Creación de un componente Paso a Paso

Índice

Introducción. 3
Objetivo. 3
Creando la estructura 3
Dibujo 6
  Bordes. 7
  Borde y tamaño 7
  Color del borde y del interior. 11
Foco. 14
Controlando al ratón 16
  Entrada y salida. 16
Cambios de color. 20
Imágenes. 26
Texto. 33
Fuentes del texto. 37
Visibilidad de propiedades, métodos y eventos. 45
Respondiendo al teclado. 55
Alineación del texto. 56
'Clavos'. Propiedad Bitmap 65
Icono para la paleta de componentes. 66
Código Completo. 69

Introducción.

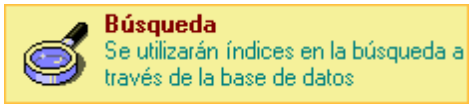
En una serie de artículos, que hoy empezamos, voy a explicar la creación de un componente (el componente es el que podéis encontrar en esta dirección web como SelPanel). Nosotros vamos a llamarlo PanelSel para que no interfiera con el que podemos tener ya instalado.

Objetivo.

En primer lugar debemos definir como queremos que se comporte nuestro nuevo componente.

Queremos un componente que cambie de color según posea o no el foco de la ventana o cuando el cursor del ratón se coloca sobre él. También debe responder a la pulsación del ratón o a la pulsación de la tecla Return o Barra Espaciadora con un evento OnClick.

Este componente podrá contener una imagen y dos tipos de texto, uno superior de una sola línea y otro posterior de varias líneas que se ajustará al tamaño del control en la ventana. En definitiva lo que queremos gráficamente es lo siguiente :



Creando la estructura

Lo primero que debemos hacer es crear la estructura de nuestro componente, para ello podemos heredar de cualquier otro de los ya definidos en delphi dependiendo de lo que queremos que nuestro componente realice, en nuestro caso vamos a partir de TCustomControl puesto que este componente puede responder a la pulsaciones de teclado y además posee un Canvas (lienzo), que vamos a necesitar para dibujar todo el contenido del componente (imagen, textos,..)

Nuestro componente vamos a crearlo dentro de un paquete (en el que más adelante podremos introducir otras definiciones de nuevos componentes). Para ello vamos al menú File--->New--->Other--->Package. Esto abrirá una ventana que representa nuestro paquete, lo guardamos en un directorio de disco (C:\ejemplo).

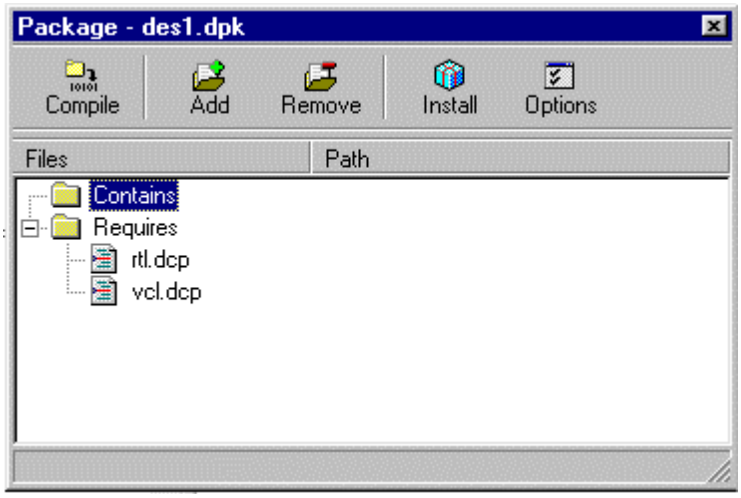
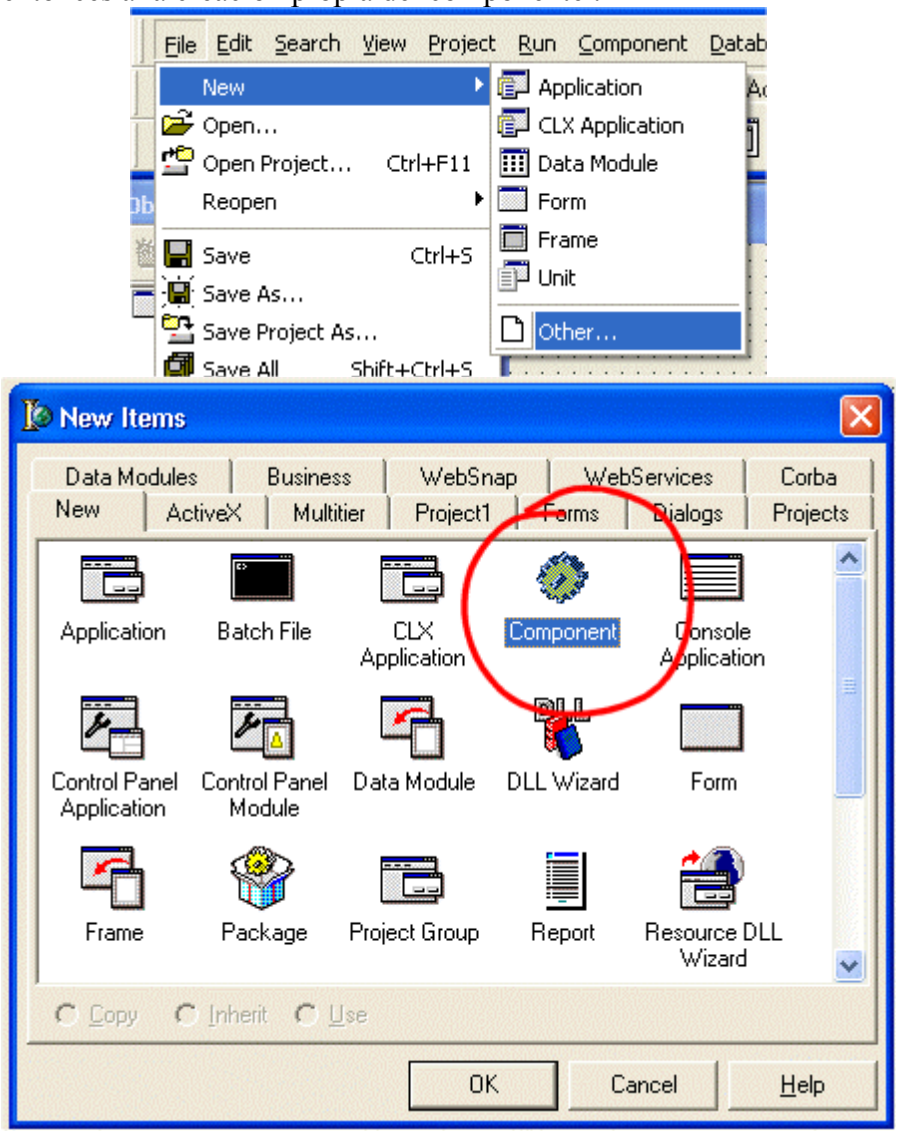
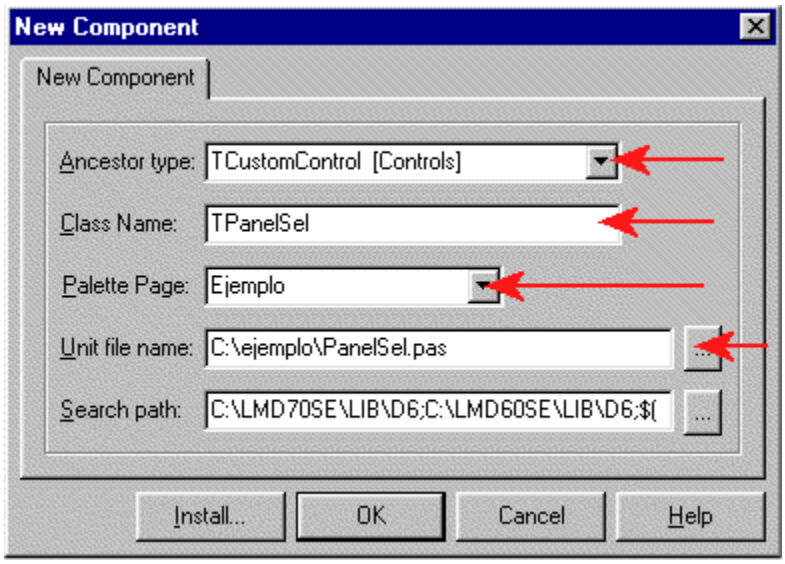


Figura 1

Pasamos entonces a la creación propia del componente :





Figura(s) 2

En la ventana que aparece (la figura anterior) cambiamos los valores que aparecen por defecto por los mostrados en la figura. Pulsamos OK. y aparecerá el siguiente texto :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TPanelSel = class(TCustomControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Este es el esqueleto básico de nuestro componente. Si pulsamos instalar (figura 1) veremos que aparece una carpeta nueva en la zona de componentes (al final) en la que hay un icono nuevo.

Si seleccionamos este icono y dibujamos sobre un form aparecerá un recuadro que se puede seleccionar Si se ejecuta el form veremos que en el mismo no aparece nada.

Dibujo

El primer paso será entonces mostrar algo en ese recuadro, para ello vamos a **sobreescribir** el procedimiento Paint (este procedimiento se encuentra el alguno de los ancestros de los cuales nuestro componente hereda propiedades y métodos. Sobreescribir significa volver a definir el comportamiento):

```
procedure Paint; override;
```

Simplemente vamos a hacer que dibuje un rectángulo que ocupe todo el área del control. El resultado del código es :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TPanelSel = class(TCustomControl)
  private
    { Private declarations }
  protected
    procedure Paint; override;
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure TPanelSel.Paint;
begin
   Canvas.Rectangle(ClientRect);
end;

procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Bordes.

Borde y tamaño

Con lo hecho hasta ahora hemos conseguido que nuestro control sea un rectángulo en el form. Lo siguiente que vamos a hacer es conseguir quitar o poner bordes a nuestra voluntad así como un ancho para esos bordes. Para ello deberemos guardar en algún sitio dentro del componente si queremos borde y el tamaño de éste, esto lo vamos a conseguir añadiendo dos variables al componente y dos propiedades que son las que van a modificar esas variables:

```
TPanelSel = class(TCustomControl)
private
  FBorder:Boolean;
  FBorderWidth:Integer;
  { Private declarations }
protected
...
```

Como podéis comprobar las variables/atributos que manejará el componente se definen en la parte privada del mismo, esto permitirá que sólo el propio componente pueda modificar sus valores. ¿Quién se encarga de modificar estos valores? El valor de estas variables se modificarán o a través de las propiedades definidas en el componente o mediante métodos o funciones.

```
...
published
   property Border:Boolean read FBorder Write FBorder default True;
   property BorderWidth:integer read FBorderWidth
            Write FBorderWidth default 1;
   { Published declarations }
...
```

Observad, en el código anterior, que la propiedad Border lee su valor de FBorder y escribe valores en FBorder, lo mismo ocurre con la propiedad BorderWidth y la variable FBorderWidth.

Si compilamos el paquete y después selecionamos nuestro componente de la paleta ejemplo y dibujamos un rectángulo en un form vemos que en pantalla aparece el mismo rectángulo, pero que en el inspector de objetos aparecen dos nuevas propiedades, las definidas por nosotros.

