Database, Heal Thyself

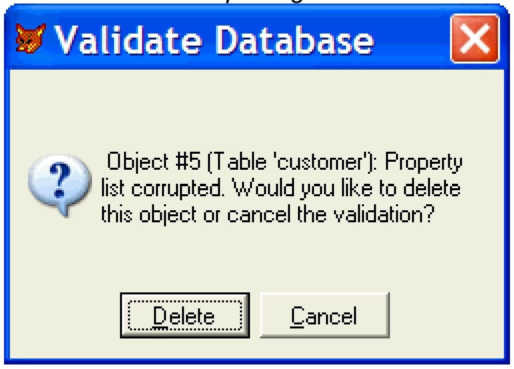
Doug Hennig

Like regualr FoxPro tables, database containers are subject to corruption. While fortunately this is rare, when it does happen, it can be the start of a very bad day. In this month's article, Doug presents a utility you can use to repair corruption in the PROPERTY memo of DBC records. In addition, this utility can be used to change read-only properties, such as the path for a table or the SQL SELECT statement of a view.

A couple of months ago, a friend (who shall rename nameless, but he knows who he is <g>) emailed me with a problem. It seems a database container he'd been working on had become corrupted, and of course he had no recent backup.

How do you fix a damaged DBC? VFP's built-in method is to use VALIDATE DATABASE RECOVER. However, as you can see in Figure 1, the fix is rather draconian: you have to remove the entire table from the DBC, which means you lose triggers, rules, long field names, default values, and all those other things we like to use a DBC for. There isn't an option to simply fix the problem.

Figure 1. VALIDATE DATABASE RECOVER prompts you to remove an invalid object from the DBC rather than repairing it.



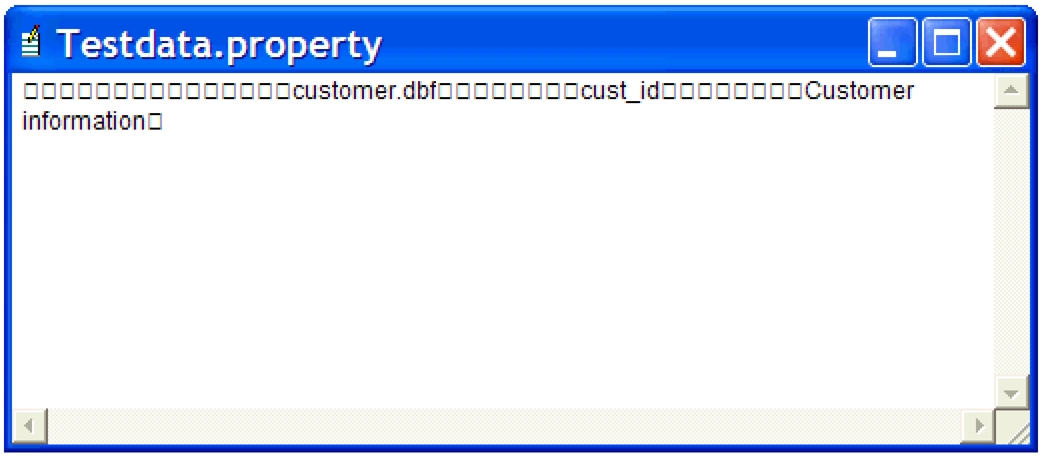
Object #5 (Table 'customer'): Property list corrupted. Would you like to delete this object or cancel the validation?

When my friend ran VALIDATE DATABASE, he got the following results:

```
Validate Database:
Rebuilding structural index....  Index rebuilt.
 Object #43 (Table 'Threads'): Property list corrupted.
 Object #76 (Table 'forums'): Missing required property.
 Object #132 (Field 'forumid'): Property list corrupted.
 Object #145 (Field 'address'): Property list corrupted.
 Object #174 (Field 'type'): Property list corrupted.
```

Notice that all of the errors regard properties. So, you might be tempted to open the DBC as a table (USE DBC() AGAIN), browse it, find the appropriate record, and modify the PROPERTY memo as necessary. Figure 2 shows the typical contents of this memo for a table. What the heck are all those little squares? They're unprintable characters, typically ASCII values less than 32. Oh well, you can probably ignore them and just edit the readable text, right? Wrong! The binary values in the memo contain important information, such as the size of a particular property. So, if you edit, say, the path of the table, and the value you enter has fewer or more characters than the original value, you've just damaged that record in the DBC.

