Beating The System: IDE Secrets, Part 2

by Dave Jewell

A couple of months back, I discussed the internals of Borland's undocumented DSNIDE50 package, and explained how it is possible to use this package to make better add-in experts and wizards for the Delphi IDE. After having written this particular article, I was feeling quite disgustingly pleased with myself, whereupon I made the fatal mistake of searching the net to satisfy myself that nobody else had written on this particular topic.

Well, it turns out that somebody has, no less a person than Allen Bauer, Staff Engineer and Delphi/C++Builder R&D Manager! Naturally, I came down to earth with a bump. But having picked myself off the floor and read through what the great man had to say, I ended up feeling a lot better. To begin with, Allen is constrained by company policy regarding what aspects of DSNIDE50 he discusses, and what he keeps secret, whereas we have no such restrictions *[Err, apart from the lawyers, Dave... Ed]*. Also, Allen doesn't seem to be in any big hurry to acquaint people with the delights of DSNIDE50 (his last article was written back in March 2000) and if my own investigations spur him on to reveal more of the goodies contained therein, then so much the better. You can download Allen's articles, and associated source code, from the Borland Community website at http://community.borland.com.

Fooling The IDE

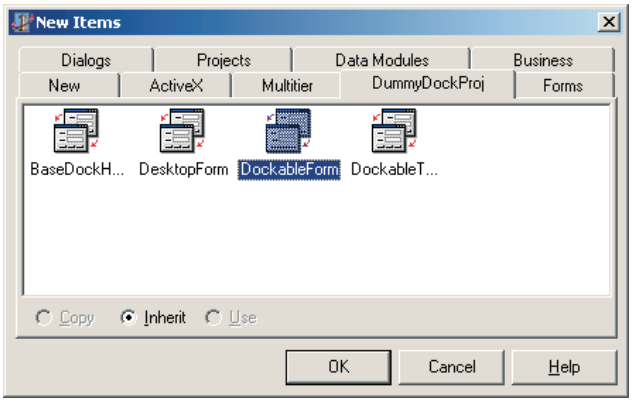
Typically, a Delphi programmer will use form inheritance to create a descendant of an existing form, where the ancestor form is contained within the *same* project. However, if you make extensive use of project groups, you will most likely have already noticed that the IDE can search for the ancestor form not only within the current project, but also within any other projects in the group. In his articles, Allen cunningly exploits this behaviour in order to fool the IDE into instantiating a form that's descended from one of the form classes inside the DSNIDE50 package. As before, I'm assuming that we're using Delphi 5 here: presumably, Delphi 6 will ship with DSNIDE60.

Once you've fooled the IDE in this way, you'll be able to see your new form within the Delphi form designer, and add additional controls, change properties, etc, in the usual way. In the previous article, I mentioned that CodeRush already gives you this sort of functionality and postulated that CodeRush was already making extensive use of the code inside DSNIDE50. After some recent email discussions with Mark Miller (the creator of CodeRush), it turns out that part of the reason why Borland decided to start shipping DSNIDEXX.DCP with Delphi is because of Mark's insistence on this point. So well done that man!

If you checked out the code from Issue 64 (in particular DUMMYFORM.PAS), you will have noticed the way in which I create controls on the fly inside the form's OnCreate event handler. Again, this restriction is analogous to the way in which users of early CodeRush implementations were forced to explicitly create controls at runtime, rather than making use of the form designer. Being able to use the form designer is obviously the preferred Delphi way of doing things, and therefore, in this next section, I'm going to walk you through the process of creating a DSNIDE50 form descendant within the IDE's form designer using the technique described by Allen Bauer.

This technique is necessarily somewhat convoluted (some might say, arcane) but I'll try and explain the rationale behind what we're doing along the way. To begin with, you'll need to download the .ZIP file associated with Allen's tutorial from Borland Code Central. What you're after is a file with the rather uninformative name of 14529.ZIP. This contains a set of stubbed-out form units along with their .DFM files. You'll find names such as DeskForm.pas, DockToolForm.pas, and so forth. Having read the previous article, these unit names should be somewhat familiar to you. We need these files in order to convince the IDE that the ancestor form class is actually contained within our project group, even though, in reality, the *real* implementation of the ancestor classes resides inside the DSNIDE50 package.

