WideString Properties

by Ray Lischner

Last month you read all about wide strings and Unicode, so now you are ready to use wide string properties on your custom components. Oops. Delphi's IDE does not support published properties of `WideString` type. Strictly speaking, the IDE supports `WideString`, but it does so by mapping all wide characters to ANSI characters. If you wanted to use ANSI characters, you would have used a plain ANSI string, right? This article explains what is wrong with the IDE and how to work around its limitations so your custom components can publish properties of type `WideString`. It's harder than it should be, and some of the code gets into Delphi internals. Do not read this article without proper supervision. You've been warned.

Unicode And WideString

If you missed last month's article on wide strings, here's a brief summary of wide strings in Delphi.

Windows NT and Windows 2000 support diverse languages and character sets using Unicode, a 16-bit character set that unifies Asian, Arabic, European and other character sets in a single character set. The character set standard is maintained by ISO and the Unicode Consortium (www.unicode.org).

Delphi supports Unicode with its `WideChar` and `WideString` types. `WideChar` is a 2-byte character that stores a Unicode character. `WideString` is a string of `WideChar` characters. Like an ANSI string (also called a narrow string or, more often, just 'string'), Delphi automatically manages the lifetime and memory of a wide string. The `WideString` type stores the string's length and automatically places a `#0` word at the end of the string's contents. Unlike the `AnsiString` type, though, `WideString` is not reference counted. When you assign one `WideString` to another `WideString`-type variable, Delphi must copy the entire string. (The lack of reference counting is to ensure compatibility with Windows. Kylix does not suffer from this requirement, so `WideStrings` in Kylix are reference counted.)

Windows NT and Windows 2000 have ANSI (narrow) and Unicode (wide) versions of their standard controls: edit boxes, list views and so on. Most Windows API functions also come in ANSI and wide varieties. Windows 9x does not support Unicode or the wide API calls. You can try to run the application, but Windows informs you that it does not support the function call.

If you look at Delphi's Source\Rtl\Win\Windows.pas file, you can see declarations for the different functions, such as `CreateFileA` and `CreateFileW`. The plain name, `CreateFile`, is the same as `CreateFileA`. The `A` version of the function takes a narrow string argument (`PChar`) whilst the `W` version takes a wide string argument (`PWideChar`).

If you often need to use the wide functions and controls, you should write your own wrapper functions (similar to those in the `SysUtils` unit) and controls. For example, suppose you want to define a Unicode label component. Listing 1 shows a simple declaration of the `TWideLabel` class. It inherits from `TLabel`, but it adds the `WideCaption` property and overrides `DoDrawText` to draw the wide string caption. Listing 2 shows the implementation of the `DoDrawText` method. Notice how it calls the `DrawTextW` API to display the wide string. As you can see, implementing a static wide control is not difficult.

Implementing a wide window control is problematic, though. The VCL makes many calls to the Windows API, always using the ANSI versions of the API functions, which you would need to replace with the wide versions of the same functions. The `TWinControl` base class is closely tied to the ANSI controls and functions, and it is difficult to change it to use wide controls. It is probably simplest to write entirely new controls that don't try to inherit any of the narrow functions from `TWinControl`, such as `CreateParams`.

The Problem With Properties

A naive implementation of the `TWideLabel` component simply

```
type
  TWideLabel = class(TLabel)
  private
    fWideCaption: WideString;
    procedure SetWideCaption(const Value: WideString);
    procedure ReadCaption(Reader: TReader);
    procedure WriteCaption(Writer: TWriter);
  protected
    procedure DoDrawText(var Rect: TRect; Flags: Longint); override;
    procedure DefineProperties(Filer: TFiler); override;
  published
    property Caption: WideString read fWideCaption write SetWideCaption
      stored False;
  end;
```