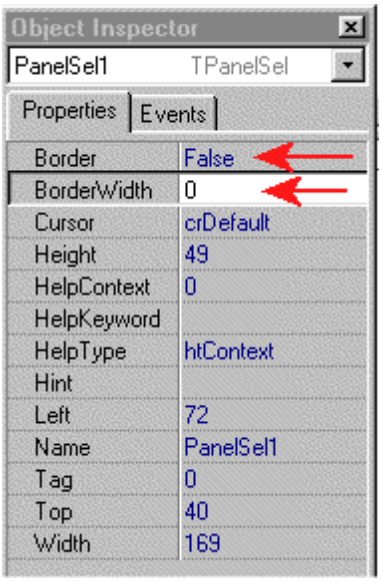


Figura 3

Pero pasa algo, si os fijáis en la definición de las propiedades habíamos puesto que por defecto el borde deberá ser true y el ancho 1, pero vemos que las propiedades son false y 0 y a pesar de ser 0 el valor del ancho del borde, el control se ha dibujado con borde.

La cláusula **default** de la definición de las propiedades no implica que éstas se inicien a ese valor, sino que está indicando que cuando se guarde la definición del componente en un fichero (.dfm por ejemplo) no guarde aquellas propiedades que tengan "su valor por defecto", ya que éste debería asignarse en la **creación** del componente, lo que implica que deberemos asignarles nosotros esos valores por defecto. ¿Dónde hacemos esto? Pues en el **Constructor** del componente. Todo componente tiene un método **Create** que es el encargado de crear el objeto. En nuestro caso, como no habíamos definido ninguno, el componente ha utilizado el de su ancestro (TCustomControl), pero ahora deberemos sobreescribir este método Create para asignar valores a las variables del componente que queramos tengan un valor inicial :

```
  ...
  public
     constructor Create(AOwner:TComponent);override;
          ...
          ...
implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited; // Hace que se llame al método Create del ancestro
(TCustomControl)
   FBorder:=True;
   FBorderWidth:=1;
end;
```

Si compilamos el paquete y dibujamos un nuevo control veremos en el inspector de objetos que ahora sí que tienen el valor inicial que deben tener, pero si cambiamos estos valores en el form no se ve reflejado nada y el control sigue apareciendo con borde de tamaño 1. Como es lógico no se hace nada porque no se le ha dicho qué tiene que hacer cuando cambie el valor de estas propiedades, para conseguir lo que queremos debemos realizar unos cambios en la definición de las propiedades :

```
  published
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth Write
SetBorderWidth default 1;
    { Published declarations }
```

Ahora las propiedades leerán el valor de las variables directamente, pero a la hora de cambiar el valor de éstas deberán utilizar dos procedimientos, estos los definiremos en la parte privada del componente :

```
  private
    FBorder:Boolean;
    FBorderWidth:Integer;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    { Private declarations }
```

Estos procedimientos tienen como parámetro un valor que es del mismo tipo que la variable que pretenden modificar. Su implementación es :

```
procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
```

Hay varias cosas comunes en ambos procedimientos, por una parte comprobar que las variables no contienen ya el mismo valor que el que se quiere cambiar, asignar el valor

e **invalidar** (invalidate) el control, invalidar el control es lo que va a indicar que el control debe redibujarse.

Por otra parte en estos procedimientos podemos *controlar* los valores que permitimos para las propiedades/variables del mismo, en el caso del Borde sólo permitimos valores mayores que 0.

Además de esto deberemos decirle que es lo que debe dibujar dependiendo del valor de estas propiedades, para ello cambiamos el procedimiento Paint visto en el capítulo anterior :

```
procedure TPanelSel.Paint;
begin
   Canvas.Pen.Width:=BorderWidth;
   if Border then
      Canvas.Rectangle(ClientRect)
   else
      Canvas.FillRect(ClientRect);
end;
```

El lienzo (Canvas) contiene dos objetos fundamentales para el dibujo : Pen y Brush, pluma y brocha. En el procedimiento anterior cambiamos el tamaño de la pluma al tamaño que nos indica la propiedad del componente y después dibujamos o no el borde (Rectangle y FillRect).

Podemos comprobar al cambiar los valores de las propiedades Border y BorderWidth que el control cambia en el form, pero hay un pequeño detalle, dependiendo del valor que se le ponga al tamaño del borde este se dibuja mal. Esto es debido a que el canvas sólo puede dibujar dentro del área cliente (ClientRect) del control, entonces ajusta el centro del ancho del pluma (Pen) con el borde del control, con lo que sólo dibuja la mitad del grosor de la pluma. Para solucionar esto definimos el método Paint de la siguiente forma:

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      Pen.Width:=BorderWidth;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      FillRect(ClientRect);
      if Border then Rectangle(X, Y, X + W, Y + H);
   end;
```

```
end;
```

O sea, que desplazamos los bordes hacia dentro del control dependiendo del tamaño de la pluma (Pen).

Color del borde y del interior.

Para cambiar el color del borde y del interior deberemos definir variables nuevas y propiedades para modificarlas. Siguiendo los pasos anteriores tenemos el código completo del componente, después de los cambios-en azul-, sería :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FBorder:Boolean;
    FBorderWidth:Integer;
    FColor:TColor;
    FBorderColor:TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetColor(Value:TColor);
    procedure SetBorderColor(Value:TColor);
    { Private declarations }
  protected
    procedure Paint; override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    { Public declarations }
  published
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth Write
SetBorderWidth default 1;
     property Color:TColor read FColor Write SetColor default
clBtnFace;
     property BorderColor:TColor read FBorderColor Write
SetBorderColor default clBlack;
    { Published declarations }
  end;

procedure Register;

implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FBorder:=True;
   FBorderWidth:=1;
   FColor:=clBtnFace;
```

```
   FBorderColor:=clBlack;
end;
procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColor(Value:TColor);
begin
   if FColor<>Value then
   begin
      FColor:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetBorderColor(Value:TColor);
begin
   if FBorderColor<>Value then
   begin
      FBorderColor:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      // Pluma
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      // Brocha
      Brush.Color:=Color;
      Brush.Style:=bsSolid;  //Relleno Sólido
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      FillRect(ClientRect);
      setbkmode(Handle,TRANSPARENT);
      Brush.Style:=bsClear;  //Relleno Sólido
      if Border then Rectangle(X, Y, X + W, Y + H);
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
```

```
end;

end.
```

La llamada `setbkmode(Handle,TRANSPARENT)`, hace que determinadas funciones de dibujo no 'dibujen' el relleno como es el caso de Rectangle, es decir, que sea transparente (siempre y cuando el estilo de la brocha no sea Sólido).

Foco.

En este apartado vamos a ver como conocemos cuando nuestro control recibe el foco de la aplicación (Focus) y como debemos reaccionar ante la consecución o pérdida del mismo en cuanto a su resultado en pantalla.

Uno de los ancestros de nuestro componente, en concreto TControl, posee un método denominado Focused que nos indica si el control posee el foco, pero nosotros necesitamos que se nos avise en el momento de conseguirlo o perderlo para no estar constantemente preguntando a este método. En este punto entran en juego los mensajes que se producen a consecuencia de ciertos eventos. Los mensajes que nos servirán para saber cuando debemos actuar son : WM_SETFOCUS, WM_KILLFOCUS que nos dicen cuando adquirimos y perdemos el foco. Debemos capturar estos mensajes para ello definimos los siguientes procedimientos en la parte protegida (Protected) de nuestro componente :

```
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    ...
```

El control en el momento que reciba cualquiera de los mensajes, reaccionará con la ejecución de estos procedimientos. Ahora, ¿qué es lo que debemos hacer nosotros ante la llegada de estos mensajes?, pues simplemente invalidar el control para que se vuelva a dibujar en la pantalla y dentro del método Paint preguntaremos a Focused para saber si poseemos el foco o acabamos de perderlo. En caso de que no hubiéramos tenido este método, podríamos haber definido una variable FFocused que pondríamos a true en el procedimiento WMSetFocus y a false en WMKillFocus.

```
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;
```

En el método paint de momento para comprobar que esto que hemos hecho funciona, haremos que escriba Focus dentro del control cuando este tenga el foco de la aplicación

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Color:=Color;
      Brush.Style:=bsSolid;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if focused then TextOut(0,0,'FOCUS');
      if Border then Rectangle(X, Y, X + W, Y + H);
   end;
end;
```

Compruebe el resultado creando un form e incluyendo varios controles de este componente. Ejecute. Verá que mediante las teclas del cursor y la de tabulación consigue cambiar el foco. Pero ¿qué ocurre con el ratón que no responde?. en ningún momento le hemos indicado que debe hacer cuando se pulsa con el ratón sobre el control. De momento en nuestro caso sólo queremos que el control sobre el que se pulsa obtenga el foco.

Para hacer esto, podemos o capturar los mensajes del ratón o podemos sobreescribir un método de uno de los ancestros : Click (como podéis ver la herencia nos ahorra mucho trabajo), el código de este método va a ser una llamada para que ejecute el código del ancestro y después asignaremos el foco al control :

```
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure Paint; override;
    procedure Click;override;
   ...
   ...
procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
```

Vuelva a ejecutar el programa y verá que el foco cambia también cuando se pulsa con el ratón.

Controlando al ratón

Entrada y salida.

Hasta ahora el único control del ratón que tiene nuestro componente es la respuesta a la pulsación del mismo. Como recordaréis lo que nos habíamos propuesto era que el control cambiara de color según tuviera el cursor del ratón encima o no.

Al igual que con el foco, su cambio provoca una cadena de mensajes, cualquier movimiento del ratón provoca también una serie de mensajes. A nosotros nos interesa capturar aquellos que nos informen sobre si el cursor del ratón se encuentra sobre el control o no, estos mensajes son : CM_MOUSEENTER, CM_MOUSEENTER, como es fácil ver uno nos avisará sobre la entrada del cursor en el control y otro sobre la salida.

```
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    ...
```

Estos mensajes nos avisan, pero el método Paint al ejecutarse, no sabe si el ratón está o no sobre el control, por esto deberemos guardar en algún sitio estas situaciones, por lo cual vamos a crear una variable/atributo que llamaremos FOver (en la parte privada de nuestro componente) que se pondrá a true cuanto se ejecute CMMouseEnter y a false cuando se ejecute CMMouseLeave. En el constructor Create la iniciamos a false.

