The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Missing Debugger Windows

Q I am having a problem debugging an application in Delphi. I was originally working at a resolution of 1024x768, and had my debugger windows (watches, local variables and call stack) positioned on the right-hand side of the screen. To keep them there for this project I used a desktop file.

Because of the target machines for this application I switched down to a resolution of 800x600 to continue development a little while ago. The problem is that I cannot see my debug windows any more, as they are now positioned off the screen! How can I retrieve them without switching back to 1024x768?

A If changing screen resolutions is not an option, for whatever reason, I see three solutions. The first would be to simply delete your desktop file (which obviously will lose all the information stored in it). Depending on which version of Delphi you are using, this may be either a project desktop file or a global desktop file (introduced in Delphi 5).

The project desktop file is a file stored in the same place as your project file, with the same name as it but with a .dsk file extension (.desk in Kylix). Once the file has been deleted, the next time you open the project, all the windows will be in their default positions and so will be accessible again.

The global desktop file affects the layout of all projects and is stored as a file in the bin directory with a .dst extension. You can delete it and the windows will become accessible next time you run Delphi.

The next possibility would be to edit the desktop file (be it a project or global desktop) and fix the positions of the missing windows. Both types of desktop file are just Windows .INI files, with individual sections for each IDE window. As you can see from Listing 1, it should be easy to modify the `Left` attribute to bring it back on screen.

Incidentally, even if you have disabled the option to auto-save a project desktop file (on the `Preferences` page of the environment options dialog), the IDE may still have a default desktop file in use if the option was ever enabled beforehand. The file is called delphi.dsk in Delphi 1 and delphi32.dsk in 32-bit versions of Delphi, and is found in the bin directory. Kylix calls the file delphi.desk and stores it in ~/.borland. This file can be similarly edited or deleted to overcome issues with the default IDE window layout.

The final solution to the problem is more a Windows solution than a Delphi solution. The first thing to do is make sure the window that you wish to retrieve is active, by selecting it from the Delphi menu (for example `View | Debug Windows | Watches`). Next, you can bring down the window's system menu by pressing `Alt+Space`. Note that the debug windows use a `bsSizeToolWin` border style, so they don't have the usual icon on the left of the caption that is used to drop down the system menu, but the keyboard shortcut still works.

From the system menu (which will probably be displayed on the screen, even though the underlying window is still off-screen), choose the `Move` option, but choose it with the keyboard by pressing `M`. This now allows the cursor keys to move the window to a new destination. Hold down the left cursor key and the window should soon make its way onto the desktop. Press `Enter` to exit this moving mode and make the window keep its new position.

➤ *Listing 1: A section of a project desktop file.*

```
[WatchWindow]
Create=1
Visible=1
State=0
Left=810
Top=226
Width=200
Height=149
MaxLeft=-1
MaxTop=-1
ClientWidth=281
ClientHeight=125
TBDockHeight=149
LRDockWidth=421
Dockable=1
```

➤ *Listing 2: Reading an SDF file.*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  S: String;
begin
  with TFileStream.Create('MyFile.sdf', fmOpenRead or fmShareDenyWrite) do
    try
      SetLength(S, Size);
      Read(S[1], Size);
      ListBox1.Items.CommaText := S
    finally
      Free
    end
end;
```

SDF Files

Q I need to access a text file in SDF format. I looked for this format on the web without significant results. How can I read SDF files from within a Delphi application?

A Fortunately this is quite straightforward as the `TStrings` class has a property dedicated to SDF (System Data

Format). Assuming you have an SDF string read from a file, you can simply assign it to any TStrings object's CommaText property, and it will be interpreted accordingly. For example, supposing you have an SDF file called MyFile.sdf. You could read the file into a listbox, translating into the actual data rather than the file contents using something like Listing 2.

RTTI To The Rescue

Q To make changing our user interfaces easier (for international customers), we have decided to go with a principle of numeric Caption properties. When a form is loaded, we iterate the form's components. If the component has a Caption property which can be converted to an integer, then we attempt to load a resource string with an identifier equal to that integer.

Is there any way we can determine whether a VCL object derived from TControl has an accessible Caption property? The best solution we've found so far is to use RTTI (via the is operator), but of course that means we have to hardcode the appropriate types (TLabel, TButton etc.) into the routines. Introducing a new component with an accessible Caption property onto a form would mean having to change the TCustomForm class that implements the resource loading; not exactly an elegant solution!

➤ *Listing 4: Backwards compatible version of Listing 3.*

