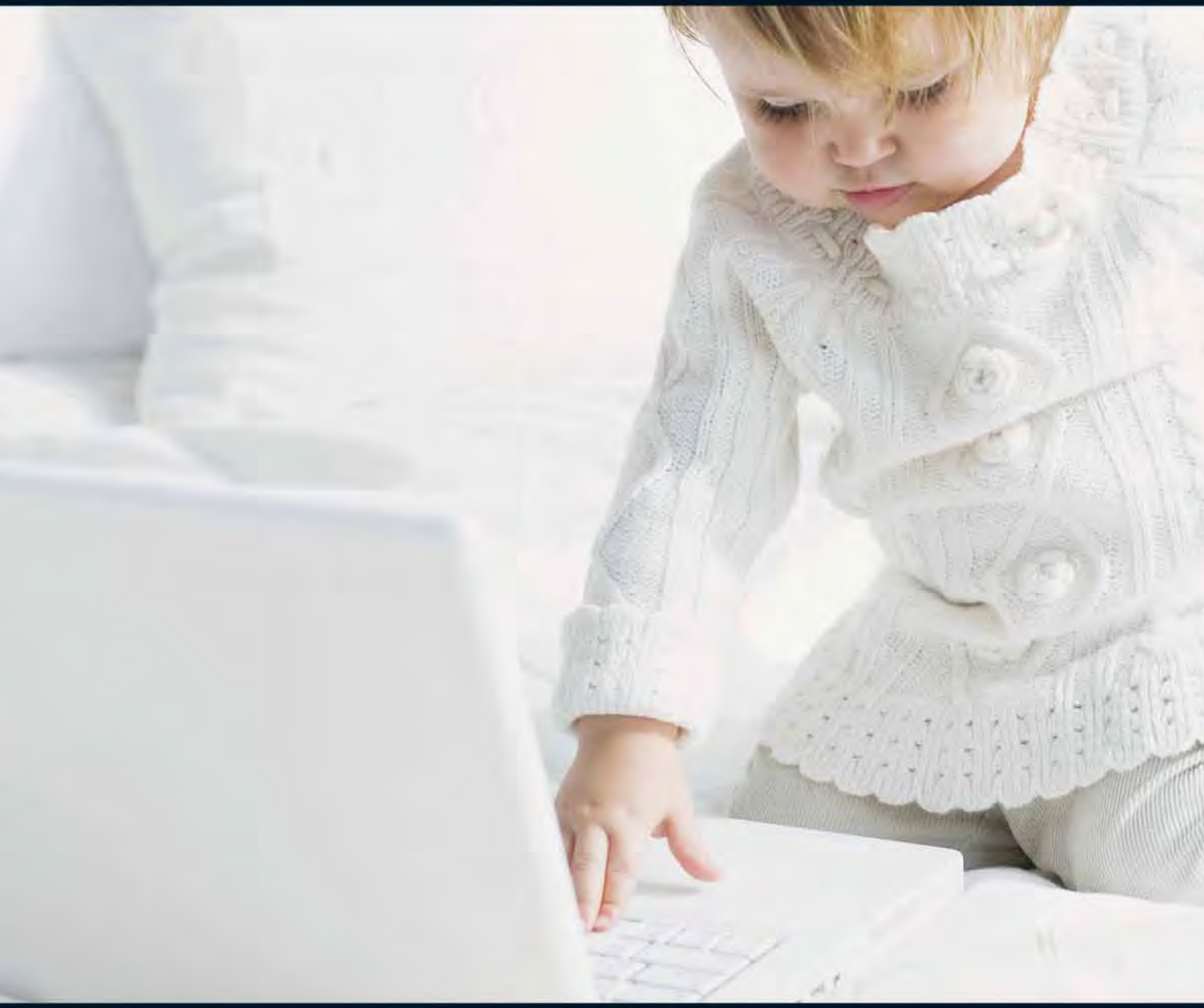
El último campo de esta estructura es un array de otra estructura llamada SIMLOCKINGPWDLENGTH, que contiene las longitudes mínimas de contraseñas (listado 11).

Su definición administrada sería como se muestra en el listado 12.

Somos compatibles.