```
procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  FOver:=True;
  Invalidate;
end;
procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  FOver:=False;
  Invalidate;
end;
```

en el método paint, para comprobar que funcionan estas modificaciones escribiremos Over cuando se encuentre el ratón sobre el control.

Hasta el momento el código del componente es el siguiente :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FBorder:Boolean;
    FBorderWidth:Integer;
    FColor:TColor;
    FBorderColor:TColor;
    FOver:Boolean;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetColor(Value:TColor);
    procedure SetBorderColor(Value:TColor);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    procedure Paint; override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    { Public declarations }
  published
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth Write
SetBorderWidth default 1;
     property Color:TColor read FColor Write SetColor default
clBtnFace;
     property BorderColor:TColor read FBorderColor Write
SetBorderColor default clBlack;
     property Tabstop;
    { Published declarations }
  end;

procedure Register;

implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FOver:=False;
```

```
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColor:=clBtnFace;
   FBorderColor:=clBlack;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  FOver:=False;
  Invalidate;
end;


procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColor(Value:TColor);
begin
   if FColor<>Value then
   begin
      FColor:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetBorderColor(Value:TColor);
begin
```

```
   if FBorderColor<>Value then
   begin
      FBorderColor:=Value;
      Invalidate;
   end;
end;
procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Color:=Color;
      Brush.Style:=bsSolid;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if focused then TextOut(0,0,'FOCUS');
      if Border then Rectangle(X, Y, X + W, Y + H);
      if FOver then TextOut(0,TextHeight('FOCUS')+2,'OVER');
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Cambios de color.

Vamos a hacer que nuestro componente una vez en el form (con lo que pasará a ser un control) cambie de color cuando se sitúe sobre él el ratón o cuando tenga el foco, es decir, tendremos tres estados posibles: Normal, Enfocado y con el Cursor del ratón sobre el control. Definiremos 2 colores por cada estado uno para el borde y otro para el fondo del control, es decir tendremos que guardar en el componente 6 colores para lograr lo que queremos hacer. Las propiedades vamos a llamarlas : Color, BorderColor para el estado normal, FocusedColor, FocusedBorderColor cuando tiene el foco y OverColor, OverBorderColor cuando tiene situado el ratón sobre él.

Es decir deberemos tener 6 variables que guarden estos valores de propiedades. Por otra parte cuando cambie el valor de estas propiedades (por ejemplo en el inspector de objetos cambiamos cualquiera de ellos), se deberá reflejar de forma automática en la ventana, como ya hemos dicho en múltiples ocasiones para que esto ocurra deberemos escribir un procedimiento Set*Propiedad*(Value) para cada propiedad. Pero en este caso observamos una cosa, por una parte queremos mantener todas las propiedades de color mencionadas anteriormente pero por otra parte vemos que el tratamiento para cada una de ellas es el mismo :

```
Procedure Set Propiedad(Value:TColor)
begin
   if value<>propiedad then
   begin
      propiedad:=value;
      invalidate;
   end;
end;
```

¿Qué hacer para no duplicar código? La solución es crear un array de tipo TColor para guardar los 6 colores, y asignar a cada una de las filas del array una de las propiedades. Por otra parte podemos crear, a mayores, otra propiedad denominada Colors que será el reflejo del array.

```
  private
     FColors:array[0..5] of TColor;
     ...
     ...
     procedure SetColors(Index:Integer;Value:TColor);
     function GetColors(Index:integer):TColor;
     ...
     ...
  public
     ...
```

```
    property Colors[Index:Integer]:TColor read GetColors
                                          Write SetColors;
  plublished
     ...
     property Color:TColor Index 0 read GetColors
            Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors
            Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
            Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
            Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors
            Write SetColors default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
            Write SetColors default clBlack;
     ...
```

En el código anterior hay varias cosas que observar:

1.- Guardamos los colores dentro de la variable FColors.

2.- La propiedad Colors la situamos en el apartado Public y no en el Published. Esto se debe a que el inspector de objetos no permite arrays como propiedades.

3.- Asignamos un índice a cada color. De esta manera nos ahorramos procedimientos distintos para el mismo tratamiento de los datos.

4.- No sólo escribimos un procedimiento SetPropiedad, sino que además debemos escribir una función que nos devuelva el color. Si los procedimientos que escriben se les pasa como parámetro un valor del tipo de la variable, la función debe devolver un valor del mismo tipo que la variable. En este caso en el que utilizamos un array para distintas propiedades, además del parámetro normal, se le pasa un índice que es el que aparece en la definición de la propiedad:

```
constructor TSelPanel.Create(AOwner:TComponent);
begin
...
...
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
...
...
end;
procedure TSelPanel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
      FColors[Index]:=Value;
```

```
      Invalidate;
   end;
end;

Function TSelPanel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;
```

Como veis el tratamiento no hace referencia más que al array y a través del índice asociado a cada propiedad se sabe que color estamos cambiando o leyendo.

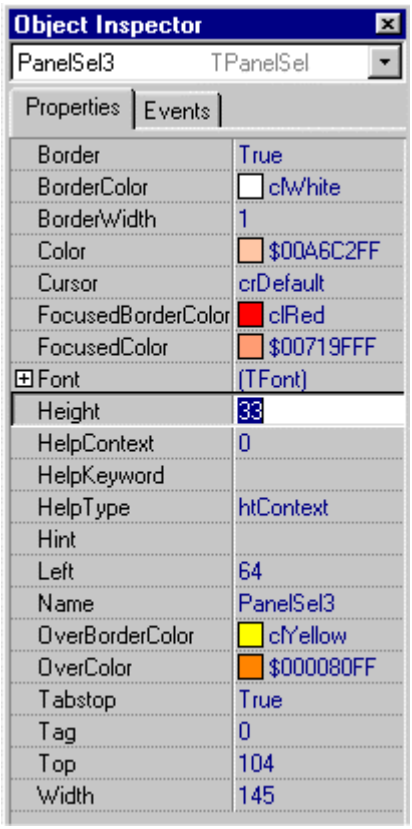


Figura 4

Una vez hecho esto ya podemos hacer que cambie el color en los casos antedichos, para ello modificaremos el método Paint para que dibuje con unos colores u otros dependiendo de la propiedad Focused y de la variable FOver, que indicarán si el control posee el foco y/o está el cursor del ratón sobre él. Si se da el caso en el que el control tiene el foco y además está el ratón sobre él, ya es decisión nuestra que debe hacer el método Paint, yo he elegido este orden de preferencia de menor a mayor : Normal -> Focused ->FOver, lo que implica que en el caso en el que el control tenga el foco y además tenga el ratón sobre él , el control aparecerá con los colores asignados a Over????Color.

El código completo del componente hasta ahora es :

```
unit PanelSel;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FColors:array[0..5] of TColor;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    procedure Paint; override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    property Colors[Index:Integer]:TColor read GetColors Write
SetColors;
    { Public declarations }
  published
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth Write
SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors
              Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors
              Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
              Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
              Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors
              Write SetColors default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
              Write SetColors default clBlack;
     property Font;
     property Tabstop;
    { Published declarations }
  end;

procedure Register;

implementation
constructor TPanelSel.Create(AOwner:TComponent);
```

```
begin
   inherited;
   FOver:=False;
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  FOver:=False;
  Invalidate;
end;

procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
```

```
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if focused then TextOut(0,0,'FOCUS');
      if Border then Rectangle(X, Y, X + W, Y + H);
      if FOver then TextOut(0,TextHeight('FOCUS')+2,'OVER');
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;
end.
```

Imágenes.

En este apartado vamos a ver como tratar imágenes en nuestro componente.

En Delphi hay varios objetos que guardan (o hacen referencia a) imágenes : TPicture, TBitmap,..., dependiendo de nosotros seleccionar aquel que más se ajuste a nuestro interés, o bien podemos optar por crear alguno nuevo. En nuestro componente vamos a utilizar dos de estos objetos, un Picture y un Bitmap (este último lo veremos más adelante, hacia el final de esta serie de artículos).

Picture nos permitirá cargar imágenes de varios tipos : bmp, jpg,ico,..., de esta manera ofrecemos al programador que utilice el componente una mayor libertad para seleccionar el tipo de gráfico que quiere mostrar.

Como hemos dicho, las imágenes se guardarán en objetos que nuestro componente debe gestionar (crear, manipular y destruir) por lo que se deben definir variables que referencien dichos objetos. Como siempre, estas variables se definirán en la parte privada y serán las *propiedades*, las encargadas de interactuar con ellas.

Como se ha dicho, al ser objetos, se deben crear en algún lugar del código de nuestro componente y deberán ser destruidos antes de que se destruya nuestro control. Crear el objeto se hará, normalmente, en el constructor (Create) del componente y para destruirlo deberemos hacerlo justo antes de la destrucción de nuestro propio objeto. Hay un procedimiento especial que se ejecuta siempre antes de la liberación de memoria del objeto, este método es el '*destructor*', se llama Destroy y deberemos sobreescribirlo:

```
Destructor Destroy;
begin
   //acciones anteriores a la liberación de memoria
   inherited;
end;
```

Este procedimiento se debe definir en el apartado Public de nuestro componente.

Resumiendo, para guardar la imagen que queremos mostrar en nuestro componente debemos : **(a)** Definir una variable que referencie al objeto que contendrá la imagen, **(b)** crear el objeto imagen y hacer que la variable definida contenga su referencia, **(c)** si queremos tener una propiedad que refleje y maneje a la variable deberemos definir esa propiedad, **(d)** tratar el objeto en cualquier sitio del código del componente y **(e)** destruir/liberar el objeto antes de liberar el control creado con nuestro componente.

```
  private
    FPicture:TPicture;
    ...
    procedure SetPicture(Value:TPicture);
    ...
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    ...
  published
    property Picture:TPicture read FPicture Write SetPicture;
    ...
implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   ...
   ...
   FPicture:=TPicture.Create;
end;
destructor TPanelSel.Destroy;
begin
   FPicture.Free;
   inherited;
end;
...
procedure TPanelSel.SetPicture(Value:TPicture);
begin
   FPicture.Assign(Value);
   repaint;
end;
...
```

Hasta aquí tenemos la imagen guardada pero todavía no hemos hecho que se dibuje en pantalla dentro del control, para ello iremos al método Paint y allí dibujaremos la posible imagen.

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
```

```
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
         Draw(BorderWidth,((Height-Picture.Graphic.Height) div
2),Picture.Graphic);
      if Border then Rectangle(X, Y, X + W, Y + H);
   end;
end;
```

El resultado en pantalla es :

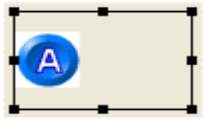


Figura 5

A la hora de dibujar la imagen hemos hecho que ésta aparezca, verticalmente, en el medio y horizontalmente justo después del borde. Nótese que antes de dibujar, nos hemos asegurado de que el objeto picture contiene algo (una imagen). La función*Assigned(puntero)* sólo comprueba que este no sea Nil

Observamos en la figura anterior un relleno blanco sobre el dibujo que a nosotros nos interesa, el círculo con la letra 'A'. La propiedad Graphic del objeto Picture de nuestro componente, es a la vez un objeto que tiene una propiedad denominada Transparent, esta propiedad cuando tiene el valor True hace que sea transparente el color de la imagen que coincida con el color del pixel (0,0) de la misma. Hay otras propiedades dentro de objetos como Bitmap que hacen referencia a esto mismo y se puede elegir el color que deseamos que sea el transparente, pero esto queda fuera del objetivo de este artículo.

```
   ...
      Brush.Style:=bsClear;
            if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth,((Height-Picture.Graphic.Height)
                           div 2),Picture.Graphic);
      end;

      if Border then Rectangle(X, Y, X + W, Y + H);
      ...
```

Vamos a dar la posibilidad de que el programador elija la coordenada X a partir de la cual dibujar la imagen, para ello añadimos una variable nueva, una propiedad y

hacemos que cualquier cambio en el valor de esta propiedad se refleje inmediatamente en pantalla (como hemos hecho en otras ocasiones) :

El código de nuestro componente hasta el momento es :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FPicture:TPicture;
    FColors:array[0..5] of TColor;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    FPosXPicture:Word;
    procedure SetPicture(Value:TPicture);
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetPosXPicture(Value:Word);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    procedure Paint; override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    property Colors[Index:Integer]:TColor read GetColors Write
SetColors;
    { Public declarations }
  published
     property Picture:TPicture read FPicture Write SetPicture;
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth
              Write SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors
              Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors
              Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
              Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
```

```
              Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors
              Write SetColors default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
              Write SetColors default clBlack;
     property PosXPicture:Word read FPosXPicture
              Write SetPosXPicture default 10;
     property Font;
     property Tabstop;
    { Published declarations }
  end;

procedure Register;

implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FOver:=False;
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
   FPicture:=TPicture.Create;
   FPosXPicture:=10;
end;
destructor TPanelSel.Destroy;
begin
   FPicture.Free;
   inherited;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  FOver:=False;
  Invalidate;
end;
```

```
procedure TPanelSel.SetPicture(Value:TPicture);
begin
   FPicture.Assign(Value);
   repaint;
end;
procedure TPanelSel.SetPosXPicture(Value:Word);
begin
   if FPosXPicture<>Value then
      // Sólo permitimos valores mayores que cero
      if value>0 then
      begin
         FPosXPicture:=Value;
         invalidate;
      end;
end;
procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
begin
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
```

```
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      Picture.Graphic.Transparent:=true;
      if Assigned(Picture.Graphic) then
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
div 2),Picture.Graphic);
      if Border then Rectangle(X, Y, X + W, Y + H);
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Texto.