Figure 2. The PROPERTY memo of the DBC contains binary information and shouldn't be edited directly.



```
□□□□□□□□□□□□□□customer.dbf□□□□□□□□cust_id□□□□□□□□Customer
information□
```

Because this type of problem arises from time to time, I decided to create a utility that allows you to edit the PROPERTY memo for records in a DBC in a safe manner. That utility is called FIXDBC. However, before we look at FIXDBC, let's examine how VFP stores properties in the DBC.

How a DBC stores properties

Each property in the PROPERTY memo is formatted as follows:

```
<L1> <L2> <L3> <L4> <N1> <N2> <B1> ... <Bn> <VALUE>
```

where:

- <L1> to <L4> represent the length of the entire property. This length is stored as a 4-byte binary number, with the least significant byte in <L1>. For example, a length of 15 would be stored as 15 0 0 0 (note: this doesn't mean the digits "1 5 0 0 0", but CHR(15) CHR(0) CHR(0) CHR(0)). A length of 260 would be stored as 255 5 0 0.
- <N1> and <N2> represent the length of the ID code for the property. This length is stored as a 2-byte binary number, with the least significant byte in <N1>. Since all current properties have an ID less than 255, these bytes contain 1 0.
- <B1> to <Bn> is the ID code for the property stored as a binary number as long as defined in <N1> and <N2>. Since all current properties have an ID less than 255, only 1 byte is used.
- <VALUE> is the actual value of the property. Strings are terminated with CHR(0).

This sequence is repeated for each property within the memo.

Here's an example: the Path property for a table, which has an ID of 1. In this case, the value of the property is "dbfs\customer.dbf":

```
25 0 0 0 1 0 1 dbfs\customer.dbf 0
```

For a complete list of properties and their IDs, see PROPS.DBF in this month's Subscriber Downloads. Table 1 shows some sample records from this table.

