When I started this newsletter back in March, I really didn't think that I would be able to keep everyone's interest for long. I decided that ultimately the material would dry up and I would run out of things to print. Well, I am happy to report that I appear to have been in error. Articles continue to flow in at a fairly comfortable pace, and just enough so I can make sure there is sufficient diversity of material in each issue. It may be a long time in coming, but I want to express my gratitude to everyone who has contributed material for this newsletter. Keep up the fine work, and keep 'em coming!

In addition, I would like to start looking into finding someone who would be willing to put together a few introductory-level programming articles for the newsletter. There are a quite a few users out there who are working on some of the more basic concepts in Delphi, and I am sure many of them would welcome some additional material to work with. Please don't read this as "I want an expert to write an article for novices", rather, I think this might best be handled by a novice who has overcome some difficult hurdle that more experienced programmers might take for granted. So, if you are new at Delphi and have learned some element of the language or environment by means of the "school of hard knocks", please let me know and I will put it in the next issue. Who knows, maybe we could even have a *Questions from Readers...* section?

Coming up in future issues will be reports on the *Delphi Study Group* conducted by Tom Theobald (74403,1324). It is a essentially a free, electronic classroom for the purpose of studying and learning about Delphi. It is conducted in the Informant Communications forum on CompuServe (ICGFORUM), and will be starting in early to mid-August. All are welcome to join, so contact Tom if you are interested.

In This Issue:

Component Create - A Review

by Robert Vivrette

"The Component Create code generator is a tool that lets Delphi developers easily produce new Delphi visual components. Need a specialized edit box, button, or image box with custom behavior? Want to convert some of your existing file-handling routines into visual components that you can drop onto a form from Delphi's component palette? Component Create makes it a point-and-click process." (from the introduction)

In a nutshell, CC is a "component expert". It is a program that allows you to define virtually all of the aspects of a new component, and, when done will generate the code for that component. I obtained a copy of CC several weeks ago and have been playing with it off and on. All in all, I am impressed. Even though the manual creation of components gets easier with experience, I found that CC removes a great deal of the overhead and drudgery. CC is also perfect as a tool for component "newbies". Those without much experience in creating components can learn a lot from seeing the code the CC makes.

To give you a good overview of how it works, I will describe the process of a creating a custom component. To keep the discussion short, I will make it a simple one. Let's say we want a graphic image control that will print some text inside it every time we click the mouse over it. Granted, not a particularly useful component, but it will illustrate a number of CC's features.

First, we click on "Create a new Delphi Component". This presents us with a expandable outline list a little like Delphi's object browser. One of the options is "Derived from Delphi Visual Component". Expanding this option extends the list to include many of the basic Delphi controls. We pick the "Images" option and then pick the TImage component. This will tell CC to descend our new component from the Delphi TImage object.

Once we have picked the parent class for the component, we are presented with a tabbed notebook dialog that allows us to define every aspect of our new component. On the main page, I will define the class name as "TMyImage", give the unit name as MYIMAGE.PAS, specify a few comments, and copyright lines, followed by specifying the palette we want it to sit on: how about "Samples". There is also a button here for editing the global declarations for the unit.

Now, we move on to the "Properties" tab. Here I can define each of the properties that the component requires. First, I will add a **Text** property. I provide a name, description, data type of "string", and default value of "Sample Text". I also check the boxes telling CC to make read and write access methods. Next, I create a **TextColor** property in much the same way. Yet this time, I pick **TColor** as the properties type, and give it a default value of **clLime**. On both of these properties, I can click on the "Code" button and see that CC has already built the access methods (which I can modify if needed).

Moving on to the "Events" tab, I see that CC has already added the inherited event handlers that come from TImage. I can choose to override these, delete those I don't want or add new ones. For this component, I want to override the *OnMouseDown* event with my own code. I simply select that event, click on the code button, and provide the code I want, namely...

```
Canvas.Font.Color := TextColor;
Canvas.TextOut(X,Y,Text);
```

At this point, I am ready to test this baby so, I tell CC to generate the code, which it does in a fraction of a second (well, this **is** a small component I guess). I go into Delphi and add the resulting TMyImage. The component compiles cleanly, and I add it to a blank form. I fiddle with my two newly added properties on the object inspector, and then run the program. By clicking inside the boundaries of the TMyImage, it prints the text I specified at the spot I clicked the mouse, in the appropriate color. Cool!

My only complaints with CC would be classified as "look and feel" issues. Some of the forms and dialogs could use a bit of reworking, and some of them when resized, covered over buttons and fields. Also, I thought that there were plenty of opportunities for "preference" options. For example, the code that is generated is **highly** commented, but it would be nice to have a configuration option that turned off the comments. Options for changing the layout, spacing, and organization of the generated code would also be helpful. I discussed these issues with the creator, David Price, and he indicated that many of these items would likely be included in the next version.

These minor items aside, Component Create is a great product. And with a little bit of tweaking, version 2.0 will be even better. If you are completely new to component creation, or even if you are a component "expert" and just need help clearing away some of the drudgery, then Component Create is for you.

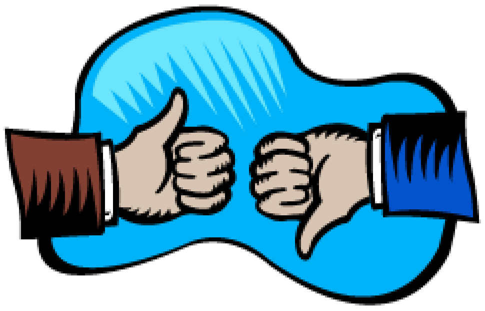
Component Create is available through Zac Catalogs, Programmer's Warehouse, and various resellers overseas. The list price is $179 though Zac and PW might be selling it for less.