Casi todos los controles de delphi que deben mostrar texto tienen un campo denominado Caption y/o un campo denominado Text. En nuestro caso vamos a tener los dos tipos de texto, como indicábamos en el primer capítulo I y vamos a llamarlos de la misma manera, uno Caption que será el texto que aparece el primero en el control y Text al segundo texto que podrá tener varias líneas y se deberá ajustar al tamaño del control de forma dinámica.

Podríamos pensar que, al igual que hemos hecho en otras ocasiones, deberíamos crear dos variables en la parte Private de nuestro componente y después dos propiedades que se encarguen de intercactuar con esas variables, pero muchas veces parte del trabajo lo tendremos hecho gracias a la herencia.

Vamos a hacer lo siguiente, en nuestro componente vamos a escribir el el apartado Published :

```
Published
   Property Capion;
   Property Text;
   ...
```

Ahora compilamos el paquete. Vemos que no nos ha dado ningún error de compilación y que estas propiedades aparecen en el inspector de objetos cuando dibujamos un control con nuestro componente, esto es gracias a la herencia. El componente PanelSel que estamos creando deriva o hereda de TCustomControl por lo que hereda todas las propiedades de él, éste a su vez hereda de TWinControl y este de TControl. Si nos fijamos en la definición de esta clase vemos que tiene estas dos propiedades.

En nuestro caso podemos aprovecharnos de esta característica para los dos textos, pero sólo lo vamos a hacer para Caption, mientras que para Text re-escribiremos la propiedad. Por lo tanto hacemos lo que siempre con esta nueva propiedad:

```
Private
   ...
   FText:TCaption;
   procedure SetText(Value:TCaption);
   ...
published
   ...
   Property Caption;
   property Text:TCaption read FText Write SetText;
...
implementation
```

```
...
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   ...
   FText:='';
end;
...
procedure TSelPanel.SetText(Value: TCaption);
begin
   if FText<>Value then
   begin
      FText:=Value;
      invalidate;
   end;
end;
```

Ya tenemos las dos propiedades de texto que necesitamos, pero aún no se dibuja nada en pantalla, tenemos varias funciones que dibujan texto en pantalla (Textout, TextRect, Drawtext) pero el estudio de las mismas queda fuera de lo que se pretende en estos artículos, sólo explicaremos la función que vamos a utilizar en este componente y esta es DrawText, esta es una llamada a la función de windows DrawTextA. Esta función escribe texto formateado dependiendo de una serie de flags que se le pasan como parámetros.

Para que nuestro componente se 'entere' de que hemos hecho cambios en la propiedad Caption heredada, debemos sobreescribir el procedimiento que responde al mensaje CM_TEXTCHANGED que es el que informa sobre un cambio en el texto :

```
procedure CMTextChanged(var Message: TMessage); message
CM_TEXTCHANGED;
```

Primero vamos a escribir nuestro Caption, queremos que aparezca en una sola línea justo después del borde superior del control, que no admita retornos de carro y que aparezcan puntos suspensivos cuando no pueda mostrarse completamente dentro del control, pues todo esto se lo diremos con los Flags antes mencionados. De momento también haremos que esté justificado a la izquierda todo lo anterior se indicaría de la siguiente manera :

```
Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
```

El formato de la llamada a Draw text está definido por Windows como :

```
int DrawText(

    HDC hDC,    // handle to device context
    LPCTSTR lpString,   // pointer to string to draw
    int nCount, // string length, in characters
```

```
    LPRECT lpRect,      // pointer to structure with formatting
dimensions
    UINT uFormat        // text-drawing flags
   );
```

El texto se dibuja en el canvas de nuestro control (Canvas.Handle) y dentro de un rectángulo cuyas coordenadas debemos pasar como parámetro, Delphi define una estructura denominada TRect que guarda los valores que determinan un rectángulo. En un primer momento vamos a suponer el rectángulo total del espacio ocupado por nuestro componente.

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
begin
   with Canvas do
   begin
   ...
   ...
      Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
      R:=ClientRect; // Devuelve el área de cliente del control
      Drawtext(handle,PChar(caption),-1,R,flags);
   end;
end;
```

El resultado es el siguiente :

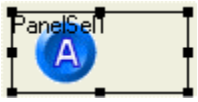


Figura 6

Observamos que el texto aparece en el componente pero no parece que quede muy estético, así que vamos a afinar, primero tendremos que tener en cuenta el ancho del borde, que no se escriba encima de la imagen y el rectángulo en el que tiene que aparecer sea como máximo el alto del texto y no el alto del control. Al igual que en el caso de picture vamos a definir una propiedad que indique una coordenada X del texto dentro del control. El código es el mismo que para la propiedad PosXPicture y la llamaremos PosXText.

El código del método Paint completo en en que se marca las partes del código que hace lo que se indica en el párrafo anterior :

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
```

```
   despv,desph:Integer;
begin
   despv:=3;
   Desph:=0;
   if border then
   begin
      despv:=despv+BorderWidth; //Desplazamiento vertical
      Desph:=BorderWidth-1;     //Desplazamiento horizontal
   end;
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
div 2),Picture.Graphic);
      end;
      if Border then Rectangle(X, Y, X + W, Y + H);
      Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
      // Tenemos en cuenta la propiedad PosXText y los desplazamientos
horizontal y vertical
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Drawtext(handle,PChar(caption),-1,R,flags);
   end;
end;
```

El resultado es el siguiente :

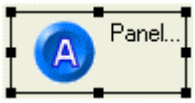


Figura 7

El segundo texto (propiedad Text) debe aparecer después del Caption pero en este caso el texto debe admitir retornos de carro y debe ajustar el texto al tamaño del control. Utilizaremos la misma función para escribir el texto, pero cambiando los Flags y el rectángulo al que el texto debe circunscribirse. Esto se consigue con el siguiente código:

```
      Flags:=DT_WORDBREAK or DT_LEFT or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,
              width-despv,height-despv);
```

Con DT_WORDBREAK, permitimos que el texto cambie de línea para ajustarse al tamaño de las coordenadas del rectángulo que se le pasa como parámetro. Para calcular estas coordenadas utilizamos el método TextHeight del canvas de devuelve el alto del texto que se le pasa en pixels, el valor depende de la fuente utilizada.

El resultado :

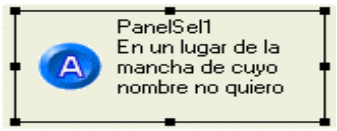


Figura 8

Fuentes del texto.

Hasta ahora no hemos tenido en cuenta el tipo de letra ni tamaño con la que escribir los textos. Si miramos en el inspector de objetos las propiedades del nuestro componente vemos que hay una que se llama Font. Vamos a utilizar ésta como fuente para escribir el texto de la propiedad Caption, para el otro texto veremos que hacer más adelante.
Para escribir con la fuente de la propiedad del componente, basta asignar ésta a la propiedad Font del objeto Canvas de nuestro componente :

```
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
   ...
   ...
      Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or DT_LEFT or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,
              width-despv,height-despv);
      DrawText(Handle, PChar(Text), -1, R, Flags);
   end;
end;
```

Si compilamos el componente y modificamos el valor de la propiedad Font vemos que en pantalla nos cambia el texto escrito con la nueva fuente, pero observamos que también se escribe con la misma fuente el segundo texto. Para evitar esto y a la vez dar la posibilidad de cambiar a una fuente distinta el texto de la propiedad Text vamos a añadir una propiedad nueva *TextFont*.

```
Private
          ...
   FTextFont:TFont;
   ...
   procedure SetTextFont(Value:TFont);
   ...
published
   ...
   property TextFont:TFont read FTextFont Write SetTextFont;
   ...
implementacion
...
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   ...
   FTextFont:=TFont.Create;
   ...
end;
destructor TPanelSel.Destroy;
begin
   FTextFont.Free;
   FPicture.Free;
   inherited;
end;
...
procedure TPanelSel.SetTextFont(Value:TFont);
begin
   FTextFont.Assign(Value);
   Invalidate;//observamos que esta orden no hace que nuestro
componente responda
end;
...
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
 ...
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or DT_LEFT or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,width-
despv,height-despv);
      Font:=self.TextFont;
      DrawText(Handle, PChar(Text), -1, R, Flags);
   end;
end;
```

La nueva propiedad hace referencia a un objeto, por lo que debemos crearlo y destruirlo antes de que nuestro control se destruya.(Observe que se asigna la fuente al canvas después de calcular el rectángulo donde se debe escribir el texto, si se asignara antes de este cálculo, TextHeight devolverían el tamaño de la nueva fuente y no la que nos interesa que es la de la fuente con que se escribió el Caption).