➤ *Below: Listing 2, Drawing the wide caption.*

➤ *Above: Listing 1, The TWideLabel class.*

```
// TLabel.DoDrawText but changed to use DrawTextW insted of DrawText.
procedure TWideLabel.DoDrawText(var Rect: TRect; Flags: Integer);
var Text: WideString;
begin
  Text := Caption;
  if (Flags and DT_CALCRECT<>0) and ((Text='') or ShowAccelChar
    and (Text[1] = '&') and (Text[2] = #0)) then
    Text := Text + ' ';
  if not ShowAccelChar then
    Flags := Flags or DT_NOPREFIX;
  Flags := DrawTextBiDiModeFlags(Flags);
  Canvas.Font := Font;
  if not Enabled then begin
    OffsetRect(Rect, 1, 1);
    Canvas.Font.Color := clBtnHighlight;
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text), Rect, Flags);
    OffsetRect(Rect, -1, -1);
    Canvas.Font.Color := clBtnShadow;
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text), Rect, Flags);
  end else
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text), Rect, Flags);
end;
```

changes the Caption property to type WideString. With that simple change, the code compiles without error. But if you try to use the control at design-time, you notice that you cannot enter a wide string value in the Object Inspector. Even if you try to trick the IDE by editing the form as text, and entering a wide string as text, the Object Inspector displays only a series of question marks because it converts all the wide characters to ANSI characters. Even worse, when you save the .DFM, Delphi stores the wide string as an ANSI string. In other words, at design-time, a WideString property acts just like an AnsiString property.

The problem is that Delphi's IDE must run on all versions of Windows from 95 onward. That means the IDE cannot use any wide functions or controls. The Object Inspector, therefore, is restricted to using ANSI strings, even if a property is of type WideString. Because the user can't enter a wide string, the code that reads and writes .DFM files does not store wide strings either.

Saving Wide Strings

The first step to solving these problems is to convince Delphi to store the wide string as a wide string. Ironically, Delphi's streaming mechanism has the native ability to store and load wide strings, but it doesn't take advantage of that capability. Instead, you must modify your component to store your property value explicitly, by overriding DefineProperties. In this method, you must call DefineProperty to define a pseudo-property. You must also supply a method to read the property value and another method to write the value. Inside the .DFM file, a pseudo-property looks and acts just like a normal property. The only difference is that Delphi calls your methods instead of using the default methods (which map wide strings to ANSI strings).

Your custom reader and writer methods have complete control over the property's format by calling the methods of TReader and TWriter, so you can easily store and retrieve a wide string, as shown in Listing 3. (It is interesting that TReader and TWriter support wide strings, but Delphi's streaming system does not use these methods, storing wide strings as narrow strings. Perhaps Borland chose to store wide strings as narrow strings because they knew the Object Inspector's limitations.)

A bigger problem is the Object Inspector. The Object Inspector uses narrow strings, so you cannot directly edit a wide string in the Object Inspector window. You can, however, use a dialog box to edit the wide string. To do this, you need to write a property editor.

Property Editor Refresher

A property editor is a class that inherits from TPropertyEditor. You register a property editor class for certain property types, property names and components. The Object Inspector checks the type and name of each property it must display and chooses an appropriate property editor class; then it creates one instance of the class for each property. When you select a different component, the Object Inspector frees all the property editor objects and creates new ones for the new component.

A property editor has a small number of methods for working with the Object Inspector. To write a new property editor, you decide which methods you need to override to provide the custom behavior for your property editor. Delphi comes with many property editors, such as TIntegerProperty, TFontProperty and so on, all of which derive from TPropertyEditor or one of its subclasses. Usually, you can choose one of these property editor classes as the base class for your custom property editor. See Source\Toolsapi\DsgnIntf.pas to find out more about the many property editor classes that you can use.