Potomac Software can be reached on CompuServe at 71726,651 or you can contact them by phone at (800) 628-5524, or Fax at (202) 244-9065.

Return to Front Page

The Delphi Magazine - Issue #2

For more information or a sample issue contact the editor *Chris Frizelle* at 70630,717. Also, note that the first issue is still available as an Adobe Acrobat format document in LIB 22 (TDM01.ZIP). Furthermore, the first 'Under Construction' article (from The Delphi Magazine #1) can now be found on the www at http://www.borland.com:80/Product/Lang/Delphi/dpconst/dp2const.html



Borland Visual Solutions Pack

By Robert Pullan - President: Lighthouse Technologies - CIS: 102162,2711

Borland's Visual Solutions Pack v 1.1 (BVSP) is perhaps a misnomer. This product is really a compilation of VBXs from a variety of third party vendors.   Borland's only involvement in this product seems to be that of a bundler. It is visually wrapped in Borland colors and the Solutions seem to be aimed at the VC++, VB, dBase crowd, rather   than at Pascal/Delphi users.

What you get is really a mixed bag of some good, some redundant, some non-current versions and some teaseware VBXs.   Don't expect product support from Borland as the purpose of this product appears to be as a sampler/teaser for the current versions, fully featured products supplied by the vendors which in itself is not unreasonable.

The product comes with examples for VC++, VB and dBase. However, with the exception of page 925 in the extensive reference manual, the word Pascal is never used... Since many of these items are not current releases, and Borland is not the developer, it seems unlikely the products will offer maintenance releases to adapt these versions to work under Delphi (I will admit I have not checked every vendor's support boards for Delphi mandated revisions which would enable these products to function more fully).

There is a text editor, some nice "Gadgets" and a comm program that can perform Xmodem only transfers.   If this is what you are looking for, and you are willing to put up with the restrictions mentioned above, then BVSP is a good deal.

It is unfortunate that this product is being sold as a Delphi add-on, as in my honest opinion, it does not live up to the considerable standards Borland has set for Delphi.

Return to Front Page

What's In Print?

A brief synopsis of the principal magazines and journals for Delphi:

Return to Front Page

Debug Box Source Code

Simply use the Edit|Copy command from the menu above to copy this code to the clipboard. Then go into Delphi, create a new Unit file, and paste the code into that empty file. Save the file as "DEBUGBOX.PAS" Then, all you have to do is pick Install Components from the Options menu.

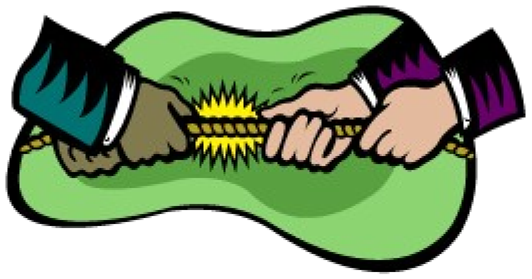
```
unit DebugBox;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;

type
  TPositions = (poTopLeft,poBottomLeft,poTopRight,poBottomRight);

  TDebugBox = class(TComponent)
  private
    DebugForm : TForm;
    DebugList : TListBox;
    FPosition : TPositions;
    FVisible  : Boolean;
    FWidth    : Integer;
    FHeight   : Integer;
    FCaption  : String;
    procedure SetPosition(A: TPositions);
    procedure SetVisible(A: Boolean);
    procedure SetWidth(A: Integer);
    procedure SetHeight(A: Integer);
    procedure SetCaption(A: String);
  public
    constructor Create(AOwner: TComponent); override;
  published
    property Caption: String read FCaption write SetCaption;
    property Position: TPositions read FPosition write SetPosition default
poTopRight;
    property Visible: Boolean read FVisible write SetVisible default True;
    property Width: Integer read FWidth write SetWidth default 250;
    property Height: Integer read FHeight write SetHeight default 200;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Dialogs', [TDebugBox]);
end;

constructor TDebugBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPosition := poTopRight;
  FVisible := False;
  FWidth := 250;
  FHeight := 200;
  FCaption := 'Debug Box';
  if not (csDesigning in ComponentState) then
    begin
```

```
      DebugForm := TForm.Create(Application);
      with DebugForm do
        begin
          Visible := FVisible;
          Caption := FCaption;
          FormStyle := fsStayOnTop;
          BorderStyle := bsSizeable;
          BorderIcons := [biSystemMenu];
        end;
      DebugList := TListBox.Create(DebugForm);
      with DebugList do
        begin
          Parent := DebugForm;
          Align := alClient;
        end;
    end;
end;

procedure TDebugBox.SetPosition(A: TPositions);
begin
  FPosition := A;
  if not (csDesigning in ComponentState) then with DebugForm do
    case A of
      poTopLeft     : SetBounds(0,0,Width,Height);
      poBottomLeft  : SetBounds(0,Screen.Height-Height,Width,Height);
      poTopRight    : SetBounds(Screen.Width-Width,0,Width,Height);
      poBottomRight : SetBounds(Screen.Width-Width,Screen.Height-
Height,Width,Height);
    end;
end;

procedure TDebugBox.SetVisible(A: Boolean);
begin
  FVisible := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Hide;
      if A then
        begin
          Width := Self.Width;
          Height := Self.Height;
          SetPosition(FPosition);
          DebugForm.Show;
        end;
    end;
end;

procedure TDebugBox.SetWidth(A: Integer);
begin
  FWidth := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Width := FWidth;
      SetPosition(FPosition);
    end;
end;

procedure TDebugBox.SetHeight(A: Integer);
begin
  FHeight := A;
  if not (csDesigning in ComponentState) then
    begin
      DebugForm.Height := FHeight;
```

```
        SetPosition(FPosition);
      end;
  end;

  procedure TDebugBox.SetCaption(A: String);
  begin
    FCaption := A;
    if not (csDesigning in ComponentState) then
      DebugForm.Caption := FCaption;
  end;

  end.
```

Return to Debug Box Component

Return to Front Page



A Call for "Delphi Compatible" Standards

By Robert Pullan - President: Lighthouse Technologies - CIS: 102162,2711

Delphi is a wonderful product, amazingly stable for a 1.x version and very rich and robust in its potential. However, as of July 1995, Delphi is sorely lacking in **true** third party support, and riddled with third party "Delphi wanna-be" providers selling products that simply don't work or don't provide a user with meaningful Delphi solutions. Each copy of Delphi contains a catalog listing more than 100 such products.