Pero sólo con esto no conseguimos que se actualice automáticamente al modificar su valor en el inspector de objetos. ¿Qué hacer para que el texto cambie en pantalla en cuanto se cambia alguna de las propiedades del objeto Font referenciado por nuestra propiedad TextFont? Si miramos la ayuda de delphi del objeto TFont vemos que aparte de propiedades y métodos tenemos también un evento : OnChange, si podemos asignar a este evento un procedimiento definido en nuestro componente, podríamos hacer que este respondiera a los cambios, pues esto es lo que hacemos :

```
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   ...
   FTextFont:=TFont.Create;
   FTextFont.OnChange:=FontChanged;
   ...
end;
```

FontChanged es un procedimiento que definimos en el apartado Private de la definición de nuestro componente y en la implementación lo único que haremos será invalidar el control :

```
procedure TPanelSel.FontChanged(Sender: TObject);
begin
   invalidate;
end;
```

Figura 9

El código del componente es hasta ahora :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FPicture:TPicture;
    FColors:array[0..5] of TColor;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    FPosXPicture:Word;
    FText:TCaption;
    FTextFont:TFont;
    FPosXText:Word;
```

```
    procedure SetPicture(Value:TPicture);
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetPosXPicture(Value:Word);
    procedure SetText(Value:TCaption);
    procedure SetPosXText(Value:Word);
    procedure SetTextFont(Value:TFont);
    procedure FontChanged(Sender: TObject);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus); message
WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus); message
WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage); message
CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage); message
CM_MOUSELEAVE;
    procedure CMTextChanged(var Message: TMessage); message
CM_TEXTCHANGED;
    procedure Paint; override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    property Colors[Index:Integer]:TColor read GetColors Write
SetColors;
    { Public declarations }
  published
     property Picture:TPicture read FPicture Write SetPicture;
     property Border:Boolean read FBorder Write SetBorder default
True;
     property BorderWidth:integer read FBorderWidth Write
SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors Write SetColors
default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors Write
SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors Write
SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors Write
SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors Write SetColors
default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors Write
SetColors default clBlack;
     property PosXPicture:Word read FPosXPicture Write SetPosXPicture
default 10;
     property PosXText:Word read FPosXText Write SetPosXText default
50;
     property Caption;
     property Text:TCaption read FText Write SetText;
     property TextFont:TFont read FTextFont Write SetTextFont;
     property Font;
     property Tabstop;
    { Published declarations }
  end;
```

```
procedure Register;

implementation
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FOver:=False;
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
   FPicture:=TPicture.Create;
   FTextFont:=TFont.Create;
   FTextFont.OnChange:=FontChanged;
   FPosXPicture:=10;
   FPosXText:=50;
   FText:='';
   Font.Style:=[fsBold];
end;
destructor TPanelSel.Destroy;
begin
   FTextFont.Free;
   FPicture.Free;
   inherited;
end;
procedure TPanelSel.CMTextChanged(var Message: TMessage);
begin
   inherited;
   invalidate;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  FOver:=False;
  Invalidate;
end;
```

```
procedure TPanelSel.SetPicture(Value:TPicture);
begin
   FPicture.Assign(Value);
   repaint;
end;
procedure TPanelSel.SetPosXPicture(Value:Word);
begin
   if FPosXPicture<>Value then
      if value>0 then
      begin
         FPosXPicture:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetPosXText(Value:Word);
begin
   if FPosXText<>Value then
      if Value>0 then
      begin
         FPosXText:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetText(Value: TCaption);
begin
   if FText<>Value then
   begin
      FText:=Value;
      invalidate;
   end;
end;

procedure TPanelSel.SetTextFont(Value:TFont);
begin
   FTextFont.Assign(Value);
end;

procedure TPanelSel.FontChanged(Sender: TObject);
begin
   invalidate;
end;

procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
```

```
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
   despv:=3;
   Desph:=0;
   if border then
   begin
      despv:=despv+BorderWidth;
      Desph:=BorderWidth-1;
   end;
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
div 2),Picture.Graphic);
```

```
      end;
      if Border then Rectangle(X, Y, X + W, Y + H);
      Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or DT_LEFT or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,width-
despv,height-despv);
      Font:=self.TextFont;
      DrawText(Handle, PChar(Text), -1, R, Flags);
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Nota : Las distintas alineaciones del texto se tratarán más adelante.

Visibilidad de propiedades, métodos y eventos.

Como se ha comentado anteriormente la herencia permite que mucha de la funcionalidad de nuestros nuevos componentes esté ya construida. Hemos visto, por ejemplo, el caso de Caption, Font o TabStop como casos de propiedades que ya estaban definidas y que nosotros sólo hemos tenido que cambiar la visibilidad (published). Pues lo mismo ocurre con otras muchas propiedades y que nos pueden resultar interesantes para nuestro componente, por ejemplo, no tenemos ninguna propiedad para habilitar/inhabilitar los controles creados con nuestro componente, tampoco tenemos una que haga que desaparezca de pantalla así que vamos a cambiar la visibilidad del siguiente grupo de propiedades (conviene que se vean los distintos ancestros de nuestro componente para ver las propiedades, métodos y eventos ya definidos):

```
published
     ...
     property BiDiMode;
     property TabOrder;
     property Action;
     property Align;
     property Anchors;
     property Visible;
     property Enabled;
     property Constraints;
     property DragCursor;
     property DragKind;
     property DragMode;
     property ParentBiDiMode;
     property ParentFont;
     property ParentShowHint;
     property PopupMenu;
     property ShowHint;
```

Alguna de ellas para que nos sean útiles debemos utilizarlas en el código de nuestro componente (BiDiMode) ya que sólo informan y no ejecutan ninguna acción, en cambio otras tienen un efecto inmediato al cambiarlas de valor (Visible).

Esto mismo ocurre con los eventos que pudieran estar definidos en alguno de los antecesores de nuestro componente (OnClick,OnKeyPress...) :

```
Published
     ...
     property OnClick;
     property OnContextPopup;
     property OnDblClick;
```

```
    property OnDragDrop;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    property OnEnter;
    property OnExit;
```

Al igual que hemos podido crear nuevas propiedades, podemos crear métodos y eventos. Los métodos van a ejecutar acciones (normalmente sobre el objeto, por ejemplo `R:=PanelSel1.ClientRect`) estos se definen como procedures o functions en el componente. Por otra parte los eventos, son sucesos provocados por alguna causa y a los que se puede asignar algo a hacer cuando ocurren, por ejemplo podemos querer que cuando se pulse sobre el control (OnClick) se abra una ventana con un mensaje.

Un ejemplo de creación de un método podría ser uno al que se le pase una cadena de caracteres y con ella asigne las propiedades caption y text de nuestro control, para ello dentro de la cadena el caption se consideraría, por ejemplo, hasta el primer carácter '|' :

```
public
   ...
   Procedure SetAllTexts(S:String);
   ...
...
implementation
...
Procedure TPanelSel.SetAllTexts(S:String);
begin
   if S='' then
   begin
      Caption:='';
      Text:='';
   end
   else
      if pos('|',S)>0 then
      begin
         caption:=copy(S,1,pos('|',S)-1);
         text:=copy(S,pos('|',S)+1,length(S));
      end
      else
         Caption:=S;
end;
...
```

Para definir eventos se hace de manera parecida a como se definen propiedades, de hecho para delphi son propiedades pero las variables sobre las que actúan estas propiedades son de un tipo especial denominado procedimiento de objeto :

```
Tipo:=procedure(...) of Object;
```

En nuestro componente aprovechando que tenemos que controlar la entrada y salida del ratón sobre nuestro componente, vamos a definir dos eventos nuevos : OnMouseEnter y OnMouseLeave. Como hemos dicho debemos definir dos variables que vamos a denominar igual que los eventos pero precedidas por la letra 'F' (para seguir el método utilizado por delphi) con lo que las variables se llamarán FOnMouseEnter y FOnMouseLeave, por otra parte lo único que nos interesa es que se se produzca el evento al entrar y salir el cursor del ratón sobre nuestro componente, por lo que no necesitamos pasar información al programador que vaya a utilizar nuestro componente, bueno como mínimo conviene pasar un parámetro Sender (como hace por ejemplo OnCLick) que sea una referencia al objeto, luego el nuevo tipo 'especial' a definir será :

```
Types
   TMouseEnterLeaveEvent=procedure(Sender:TObject) of Object;
```

Ahora nuestras variables serán de este nuevo tipo :

```
FOnMouseEnter, FOnMouseLeave:TMouseEnterLeaveEvent;
```

En párrafo anterior hemos dicho que el evento OnClick sólo envía como parámetro Sender como en nuestro caso, por lo que en vez de definir un tipo nuevo (que se puede definir y funcionará correctamente) podemos utilizar el mismo tipo que utiliza OnClick este tipo en delphi se llama TNotiFyEvent entonces :

```
  private
     ...
     FOnMouseEnter,
     FOnMouseLeave:TNotifyEvent;
     ...
     ...
  published
     ...
     property OnMouseEnter:TNotifyEvent read FOnMouseEnter
                                        Write FOnMouseEnter;
     property OnMouseLeave:TNotifyEvent read FOnMouseLeave
                                        Write FOnMouseLeave;
     ...
```

Como vemos se definen como cualquier otra propiedad (que leen y escriben las variables declaradas anteriormente).

Con esto hemos conseguido que el inspector de objetos tenga una referencia de dónde leer y guardar el código que los programadores que utilicen nuestro componente escriban, pero ¿cómo hacer que éste se ejecute cuando se produzca el evento?, pues observe los cambios en el código de nuestro componente :

```
...
procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseEnter) then FOnMouseEnter(Self);
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseLeave) then FOnMouseLeave(Self);
  FOver:=False;
  Invalidate;
end;
...
```

Sencillo, ¿no?, simplemente comprobamos que la variable(que es un puntero a un procedimiento) tenga asignado algún valor y si es así se llama al procedimiento con el parámetro que estera, en este caso él mismo.

El código completo es hasta aquí :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FPicture:TPicture;
    FColors:array[0..5] of TColor;
    FOnMouseEnter,
    FOnMouseLeave:TNotifyEvent;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    FPosXPicture:Word;
    FText:TCaption;
    FTextFont:TFont;
    FPosXText:Word;
    procedure SetPicture(Value:TPicture);
```

```
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetPosXPicture(Value:Word);
    procedure SetText(Value:TCaption);
    procedure SetPosXText(Value:Word);
    procedure SetTextFont(Value:TFont);
    procedure FontChanged(Sender: TObject);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    procedure CMTextChanged(var Message: TMessage);
              message CM_TEXTCHANGED;
    procedure Paint; override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    property Colors[Index:Integer]:TColor read GetColors Write
SetColors;
    Procedure SetAllTexts(S:String);
    { Public declarations }
  published
     property Picture:TPicture read FPicture Write SetPicture;
     property Border:Boolean read FBorder
              Write SetBorder default True;
     property BorderWidth:integer read FBorderWidth
              Write SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors
              Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors
              Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
              Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
              Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors
              Write SetColors default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
              Write SetColors default clBlack;
     property PosXPicture:Word read FPosXPicture
              Write SetPosXPicture default 10;
     property PosXText:Word read FPosXText
              Write SetPosXText default 50;
     property Caption;
     property Text:TCaption read FText Write SetText;
     property TextFont:TFont read FTextFont Write SetTextFont;
     property Font;
     property Tabstop;
     property BiDiMode;
     property TabOrder;
     property Action;
```

```
      property Align;
      property Anchors;
      property Visible;
      property Enabled;
      property Constraints;
      property DragCursor;
      property DragKind;
      property DragMode;
      property ParentBiDiMode;
      property ParentFont;
      property ParentShowHint;
      property PopupMenu;
      property ShowHint;
      property OnMouseEnter:TNotifyEvent read FOnMouseEnter
               Write FOnMouseEnter;
      property OnMouseLeave:TNotifyEvent read FOnMouseLeave
               Write FOnMouseLeave;
      property OnClick;
      property OnContextPopup;
      property OnDblClick;
      property OnDragDrop;
      property OnDragOver;
      property OnEndDock;
      property OnEndDrag;
      property OnMouseDown;
      property OnMouseMove;
      property OnMouseUp;
      property OnStartDock;
      property OnStartDrag;
      property OnEnter;
      property OnExit;
    { Published declarations }
  end;

procedure Register;

implementation
Procedure TPanelSel.SetAllTexts(S:String);
begin
   if S='' then
   begin
      Caption:='';
      Text:='';
   end
   else
      if pos('|',S)>0 then
      begin
         caption:=copy(S,1,pos('|',S)-1);
         text:=copy(S,pos('|',S)+1,length(S));
      end
      else
         Caption:=S;
end;
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FOver:=False;
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
```

```
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
   FPicture:=TPicture.Create;
   FTextFont:=TFont.Create;
   FTextFont.OnChange:=FontChanged;
   FPosXPicture:=10;
   FPosXText:=50;
   FText:='';
   Font.Style:=[fsBold];
end;
destructor TPanelSel.Destroy;
begin
   FTextFont.Free;
   FPicture.Free;
   inherited;
end;
procedure TPanelSel.CMTextChanged(var Message: TMessage);
begin
   inherited;
   invalidate;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseEnter) then FOnMouseEnter(Self);
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseLeave) then FOnMouseLeave(Self);
  FOver:=False;
  Invalidate;
end;

procedure TPanelSel.SetPicture(Value:TPicture);
begin
   FPicture.Assign(Value);
   repaint;
end;
procedure TPanelSel.SetPosXPicture(Value:Word);
begin
   if FPosXPicture<>Value then
      if value>0 then
```

```
      begin
         FPosXPicture:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetPosXText(Value:Word);
begin
   if FPosXText<>Value then
      if Value>0 then
      begin
         FPosXText:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetText(Value: TCaption);
begin
   if FText<>Value then
   begin
      FText:=Value;
      invalidate;
   end;
end;

procedure TPanelSel.SetTextFont(Value:TFont);
begin
   FTextFont.Assign(Value);
end;

procedure TPanelSel.FontChanged(Sender: TObject);
begin
   invalidate;
end;

procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
```

```
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.Paint;
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
   despv:=3;
   Desph:=0;
   if border then
   begin
      despv:=despv+BorderWidth;
      Desph:=BorderWidth-1;
   end;
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
div 2),Picture.Graphic);
      end;
      if Border then Rectangle(X, Y, X + W, Y + H);
      Flags:=DT_LEFT or DT_NOPREFIX or DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or DT_LEFT or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,width-
despv,height-despv);
```

```
      Font:=self.TextFont;
      DrawText(Handle, PChar(Text), -1, R, Flags);
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

Respondiendo al teclado.