La compatibilidad existe gracias a la comprensión. Toda relación funciona si hay entendimiento entre las dos partes. Nosotros conocemos la informática de la misma manera que conocemos a las personas. Las comprendemos a ambas. En Raona creemos que son tan importantes las personas como su trabajo. Somos una consultoría de software que se renueva constantemente para avanzar en consonancia con nuestros clientes. Un equipo de ingenieros que nos apasionamos con nuestro trabajo.

Raona. Pasión por el software.



www.raona.com
Tel.: + 34 902 362 918

raona
pasión por el software

```
typedef struct simcaps_tag {
 DWORD cbSize;
 DWORD dwParams
 DWORD dwPBStorages;
 DWORD dwMinPBIndex
 DWORD dwMaxPBIndex;
 DWORD dwMaxPBEAddressLength;
 DWORD dwMaxPBETextLength;
 DWORD dwLockFacilities;
 DWORD dwReadMsgStorages;
 DWORD dwWriteMsgStorages;
 DWORD dwNumLockingPwdLengths;
 SIMLOCKINGPWDLENGTH
    rgLockingPwdLengths
       [SIM_NUMLOCKFACILITIES];
} SIMCAPS, FAR *LPSIMCAPS;
```

Listado 10

```
typedef struct simlockingpwdlength
{
    DWORD dwFacility;
    DWORD dwPasswordLength;
}
```

Listado 11

```
/// <summary>
/// Estructura para almacenar los
/// valores mínimos de longitud
/// de password
/// </summary>
struct SIMLOCKINGPWDLENGTH
{
  public uint dwFacility;
  public uint dwPasswordLength;
}
```

Listado 12

```
struct SimCaps
{
  public int cbSize;
  public int dwParams;
  public int dwPBStorages;
  public int dwMinPBIndex;
  public int dwMaxPBIndex;
  public int dwMaxPBEAddressLength;
  public int dwMaxPBETextLength;
  public int dwLockFacilities;
  public int dwReadMsgStorages;
  public int dwWriteMsgStorages;
  [MarshalAs(UnmanagedType.ByValArray,
             SizeConst = 10)]
  public SIMLOCKINGPWDLENGTH
             rgLockingPwdLengths;
  }
```

Listado 13

Finalmente, en el listado 13 vemos la definición de la estructura `SimCaps` en código administrado.

Como el último parámetro es un array de estructuras, utilizamos el atributo `MarshalAs`, que nos permite indicar al CLR cómo calcular las referencias de tipos. En el listado 13 estamos indicando que se nos guarde la información en un array de 10 posiciones.

En el listado 14, vemos ahora el prototipo de la función `SimGetDevCaps`.

```
[DllImport("CellCore.dll", SetLastError = true)]
    internal static extern int SimGetDevCaps(IntPtr hSim,
      CapsType dwCapsType,
      ref SimCaps lpSimCaps);
```

Listado 14

Y el listado 15 muestra el cuerpo correspondiente a la función que encapsula la funcionalidad para recuperar las capacidades de la SIM.

Algo que no he comentado anteriormente, y que es también bastante común en las llamadas a la API de Windows: al pasar la estructura de datos donde se va a almacenar la información de la SIM, es necesario previamente asignar el tamaño en bytes de la estructura en el campo `cbSize`. Este tamaño se obtiene mediante la llamada `Marshal.SizeOf(typeof(SimCaps))`.

Agregar un contacto a la SIM

SIM Manager nos ofrece la función `SimWritePhoneBookEntry` para agregar contactos a la SIM. La definición en `simmgr.h` la podemos ver en el listado 16.

```
HRESULT SimWritePhonebookEntry(
    HSIM hSim,
    DWORD dwLocation,
    DWORD dwIndex,
    LPSIMPHONEBOOKENTRY
                lpPhonebookEntry
);
```

Listado 16

```
public SimCaps getCapacidadesSIM()
{
  SimCaps objcaps;
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      objcaps = new SimCaps();
      objcaps.cbSize = Marshal.SizeOf(typeof(SimCaps));
      int hresult = FuncionesNativas.SimGetDevCaps(handleSIM,
        CapsType.SIM_CAPSTYPE_ALL, ref objcaps);
       if (hresult != 0)
       {
         throw new Exception("No se ha podido recuperar las capacidades de la SIM");
       }
    }
    else
    {
      throw new Exception("La SIM no está inicializada.");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
  return objcaps;
}
```