```
procedure TForm1.Button1Click(Sender: TObject);
const
  PropName = 'Caption';
  Default = -1;
var
  Loop, CaptionVal: Integer;
  Comp: TComponent;
  PropInfo: PPropInfo;
begin
  for Loop := 0 to ComponentCount - 1 do begin
    Comp := Components[Loop];
    PropInfo := GetPropInfo(Comp.ClassInfo, PropName);
    if Assigned(PropInfo) then begin
      CaptionVal := StrToIntDef(GetStrProp(Comp, PropInfo), Default);
      if CaptionVal <> Default then
        ListBox1.Items.Add(Format(
          'The %s component %s has a numeric caption: %d',
          [Comp.ClassName, Comp.Name, CaptionVal]))
    end
  end
end;
```

```
uses
  Typinfo;
procedure TForm1.Button1Click(Sender: TObject);
const
  PropName = 'Caption';
  Default = -1;
var
  Loop, CaptionVal: Integer;
  Comp: TComponent;
begin
  for Loop := 0 to ComponentCount - 1 do begin
    Comp := Components[Loop];
    if IsPublishedProp(Comp, PropName) then begin
      CaptionVal := StrToIntDef(GetStrProp(Comp, PropName), Default);
      if CaptionVal <> Default then
        ListBox1.Items.Add(Format(
          'The %s component %s has a numeric caption: %d',
          [Comp.ClassName, Comp.Name, CaptionVal]))
    end
  end
end;
```

➤ *Listing 3: Simplifying common property access.*

A RTTI does answer your question, but not by using the is operator. As you say, is allows you to find if a component has a Caption property, but you then typically need to typecast to the right type to assign to the Caption property.

Relying on a common ancestor to avoid the typecast does not work, for two reasons. Firstly, TControl (which does define the Caption property common to all visual components) defines Caption as protected, so it is not directly accessible: descendants will publish it as required. Secondly, some components introduce Caption as a new property and some components inherit it from TControl. For example, TMenuItem defines its own Caption but TButton inherits it from TControl and publishes it.

Instead, I would recommend the RTTI support unit, Typinfo. A simple example should show what can be achieved. Listing 3 shows some code that works in Delphi 5 and later. It loops through each component on the form using RTTI to read the value of the Caption string property, if it exists as a published property. Once a caption has been read, it is translated into an integer if possible. If the string does represent an integer (meaning StrToIntDef didn't return its specified default value), some information is added to a listbox to describe the results.

The questioner can replace the statement that adds the string to the listbox with code that reads a resource string and assigns it to the component's caption using SetStrProp.

IsPublishedProp and the version of GetStrProp that can take an object instance as a parameter were added in Delphi 5. For earlier versions, use the code in Listing 4, which uses property information records (pointed to by PPropInfo pointers). Both sections of code can be found in the RTTIEg.dpr project on this month's disk.

Type Libraries And COM Parameters

Q I am faced with the task of writing a simple IDE for our in-house language. So far I have written a framework that can handle streams and register plug-ins. The framework can handle COM plug-ins, which operate on a stream. Since I can easily copy from one stream to another the solution seemed quite straightforward: I should use OLE streams. The help states that those are used by OLE to read and write data.