By default, Delphi uses TStringProperty for all string-type properties, including WideString-type properties. TStringProperty converts a WideString property value to a narrow string. If you look under the hood, the TStringProperty editor calls the GetStrProp and SetStrProp routines in the TypInfo unit. That's where the conversion actually takes place. GetStrProp can read any string-valued property, and it converts the property value to an AnsiString. This can be convenient if the property's type is a ShortString, but it is not helpful when it converts a WideString to an AnsiString. Clearly, you need a GetWideStrProp function. Too bad Delphi doesn't have one.

WideString Properties

In order to get and set the value of a WideString-type property, you need to understand how Delphi stores information about published properties of all types. Now the code starts to get ugly.

Every published declaration has some information stored for use at runtime. This information is called Runtime Type Information (RTTI). For a published property, Delphi stores the property name, type, reader, writer, default value, index value, and value of the stored directive. The reader and writer can be methods, field names, or parts of aggregate fields (arrays and records). You can use static and virtual methods, but not class or dynamic methods. The reader must be a function that returns the same type as the property type,

➤ *Listing 3*

```
// Delphi stores a WideString by converting it to a narrow string,
// which discards information. Store the actual WideString data.
procedure TWideLabel.DefineProperties(Filer: TFiler);
begin
  inherited;
  Filer.DefineProperty('WideCaption', ReadCaption, WriteCaption, Caption <> '');
end;
procedure TWideLabel.ReadCaption(Reader: TReader);
begin
  fWideCaption := Reader.ReadWideString;
end;
procedure TWideLabel.WriteCaption(Writer: TWriter);
begin
  Writer.WriteWideString(Caption);
end;
```

and the writer must be a procedure that takes an argument of the same type as the property type.

Delphi stores a 32-bit value for the reader and another for the writer. The value can be a pointer or an integer, depending on how the property value is stored:

Field: The first byte of the pointer value is `$FF`, and the rest of the value is a byte offset of the field that stores the `WideString`. The compiler lets you specify the name of a scalar field, a constant index into an array-type field, a member of a record-type field, or any combination. The field reference must be a constant, though, so you cannot use a variable to index an array-type field.

Virtual method: The first byte is `$FE`, and the low-order word is a byte offset in the class's virtual method table (VMT) that contains the method pointer.

Static method: If the first byte is not `$FE` or `$FF`, the entire value points to the start of the reader or writer method. The Windows memory architecture ensures static methods can never have a pointer value starting with `$FE` or `$FF`: these addresses are reserved for the Windows kernel.

A reader method returns a `WideString` value, and the writer method takes a `WideString` argument. If the property is indexed, the index value is passed as the sole argument to a reader method or as the first argument to a writer method. An indexed property stores an integer as the index value. Delphi stores the smallest negative integer for a property that is not indexed.

For example, the declaration of the `Info` property in Listing 4 might return a reader of `$FF000004` if `FInfo` is the object's first field (the first four bytes of the object point to its VMT). Suppose also that `SetInfo` is an ordinary (non-virtual) method. The writer value would be a code pointer to the method's entry point, such as `$402004C`. Because the property is not indexed, the index value is -2,147,483,648. The `Stuff` property is indexed, with an index value of 2.

To get the RTTI for a published property, call `TypInfo.GetPropInfo` (for example). With the property RTTI and the component instance you can interpret the reader information, look up the field or method information, and obtain the property value. This is shown in `GetWideStrProp`, which gets the property's value as a `WideString`, as shown in Listing 5. The helper function `GetPropGetter` examines the property information to get a field or method address, so hiding from `GetWideStrProp` the details of virtual or static methods. Listing 6 shows the corresponding function, `SetWideStrProp` and its helper, `GetPropSetter`.

Note that the read and write functions for a property are methods, so the first argument must be the object reference (that is, `Self`). Delphi implicitly passes the object reference as the first argument to any method call. The `GetWideStrProp` and `SetWideStrProp` functions must emulate this behavior by explicitly passing the object reference. Thus, the reader function is actually called with one or two arguments: the object reference and the optional index value. The write method is called with two or three arguments: the object reference, the optional index value, and the new wide string value.