When you browse these files, you'll discover that Allen has rather generously left a lot more of the implementation code inside them than was strictly necessary, thus providing some fascinating insights into the inner workings of the IDE. The fact that these files have been made available by



➤ *Figure 1: Using Allen Bauer's technique for fooling the IDE, it's possible to make use of form inheritance to load descendants of the various IDE docking form classes into the form designer.*

Borland is very good news for me, because I'd been planning to do something similar, but was worried about falling foul of Borland's lawyers and didn't want to provide any more of my own 'decompiled' findings than was strictly necessary in order to make this technique work. Happily, Allen's files solve the problem and give us a few extra little gems along the way.

Once you've unzipped 14529.ZIP into a directory of your choosing, you can fire up the Delphi IDE and open the DUMMYDOCKPROJ.DPR project which Allen supplies. As you'll see, this includes the various stubbed-out units mentioned earlier. If you try and build this project immediately after a fresh installation of the Delphi IDE, the build will probably fail, with the compiler complaining that it can't find DSGINTF, which is referenced from some of the supplied units. In any event, it's not really necessary that this project *should* be built, the important thing is simply to include it in the same project group as the Delphi expert project that you're working on.

The next step is therefore to right-click on the project group name and choose Add New Project... from the resulting context menu. When asked what type of project you want, you should select Package in the usual way because we're obviously in the business of creating a Delphi add-in. I'll refer to this specifically as the expert project, while the Borland-supplied files constitute the dummy project. The real trick is to create the DSNIDE50 descendant in the dummy project, using the stub units provided for the purpose. It can then be sneakily copied over to the expert project, replacing the stub units with a reference to the DSNIDE50 package at the same time. Here's how to do it.

First, make sure that the dummy project is the active project and then select New... from the File menu. If you've used form inheritance in the past, you'll know that the New Item dialog automatically adds a tab which is named according to the project, and all the currently defined forms sit on that dialog tab, waiting to be inherited from. Sure enough, if you go to the DummDockProj tab of the dialog, you'll find four form classes awaiting your pleasure: see Figure 1. These are: BaseDockHostForm, DesktopForm, DockableForm and DockableToolbarForm.

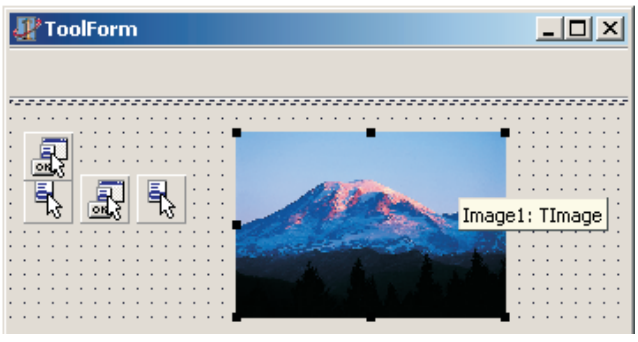
As I mentioned last month, the final form class, DockableToolbarForm, is used to create a dockable form that includes an integral, resizable toolbar such as the package manager window or the project manager. If you don't need this facility, it makes sense to inherit from DockableForm most of the time.