En el capítulo III vimos como podíamos responder a la pulsación con el ratón sobre nuestro componente (Click) este método estaba definido en el ancestro de nuestro componente TControl, si vais a la ayuda de delphi de esta clase lo podéis ver.

En nuestro componente queremos, además, que cuando se pulse la barra espaciadora o la tecla Return, el control se comporte como si se hubiera pulsado con el ratón, es decir, que se hubiera hecho Click. Si, como vimos, ya existía un método definido para el ratón, podemos intuir que también lo puede haber para la pulsación del teclado, para ello buscamos en sus ancestros un método que pueda servirnos y hayamos que en TWinControl tenemos tres métodos que podrían servirnos :

```
procedure KeyDown(var Key: Word; Shift: TShiftState); dynamic;
procedure KeyUp(var Key: Word; Shift: TShiftState); dynamic;
procedure KeyPress(var Key: Char); dynamic;
```

A nosotros nos basta con saber si las teclas Return y Barra espaciadora se han pulsado, independientemente de si las teclas de mayúsculas, control o Alt están también pulsadas, así que lo que haremos será sobreescribir el método KeyPress para que responda a la pulsación de estas dos teclas y haga caso omiso al resto.

```
  protected
    ...
    procedure KeyPress(var Key: Char);override;
    ...
implementation
...
procedure TPanelSel.KeyPress(var Key: Char);
begin
   if (Key=#13) or (Key=#32) then
      Click;
   Key:=#0;
end;
...
```

Como podéis ver, lo único que hace este procedimiento es comprobar si se ha pulsado Return (#13) o espacio (#32), si es así se llama al método Click. Después se pone a nulo (#0) la tecla pulsada.

Alineación del texto.

Hasta este momento el texto de las propiedades Caption y Text sólo está alineado a la izquierda, y si nos fijamos en el método Paint vemos que esto lo fija el Flag DT_LEFT, yendo a la ayuda vemos que además existe DT_RIGHT y DT_CENTER, por lo que usando cualquiera de estos flags podemos cambiar la justificación del texto. Por otra parte si nos fijamos en las propiedades del control Label, podemos observar que tiene una propiedad Alignment que es la misma que nos interesa a nosotros, así que utilizaremos su misma denominación y tipo para nuestro componente, esta propiedad es de tipo TAlignment que delphi define así :

```
TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

En nuestro componente deberemos tener dos propiedades distintas, una para Caption y otra para Text, la de Caption la llamaremos Alignment y la de Text la denominaremos TextAlign las dos del tipo TAlignment :

```
   private
     ...
     FAlignment,
     FTextAlign: TAlignment;
     ...
     procedure SetAlignment(Value: TAlignment);
     procedure SetTextAlign(Value: TAlignment);
     ...
   published
     ...
     property Alignment: TAlignment read FAlignment write SetAlignment
default taLeftJustify;
     property TextAlign: TAlignment read FTextAlign write SetTextAlign
default taLeftJustify;
     ...
implementation
...
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   ...
   Alignment:=taLeftJustify;
   TextAlign:=taLeftJustify;
   ...
end;
...
procedure TPanelSel.SetAlignment(Value: TAlignment);
begin
   if FAlignment<>Value then
   begin
      FAlignment:=Value;
```

```
      Invalidate;
   end;
end;
procedure TPanelSel.SetTextAlign(Value: TAlignment);
begin
   if FTextAlign<>Value then
   begin
      FTextAlign:=Value;
      Invalidate;
   end;
end;
...
```

Ya tenemos definidas las propiedades, ahora nos falta tenerlas en cuenta cuando dibujamos el texto en el método Paint de nuestro control. Debemos hacer algo para transformar los valores de las propiedades a los que nosotros necesitamos, o sea, de taLeftJustity a DT_LEFT e igual para el resto. Una forma sería mediante la estructura **'Case *var* of'** de delphi, pero en este caso que tenemos dos propiedades que hacen lo mismo, nos veríamos duplicar código o sacar el código a una función y hacer dos llamadas, una con cada una de las propiedades. Hay una forma más 'elegante' de hacerlo:

```
const
   AAlignment : array[taLeftJustify..taCenter] of
uchar=(DT_LEFT,DT_RIGHT,DT_CENTER);
```

¿Qué hace esto?, pues define una constante que es un array unidimensional con tres valores, los índices de este array son los contenidos de las propiedades de Alineación y los valores del mismo son los Flags que nosotros necesitamos, así que bastará hacer `AAlignment[TextAlign]` para obtener el flag que haga que el texto contenido en la propiedad Text esté justificado de acuerdo a esa propiedad:

```
procedure TPanelSel.Paint;
const
   AAlignment : array[taLeftJustify..taCenter] of
uchar=(DT_LEFT,DT_RIGHT,DT_CENTER);
var
   X, Y, W, H: Integer;
   ...
   ...
      Flags:=AAlignment[Alignment] or DT_NOPREFIX or DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or AAlignment[TextAlign] or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,width-
despv,height-despv);
      Font:=self.TextFont;
      DrawText(Handle, PChar(Text), -1, R, Flags);
   ...
```

```
end;
...
```

Figura 10

Podemos pensar que ya hemos conseguido lo que queríamos, pero nos falta un detalle, si observamos las propiedades de nuestro componente, vemos una propiedad que tiene que ver con la alineación del texto y que nosotros no hemos tenido en cuenta : BiDiMode (Bidirectional Mode) esta propiedad ajusta la apariencia del texto cuando el componente se ejecuta en un país en el que se lee de derecha a izquierda, por ello, en el caso de que esto sea así, lo que nosotros llamaremos justificado a la izquierda para otros países será a la derecha y viceversa. Vea los cambios hechos en el código para tener en cuenta esta propiedad :

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FPicture:TPicture;
    FColors:array[0..5] of TColor;
    FOnMouseEnter,
    FOnMouseLeave:TNotifyEvent;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    FPosXPicture:Word;
    FText:TCaption;
    FTextFont:TFont;
    FPosXText:Word;
    FAlignment,
    FTextAlign: TAlignment;
    procedure SetPicture(Value:TPicture);
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetPosXPicture(Value:Word);
    procedure SetText(Value:TCaption);
    procedure SetPosXText(Value:Word);
    procedure SetTextFont(Value:TFont);
    procedure FontChanged(Sender: TObject);
    procedure SetAlignment(Value: TAlignment);
    procedure SetTextAlign(Value: TAlignment);
    { Private declarations }
  protected
```

```
    procedure WMSetFocus(var Message: TWMSetFocus);
              message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus);
              message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage);
              message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage);
              message CM_MOUSELEAVE;
    procedure CMTextChanged(var Message: TMessage);
              message CM_TEXTCHANGED;
    procedure Paint; override;
    procedure KeyPress(var Key: Char);override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    property Colors[Index:Integer]:TColor read GetColors Write
SetColors;
    Procedure SetAllTexts(S:String);
    { Public declarations }
  published
     property Alignment: TAlignment read FAlignment
              write SetAlignment default taLeftJustify;
     property TextAlign: TAlignment read FTextAlign
              write SetTextAlign default taLeftJustify;
     property Picture:TPicture read FPicture Write SetPicture;
     property Border:Boolean read FBorder
                             Write SetBorder default True;
     property BorderWidth:integer read FBorderWidth
              Write SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors
              Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors
              Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
              Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
              Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors
              Write SetColors default clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
              Write SetColors default clBlack;
     property PosXPicture:Word read FPosXPicture
              Write SetPosXPicture default 10;
     property PosXText:Word read FPosXText
              Write SetPosXText default 50;
     property Caption;
     property Text:TCaption read FText Write SetText;
     property TextFont:TFont read FTextFont Write SetTextFont;
     property Font;
     property Tabstop;
     property BiDiMode;
     property TabOrder;
     property Action;
     property Align;
     property Anchors;
     property Visible;
     property Enabled;
     property Constraints;
     property DragCursor;
```

```
      property DragKind;
      property DragMode;
      property ParentBiDiMode;
      property ParentFont;
      property ParentShowHint;
      property PopupMenu;
      property ShowHint;
      property OnMouseEnter:TNotifyEvent read FOnMouseEnter
                                         Write FOnMouseEnter;
      property OnMouseLeave:TNotifyEvent read FOnMouseLeave
                                         Write FOnMouseLeave;
      property OnClick;
      property OnContextPopup;
      property OnDblClick;
      property OnDragDrop;
      property OnDragOver;
      property OnEndDock;
      property OnEndDrag;
      property OnMouseDown;
      property OnMouseMove;
      property OnMouseUp;
      property OnStartDock;
      property OnStartDrag;
      property OnEnter;
      property OnExit;
     { Published declarations }
  end;
Function _BiDiMode(Alignment: TAlignment;BiDi:TBiDiMode):TAlignment;
procedure Register;

implementation
Function _BiDiMode(Alignment: TAlignment;BiDi:TBiDiMode):TAlignment;
begin
  Result :=Alignment;
  if (SysLocale.MiddleEast) and (BiDi= bdRightToLeft) then
  case Alignment of
    taLeftJustify:  Result := taRightJustify;
    taRightJustify: result := taLeftJustify;
  end;
end;
Procedure TPanelSel.SetAllTexts(S:String);
begin
   if S='' then
   begin
      Caption:='';
      Text:='';
   end
   else
      if pos('|',S)>0 then
      begin
         caption:=copy(S,1,pos('|',S)-1);
         text:=copy(S,pos('|',S)+1,length(S));
      end
      else
         Caption:=S;
end;
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FOver:=False;
   Tabstop:=True;
```

```
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
   FPicture:=TPicture.Create;
   FTextFont:=TFont.Create;
   FTextFont.OnChange:=FontChanged;
   FPosXPicture:=10;
   FPosXText:=50;
   FText:='';
   Font.Style:=[fsBold];
   FAlignment:=taLeftJustify;
   FTextAlign:=taLeftJustify;
end;
destructor TPanelSel.Destroy;
begin
   FTextFont.Free;
   FPicture.Free;
   inherited;
end;
procedure TPanelSel.CMTextChanged(var Message: TMessage);
begin
   inherited;
   invalidate;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseEnter) then FOnMouseEnter(Self);
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseLeave) then FOnMouseLeave(Self);
  FOver:=False;
  Invalidate;
end;

procedure TPanelSel.SetPicture(Value:TPicture);
begin
   FPicture.Assign(Value);
   repaint;
```

```
end;
procedure TPanelSel.SetPosXPicture(Value:Word);
begin
   if FPosXPicture<>Value then
      if value>0 then
      begin
         FPosXPicture:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetPosXText(Value:Word);
begin
   if FPosXText<>Value then
      if Value>0 then
      begin
         FPosXText:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetText(Value: TCaption);
begin
   if FText<>Value then
   begin
      FText:=Value;
      invalidate;
   end;
end;

procedure TPanelSel.SetTextFont(Value:TFont);
begin
   FTextFont.Assign(Value);
end;

procedure TPanelSel.FontChanged(Sender: TObject);
begin
   invalidate;
end;

procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
```

```
   begin
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
end;
procedure TPanelSel.KeyPress(var Key: Char);
begin
   if (Key=#13) or (Key=#32) then
      Click;
   Key:=#0;
end;
procedure TPanelSel.SetAlignment(Value: TAlignment);
begin
   if FAlignment<>Value then
   begin
      FAlignment:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetTextAlign(Value: TAlignment);
begin
   if FTextAlign<>Value then
   begin
      FTextAlign:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.Paint;
const
   AAlignment : array[taLeftJustify..taCenter] of
uchar=(DT_LEFT,DT_RIGHT,DT_CENTER);
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
   despv:=3;
   Desph:=0;
   if border then
   begin
      despv:=despv+BorderWidth;
      Desph:=BorderWidth-1;
   end;
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
```