➤ *Listing 4*

```
procedure SetInfo(
  const Info: WideString);
procedure SetStuff(Index: Integer;
  const Stuff: WideString);
function GetStuff(Index: Integer):
  WideString;
...
property Info: WideString
  read fInfo write SetInfo;
property Stuff: WideString index 2
  read GetStuff write SetStuff;
```

➤ *Listing 5: Getting a WideString-type property.*

```
const
  NoIndex = Low(Integer);
type
  TWideStrProc = procedure(Instance: TObject;
    const Value: WideString) register;
  TWideStrIndexProc = procedure(Instance: TObject; Index:
    Integer; const Value: WideString) register;
  TWideStrFunc = function(Instance: TObject):
    WideString register;
  TWideStrIndexFunc = function(Instance: TObject;
    Index: Integer): WideString register;
  PPChar = ^PChar;
  PPointer = ^Pointer;
// To help access the property value, GetPropValue gets a
// pointer to the field or method. PtrType says what kind of
// pointer it is. An exception is raised for any error.
procedure GetPropGetter(Instance: TObject; PropInfo:
  PPropInfo; var PtrType: TPtrType; var Ptr: Pointer);
var Mask: LongWord;
begin
  // High word of GetProc determines how to interpret it.
  Mask := LongWord(PropInfo.GetProc) and $FF000000;
  if Mask = $FF000000 then begin
    // GetProc is a field offset in Instance. The low-order
    // 3 bytes specify the byte offset from the start of
    // Instance. Treat Instance as a pointer to add the
    // offset, then dereference that pointer to perform the
    // simple WideString assignment.
    PtrType := ptData;
    Ptr := PChar(Instance) + LongInt(PropInfo.GetProc)
      and $FFFFFF;
  end else begin
    // Otherwise, GetProc is a reference to a method,
    // either virtual or static.
    PtrType := ptCode;
    if Mask = $FE000000 then begin
      // GetProc is a virtual function offset. Only the
      // low-order 2 bytes are used for the offset into the
      // VMT. The first field in Instance is a pointer to a
      // VMT, which is a list of pointers to functions. Use
      // offset into the VMT to get actual method pointer.
      Ptr := PPChar(Instance)^ +
        LongRec(PropInfo.GetProc).Lo;
      Ptr := PPointer(Ptr)^;
    end else begin
      // GetProc is a static method pointer.
      Ptr := PropInfo.GetProc;
      if Ptr = nil then
        // No GetProc at all!
        raise EPropWriteOnly.Create(sWriteOnlyProperty);
    end;
  end;
end;
// Delphi always converts a wide string to an ANSI string
// when setting a property value. Call GetWideStrProp and
// SetWideStrProp to access the property value as a real
// WideString.
function GetWideStrProp(Instance: TObject; PropInfo:
  PPropInfo): WideString;
var
  PtrType: TPtrType;
  Ptr: Pointer;
begin
  GetPropGetter(Instance, PropInfo, PtrType, Ptr);
  if PtrType = ptData then
    Result := PWideString(Ptr)^
  else if PropInfo.Index <> NoIndex then
    // Indexed property, so call GetProc with index value.
    Result :=
     TWideStrIndexFunc(Ptr)(Instance, PropInfo.Index)
  else
    // Not an indexed property, so just call the GetProc.
    Result := TWideStrFunc(Ptr)(Instance);
end;
```

```
procedure GetPropSetter(Instance: TObject; PropInfo:
  PPropInfo; var PtrType: TPtrType; var Ptr: Pointer);
var Mask: LongWord;
begin
  // High word of SetProc determines how to interpret it.
  Mask := LongWord(PropInfo.SetProc) and $FF000000;
  if Mask = $FF000000 then begin
    // SetProc is a field offset in Instance. Low-order 3
    // bytes specify byte offset from start of Instance.
    // Treat Instance as a pointer to add the offset, then
    // dereference pointer to perform WideString assignment.
    PtrType := ptData;
    Ptr := PChar(Instance) + LongInt(PropInfo.SetProc)
      and $FFFFFF;
  end else begin
    // SetProc is reference to virtual or static method
    PtrType := ptCode;
    if Mask = $FE000000 then begin
      // SetProc is a virtual function offset. Only the
      // low-order 2 bytes are used for offset into VMT.
      // First field in Instance is pointer to a VMT, which
      // is a list of pointers to functions. Use offset into
      // the VMT to get the actual method pointer.
      Ptr := PPChar(Instance)^ +
        LongRec(PropInfo.SetProc).Lo;
      Ptr := PPointer(Ptr)^;
    end else begin
      // SetProc is a static method pointer.
      Ptr := PropInfo.SetProc;
      if Ptr = nil then          // No SetProc at all!
        raise EPropReadOnly.Create(sReadOnlyProperty);
    end;
  end;
end;
procedure SetWideStrProp(Instance: TObject; PropInfo:
  PPropInfo; const Value: WideString);
var
  PtrType: TPtrType;
  Ptr: Pointer;
begin
  GetPropSetter(Instance, PropInfo, PtrType, Ptr);
  if PtrType = ptData then
    PWideString(Ptr)^ := Value
  else if PropInfo.Index <> NoIndex then
    // Indexed property, so call SetProc with index value.
    TWideStrIndexProc(Ptr)(Instance, PropInfo.Index, Value)
  else
    // Not an indexed property, so just call the SetProc.
    TWideStrProc(Ptr)(Instance, Value);
end;
```