So let's say you've created a new DockableForm descendant, naming the new form as ToolForm, and the form unit as TestDock.pas (for the sake of argument). Next, swap over to the expert project, making it the active project. In the usual way, add the form unit that you've just created to the expert project and then remove it from the dummy project. Finally, in the package manager window, right-click on the Requires tree node (as I showed you last time) and add a reference to the DSNIDE50.DCP file. Using this technique, we have 'grafted' the required form unit into the expert project and told the IDE where to find the real implementation of the ancestor class that we used. Obviously, you'll also need to designate your expert project as being a design-time only package, and you'll want to add code similar to that in Listing 1, in order to ensure that the expert form is created and deleted as appropriate. (Be sure to reference the Register routine from the interface part of your unit!)

```
procedure Register;
begin
  if ToolForm = nil then
    ToolForm := TToolForm.Create(
      Application);
  ToolForm.Show;
end;
initialization
finalization
  ToolForm.Free;
end.
```

➤ *Listing 1*

Figure 2 shows the result of using the above techniques on a derivative of DockableToolbarForm. As you can see, the integral toolbar, splitter and assorted other components are all visible in the normal way, and you can add your own custom controls using the form designer. Similarly, Figure 3 shows what happens when we build the expert project and then double-click the DSNIDE50.DCP file under the Requires node of the package editor window. This opens another instance of the package editor, and we get treated to a full blow-by-blow list of all the units contained inside this package. As you can see, we have barely scratched the surface of the various units that are available!



➤ *Figure 2: And here's the proof of the pudding, a live dockable toolbar form which we can manipulate inside the form designer just as if it was a regular Delphi form. As with any other example of form inheritance, you can't delete the inherited bits!*

To Boldly Go

So what other creepy-crawlies are lurking undiscovered inside DSNIDE50? There are a few tools around which will pick open a .DCP or .DCU file and give a fair representation of the Pascal source contained therein, or at least the interface part of it! (Well, what do you think this is, Christmas?) However, for our purposes, an even more direct approach is possible. If you follow the above steps to create an expert project which uses the DSNIDE50 unit and then

build the project, you'll have essentially injected the contents of DSNIDE50 into the symbol space of the project. Or, to put things differently, once you've built the package, you can use the IDE's built-in Object Browser to browse DSNIDE50 to your heart's content. For maximum amusement value, right-click the Object Browser window, select Properties and be sure that the Show Declaration Syntax checkbox is checked. With the Browser in 'Units' mode, you should then see something like Figure 4. In this particular case, we're looking at the interface declaration for TDockableForm.

Referring back to the Borland-supplied 14529.ZIP file, Allen Bauer only gave us the interface declarations for the units and classes that were relevant to his tutorial on dockable IDE windows. However, using the Object Browser, we can effectively examine *everything* that is exported from the DSNIDE50 package! Now, admittedly, you won't see a particular unit unless that unit is referenced from the current project and the package rebuilt, but that's easy enough to do. And if you need a list of all the units contained inside the package, just look back to Figure 3.

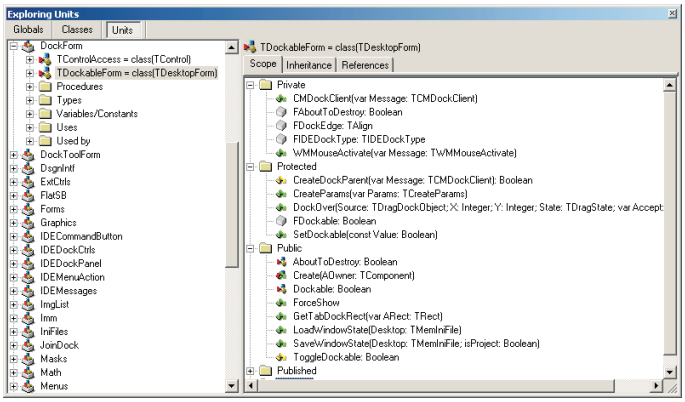
As you pick your way through this little lot, you will find a number of units with rather intriguing names. For example, there are no less than nine different units whose names are prefixed by Model: ModelCaps, ModelClasses, ModelControl, ModelDesigner, ModelFactories, ModelPrimitives, ModelTypes, ModelUtils and ModelViews. Phew! So what's all that lot, then? Much of the code inside DSNIDE50 is actually concerned with the implementation of the Data Module Designer, and that's basically what these nine units are about. The same is true of DMDesigner, DMStrs, and all the units whose names are prefixed by DataModel.