Listado 15

El primer parámetro que necesita esta función es el mismo manejador que hemos venido usando hasta ahora. Con el segundo parámetro determinamos en qué lugar se debe almacenar el contacto. Como siempre, he encapsulado todos los posibles valores en la enumeración que se presenta en el listado 17.

```
public enum SIM_PBSTORAGE
  {
    SIM_PBSTORAGE_EMERGENCY     = (0x00000001),
    SIM_PBSTORAGE_FIXEDDIALING  = (0x00000002),
    SIM_PBSTORAGE_LASTDIALING   = (0x00000004),
    SIM_PBSTORAGE_OWNNUMBERS    = (0x00000008),
    SIM_PBSTORAGE_SIM           = (0x00000010)
  }
```

Listado 17

Con el tercer parámetro indicamos la posición (índice) donde queremos almacenar el contacto. Si queremos guardarlo en la primera posición disponible, debemos enviar como valor `0xffffffff`. Y el último parámetro es un puntero a una estructura `SimPhoneBookEntry`, cuya definición en `simmgr.h` vemos en el listado 18.

```
typedef struct simphonebookentry_tag {
    DWORD cbSize;
    DWORD dwParams;
    TCHAR lpszAddress[MAX_LENGTH_ADDRESS];
    DWORD dwAddressType;
    DWORD dwNumPlan;
    TCHAR lpszText[MAX_LENGTH_PHONEBOOKENTRYTEXT];
} SIMPHONEBOOKENTRY, *LPSIMPHONEBOOKENTRY;
```

Listado 18

El campo `cbSize` indica el tamaño de la estructura, y habrá que asignarlo antes de enviar la estructura a la función. `dwParams` indica cuál de los campos de la estructura contiene datos válidos. Se corresponde con una serie de valores constantes, como se puede apreciar en el listado 19.

```
public enum SimParamPBE
  {
    SIM_PARAM_PBE_ADDRESS      = (0x00000001),
    SIM_PARAM_PBE_ADDRESS_TYPE = (0x00000002),
    SIM_PARAM_PBE_NUMPLAN      = (0x00000004),
    SIM_PARAM_PBE_TEXT         = (0x00000008),
    SIM_PARAM_PBE_ALL          = (0x0000000f),
  }
```

Listado 19

`dwAddressType` y `dwNumPlan` pueden ser una serie de valores constantes. Definen el tipo de teléfono que se va a agregar y el esquema de numeración (conjunto de reglas usadas para generar números) respectivamente. En los listados 20 y 21 se pueden ver los posibles valores de estas constantes, encapsulados en enumeraciones. Y en los listados 22, 23 y 24 se muestran la estructura administrada, el prototipo de la función `SimWritePhoneBookEntry` y el código del método del componente que encapsula la operación.

```
public enum AddressType
  {
    SIM_ADDRTYPE_UNKNOWN       = (0x00000000),
    SIM_ADDRTYPE_INTERNATIONAL = (0x00000001),
    SIM_ADDRTYPE_NATIONAL      = (0x00000002),
    SIM_ADDRTYPE_NETWKSPECIFIC = (0x00000003),
    SIM_ADDRTYPE_SUBSCRIBER    = (0x00000004),
    SIM_ADDRTYPE_ALPHANUM      = (0x00000005),
    SIM_ADDRTYPE_ABBREV        = (0x00000006),
  }
```

Listado 20

```
public enum NumPlan
{
  SIM_NUMPLAN_UNKNOWN   = (0x00000000),
  SIM_NUMPLAN_TELEPHONE = (0x00000001),
  SIM_NUMPLAN_DATA      = (0x00000002),
  SIM_NUMPLAN_TELEX     = (0x00000003),
  SIM_NUMPLAN_NATIONAL  = (0x00000004),
  SIM_NUMPLAN_PRIVATE   = (0x00000005),
  SIM_NUMPLAN_ERMES     = (0x00000006),
}
```

Listado 21

```
public struct SimPhoneBookEntry
{
  public uint cbSize;
  public uint dwParams;
  [MarshalAs(UnmanagedType.ByValTStr,SizeConst = 256)]
  public string lpszAddress;
  public AddressType dwAddressType;
  public NumPlan dwNumPlan;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)]
  public string lpszText;
}
```