➤ *Listing 6: Changing a WideString-type property.*

Wide String Property Editor

Now that you can read and write `WideString` property values, it's time to finish the wide string property editor. The class `TPropertyEditor` has some convenience methods to set a property for all selected components. For example, `SetStrValue` calls `SetStrProp` for all selected components. As I mentioned earlier, `SetStrProp` takes an `AnsiString` argument, which isn't what you want for a `WideString`-type property. So, you must write your own `SetWideStrValue`, which calls `SetWideStrProp` (see the previous section). Other related methods include `GetWideStrValue` and `GetWideStrValueAt`. Listing 7 shows the wide string methods of `TWideStringProperty`.

The Object Inspector calls `GetValue` to obtain an ANSI string to display, and it calls `SetValue` when the user modifies that ANSI string. Because the Object Inspector cannot handle wide strings, there is no need to override these methods.

A new feature in Delphi 5 is owner-drawn property editors. `TWideStringProperty` overrides the `PropDrawValue` method to display the wide string property value. To set the value, though, you need to pop up a dialog box. There is no convenient way to force the Object Inspector to accept a wide string as input. Thus, `TWideStringProperty` sets the property attributes to read only and to have a dialog box. It overrides the `Edit` method to show the dialog box and let the user edit the wide string.

```
function TWideStringProperty.GetWideStrValue: WideString;
begin
  Result := GetWideStrValueAt(0);
end;
function TWideStringProperty.GetWideStrValueAt(Index: Integer): WideString;
begin
  Result := GetWideStrProp(GetComponent(Index), GetPropInfo);
end;
procedure TWideStringProperty.SetWideStrValue(const Value: WideString);
var
  I: Integer;
begin
  for I := 0 to PropCount-1 do
    SetWideStrProp(GetComponent(I), GetPropInfo, Value);
  Modified;
end;
```

➤ *Listing 7: WideString-access methods for TWideStringProperty.*

The dialog box is like a wide version of Delphi's input query dialog box. To encourage code reuse, you can write a general-purpose function, `InputQueryW`, which is similar to Delphi's `InputQuery` function, but it lets the user enter a wide string. For the sake of simplicity, you can use ANSI strings for the caption and prompt. The `InputQueryW` function creates a `TInputQueryWForm` (Listing 8). The form is an ordinary Delphi form, with a label for the prompt, and two buttons: `OK` and `Cancel`. The only trick is that the form's `OnCreate` handler creates a wide edit control using the Windows API, as shown in Listing 9. The form also has some access methods to make it easier to get and set the value of the wide edit control. Listing 10 shows these methods.