```
//This is the C declaration:
//DWORD RegisterServiceProcess(DWORD dwProcessId, DWORD dwType);
//This is a direct translation from C to ObjectPascal
//function RegisterServiceProcess(dwProcessId, dwType: DWord): DWord; stdcall;
//external kernel32;
//This is a more appropriate translation
function RegisterServiceProcess(dwProcessId, dwType: DWord): LongBool; stdcall;
external kernel32;
```

➤ *Listing 5: An API declaration translated from C to ObjectPascal.*

The problem appears when I use the Type Library Editor. I want to declare a method in my interface called `Execute` that takes and returns a stream but `OLEStream` doesn't appear in the list of valid parameter types. Is there any way to send and receive streams over a COM interface? Writing my own marshaling code is beyond me.

A When a COM method is intended to take a reference to some interface that is not accessible, the most common way of proceeding is to declare the parameter as type `IUnknown`, but document that it takes an `IStream` (the interface type for an OLE stream). The method can query the interface for `IStream` when it comes in (using either `QueryInterface` or the as operator) and return an error value if `IStream` is not supported.

Other options could include trying to track down whether there is a standard type library that defines the `IStream` interface. If there is, you should make your type library use that type library (the `Uses` page of the Type Library Editor, when the root node in the `Object` pane is selected). You will then be able to refer to types from the other type library meaning you can define your parameter as `IStream`.

Hiding From The Windows 95/98/Me Task Manager

Q I've a problem I've been trying to find a solution for, but with no results so far. How do I stop Windows Task Manager from showing my program (and therefore allowing it to be closed by the user)?

A The answer to this question is to make your application look like the Windows 95/98/Me version of a service. On Windows NT/2000, service applications do not show up in the Task Manager's `Applications` tab (which is much the same as the Windows 9x Task Manager dialog). However, service applications are specific to the Windows NT/2000 architecture.

To cater for this requirement, an API exists in Windows 9x/Me called `RegisterServiceProcess`, which is implemented in Kernel32.dll. This can be called to turn your own application (or any other one, for that matter) into a *simple service* application. There are two implications of this. Firstly, the name of the process will not appear in the Task Manager. Secondly, the application has the capability of surviving a user logoff. Normally, when a user logs off, all programs are closed. However, an application can tell the difference between a system shutdown and a user logoff. Simple service applications can avoid closing down if the user is logging off, and they will be left running by Windows.

`RegisterServiceProcess` does not exist in Windows NT/2000, so Microsoft advice from the Platform SDK is: *'To call* `RegisterServiceProcess`*, retrieve a function pointer using* `GetProcAddress` *on KERNEL32.DLL. Use the function pointer to call* `RegisterServiceProcess`*'*. This is wise if your application may be launched on either platform. If, however, you know it will only be launched on Windows 9x/Me, you can write a normal import declaration for it.

Before starting with this API, we need to know how it is declared. Listing 5 shows the C declaration in the first comment. The direct ObjectPascal equivalent is then shown in the next comment. However, the documentation states that the return value is either 1 for success or 0 for failure. Clearly the 32-bit return value is used as a `Boolean` success indicator, so a more useful translation is at the end of Listing 5. `LongBool` is a 32-bit wide `Boolean` type whose value is interpreted using the same semantics as C does.

When calling the function, the first parameter identifies the process to be turned into a simple service. You can pass 0 to identify the current process. The second parameter should be 1 to register the specified process as a simple service or 0 to unregister it. The Platform SDK has constants for these values, but Delphi 5 does not define them.

With this information, Listing 6 shows how a Windows 9x/Me-specific application can be hidden from the Task Manager. However, because of the import declaration, trying to run such a program on Windows NT/2000 would give a fatal error on startup (since the specified API does not exist in Kernel32.dll on Windows NT and 2000).

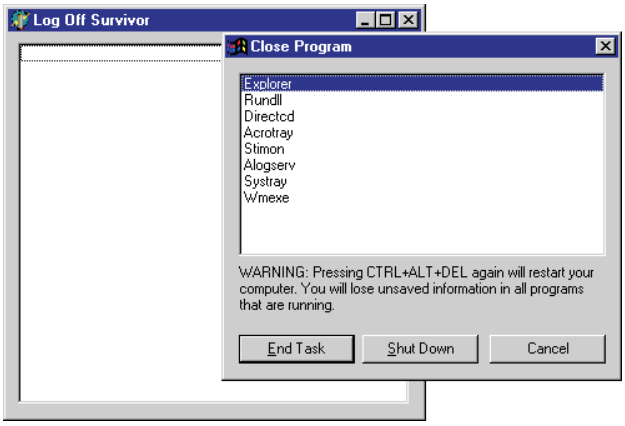
➤ *Listing 6: Hiding from the Windows 9x/Me Task Manager.*