Listado 22

```
[DllImport("Cellcore.dll", SetLastError = true)]
  internal static extern int SimWritePhonebookEntry(
    IntPtr hSim,
    SIM_PBSTORAGE dwLocation,
    uint dwIndex,
    ref SimPhoneBookEntry lpPhonebookEntry);
```

Listado 23

```
public void agregarContactoSIM(string nombre, string telefono)
{
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      SimPhoneBookEntry entrada ;
      entrada               = new SimPhoneBookEntry();
      entrada.cbSize        = (uint)Marshal.SizeOf(typeof(SimPhoneBookEntry));
      entrada.dwAddressType = AddressType.SIM_ADDRTYPE_UNKNOWN;
      entrada.dwNumPlan     = NumPlan.SIM_NUMPLAN_TELEPHONE;
      entrada.lpszAddress   = telefono;
      entrada.lpszText      = nombre;
      entrada.dwParams      = SimParamPBE.SIM_PARAM_PBE_ALL;
      int hresult = FuncionesNativas.SimWritePhonebookEntry(handleSIM,
        SIM_PBSTORAGE.SIM_PBSTORAGE_SIM, 0xffffffff, ref entrada);
      if (hresult != 0)
      {
        throw new Exception("Se ha producido un error al agregar el contacto.");
      }
    }
    else
    {
      throw new Exception("La SIM no está inicializada.");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
}
```

Listado 24

```
public SimPhoneBookEntry getInformacionContactoSIM(uint pos)
{
  SimPhoneBookEntry infoContacto;
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      infoContacto = new SimPhoneBookEntry();
      int hresult = FuncionesNativas.SimReadPhonebookEntry(
        handleSIM,
        SIM_PBSTORAGE.SIM_PBSTORAGE_SIM,
        pos,
        ref infoContacto);
      if (hresult != 0)
      {
        throw new Exception(
          "Se ha producido un error al recuperar el contacto.");
      }
    }
    else
    {
      throw new Exception("La SIM no está inicializada.");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
  return infoContacto;
}
```

Listado 27

Leer un contacto

Al igual que en el apartado anterior, SIM Manager nos ofrece la función **SimReadPhonebookEntry** para acceder a los contactos almacenados en la SIM. En el listado 25 vemos su definición en **simmgr.h**.

```
HRESULT SimReadPhonebookEntry(
    HSIM hSim,
    DWORD dwLocation,
    DWORD dwIndex,
    LPSIMPHONEBOOKENTRY lpPhonebookEntry
);
```

Listado 25

Esta función es muy similar a **SimWritePhonebookEntry**. El primer parámetro es el manejador asociado a la SIM. Con el segundo determinamos de dónde queremos leer: de la SIM, números propios, etc. Para ser exactos, son los valores constantes que ya tenemos definidos en **SIM_PBSTORAGE**. El tercer parámetro define la posición del contacto que queremos obtener, y el último de todos es un puntero a la estructura **SimPhonebookEntry** que SIM Manager nos rellenará con la información del contacto.

En el listado 26 vemos el prototipo administrado de la función, y en el listado 27, la llamada desde el componente.

```
[DllImport("CellCore.dll",SetLastError = true)]
    internal static extern int SimReadPhonebookEntry(
      IntPtr hSim,
      SIM_PBSTORAGE dwLocation,
      uint dwIndex,
      ref SimPhoneBookEntry lpPhonebookEntry);
```

Listado 26

Eliminar un contacto

Para eliminar un contacto utilizaremos la función **SimDeletePhonebookEntry**. Su firma se muestra en el listado 28.

```
HRESULT SimDeletePhonebookEntry(
    HSIM hSim,
    DWORD dwLocation,
    DWORD dwIndex
);
```

Listado 28

Para que la función realice su acción correctamente, basta con enviarle el manejador, el lugar de donde queremos eliminar el contacto (que puede ser del contenedor de números propios o de todos los existentes en la SIM), y la posición a eliminar. En los listados 29 y 30 se presentan el prototipo administrado de la función y el método del componente que encapsula la operación, respectivamente.

```
[DllImport("CellCore.dll",SetLastError=true)]
    internal static extern int
          SimDeletePhonebookEntry(
              IntPtr hSim,
              SIM_PBSTORAGE dwLocation,
              uint dwIndex);
```