The power of third party products/libraries is unmistakable and a potent way to multiply programming productivity and a great way to add substantial functionality to an app at relatively minimal effort. I will be reviewing a few of these products in upcoming issues of this publication and want to establish some standards upon which we can benchmark at least suitability as a Delphi product.

The bulk of the third party GUI support seems to be aimed at the VBX, at present. A Visual Basic v.1 compliant VBX standard has evolved and this standard has allowed third party developers to produce products that potentially work in Visual Basic (VB), Delphi, Powerbuilder and other GUI development systems. There are better ways for this market to have evolved, but it hasn't. However, this VB 1 standard comes at a price because these VBXs have three key limitations:

1) VB 1 standards have evolved, rather than being strictly documented and regulated. This means this spec is not absolute and the degree to which a VBX works in Delphi, or other products, can vary.
2) Adherence to VB 1 compatibility means VBXs will always be 16 bit add-ons... which have implications for Win95 apps and migration among others,
3) VB 1 did not allow for data-aware VBXs, like the powerful TrueGrid, or Smithware's Controls for Btrieve. If comments on the Delphi CIS forum are any indication, Delphi's Database draws the greatest amount of traffic by far for any other Delphi topic... suggesting database development is a major use for Delphi and this constraint probably is a greater limitation than might otherwise be obvious.

Delphi does provide version 1 VBX compatibility. However, that is not the only concern for meaningful functionality in a Delphi environment.

DLLs and Declarations Files

Many VBXs not only provide the user with properties and events, but also offer methods upon which one can perform various actions (e.g. Printing, etc.). These methods are typically and ideally stored in DLLs. Delphi requires functions and procedures stored in DLLs be declared before they can be used. Trying to translate VB or "C" declarations into Pascal, however, becomes a very involved guessing game if you are trying to decipher DLL declarations for something one did not have a hand in developing or access to source code.

Documentation

Third party product developers provide documentation not only in the form of written manuals, but also as Windows Help files and examples. Filing to make this information available in Pascal/Delphi format certainly makes the task of working with these products more difficult, and expensive. A programmer then needs to have proficiency in both Delphi and a second programming language to even get a clue as to how these products may be used.

Tech Support

We need someone in tech support that speaks the Delphi language. It is nice to know how something works in VB or "C", but it is often futile and frustrating when confronted with a tech staff that simply hasn't a clue how (or often if) the product works in Delphi.

The end result is that one can buy a third party VBX that does load itself onto the Delphi palette, but if it uses a DLL, and the VBX developer has not provided a Delphi/Pascal declaration (.PAS) file and or Delphi/Pascal oriented documentation or support, then you have bought yourself a canoe without a paddle and are about to head down the old creek and into some raging rapids with the vendor promising help "soon".

Absolute "Delphi Compatible" Standards

I would like to propose a minimal standard for third party providers who want to sell Delphi compliant products. Further I will suggest that buyers rigorously insist upon products that comply with these standards:

1) The VBXs should be tested with Delphi to ensure that they operate at a level comparable to VB. Ideally this should happen before ad and or packaging copy is put into production... many seem to like waiting to see what happens before they make the considerable commitment to produce a "Delphi Compatible" product.
2) If the product requires declarations to fully function, the requisite .PAS files be provided.
3) The product should ship with documentation and examples oriented toward or with the Pascal user in mind and comparable to that for other languages supported.
4) The provider have a Pascal-conversant tech support capability to resolve bugs and Delphi customer queries.

Each of these issues will figure prominently in my reviews.

It is debatable whether or how soon a large selection of Delphi VCL components will evolve (obviously native VCLs are the optimal choice for Delphi developers.) Perhaps when all of the hype about Win95 cools down, people will realize the market for new 16-bit Win 3.x applications will remain robust for many years. Remember, most existing end users lack the hardware to run Win95 in the real world (Microsoft specs Win 3.x hardware minimums at a 286 with at least 3 meg) and if Win 3.x migration is a precedent, few end-users will trade-up until Win95 apps become more functional than Win 3.x apps... no, I am not saying Win 95 will never happen, simply it won't happen overnight and the transition to a Win95 dominated world will take time.

Microsoft has revealed Visual Basic 4 will be a 32-bit product designed for Win95 development. This represents a real opportunity for a dramatic expansion of the Delphi user base. Clearly Delphi is the best 16-bit RAD product and there will be no 16-bit alternative for VB programmers.16-bit programs will run on both Win95 and Win 3.x machines providing needed transitional consistency for those end users with multiple PCs. Hopefully this will provide greater incentives for third party Delphi product vendors to produce more and better Delphi native (VCL) and compliant (VBX) add-ons.

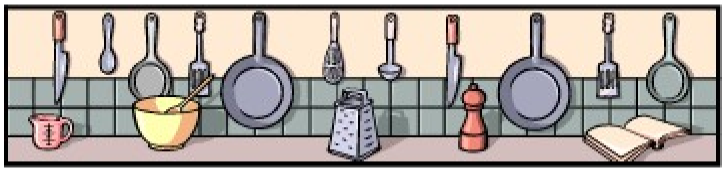
Finally, I want to be certain those vendors who's products do meet these minimal standards are not forgotten. There are some straight shooters out there and these are the folks we should be encouraging with our Delphi add-on purchases.