```
function RegisterServiceProcess(dwProcessId, dwType: DWord): LongBool; stdcall;
  external kernel32;
procedure RegisterMe(Register: Boolean);
const
  RSP_SIMPLE_SERVICE = 1;
  RSP_UNREGISTER_SERVICE = 0;
  Types: array[Boolean] of DWord = (RSP_UNREGISTER_SERVICE, RSP_SIMPLE_SERVICE);
begin
  Win32Check(RegisterServiceProcess(0, Types[Register]))
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  RegisterMe(True);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  RegisterMe(False)
end;
```

An alternative implementation of the `RegisterMe` routine gets round that by following the Microsoft advice, as shown in Listing 7.

This now works quite nicely, but doesn't address the other aspect of this API. The program should, if needed, be able to survive the current user logging off and any user then logging on. A Delphi application won't demonstrate this behaviour without further changes, because of the way the VCL behaves. When Windows is closing, or the user logs off, Windows sends a `WM_ENDSESSION` message to the application. The VCL picks this message up and sets `Application.Terminated` to `True`, which stops the message loop running and lets the program quickly shut down.

We need to trap this message ourselves and carefully examine one of the values passed along with it (the `LParam` value). If `LParam` has a value of `ENDSESSION_LOGOFF` then the user is logging off rather than shutting the machine down, and so we should stop the `Application` object doing its normal job of deciding to shut.



➤ *Figure 1: A program running which is not in the Task Manager.*

To modify the `Application` object's handling of a message we can write an application message hook routine, installed with `Application.HookMainWindow` and uninstalled with `Application.UnhookMainWindow`. Application message hook routines intercept messages targeted at the `Application` object's internal window (the top level window in the application) and can stop them reaching their destination.

Listing 8 shows the hook method along with the code that installs it and uninstalls it, and comes from the project LogOffSurvivor.dpr on the disk. The key thing is that the hook routine returns `True` to stop the message being processed by any more hook routines or the `Application` object itself. Figure 1 shows the program running with the Task Manager not listing it.

Note that the code in Listing 8 only affects users logging off of Windows 95/98/Me. The program will still be terminated when logging off Windows NT/2000. To survive a log off on those platforms, you should write a proper NT service application.

➤ *Listing 7: A rewrite of Listing 6 that won't crash out on WinNT/2000.*

```
procedure RegisterMe(Register: Boolean);
type
  TRegisterServiceProcess = function(dwProcessId, dwType: DWord): LongBool;
    stdcall;
const
  RegisterServiceProcess: TRegisterServiceProcess = nil;
  RSP_SIMPLE_SERVICE = 1;
  RSP_UNREGISTER_SERVICE = 0;
  Types: array[Boolean] of DWord = (RSP_UNREGISTER_SERVICE, RSP_SIMPLE_SERVICE);
begin
  if not Assigned(@RegisterServiceProcess) then
    RegisterServiceProcess :=
      GetProcAddress(GetModuleHandle(kernel32), 'RegisterServiceProcess');
  if Assigned(@RegisterServiceProcess) then
    Win32Check(RegisterServiceProcess(0, Types[Register]))
//  API won't be found on Windows NT/2000, but that's okay
//  else
//    raise EWin32Error.Create('RegisterServiceProcess API not located');
end;
```

➤ *Listing 8: Making the program survive a user logging off.*