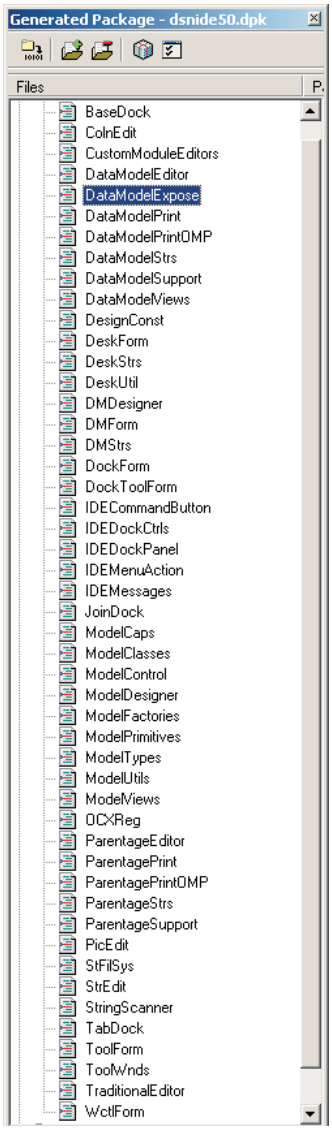
Another interesting unit is IDEMessages. This unit defines a whole host of internal IDE messages, which could potentially be very useful for those who have come up against the limitations of the published Open Tools architecture. Here's a small snippet taken from this file (you can use the Browser to get the full picture):

```
IDEM_StartCompile            = 1295;
CM_CompleteChangeProperty    = 5125;
CM_UpdateInspector           = 5127;
```

PicEdit and StrEdit are of course the built-in property editors for the picture and string classes. Don't waste time trying to reverse engineer these units because Borland have been providing us with the source for some time. With a standard installation, the source code is at C:\Program Files\Borland\Delphi5\Source\Property Editors. Similarly, ColnEdit is the standard collection editor used by TListView. Again, Borland already provide us with full source.

Even so, it is likely that most budding expert writers will want to focus on the various units and classes that I introduced a couple of months ago. With this in mind, let's revisit some of those topics, taking a more detailed look than before. As should be clear by now, most of your experts will be descendants of TDockableForm, so let's concentrate on that class, because it's central to the whole IDE implementation of dockable windows.





➤ *Figure 3: Here are all the units inside DSNIDE50.*

➤ *Figure 4: Once you've built your add-in expert, you can use the browser to examine the interface part of any of the units that are referenced from your project. In this case, we're looking at the interface definition for TDockableForm.*

Delving Into Docking Details

As you'll no doubt appreciate, the IDE permits two different forms of docking: a window can be docked up against another window, or else it can be dropped *onto* another window, thus creating a sort of nested arrangement whereby tabs are used to select between the different windows. The former possibility is referred to as a joined dock, and the latter as a tabbed dock.

Thus, if you peruse the various units in DSNIDE50, you'll find two

units called JoinDock and TabDock which correspond to these two possibilities. The JoinDock unit implements a descendant of TDockableForm called TJoinDockForm, and the TabDock unit implements another TDockableForm descendant called TTabDockHostForm. Thus, the arrangement is something like that shown in Figure 5.

Notice the phrase 'host' in TBaseDockHostForm and TTabDockHostForm. Strictly speaking, the TJoinDockForm should likewise be called TJoinDockHostForm, but Borland weren't as consistent as they should have been here. You'll just have to imagine the 'host' in this particular case! The key point is that the bottom two classes (in Figure 5) are 'container' windows which contain any IDE forms that are docked into them. To make this clearer, let's suppose, for the sake of argument, that you have two dockable forms, the Object Inspector and the Project Manager which are both linked together with a joined dock. In this particular case, the IDE will automatically create a window of class TJoinDockForm, and the two aforementioned IDE windows will both be regarded as dock clients within this enclosing window.