Listado 29

```
public void eliminarContactoSIM(uint pos)
{
 try
 {
  if (handleSIM != IntPtr.Zero)
  {
   int hresult =
       FuncionesNativas.SimDeletePhonebookEntry(
           handleSIM,
           SIM_PBSTORAGE.SIM_PBSTORAGE_SIM, pos);
   if (hresult != 0)
     throw new Exception(
           "No se ha podido eliminar el contacto.");
  }
  else
  {
   throw new Exception("La SIM no está inicializada.");
  }
 }
 catch (Exception ex)
 {
  throw ex;
 }
}
```

Listado 30

Escribir un SMS en la tarjeta SIM

Para esta funcionalidad, SIM Manager ofrece la función **SimWriteMessage**. La firma de la función se puede ver en el listado 31, y su prototipo administrado en el listado 32. Como antes, los tres primeros parámetros ya nos son conocidos; no así el último de ellos, que es nuevo para nosotros. Es un puntero a una estructura llamada **SimMessage** que es necesario rellenar con la información del mensaje a escribir en la SIM. En el listado 33 podemos ver la definición de la misma.

```
HRESULT SimWriteMessage(
    HSIM hSim,
    DWORD dwStorage,
    LPDWORD lpdwIndex,
    LPSIMMESSAGE lpSimMessage
);
```

Listado 31

```
[DllImport( "CellCore.dll", SetLastError = true)]
 internal static extern int SimWriteMessage(
         IntPtr hSim,
         SIM_SMSSTORAGE dwStorage,
         out uint lpdwIndex,
         ref SimMessage lpSimMessage );
```

Listado 32

```
typedef struct simmessage_tag {
    DWORD cbSize;
    DWORD dwParams;
    TCHAR lpszAddress[MAX_LENGTH_ADDRESS];
    DWORD dwAddressType;
    DWORD dwNumPlan;
    SYSTEMTIME stReceiveTime;
    DWORD cbHdrLength;
    BYTE rgbHeader[MAX_LENGTH_HEADER];
    TCHAR lpszMessage[MAX_LENGTH_MESSAGE];
} SIMMESSAGE, FAR *LPSIMMESSAGE;
```

Listado 33

De todos los campos de la estructura, nos interesa **lpszAddress**, que almacenará en un array el número de teléfono del que procede el SMS, **lpszMessage**, el contenido del mensaje, y **dwParams**, que se utiliza para indicar qué campos son válidos en la estructura. En el listado 34 se presenta la versión administrada de la estructura.

```
public struct SimMessage
{
  public uint cbSize;
  public Sim_Param_Msg dwParams;
  [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 256)]
  public string lpszAddress;
  public Sim_AddrType dwAddressType;
  public NumPlan dwNumPlan;
  public SYSTEMTIME stReceiveTime;
  public uint cbHdrLength;
  [MarshalAs(UnmanagedType.ByValArray,SizeConst = 256)]
  public byte[] rgbHeader;
  [MarshalAs(UnmanagedType.ByValTStr,SizeConst = 256)]
  public string lpszMessage;
}
```

Listado 34

Es importante comentar que esta función siempre almacena el SMS en la primera posición libre que encuentre. El parámetro **lpdwIndex** es un puntero a la posición en la que se ha agregado el mensaje. El parámetro **lpSimMessage** es un puntero a la estructura **SimMessage** que contiene la información del SMS a agregar a la SIM. En el listado 35 vemos el cuerpo del método del componente que encapsula la funcionalidad descrita. Observe también que en el listado 34 hay una estructura llamada **SYSTEMTIME**, que se utiliza para representar fechas y horas mediante elementos individuales. Su versión administrada se presenta en el listado 37, y en ella hemos redefinido el método **ToString** (como no podía ser de otra forma) para ofrecer una representación de cadena de la estructura.

```
public void escribirMensajeSIM(String telefono, String msg)
{
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      SimMessage sms = new SimMessage();
      sms.cbSize = (uint)Marshal.SizeOf(typeof(SimMessage));
      sms.dwAddressType = Sim_AddrType.SIM_ADDRTYPE_UNKNOWN;
      sms.dwNumPlan   = NumPlan.SIM_NUMPLAN_UNKNOWN;
      sms.lpszAddress = telefono;
      sms.lpszMessage = msg;
      sms.dwParams = Sim_Param_Msg.SIM_PARAM_MSG_ALL;
      SYSTEMTIME s = new SYSTEMTIME();
      s.day = 14; s.month = 4; s.year = 2008;
      s.hour = 22; s.minute = 30; s.second = 0;
      sms.stReceiveTime = s;
      uint pos;
      int hresult = FuncionesNativas.SimWriteMessage(
            handleSIM,SIM_SMSSTORAGE.SIM_SMSSTORAGE_SIM,
            out pos, ref sms);
      if (hresult != 0)
      {
        throw new Exception("No se ha podido agregar el SMS");
      }
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
}
```