Review of Borland Visual Solutions Pack

Return to Front Page

Delphi Developer - July '95

Delphi Developer, PO Box 888, Kent, MA   98035-0888
CIS: 76064,51   /   Voice: (800) 788-1900   /   Fax: (206) 251-5057



Cooking Up Components

As I promised last issue, I am continuing our discussion on the **StatusBar** component. This issue extends the components behavior to include a resource gauge and enhances its look a bit. In addition, I have included a little component called **DebugBox** that I am using to allow the presentation of debugging information while your program runs. Check 'em out!

Return to Front Page

Status Bar Source Code

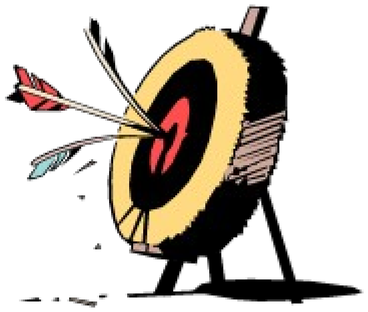
Simply use the Edit|Copy command from the menu above to copy this code to the clipboard. Then go into Delphi, create a new Unit file, and paste the code into that empty file. Save the file as "STATUS.PAS" Then, all you have to do is pick Install Components from the Options menu.

```
unit Status;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, Menus, Gauges;

type
  TStatus = class(TCustomPanel)
  private
    FDate         : Boolean;
    FKeys         : Boolean;
    FTime         : Boolean;
    FResources    : Boolean;
    DateTimePanel : TPanel;
    ResPanel      : TPanel;
    ResGauge      : TGauge;
    CapPanel      : TPanel;
    NumPanel      : TPanel;
    InsPanel      : TPanel;
    HelpPanel     : TPanel;
    UpdateWidth   : Boolean;
    FTimer        : TTimer;
    procedure SetDate(A: Boolean);
    procedure SetKeys(A: Boolean);
    procedure SetTime(A: Boolean);
    procedure SetResources(A: Boolean);
    procedure SetCaption(A: String);
    Function  GetCaption: String;
    procedure CMFontChanged(var Message: TMessage); message CM_FONTCHANGED;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SetupPanelFields(ThePanel: TPanel);
    procedure SetupPanel(ThePanel: TPanel; WidthMask: String);
    procedure UpdateStatusBar(Sender: TObject);
  published
    property ShowDate: Boolean read FDate write SetDate default True;
    property ShowKeys: Boolean read FKeys write SetKeys default True;
    property ShowTime: Boolean read FTime write SetTime default True;
    property ShowResources: Boolean read FResources write SetResources default
True;
    property BevelInner;
    property BevelOuter;
    property BevelWidth;
    property BorderStyle;
    property BorderWidth;
    property Caption: string read GetCaption write SetCaption;
    property Color;
    property Ctl3D;
    property DragCursor;
    property DragMode;
    property Enabled;
    property Font;
    property ParentColor;
    property ParentCtl3d;
    property ParentFont;
    property ParentShowHint;
    property PopUpMenu;
    property ShowHint;
    property Visible;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Additional', [TStatus]);
end;

procedure TStatus.SetupPanelFields(ThePanel: TPanel);
begin
  with ThePanel do
    begin
      Alignment := taCenter;
      Caption := '';
      BevelInner := bvLowered;
      BevelOuter := bvNone;
      {Set all these true so they reflect the settings of the TStatus}
      ParentColor := True;
      ParentFont := True;
      ParentCtl3D := True;
    end;
end;

procedure TStatus.SetupPanel(ThePanel: TPanel; WidthMask: String);
begin
  SetupPanelFields(ThePanel);
  with ThePanel do
    begin
      Width := Canvas.TextWidth(WidthMask);
      Align := alRight;
    end;
end;

constructor TStatus.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Parent := TWinControl(AOwner);
  FTime := True;
  FDate := True;
  FKeys := True;
  FResources := True;
  {Force the status bar to be aligned bottom}
  Align := alBottom;
  Height := 19;
  BevelInner := bvNone;
  BevelOuter := bvRaised;
  {When UpdateWidth is set TRUE, status bar will recalculate panel widths once}
  UpdateWidth := True;
  Locked := True;
  TabOrder := 0;;
  TabStop := False;
  Font.Name := 'Arial';
  Font.Size := 8;
  {Create the panel that will hold the date & time}
  DateTimePanel := TPanel.Create(Self);
  DateTimePanel.Parent := Self;
  SetupPanel(DateTimePanel,'  00/00/00 00:00:00 am  ');
  {Create the panel that will hold the resources graph}
  ResPanel := TPanel.Create(Self);
  ResPanel.Parent := Self;
  SetupPanel(ResPanel,'                    ');
  {Create the 2 Gauges that will reside within the Resource Panel}
  ResGauge := TGauge.Create(Self);
  ResGauge.Parent := ResPanel;
  ResGauge.Align := alClient;
  ResGauge.ParentFont := True;
  ResGauge.BackColor := Color;
  ResGauge.ForeColor := clLime;
  ResGauge.BorderStyle := bsNone;
  {Create the panel that will hold the CapsLock state}
  CapPanel := TPanel.Create(Self);
  CapPanel.Parent := Self;
  SetupPanel(CapPanel,'  Cap  ');
  {Create the panel that will hold the NumLock state}
  NumPanel := TPanel.Create(Self);
  NumPanel.Parent := Self;
  SetupPanel(NumPanel,'  Num  ');
  {Create the panel that will hold the Insert/Overwrite state}
  InsPanel := TPanel.Create(Self);
  InsPanel.Parent := Self;
  SetupPanel(InsPanel,'  Ins  ');
  {Create the panel that will hold the status text}
  HelpPanel := TPanel.Create(Self);
  HelpPanel.Parent := Self;
  SetupPanelFields(HelpPanel);
  {Have the help panel consume all remaining space}
  HelpPanel.Align := alClient;
  HelpPanel.Alignment := taLeftJustify;
  {This is the timer that will update the status bar at regular intervals}
  FTimer := TTimer.Create(Self);
  If FTimer <> Nil then
    begin
      FTimer.OnTimer := UpdateStatusBar;
      {Updates will occur twice a second}
      FTimer.Interval := 500;
      FTimer.Enabled := True;
    end;
end;

destructor TStatus.Destroy;
begin
  FTimer.Free;
  HelpPanel.Free;
  InsPanel.Free;
  NumPanel.Free;
  CapPanel.Free;
  ResGauge.Free;
  ResPanel.Free;
  DateTimePanel.Free;
  inherited Destroy;
end;

procedure TStatus.SetDate(A: Boolean);
begin
  FDate := A;
  UpdateWidth := True;
```

```
end;

procedure TStatus.SetKeys(A: Boolean);
begin
  FKeys := A;
  UpdateWidth := True;
end;

procedure TStatus.SetTime(A: Boolean);
begin
  FTime := A;
  UpdateWidth := True;
end;

procedure TStatus.SetResources(A: Boolean);
begin
  FResources := A;
  UpdateWidth := True;
end;

{When we set or get the TStatus caption, it is affecting the HelpPanel caption
instead}
procedure TStatus.SetCaption(A: String);
begin
  HelpPanel.Caption := ' '+A;
end;

function TStatus.GetCaption: String;
begin
  GetCaption := HelpPanel.Caption;
end;

{This procedure sets the captions appropriately}
procedure TStatus.UpdateStatusBar(Sender: TObject);
begin
  if ShowDate and ShowTime then
    DateTimePanel.Caption := DateTimeToStr(Now)
  else
    if ShowDate and not ShowTime then
      DateTimePanel.Caption := DateToStr(Date)
    else
      if not ShowDate and ShowTime then
        DateTimePanel.Caption := TimeToStr(Time)
      else
        DateTimePanel.Caption := '';
  if UpdateWidth then with DateTimePanel do
    if ShowDate or ShowTime then
      Width := Canvas.TextWidth('  '+Caption+'  ')
    else
      Width := 0;
  if ShowResources then
    begin
      ResGauge.Progress := GetFreeSystemResources(GFSR_SYSTEMRESOURCES);
      if ResGauge.Progress < 20 then
        ResGauge.ForeColor := clRed
      else
        ResGauge.ForeColor := clLime;
    end;
  if UpdateWidth then
    if ShowResources then
      ResPanel.Width := Canvas.TextWidth('                     ')
    else
```

```
      ResPanel.Width := 0;
    if ShowKeys then
      begin
        if (GetKeyState(vk_NumLock) and $01) <> 0 then
          NumPanel.Caption := '  Num  ' else NumPanel.Caption := '';
        if (GetKeyState(vk_Capital) and $01) <> 0 then
          CapPanel.Caption := '  Cap  ' else CapPanel.Caption := '';
        if (GetKeyState(vk_Insert) and $01) <> 0 then
          InsPanel.Caption := '  Ins  ' else InsPanel.Caption := '';
      end;
    if UpdateWidth then
      if ShowKeys then
        begin
          NumPanel.Width := Canvas.TextWidth(' Num ');
          InsPanel.Width := Canvas.TextWidth(' Ins ');
          CapPanel.Width := Canvas.TextWidth(' Cap ');
        end
      else
        begin
          NumPanel.Width := 0;
          InsPanel.Width := 0;
          CapPanel.Width := 0;
        end;
    UpdateWidth := False;
  end;

  {This allows font changes to be detected so the panels will be adjusted}
  procedure TStatus.CMFontChanged(var Message: TMessage);
  begin
    inherited;
    UpdateWidth := True;
  end;

  end.
```