When you grab one of the docked forms and drag it away from the containing window, the code inside TJoinDockForm.FormUnDock looks to see if the current DockClientCount is equal to two, in other words, if there are currently only two windows in the 'join'. If so, we know that it's time to kiss goodbye to the enclosing TJoinDockForm window! Accordingly, the code posts a special UM_FinalUnDock message back to the enclosing window, this is one of the internal IDE messages that I mentioned earlier as being defined inside IDEMessages.Pas. The UM_FinalUnDock message gets fielded inside the TBaseDockHostForm.UMFinalUndock message handler where the containing window is first hidden to prevent flicker, the last remaining dock client is 'floated off' via a call to TControl.ManualFloat and finally the containing window commits suicide by calling Release on itself! The net effect is that the last remaining dock client is freed from captivity, with the containing window silently exiting stage left.

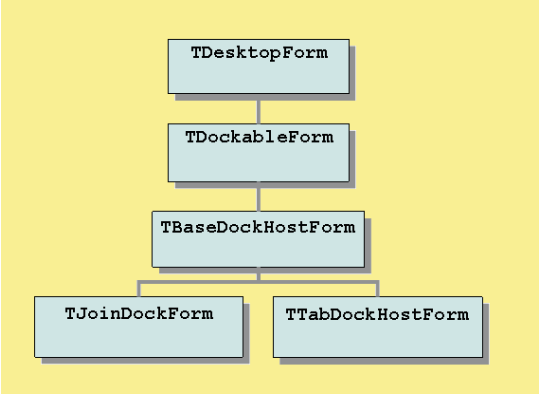
The same sort of thing happens with a tabbed dock. In this case, of course, the containing window is an object of class TTabDockHostForm which implements some additional goodies such as a special popup menu that only ever appears when you right-click on the tabs themselves, this allows the tabs to be repositioned to any of the four edges of the containing window. There's also a property, DockPage, which provides access to a private field of type TTabDockPageControl. This is a specialised descendant of the familiar TPageControl component, implemented in the IDEDockCtrls unit. Inside the implementation of TTabDockPageControl, the DoRemoveDockClient method tests to see if the page count has dropped down to one. If so, it fires off a UM_FinalUnDock message to its parent window, the TTabDockHostFormObject. The most important aspect of this elegant architecture is the fact that both the 'container' classes, TJoinDockForm and TTabDockHostForm, are themselves derived from TDockableForm, and can therefore be docked to one another, or with another 'standalone' dockable form.

The thing that rather irritates me about all this is the fact that Borland have never made these routines available to ordinary Delphi developers, despite the fact that many programmers are understandably keen to emulate the same dual-mode (join and tab) docking system that's used by the IDE. Why have Borland never made this stuff available? Why isn't it part of the VCL component library? What's the big deal? Maybe in Delphi 6?

There's a lot more that could be said about DSNIDE50, but I think I'll leave things there for now. In a future article, perhaps we'll explore the way in which a dockable form can persistently save its state using the TMemIniFile argument, which is used within the IDE to store desktop information. Or maybe I'll go mad and present the full source code for the IDE's dual-mode docking system? Perhaps if Borland are too busy playing with Kylix, they won't notice. ☺

Incidentally, if you're serious about building Delphi add-in experts, I'd definitely encourage you to get a copy of CodeRush from www.eagle-software.com, the latest version of which is 5.03h. This isn't to suggest that you always create CodeRush experts (ie packages that are written for, and depend upon CodeRush API), but whether you go down this route or create 'native' experts, you'll generally find that CodeRush brings a big productivity boost to your development work. In addition to that, the add-ins that come with CodeRush are themselves fascinating examples of just what can be done when you get into a sufficiently intimate relationship with the IDE internals.

➤ *Figure 5: The dual-mode (joined versus tabbed) docking architecture used by the IDE accounts for the split in the road when viewing the class hierarchy. Most of the clever stuff happens in TDockableForm and TBaseDockHostForm.*



Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level work. You can contact Dave at TechEditor@itecuk.com