```
function TForm1.ApplicationHook(var Message: TMessage): Boolean;
begin
  Result := False;
  //If this is a user logoff...
  if (Message.Msg = WM_ENDSESSION) and
     (TWMEndSession(Message).Unused = LParam(ENDSESSION_LOGOFF)) then
  begin
    //...Stop this message being processed by the
    //Application object, which would shut the application
    Result := True;
    Message.Result := 0;
  end
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  RegisterMe(True);
  Application.HookMainWindow(ApplicationHook)
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Application.UnhookMainWindow(ApplicationHook);
  RegisterMe(False)
end;
```

Problems With Packages

Q I have written some components for my own use and I have added them to a package so they can be installed into Delphi. The problem is that each time I build a project that uses these components, all the source files from the package are also rebuilt. I was expecting that the files belonging to the package would be linked in, but not recompiled each time. How can I set things up so that only my actual project code is compiled during a build?

A Before covering the problem itself, I think it might be a good idea to have an overview of how Delphi packages are supposed to be used both in the IDE and also in your programs. The last time I mentioned packages, in

the *Transfer Efficiency* Clinic entry from Issue 67, I avoided looking at the general subject, but feel this time we should lay some groundwork before we proceed.

Firstly, some definitions. These come from the Delphi help, but are equally applicable to C++Builder and Kylix:

'A package is a special dynamic link library used by Delphi applications, the IDE, or both. Runtime packages provide functionality when a user runs an application. Design-time packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design-time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland Package Library) extension.'

A package is defined in terms of the units compiled directly into it and the additional packages it requires to operate (in other words, the packages that contain the units used by units compiled into the package). Unlike a normal executable, when the compiler compiles a package, every line of code is compiled and linked into the package: there is no smart linking. This is because the package might be used by many different applications which access various parts of various units contained in the package. In order for them to work, absolutely all the code must be present in the binary package.

The primary reason packages exist is to allow applications to compile to much smaller executables. Code that is common to many applications (such as VCL/RTL code) can be housed in some packages (runtime packages) which can be used by all applications. The applications themselves end up much smaller, as all the VCL/RTL code is stripped out of them. Thanks to the nature of packages, an application developer can switch between using them and not using them with one single project option, making it very easy to try them out.

When an application developer compiles their application, they can disable or enable the use of runtime packages. When enabled, they also have control over which runtime packages are used. If a unit referenced by a program resides in a runtime package, but that package is not in the application's runtime packages list, that unit will be compiled directly into the application just as in the case of a traditional non-packaged application.

However, as described above, the IDE also uses packages as a means to have components, property editors, component editors and wizards installed (these are design-time packages). In general, professional component suites are distributed as a runtime package and a design-time package. The runtime package contains the units which implement the components and any associated support routines. The design-time package contains the implementation of any design-time support, such as component registration or property editors, but relies on the runtime package for the implementation of the actual components.

This split means that the running program uses a package that contains only code that is intended to be called by the program. It is not inflated by code that is superfluous at runtime (the IDE's design-time support code).

If you look at some of the packages supplied by Borland, you can see the general idea. For example, vclmid50.bpl is a runtime package that contains all the MIDAS components. This file is installed in the Windows system directory so it can be accessed by all applications. Dclmid50.bpl is the design-time package, installed in Delphi's bin directory, so only Delphi can locate it. This package contains all the property editors and component registrations for the MIDAS components. dclmid50.bpl is compiled to require vclmid50.bpl, as that package contains the MIDAS component implementations.

The IDE loads dclmid50.bpl when it starts and, because of the aforementioned built-in requirement, vclmid50.bpl is automatically loaded as well. The IDE therefore has access to the components and all their associated design-time paraphernalia. Applications, on the other hand, will load vclmid50.bpl, gaining access to the components and carrying no design-time stuff with them.