Listado 35

```
HRESULT SimWriteMessage(
    HSIM hSim,
    DWORD dwStorage,
    LPDWORD lpdwIndex,
    LPSIMMESSAGE lpSimMessage
);
```

Listado 36

```
public struct SYSTEMTIME
{
  public short year;
  public short month;
  public short dayOfWeek;
  public short day;
  public short hour;
  public short minute;
  public short second;
  public short millisecond;
  public override string ToString()
  {
    DateTime d =
        new DateTime(year, month, day, hour, minute, second);
    return d.ToString();
  }
}
```

Listado 37

Leer un SMS de la tarjeta SIM

La función para leer un SMS (**SimReadMessage**) es similar a la de escribir. En el listado 38 podéis ver el proto-

```
[DllImport("CellCore.dll",SetLastError = true)]
    internal static extern int SimReadMessage(
      IntPtr hSim,
      SIM_SMSSTORAGE dwStorage,
      uint dwIndex,
      ref SimMessage lpSimMessage);
```

Listado 38

```
public SimMessage getMensajeSIM(uint pos)
{
  SimMessage sms;
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      sms = new SimMessage();
      sms.cbSize = (uint)Marshal.SizeOf(typeof(SimMessage));
      int hresult = FuncionesNativas.SimReadMessage(handleSIM,
            SIM_SMSSTORAGE.SIM_SMSSTORAGE_SIM, pos, ref sms);
      if (hresult != 0)
      {
        throw new Exception("No se ha podido leer el mensaje");
      }
    }
    else
    {
      throw new Exception("La SIM no está inicializada.");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
  return sms;
}
```

Listado 39

tipo de la función administrada, y en el listado 39, el código correspondiente a la función del componente.

Eliminar un SMS

Para eliminar un SMS de la SIM, solo necesitamos indicarle a SIM Manager el manejador, el lugar de donde debe eliminar el mensaje y la posición. En los listados 40 y 41 se presentan el prototipo de la función administrada y el encapsulado del método que implementa la funcionalidad, respectivamente.

```
[DllImport( "CellCore.dll", SetLastError = true)]
 internal static extern int SimDeleteMessage( IntPtr hSim,
                                    SIM_SMSSTORAGE dwStorage,
                                    uint dwIndex);
```

Listado 40

```
public void eliminarSMS(uint pos)
{
  try
  {
    if (handleSIM != IntPtr.Zero)
    {
      int hresult = FuncionesNativas.SimDeleteMessage(handleSIM,
        SIM_SMSSTORAGE.SIM_SMSSTORAGE_SIM,
        pos);
      if (hresult != 0)
      {
        throw new Exception("No se ha podido eliminar el SMS.");
      }
    }
    else
    {
      throw new Exception("La SIM no está inicializada.");
    }
  }
  catch (Exception ex)
  {
    throw ex;
  }
}
```

Listado 41

Recuperar la infomación del almacén de SMS

Nos puede ser de utilidad conocer la capacidad de nuestra tarjeta SIM, las posiciones ocupadas y libres que tiene, etc. Para ello disponemos de la función `SimGetSmsStorageStatus`, que nos rellena en dos parámetros de salida la información solicitada. En el listado 42 se presenta el mapeo de la función a un método administrado. En el parámetro `lpdwUsed` recibiremos el número de posiciones ocupadas, y en `lpdwTotal` el número de posiciones totales de la SIM.

```
[DllImport("CellCore.dll", SetLastError = true)]
    internal static extern int SimGetSmsStorageStatus(
      IntPtr hSim,
      SIM_SMSSTORAGE dwStorage,
      out uint lpdwUsed,
      out uint lpdwTotal);
```