Packaging The Parts

Put all the pieces together by putting the wide label component in a runtime package (Wide50.bpl) and the design-time code in a design-time package (WideCtls.bpl): the code is on the disk. Install the design-time package in Delphi's IDE to register the `TWideLabel` control and the `WideString` property editor. Put the runtime package anywhere in your `PATH`, so Windows can find it. You can now use wide strings in components, and have full access to wide strings at design-time and at runtime.

```
function InputQueryW(const Caption,
  Prompt: string; var Value:
  WideString): Boolean;
var
  Form: TInputQueryWForm;
begin
  Form := TInputQueryWForm.Create(
    Application);
  try
    Form.Caption := Caption;
    Form.Label1.Caption := Prompt;
    Form.Text := Value;
    // Show form modally so user
    // can edit the text.
    Result :=
      Form.ShowModal = mrOK;
    if Result then
      Value := Form.Text;
  finally
    Form.Free;
  end;
end;
```

➤ *Listing 8: InputQueryW.*

```
procedure TInputQueryWForm.FormCreate(Sender: TObject);
const Margin = 8;
      EditWidth = 300;
begin
  // Create the wide edit control.
  fEditWnd := CreateWindowExW(Ws_Ex_ClientEdge, 'EDIT', '',
    Ws_Child or Ws_Visible or Ws_Border or Ws_TabStop or Es_AutoHScroll,
    Margin, Label1.BoundsRect.Bottom+Margin, EditWidth, Label1.Height + Margin,
    Handle, 0, hInstance, nil);
  // The default font uses the OEM character set, not Unicode.
  // The form uses Arial, which supports Unicode on NT.
  SendMessageW(EditWnd, Wm_SetFont, WParam(Font.Handle), 0);
end;
```

➤ *Listing 9: Creating the control.*

```
function TInputQueryWForm.GetText:
  WideString;
begin
  // Get text, return it to caller.
  SetLength(Result,
    GetWindowTextLengthW(EditWnd));
  GetWindowTextW(EditWnd,
    PWideChar(Result),
    Length(Result)+1);
end;
procedure TInputQueryWForm.SetText(
  const Value: WideString);
begin
  SetWindowTextW(EditWnd,
    PWideChar(Value));
end;
```

➤ *Listing 10: Wide text access.*

Delphi has one last trick up its sleeve, though. You must disable the Text DFM feature: from the form editor's context menu, make sure Text DFM is not checked. Binary DFMs store wide strings correctly, but a text DFM does not distinguish between a wide string and an ANSI string. If a string contains at least one wide character, Delphi treats it as a wide string. If the string contains all ANSI characters, Delphi thinks the string is an ANSI string, regardless of the property's type. This is a bug. A wide string that contains all ANSI characters should still be treated as a wide string. You won't notice the error when you save the DFM, but when you load it, you will get an 'Invalid property value' error because Delphi reads an ANSI string from the DFM, but the form expects a wide string.

Supporting wide strings as published properties is harder than it should be, but you need to do the dirty work only once (or, in this case, let me do the dirty work for you). Once you register the `TWideStringProperty` editor, it automatically works for all properties of type `WideString`. You can also call the `InputQueryW` function anytime you want to get a wide string from the user. The only work you need to repeat is loading and storing the wide string value in your component, but that's just a couple of one-line methods. Perhaps future versions of Delphi will provide better support for wide strings at design-time (but don't hold your breath).

Ray Lischner is the author of *Delphi In A Nutshell* and other books and articles. He speaks about Delphi at conferences and teaches Computer Science at Oregon State University. Email Ray at lisch@tempest-sw.com