That said, the IDE's New Component wizard does not try and encourage this type of split at all. It manufactures a single unit which ends up containing the implementation of a component and also the code that registers the component with the IDE so it will appear on the Component Palette (design-time specific code). Unless you split the registration routine into a separate unit, you will be forced to build a package that can work both at design-time and at runtime. This is no big problem but, as has been mentioned, the package will contain a certain amount of pointless code which will swell its size.

The questioner has apparently made a package that contains units generated by the New Component wizard. This means that their package will also function both at design-time and runtime. But it appears the questioner only made the package in order to install the components into the IDE. I gather he is making a standalone application and is perplexed by the component units being compiled repeatedly whereas the pre-supplied Borland components never get recompiled.

To overcome the problem, we should understand how the standard components are laid out. In a standard Delphi installation, one of the most important subdirectories is Lib, where all the compiled units and packages live. If you browse this directory you will find many DCU files (compiled units) and quite a few DCP files (compiled package files) but no PAS files (Pascal units). This is the key to successful linking without recompilation. The compiler needs to be told where the compiled versions of your files are, and the source code lives somewhere else.

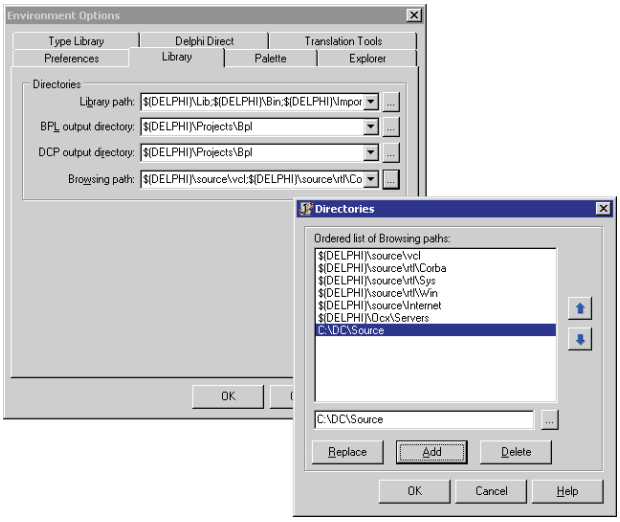
Borland supplies its source code in a variety of subdirectories under the Source directory, notably Source\VCL and Source\ RTL\Sys. During recompilation, the Delphi compiler does not find this source code since it does not know where it lives.

On the other hand, the IDE *can* locate these source files, and uses them for Code Browsing. This is where you hold the *Ctrl* key down, move the mouse over an identifier and click on the resultant hyperlink that is shown. Alternatively, you can do menu-driven code browsing by right-clicking an any identifier and choosing `Find declaration`. There is a separate setting that tells the IDE about the locations of these source files, but the compiler does not get told.

To answer the question we need to know how to set up a custom package so that the compiled files and the source are also kept separate, so let's start the ball rolling. Before doing anything, you should make some directories somewhere on your system, one for package source file, one for your component source code and one for the compiled code. They can be in the same directory tree, or in completely separate locations: the choice is yours. For the rest of this discussion, I'll refer to them as C:\DC\Package, C:\DC\Source and C:\DC\Lib.

When you create your package (from the `File | New...` dialog) you should save it in C:\DC\Package. Similarly, when you create components, they should also be saved in C:\DC\Source.

In order to tell the Delphi compiler that compiled files should be placed in a different directory, press the package editor's `Options` button (note that the package has its own set of options as distinct from those of any open project, which are accessed by `Project | Options...`). On the `Directories/Conditionals` page of the package's options dialog, change `Output directory:` to be `$(DELPHI)\Projects\Bpl` (that's where the compiled package will be placed), and change both `Unit output directory:` (which is where DCU files go)



➤ *Figure 2: Telling the IDE where to find source files.*

and `DCP output directory:` to C:\DC\Lib.

When you press the package editor's `Compile` button, C:\DC\Lib will contain compiled versions of each component unit, as well as a DCU for the package source file. It will also contain a DCP file for the package as a whole. The final compiled binary package (the BPL file) will be found in the BPL sub- directory off Delphi's Projects subdirectory.

At this stage, you can press the package editor's `Install` button and the components in the package will make their way onto the Component Palette. The problem now is how to tell the compiler about the location of the compiled component units, and the IDE about the component source files. To do this, you go to the environment options dialog (`Tools | Environment Options...`). On the `Library` page of the dialog, there are two key entries: `Library path` and `Browsing path`. You should add C:\DC\Lib to `Library path` and add C:\DC\Source to `Browsing path` (see Figure 2).

You should now be able to use your newly installed components and build your application without having the component units rebuilt. You should also be able to use Code Browsing in the editor. Hold down the `Ctrl` key and click on a reference to one of your component classes (or anything from one of your component units). You will be taken to the appropriate location in the source file.

Update

In Issue 69 (May 2001) I discussed the subject of published interface properties (the *RTTI and Interfaces* entry). Shortly after it hit your letterboxes I received a message from Phil Webb who explained that anyone using Delphi 5 or earlier can also make use of published properties that refer to interfaces. You can define the property as type `TComponent`, and implement the relevant interface in classes inherited from `TComponent`. A custom property editor can be written quite easily to restrict the number of components shown in the Object Inspector for the property in question to those that actually do implement the interface.

This information might be useful for anyone not immediately upgrading to Delphi 6 where published interface properties are natively supported, so thanks Phil.