Listado 42

Obtener los mensajes del Sistema Operativo

Desde el principio del artículo, hemos comentado que SIM Manager es capaz de capturar y notificarnos los cambios que se producen en la SIM. Para ello, debemos suministrar un método de respuesta en la función `SimCallback` cuando se inicializa la SIM. El listado 43 muestra el tipo del delegado que se debe registrar para capturar las notificaciones. En el listado 44 se presenta el esqueleto de un método privado que utilizaremos como destino de las notificaciones, y en el listado 45, la instanciación del delegado y asignación al campo de la clase en el que se almacenará. Finalmente, en el listado 46 se ofrece una nueva definición para la función `SimInitialize` y se muestra su llamada desde el método del componente.

```
//Delegado que se registrará para capturar
//las llamadas del sistema
internal delegate void SIMCallBackDelegate (
       uint dwNotifyCode,
       IntPtr pData,
       int dwDataSize,
       int dwParam);
```

Listado 43

```
private void SimCallback(uint dwNotifyCode,
       IntPtr pData, int dwDataSize, int dwParam)
{
        // Gestión de las notificaciones
}
```

Listado 44

```
public SIM()
{
  InitializeComponent();
  d = new SIMCallBackDelegate(SimCallback);
}
```

Listado 45

Las posibles notificaciones que podemos capturar son las que se presentan en el listado 47. Sus nombres son autoexplicativos y no requieren traducción.

```
//Cambiamos el prototipo
[DllImport("CellCore.dll", SetLastError = true)]
internal static extern int SimInitialize( SIM_INIT_NOTIFICATIONS dwFlags,
                                          SIMCallBackDelegate lpfnCallBack, uint dwParam,
                                          out IntPtr lphSim);
public void Inicializar()
{
  try
  {
    handleSIM = IntPtr.Zero;
    int hresult = 0;
    hresult = FuncionesNativas.SimInitialize(SIM_INIT_NOTIFICATIONS.SIM_INIT_SIMCARD_NOTIFICATIONS,
                                             d, 0, out handleSIM);
    if (hresult != 0)
       throw new Exception("No se ha podido inicializar la SIM");
  }
  catch (Exception ex)
  {
     throw ex;
  }
}
```

Listado 46

Es importante observar que uno de los parámetros del delegado **SimCallBackDelegate** es de tipo **IntPtr**. Esto lo hemos hecho así porque en el prototipo de la función nativa **SimCallback** (listado 48) es un puntero a **void**; para que nos entendamos, un puntero a cualquier cosa. En dependencia del tipo de notificación de que se trate, en **pData** recibiremos una estructura

```
public enum Sim_Notify
  {
    SIM_NOTIFY_CARD_REMOVED = (0x100),
    SIM_NOTIFY_FILE_REFRESH = (0x101),
    SIM_NOTIFY_MSG_STORED   = (0x102),
    SIM_NOTIFY_MSG_DELETED  = (0x103),
    SIM_NOTIFY_PBE_STORED   = (0x104),
    SIM_NOTIFY_PBE_DELETED  = (0x105),
    SIM_NOTIFY_MSG_RECEIVED = (0x106),
    SIM_NOTIFY_RADIOOFF     = (0x107),
    SIM_NOTIFY_RADIOON      = (0x108),
    SIM_NOTIFY_RADIOPRESENT = (0x109),
    SIM_NOTIFY_RADIOREMOVED = (0x10A),
  }
```

Listado 47

```
typedef void (*SIMCALLBACK)(
    DWORD dwNotifyCode,
    const void* pData,
    DWORD dwDataSize,
    DWORD dwParam);
```

Listado 48

```
internal struct SimPBEChange
{
  /// <summary>
  /// Aqui veremos el índice de
  /// la entrada
  /// añadida o eliminada
  /// </summary>
  public uint dwEntry;
  /// <summary>
  /// Nos indica donde se ha agregado
  /// o eliminado
  /// la entrada
  /// </summary>
  public SIM_PBSTORAGE dwStorage;
}
```

Listado 49

de un tipo u otro. Por ejemplo, si se recibe una notificación de tipo **SIM_NOTIFY_PBE_DELETED, pData** apuntará a una estructura de tipo **SIMPBECHANGE**. En el listado 49 se presenta el equivalente manejado de esta estructura, y en el listado 50, un ejemplo de cómo tratar las notificaciones de ese tipo. La gestión del resto de las posibles notificaciones es similar y se le deja al lector como propuesta.