Return to Cooking Up Components
Return to Front Page



Tips & Tricks

by Robert Vivrette

Return to Front Page

Delphi Developer - Premier Issue - June '95

Delphi Developer, PO Box 888, Kent, MA 98035-0888
CIS: 76064,51 / Voice: (800) 788-1900 / Fax: (206) 251-5057

Delphi Developer's Journal - June 95

Delphi Developer's Journal
PO Box 35160, Louisville, KY   40232
EMail: delphi_dev@merlin.cobb.ziff.com   /   Voice: (800) 223-8720   /   Fax: (502) 491-8050

Delphi Developer's Journal July 95

Delphi Developer's Journal

PO Box 35160, Louisville, KY 40232

EMail: delphi_dev@merlin.cobb.ziff.com / Voice: (800) 223-8720 / Fax: (502) 491-8050

Delphi Informant June 95

Delphi Informant, 10519 E. Stockton Blvd., Suite 142, Elk Grove, CA   95624-9704

CIS: 70304,3633   /   Voice: (916) 686-6610   /   Fax: (916) 686-8497

Delphi Informant March 95

Visual Programming

Provides a brief visual tour of the Delphi IDE discussing the various features such as the Component Palette, Object Inspector, Code editor, and more.

The Way of Delphi

Illustrates the fundamentals of component design and begins with a sample component. A very good tutorial for users who have not yet stepped into component creation.

DataSource1

Demonstrates a number of Delphi's capabilities including modifying a Query component at run-time, sharing an event handler among multiple components, and creating a dynamic tabbed interface to filter database information.

Sights & Sounds

Discusses the creation of a simple Audio CD player application. Very good introduction to the Media Player component as well as connections to the Windows API.

Delphi C/S

Presents a discussion of Delphi Client/Server capabilities covering issues such as ODBC, the BDE, and the Delphi Database Form Expert.

DBNavigator

Discusses Delphi's database features covering topics including InterBase, Paradox for Windows, and dBase for Windows, as well as the BDE.