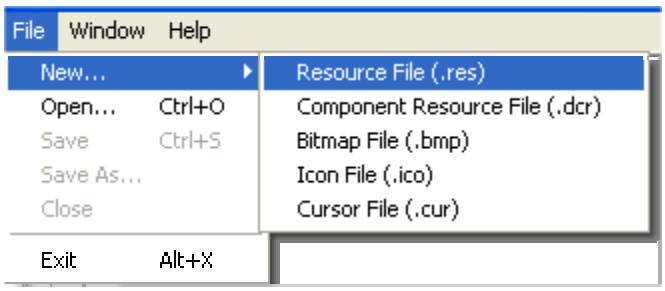
```
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
              div 2),Picture.Graphic);
      end;
      if Border then Rectangle(X, Y, X + W, Y + H);
      Flags:=AAlignment[_BiDiMode(Alignment,BiDiMode)] or DT_NOPREFIX
             or DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or AAlignment[_BiDiMode(TextAlign,BiDiMode)]
             or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,
              width-despv,height-despv);
      Font:=self.TextFont;
      DrawText(Handle, PChar(Text), -1, R, Flags);
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```

'Clavos'. Propiedad Bitmap

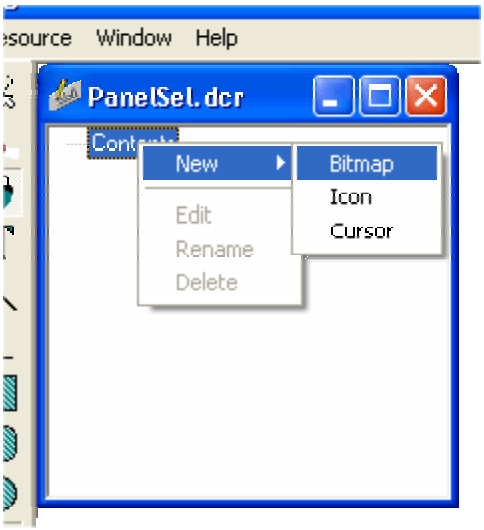
Anteriormente dijimos que íbamos a utilizar dos tipos de objetos de imagen, TPicture ya utilizado y uno TBitmap. Éste último lo utilizaremos para dibujar 4 clavos alrededor del control cuando éste tiene el foco (que se muestren estos 'clavos' va a ser opcional dependiendo de otra propiedad). Por lo tanto definiremos dos propiedades Screw y ShowScrew, la primera será el bitmap que queremos utilizar como 'clavo' y la segunda si queremos mostrarlos o no.

Los pasos a seguir con los de siempre, definir dos variables en la parte 'Private' de nuestro componente, definir las propiedades basadas 'manejadoras' de las variables anteriores. En el constructor crearemos el objeto imagen, que destruiremos en el destructor.

Pero queremos algo más, por defecto y en el caso de que no se elija ninguna imagen como 'clavo' haremos que aparezca uno por defecto (sólo si ShowCrew es true, claro). Para conseguir esto, vanos a crear un archivo de tipo recurso(.res) que añadiremos en nuestro componente, para ello utilizaremos ImageEdit, que es una herramienta que se encuentra en directorio Bin de delphi : "C:\Archivos de programa\Borland\Delphi6\Bin\ imagedit.exe" :

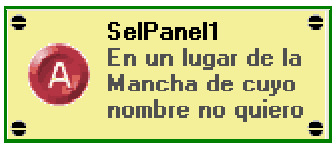


Al fichero los llamaremos "panselre.res" y lo guardaremos en el mismo directorio que el código de nuestro componente. Después creamos el bitmap (de 8x8) en este fichero de recursos, al bitmap le pondremos de nombre 'CLAVO':



Ahora, para que al compilar se incluya el recurso dentro del código compilado, habrá que añadir la directiva `{$R selpanre}`, donde panselre es el fichero de recursos, para cargar un recurso de bitmap dentro de un objeto usaremos el método de la Clase TBitmap LoadFromResourceName:

```
FScrew.LoadFromResourceName(Hinstance,'CLAVO');
```



(El código completo del componente después del siguiente apartado)

Icono para la paleta de componentes.

Una vez completado el componente (o si se prefiere en cualquier momento de la creación del mismo) deberíamos dibujar un icono que represente a nuestro componente en la paleta de componentes de delphi :

Figura 11- Paleta de componentes

Nosotros cuando en la definición de nuestro componente escribimos en el procedimiento Register el siguiente código :

```
RegisterComponents('Ejemplo', [TPanelSel]);
```

estamos diciendo al entorno de Delphi que añada nuestro componente a la carpetilla (pestaña) 'Ejemplo' de la paleta de componentes, en caso de no existir, el entorno (IDE)

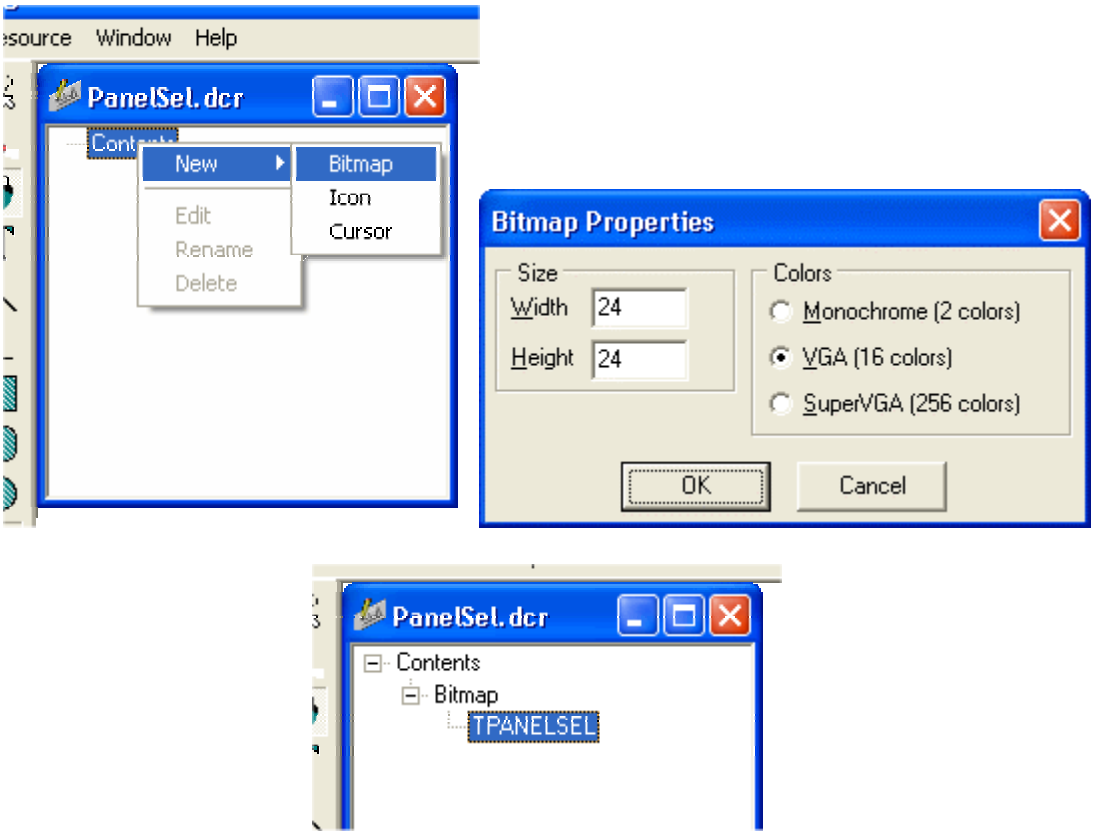
de delphi creará una con dicho nombre y dentro de ella aparecerá un icono como representación de nuestro componente :

Figura 12 - Componente PanelSel

El icono mostrado es que que el IDE de delphi asigna por defecto cuando no le hemos indicado otro.

Para crear uno propio debemos seguir los siguientes pasos :

1.- Mediante ImageEdit debéis crear un archivo nuevo de tipo .dcr (Component Resource File) que debe llamarse igual que la unidad de nuestro componente (PanelSel.dcr), y dentro de este fichero crear un recurso Bitmap de 24x24 pixels y cuyo nombre debe ser el del tipo de nuestro componente en mayúsculas : TPANELSEL :



Figuras 13,14,15

2.- Edite el bitmap y haga el dibujo que crea que mejor representa al componente, guarde el archivo en el mismo directorio que nuestro componente.

3.- Para que el IDE acepte esto cambios debemos añadir al paquete dentro del cual definimos nuestro componente (figura 1), para ello lo mejor es que eliminemos nuestro

componente del paquete y después volvamos a añadirlo, con esto conseguimos que los ficheros PanelSel.pas y PanelSel.dcr se incorporen al paquete, después compilar y 'voilá' en la pestaña ejemplo aparecerá nuestro bitmap :

Figura 16

Código Completo.