El resultado final del proyecto entero se presenta en la figura 4.

```
private void SimCallback(Sim_Notify dwNotifyCode, IntPtr pData, int dwDataSize, int dwParam)
    {
      switch (dwNotifyCode)
      {
        case Sim_Notify.SIM_NOTIFY_CARD_REMOVED:
          break;
        case Sim_Notify.SIM_NOTIFY_FILE_REFRESH:
          break;
        case Sim_Notify.SIM_NOTIFY_MSG_STORED:
          break;
        case Sim_Notify.SIM_NOTIFY_MSG_DELETED:
          break;
        case Sim_Notify.SIM_NOTIFY_PBE_STORED:
          break;
        case Sim_Notify.SIM_NOTIFY_PBE_DELETED:
          {
            //Recuperar una estructura desde el Puntero
            SimPBEChange aux;
            aux = (SimPBEChange)Marshal.PtrToStructure(pData, typeof(SimPBEChange));
            //Lanzaremos un evento
            if (EntradaCambiadaEvent != null)
              EntradaCambiadaEvent(aux.dwEntry, aux.dwStorage);
          }
          break;
        case Sim_Notify.SIM_NOTIFY_MSG_RECEIVED:
          break;
        case Sim_Notify.SIM_NOTIFY_RADIOOFF:
          break;
        case Sim_Notify.SIM_NOTIFY_RADIOON:
          break;
        case Sim_Notify.SIM_NOTIFY_RADIOPRESENT:
          break;
        case Sim_Notify.SIM_NOTIFY_RADIOREMOVED:
          break;
        default:
          break;
      }
    }
```

Listado 50

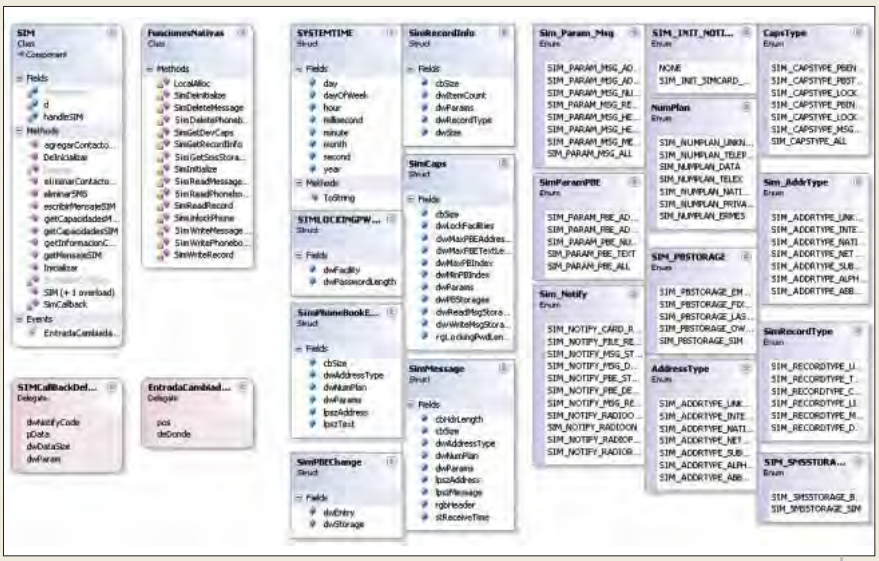


Figura 4. Diagrama final del componente

Posibles mejoras al componente

Como todas las cosas, el componente que hemos presentado aquí podría ser susceptible de mejoras y ampliaciones. Algunas ideas que me vienen a la mente son agregarle propiedades de tiempo de diseño al componente, dotarlo de un conjunto completo de eventos o permitir el acceso indexado al contenido de la SIM a partir de nombres de contactos además de posiciones. Desde el punto de vista de la funcionalidad, validar o cambiar el PIN, asociar al componente un icono de la barra de herramientas o controlar todos los posibles errores podrían ser extensiones potenciales. Todo ello se deja como ejercicio práctico al lector.