From The Palette

Introduces the basics of Component design, including property and method basics and the proper use of class directives.

At Your Fingertips

A "tips & tricks" column devoted to help programmers quickly get up to speed on Delphi and Object Pascal. This issue's column includes: Implementing a status bar, locating values in a table, and filtering records in a DBGrid.

API Calls

Provides a very good discussion of dynamically calling DLL functions from within Delphi.

Delphi Informant, 10519 E. Stockton Blvd., Suite 142, Elk Grove, CA   95624-9704

CIS: 70304,3633   /   Voice: (916) 686-6610   /   Fax: (916) 686-8497

Delphi Informant July 95

The DBTracker Utility - Inside Delphi's Database Control Events - *John O'Connell*

Delphi's database connectivity is based on a number of components that work together to allow your applications access to database tables. At a basic level, database access is provided by a component descended from TDataSet, *such as* TTable, TQuery, *and* TStoredProc.

Moving to Local Interbase - The First Step Towards Upsizing A Delphi Application to Client Server - *Marco Cantu'*

Upsizing is the process of taking an application that works on a PC and making it more robust by moving it to client/server. This is the opposite of downsizing, or taking a mainframe or mini-computer application and moving it to the client/server model.

The TField Class: Part II - Using Calculated Fields - *Cary Jensen, Ph.D.*

TField components are a class of objects that permit you to directly manipulate the data displayed on a form. These objects are created automatically at run-time or manually at design time. By creating TFields at design time, you can exert greater control over the default characteristics of the corresponding fields. For example, you can display only some of the fields from a table in a DBGrid component.

A Question of Size: Or, Why Are Delphi Executables So Big? - *Robert Vivrette*

Everyone loves small, efficient programs. Perhaps that explains all the hubbub. Since the release of Delphi, there has been quite a bit of discussion about the size of executable files created with Delphi. In this article we'll discuss some of the issues relating to executable size, and present a detailed picture of what exactly goes into a Delphi executable.

A Little Memo Help: Manipulating the Cursor in Delphi's Memo Components - *Bill Todd*

Delphi includes two classes of memo control components, TMemo and TDBMemo, that enable the user to edit text. Since both of these controls encapsulate the Windows multi-line edit control, they are limited to a maximum of 32kb of text.

Meet The .DFM File: Modifying Delphi's "File Behind The Form" - *Sedge Simons, Ph.D.*

The graphical properties of a Delphi form and its components are stored in a .DFM file. In this article, we'll see how to open and edit this file, making short work of global changes or component creation, and giving you an additional approach for troubleshooting.

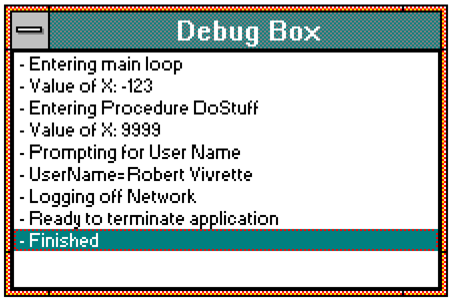
The Developers Visual Suite Deal - *Review by Doug Horn*

Instant Delphi Programming - *Review by Jerry Coffey*

Delphi By Example - *Review by Jerry Coffey*

Delphi Informant, 10519 E. Stockton Blvd., Suite 142, Elk Grove, CA 95624-9704

CIS: 70304,3633 / Voice: (916) 686-6610 / Fax: (916) 686-8497



- Entering main loop
- Value of X: -123
- Entering Procedure DoStuff
- Value of X: 9999
- Prompting for User Name
- UserName=Robert Vivrette
- Logging off Network
- Ready to terminate application
- Finished

The DebugBox Component

by Robert Vivrette

Sometimes when I write programs, I like to know what is going on inside the program. I may want to know what particular piece of code I am in, and I may also want to examine particular variables as they change during the time the program is executing. To help me out in this area, I created the DebugBox. Originally developed in Borland Pascal for Windows, the DebugBox enabled me to do all of these things without having to resort to using the debugger, and also not having to use message boxes to display run-time information.

Now with Delphi, we have a fully-integrated debugger that makes most of this seem unneccesary. However, with DebugBox (and with some of the modifications I will make to is in later issues), I can gain a number of advantages:

1. I can examine information while the program is running at virtually top speed. (i.e. no need to "step" through the program with the debugger just to examine a value). Further, I don't have to use message boxes, which disturb the visual flow of the program.
2. I can feed a great deal of data into the debug box and then look at it later before the program quits. In a future issue, I will put in a procedure to save the information in the debug box out to a disk file.
3. I can programatically hide or show the box. In this way, I can even keep the debugging lines in the final program and just turn off the box so the user never notices that it even exists. I do this in one network application where the DebugBox keeps a log of activity and then, in the event of an error, copies that log to me across the network.

The DebugBox component is a non-visual component. To use it, simply drop it onto your main form. You only need one TDebugBox, so don't go scattering them all over your application. The box is initially hidden, so you can assign "True" to it's **Visible** property to show it. To add strings into the debug box, simply call the **Add** method, passing in the string that you wish to append. To clear the entire list, call the **Clear** method. You can also control its initial placement, size, window caption, and other properties at design-time with the Object Inspector. Below is an example of how it might be used in a section of code:

```
  {Attempt to retrieve users Banyan ID}
  if GetBanyanID then
    DebugBox1.Add('- Banyan ID retrieved: '+ID)
  else
    begin
      DebugBox1.Add('x Could not get users Banyan ID.');
      MessageBox(0,'You must be logged on to Banyan to run this
program.','Error',MB_ICONSTOP);
      Exit;
    end;
```

Next issue I will extend some of the features of the DebugBox component. If you have suggestions, please send them in! By the way, this is the bitmap I use for the DebugBox on the Component Palette:

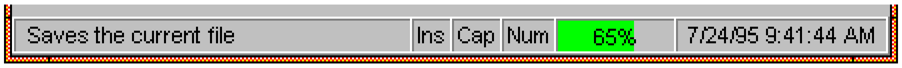
so you can clip it off using SnagIt or some other clipboard utility!
Debug Box Source Code

Return to Front Page

Status Bar Component

by Robert Vivrette

As I promised last month, this issue I am getting back to the **Status Bar** project first presented in issue #4 . The key changes that have been made to the component is the addition of a resource gauge, and the separation of each of the key states into their own panels.