```
unit PanelSel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, Graphics;

type
  TPanelSel = class(TCustomControl)
  private
    FPicture:TPicture;
    FColors:array[0..5] of TColor;
    FOnMouseEnter,
    FOnMouseLeave:TNotifyEvent;
    FBorder:Boolean;
    FBorderWidth:Integer;
    FOver:Boolean;
    FPosXPicture:Word;
    FText:TCaption;
    FTextFont:TFont;
    FPosXText:Word;
    FAlignment,
    FTextAlign: TAlignment;
    FScrew:TBitmap;
    FShowScrew:Boolean;
    procedure SetPicture(Value:TPicture);
    procedure SetColors(Index:Integer;Value:TColor);
    function GetColors(Index:integer):TColor;
    procedure SetBorder(Value:Boolean);
    procedure SetBorderWidth(Value:integer);
    procedure SetPosXPicture(Value:Word);
    procedure SetText(Value:TCaption);
    procedure SetPosXText(Value:Word);
    procedure SetTextFont(Value:TFont);
    procedure FontChanged(Sender: TObject);
    procedure SetAlignment(Value: TAlignment);
    procedure SetTextAlign(Value: TAlignment);
    procedure SetScrew(Value:TBitmap);
    { Private declarations }
  protected
    procedure WMSetFocus(var Message: TWMSetFocus); message WM_SETFOCUS;
    procedure WMKillFocus(var Message: TWMSetFocus); message WM_KILLFOCUS;
    procedure CMMouseEnter(var Message: TMessage); message CM_MOUSEENTER;
    procedure CMMouseLeave(var Message: TMessage); message CM_MOUSELEAVE;
    procedure CMTextChanged(var Message: TMessage); message CM_TEXTCHANGED;
    procedure Paint; override;
    procedure KeyPress(var Key: Char);override;
    procedure Click;override;
    { Protected declarations }
  public
    constructor Create(AOwner:TComponent);override;
    destructor Destroy;override;
    property Colors[Index:Integer]:TColor read GetColors Write SetColors;
    Procedure SetAllTexts(S:String);
    { Public declarations }
  published
     property Alignment: TAlignment read FAlignment
                write SetAlignment default taLeftJustify;
     property TextAlign: TAlignment read FTextAlign
                write SetTextAlign default taLeftJustify;
     property Picture:TPicture read FPicture Write SetPicture;
     property Border:Boolean read FBorder Write SetBorder default True;
     property BorderWidth:integer read FBorderWidth Write SetBorderWidth default 1;
     property Color:TColor Index 0 read GetColors Write SetColors default clBtnFace;
     property BorderColor:TColor Index 1 read GetColors Write SetColors default clBlack;
     property FocusedColor:TColor Index 2 read GetColors
                Write SetColors default clBtnHighlight;
     property FocusedBorderColor:TColor Index 3 read GetColors
                Write SetColors default clBlack;
     property OverColor:TColor Index 4 read GetColors Write SetColors default
clBtnShadow;
     property OverBorderColor:TColor Index 5 read GetColors
                 Write SetColors default clBlack;
```

```
    property PosXPicture:Word read FPosXPicture Write SetPosXPicture default 10;
    property PosXText:Word read FPosXText Write SetPosXText default 50;
    property Caption;
    property Text:TCaption read FText Write SetText;
    property TextFont:TFont read FTextFont Write SetTextFont;
    property Screw:TBitmap read FScrew Write SetScrew;
    property ShowScrew:Boolean read FShowScrew Write FShowScrew default false;
    property Font;
    property Tabstop;
    property BiDiMode;
    property TabOrder;
    property Action;
    property Align;
    property Anchors;
    property Visible;
    property Enabled;
    property Constraints;
    property DragCursor;
    property DragKind;
    property DragMode;
    property ParentBiDiMode;
    property ParentFont;
    property ParentShowHint;
    property PopupMenu;
    property ShowHint;
    property OnMouseEnter:TNotifyEvent read FOnMouseEnter Write FOnMouseEnter;
    property OnMouseLeave:TNotifyEvent read FOnMouseLeave Write FOnMouseLeave;
    property OnClick;
    property OnContextPopup;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
    property OnStartDock;
    property OnStartDrag;
    property OnEnter;
    property OnExit;
   { Published declarations }
  end;
Function _BiDiMode(Alignment: TAlignment;BiDi:TBiDiMode):TAlignment;
procedure Register;

implementation
{$R panselre}
Function _BiDiMode(Alignment: TAlignment;BiDi:TBiDiMode):TAlignment;
begin
  Result :=Alignment;
  if (SysLocale.MiddleEast) and (BiDi= bdRightToLeft) then
  case Alignment of
    taLeftJustify:  Result := taRightJustify;
    taRightJustify: result := taLeftJustify;
  end;
end;
Procedure TPanelSel.SetAllTexts(S:String);
begin
   if S='' then
   begin
      Caption:='';
      Text:='';
   end
   else
      if pos('|',S)>0 then
      begin
         caption:=copy(S,1,pos('|',S)-1);
         text:=copy(S,pos('|',S)+1,length(S));
      end
      else
         Caption:=S;
end;
constructor TPanelSel.Create(AOwner:TComponent);
begin
   inherited;
   FScrew:=TBitmap.Create;
```

```
   FScrew.LoadFromResourceName(Hinstance,'CLAVO');
   FShowScrew:=False;
   FOver:=False;
   Tabstop:=True;
   FBorder:=True;
   FBorderWidth:=1;
   FColors[0]:= clBtnFace;
   FColors[1]:=clBlack;
   FColors[2]:=clBtnHighlight;
   FColors[3]:=clBlack;
   FColors[4]:= clBtnShadow;
   FColors[5]:=clBlack;
   FPicture:=TPicture.Create;
   FTextFont:=TFont.Create;
   FTextFont.OnChange:=FontChanged;
   FPosXPicture:=10;
   FPosXText:=50;
   FText:='';
   Font.Style:=[fsBold];
   FAlignment:=taLeftJustify;
   FTextAlign:=taLeftJustify;
end;
destructor TPanelSel.Destroy;
begin
   FTextFont.Free;
   FPicture.Free;
   inherited;
end;
procedure TPanelSel.SetScrew(Value:TBitmap);
begin
   if Assigned(Value) then
   begin
      if (Value.Width>8) or (Value.Height>8) then
         raise Exception.Create('La imagen debe ser como máximo de 8x8 pixels')
      else
         FScrew.Assign(Value);
   end
   else
      FScrew.LoadFromResourceName(Hinstance,'CLAVO');
   Invalidate;
end;

procedure TPanelSel.CMTextChanged(var Message: TMessage);
begin
   inherited;
   invalidate;
end;
procedure TPanelSel.WMSetFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.WMKillFocus(var Message: TWMSetFocus);
begin
   inherited;
   Invalidate;
end;

procedure TPanelSel.CMMouseEnter(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseEnter) then FOnMouseEnter(Self);
  FOver:=True;
  Invalidate;
end;

procedure TPanelSel.CMMouseLeave(var Message: TMessage);
begin
  inherited;
  if Assigned(FOnMouseLeave) then FOnMouseLeave(Self);
  FOver:=False;
  Invalidate;
end;

procedure TPanelSel.SetPicture(Value:TPicture);
begin
```

```
   FPicture.Assign(Value);
   repaint;
end;
procedure TPanelSel.SetPosXPicture(Value:Word);
begin
   if FPosXPicture<>Value then
      if value>0 then
      begin
         FPosXPicture:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetPosXText(Value:Word);
begin
   if FPosXText<>Value then
      if Value>0 then
      begin
         FPosXText:=Value;
         invalidate;
      end;
end;

procedure TPanelSel.SetText(Value: TCaption);
begin
   if FText<>Value then
   begin
      FText:=Value;
      invalidate;
   end;
end;

procedure TPanelSel.SetTextFont(Value:TFont);
begin
   FTextFont.Assign(Value);
end;

procedure TPanelSel.FontChanged(Sender: TObject);
begin
   invalidate;
end;

procedure TPanelSel.SetBorder(Value:Boolean);
begin
   if FBorder<>Value then
   begin
      FBorder:=Value;
      Invalidate;
   end;
end;

procedure TPanelSel.SetBorderWidth(Value:integer);
begin
   if FBorderWidth<>Value then
   begin
      if Value>0 then
         FBorderWidth:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetColors(Index:Integer;Value:TColor);
begin
   if FColors[Index]<>Value then
   begin
      FColors[Index]:=Value;
      Invalidate;
   end;
end;
Function TPanelSel.GetColors(Index:Integer):TColor;
begin
   Result:=FColors[Index];
end;

procedure  TPanelSel.Click;
begin
   inherited;
   SetFocus;
```

```
end;
procedure TPanelSel.KeyPress(var Key: Char);
begin
   if (Key=#13) or (Key=#32) then
      Click;
   Key:=#0;
end;
procedure TPanelSel.SetAlignment(Value: TAlignment);
begin
   if FAlignment<>Value then
   begin
      FAlignment:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.SetTextAlign(Value: TAlignment);
begin
   if FTextAlign<>Value then
   begin
      FTextAlign:=Value;
      Invalidate;
   end;
end;
procedure TPanelSel.Paint;
const
   AAlignment : array[taLeftJustify..taCenter] of uchar=(DT_LEFT,DT_RIGHT,DT_CENTER);
var
   X, Y, W, H: Integer;
   r:TRect;
   Flags:Cardinal;
   despv,desph:Integer;
begin
   despv:=3;
   Desph:=0;
   if border then
   begin
      despv:=despv+BorderWidth;
      Desph:=BorderWidth-1;
   end;
   with Canvas do
   begin
      setbkmode(Handle,TRANSPARENT);
      Pen.Width:=BorderWidth;
      Pen.Color:=BorderColor;
      Brush.Style:=bsSolid;
      Brush.Color:=Color;
      X := Pen.Width div 2;
      Y := X;
      W := Width - Pen.Width + 1;
      H := Height - Pen.Width + 1;
      if Focused then
      begin
         Pen.Color:=FocusedBorderColor;
         Brush.Color:=FocusedColor;
      end;
      if FOver then
      begin
         Pen.Color:=OverBorderColor;
         Brush.Color:=OverColor;
      end;
      FillRect(ClientRect);
      Brush.Style:=bsClear;
      if Assigned(Picture.Graphic) then
      begin
         Picture.Graphic.Transparent:=true;
         Draw(BorderWidth+PosXPicture,((Height-Picture.Graphic.Height)
                                       div 2),Picture.Graphic);
      end;
      if Border then Rectangle(X, Y, X + W, Y + H);
      Flags:=AAlignment[_BiDiMode(Alignment,BiDiMode)] or DT_NOPREFIX or
             DT_END_ELLIPSIS;
      R:=Rect(posxText+desph,despv,width-desph,height-despv);
      Font:=self.Font;
      Drawtext(handle,PChar(caption),-1,R,flags);
      Flags:=DT_WORDBREAK or AAlignment[_BiDiMode(TextAlign,BiDiMode)] or DT_NOPREFIX;
      R:=Rect(posxText+desph,TextHeight(Caption)+despv,width-despv,height-despv);
      Font:=self.TextFont;
```

```
      DrawText(Handle, PChar(Text), -1, R, Flags);
      if Focused and ShowScrew then
      begin
         FScrew.Transparent:=True;
         FScrew.TransparentMode:=tmAuto;
         Draw(desph+2,desph+2,FScrew);
         Draw(Width-desph-2-FScrew.Width,desph+2,FScrew);
         Draw(desph+2,Height-desph-2-FScrew.Height,FScrew);
         Draw(Width-desph-2-FScrew.Width,Height-desph-2-FScrew.Height,FScrew);
      end;
   end;
end;
procedure Register;
begin
  RegisterComponents('Ejemplo', [TPanelSel]);
end;

end.
```