The entire status bar is actually a collection of panels strung together. Each panel has the ability to show some bit of information (such as the date/time, key states, or hint information). The component creates each one of the panels in sequence, marking the **TStatus** as its owner. Then all of the panels except for the hint panel are aligned as alRight which snugs each one up against the one created before it. Lastly, the hint panel is set to align as alClient which tells it to use up the remainder of the room across the form. In this way, when the form is resized, the hint panel will resize to take advantage of the extra room. In addition, the panels all have their ParentColor, ParentFont, and ParentCtl3D properties set to True so that the user of the component can alter the Font, Color, or Ctl3D property of the TStatus, and all the panels will individually reflect these changes as well.
In addition, the resource panel itself contains its own child component... a progress bar component called **TGauge.** This gauge component was included with Delphi on the Samples page, so you will need to make sure that you have not removed it from the component library. The gauge is aligned within its parent panel with the alClient align property. That way, if the status bar is adjusted either horizontally (by resizing the form), or vertically (by resizing the TStatus at design time), the gauge will continue to fill the panel appropriately. The key to the resource gauge is a windows API procedure called *GetFreeSystemResources*. This procedure can return the number of remaining GDI resources as well as the USER resources. In this component, I am using the GFSR_SYSTEMRESOURCES parameter to return the lower of these two numbers, which is generally referred to as "free system resources". The value that is returned is a WORD value from 0-100 which represents the percentage of the resource heaps that are currently available.

Adding the component to your project is easy. Simply place the status bar on the form, and it will instantly drop to the bottom edge and align itself appropriately. Then make sure the following code is in that forms unit:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := ShowHint;
end;

procedure TForm1.ShowHint(Sender: TObject);
begin
  Status1.Caption := Application.Hint;
end;
```

You will have to manually add a forward declaration for the ShowHint procedure to the TForm1 class definition at the top, which would look like this:

```
type
  TForm1 = class(TForm)
    procedure ShowHint(Sender: TObject);
    (etc...)
  end;
```

<u>Status Bar Source Code</u>

Return to Cooking Up Components
Return to Front Page

Dynamic Connections to A DLL

by Robert Vivrette

Recently, I went through a bit of work connecting to some DLL's across our network, and I thought I would share some of my experiences so that others might understand the issue a bit better. Here at work, we use the Banyan Vines networking system, and I needed the ability to access some of the network features from within my programs, such as getting information on the user, or establishing drive connections to network services. In the past I had gone through some similar steps while programming in Borland Pascal for Windows. When Delphi came out, I decided that it was time to rethink this code and to make native Banyan components that would achieve the effects I needed.

With Banyan, most of the network DLL's that you will ever need are automatically located on your "Z" drive when you are logged in. When my PC boots up, it loads some network drivers that creates a Z drive on my system, which is then mapped over to some network server somewhere. On this drive is the DLL I will need to use to make these components work.

So... no big deal... to gain access to a function or procedure within a DLL, you simply do something like this:

```
Unit NotAGoodIdea;
interface
  function MyDLLFunction: Integer;
implementation
  function MyDLLFunction; external 'MYFUNC.DLL' index 1
end.
```

The problem is that when the program launches, one of its first actions is going to be to establish the connection to the DLL. If it is loaded, no problem. But if it is not loaded for some reason, Windows will generate a big, ugly error box and bomb out of the program. With the code put together this way, there is little you can do about it because you're code really has not even begun to execute yet!

In the case of the Banyan component then, what happens if, for example, you have started the PC without the Banyan drivers loaded? Without the drivers loaded, you have no Z drive, and so the DLL will be missing. You would not want your finely crafted component to generate this ugly system error if it is placed on a form when the PC does not have access to the DLL.

This is where the "dynamic" portion of this procedure comes in. Connecting dynamically, means that you are going to programatically connect to the DLL after determining if it is safe to do so, and handling any adverse conditions that might arise. This effect can be achieved by means of the *LoadLibrary* and *FreeLibrary* Windows API calls. Lets look at how it is done now... (this is not the complete code, just the interesting pieces):

```
type
  ProcGetUserName        = FUNCTION(ID : PChar): Integer;
var
  VnsGetUserName         : ProcGetUserName;

constructor TVines.Create(AOwner: TComponent);
var
  LastState : Word;
  ThePtr    : Pointer;
begin
  inherited Create(AOwner);
  {Suppress the 'file not found' system error box from Windows}
  LastState := SetErrorMode(sem_NoOpenFileErrorBox);
  {Load the vines API DLL library}
  hVinesDLL := LoadLibrary('Z:\VNSAPI.DLL');
  {Restore the windows system error state}
  SetErrorMode(LastState);
  {If the return value from LoadLibrary is greater than the
   constant HINSTANCE_ERROR, then the load was sucessful.}
```

```
    VinesAvailable := (hVinesDLL > HINSTANCE_ERROR);
    {Go get a pointer to the address of the VnsGetUserName procedure}
    ThePtr := GetProcAddress(hVinesDLL,'VnsGetUserName');
    {Typecast the pointer as a procedure of type ProcGetUserName}
    VnsGetUserName := ProcGetUserName(ThePtr);
  end;
```

Now later on, when you want to access the function in the DLL, you would do something like this:

```
  Procedure TVines.GetTheUser;
  var
    ID : Array[0..80] of Char;
  begin
    {Setup a array to hold the users name}
    FillChar(ID,SizeOf(ID),#0);
    {Don't call the function if the DLL was not loaded correctly!}
    if VinesAvailable then
      begin
        {Call the DLL to retrieve the users name}
        FError := VnsGetUserName(ID);
        if (VinesError = 0) then
          begin
            {function call was good, clear any error message...}
            FMessage := '';
            {... and save the user name}
            FUser := StrPas(ID);
          end
        else
          begin
            {DLL call failed, set an error message...}
            FMessage := 'Unable to identify user';
            {... and call any user-defined error handler}
            If Assigned(FOnError) then FOnError(Self);
          end;
      end;
  end;
```

Lastly, when the component is destroyed, you need to free the DLL:

```
  destructor TVines.Destroy;
  begin
    If VinesAvailable Then FreeLibrary(hVinesDLL);
    inherited Destroy;
  end;
```

In practice, the final component is very well behaved. If you do not have the network drivers loaded, the component doesn't crash on you, but instead reports an error message in a slot in the object inspector. At runtime, you can provide a message to the user that he must have the Banyan drivers loaded in order to run the program.

The final components are a bit more complex than this, but I hope this does give you some insight on how a dynamic DLL connection can be achieved.

Return to Tips & Tricks
Return to Front Page

Initializing After Loading

When I was working on my Banyan Vines components, I decided that I was going to add the ability to have my SetDrive component automatically connect to a network service when the program executes. The only problem was determining where this automatic connection would occur. I couldn't do it in the Create method, because the SetDrive component is not done being built at that point. The solution is the **Loaded** method.

After a component reads all its property values from its stored description, it calls a virtual method called **Loaded**, which provides a chance to perform any initializations that might be required. In my case, it would perform the connection to the network service and map a new drive letter to that service.

Now, the user of the component can simply plop one of the SetDrive components onto his application, set some of the properties to indicate the service to connect to, and set the AutoConnect property to True. Now, when the program runs, it will automatically perform the connection without the need to have ANY code to control the component in the program. When the program ends, it automatically terminates the connection too! Pretty Cool!

This is a little bit of the code to show how it is done...

```
    TVinesDrives = class(TVinesLogin)
    private
      FAutoConnect  : Boolean;
      FDrive        : Char;
      FService      : String;
      FRoot         : String;
      FOnConnect    : TNotifyEvent;
      FOnDisconnect : TNotifyEvent;
      IsConnected   : Boolean;
      {other private declarations here}
    protected
      procedure Loaded; override;
    public
      {other public declarations here}
      procedure Connect;
      procedure Disconnect;
    published
      {other published declarations here}
      property AutoConnect: Boolean read FAutoConnect write FAutoConnect default
  False;
    end;

  procedure TVinesDrives.Loaded;
  begin
    inherited Loaded;
    if (not (csDesigning in ComponentState)) and FAutoConnect then Connect;
  end;
```

Return to Tips & Tricks

Return to Front Page

Need to find out how many components are on a form? You can use the ComponentCount method. As an example, you can change all panels on a form to be blue as follows:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a : Integer;
begin
  for a := 1 to ComponentCount do
     if Components[a-1] is TPanel then
       TPanel(Components[a-1]).Color := clBlue;
end;
```

ComponentCount is also a property of the Application object so you can use it to find out how many things are owned by the main TApplication object.

You can find a component by name by using the **FindComponent** method. The example code below will allow a user to enter the name of a component in an edit box, and the program will then use that name to change that component a different color.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyComponent: TComponent;
begin
  MyComponent := FindComponent(Edit1.Text);
  If MyComponent is TPanel then
    TPanel(MyComponent).Color := clBlue;
end;
```

Have you ever wanted to change the name of an application when it is minimized? Assign a value to the **Application.Title** property. This does not affect the caption of the main form, but when you minimize the application, the minimized icon will be named according to the Title property. You could also use this to show a current file in use as follows:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  Application.Title := 'Accounting Module: '+CurrentFileName;
end;
```

This actually has nothing at all to do with Delphi, but I thought it was **cool** and wanted to mention it anyway. About 2 weeks ago, I started subscribing to a service called **TVHost** which is essentially an electronic *TV Guide*. You run a Windows program on your PC and once a week, you retrieve a data file of detailed program listings from the Internet (about 700K zipped) which automatically attaches itself into the program. There are dozens of files available, each one covering a different geographic area.

The program is solid and easy to use, and I haven't had a single problem with it. It allows you to mark shows as a "favorite", so each week, it will gracefully notify what specific episode of your favorite shows are coming up. You can customize which channels you are interested in, or even for what time frames you are interested, and you can print out a variety of schedules and information. About 90-95% of the shows have textual descriptions of the specific episode, and most movies have cast lists attached.

You can perform **very** detailed searches that allow you to find when and where that particular movie is playing (you know... that one you've been **dying** to see...) The cost is $29.95 for 6 months, or $49.95 for a full year. You can download the program (with demo data files) from CompuServe or the Net. Then when you choose to subscribe, you are given a code which immediately activates the program for live data (for the full term of your subscription).

You can reach TVHost at `etv@tvhost.com` or by phone at (717) 657-1700 (make sure you press '4' to get to the *electronic* folks, as other options are for the *non-electronic* subscription services they offer as well). You can also get a copy of the program on the **TVZONE** forum on CompuServe. The file is called ETVDEMO.EXE and is in the "Episode Guides" file section. They have mentioned to me that they have set up a **free** 3-week trial offer. When you download the program, you can enter a code to activate it for the time period of **Aug 1st** to **Aug 18th**, 1995. The code is: **5784-8788-9028-2905**.

Very cool system and highly recommended!