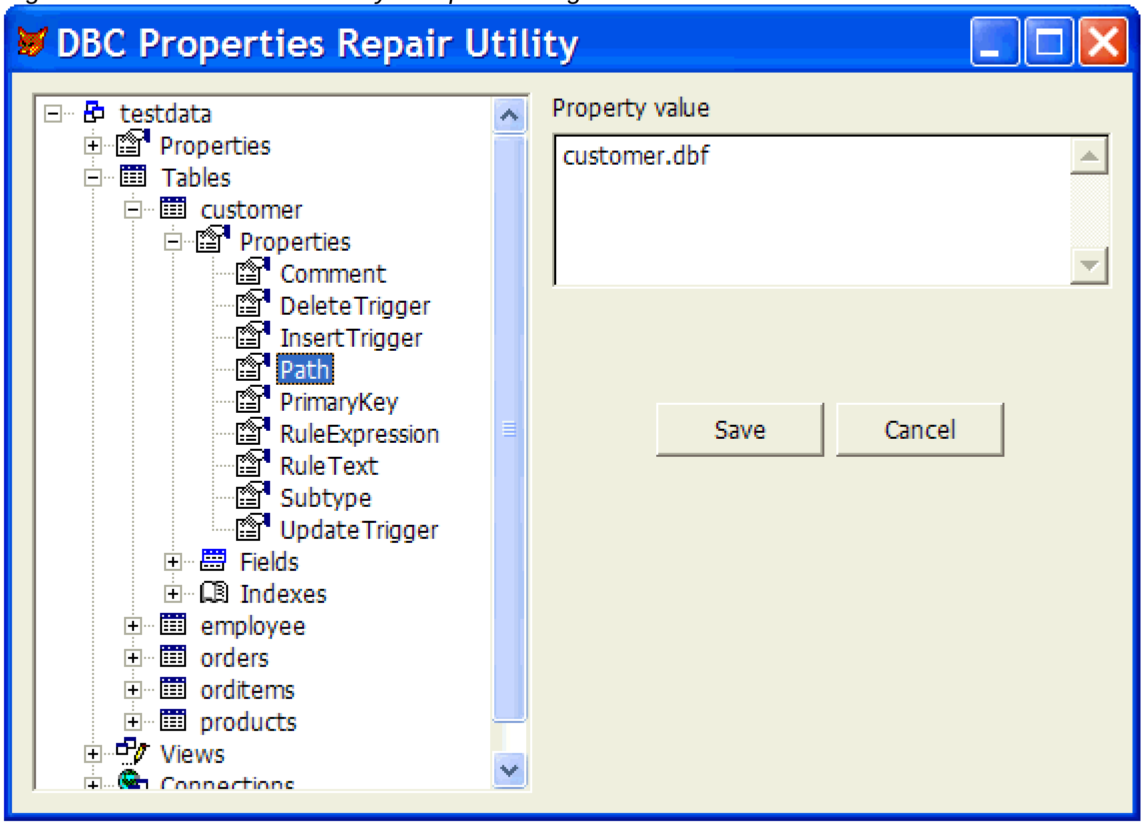
TYPE	SUBTYPE	PROPERTY	ID	BINARY	COMMENT
Database		Comment	7	F	
Table		Path	1	F	
Field		Caption	56	F	
Field	View	UpdateName	35	F	
Index		Primary/Candidate	17	T	0 = no, 1 = yes

Table 1. PROPS.DBF contains information about every type of property available.

FIXDBC to the rescue

Now that we know how properties are stored, let's take a look at the utility to edit them. Figure 3 shows FIXDBC in action. To use this tool, open the DBC that's corrupted, then DO FORM FIXDBC. The TreeView at the left shows all the tables, fields, views, and connections in the selected database, and the properties stored in the PROPERTY memo for each item. The editbox at the right allows you to see and edit the value of the selected property. Some properties have comments about the values they hold (for example, the comment for a view's WhereType property is "1 = key fields only, 2 = key and updatable fields, 3 = key and modified fields, 4 = key and timestamp"); these comments are displayed below the editbox. To change the value of a property, simply edit the value in the editbox and click on the Save button. To cancel any changes you've made, click on another item in the TreeView or click on Cancel.

Figure 3. FIXDBC makes is easy to repair damaged PROPERTY memo values.



customer.dbf

Let's start our examination of FIXDBC.SCX by looking at how the TreeView is loaded with the various objects in the DBC. If you haven't worked with a TreeView control before, don't worry – it's pretty simple. To add a TreeView control to a form, drop an OLEControl on the form and select Microsoft TreeView Control (there may be several versions; choose version 6 if so) in the Insert Object dialog. You add a node to the TreeView using the following syntax:

```
TreeView.Nodes.Add(Relative, RelationShip, Key, Text, ;
  Image, SelectedImage)
```

where:

- *TreeView* is the object reference to the TreeView control.
- *Relative* is the index or key of an existing node object. If it isn't specified, the new node is placed at the end of the top node hierarchy.

- *Relationship* is where the new node should be placed relative to the node specified in the first parameter:
    - 1: the node is placed at the end of all other nodes at the same level of the relative node.
    - 2: the node is placed after the relative node.
    - 3: the node is placed before the relative node.
    - 4: the node becomes a child of the relative node.
- *Key* is a unique string used to identify the node.
- *Text* is the text displayed in the TreeView for the node.
- *Image* is the index or key of an image in the associated ImageList control.
- *SelectedImage* is the index or key of an image in the associated ImageList control that's shown when the node is selected.

Notice that TreeView images come from an ImageList control. As with the TreeView control, drop an OLEControl on the form and choose Microsoft ImageList Control. To load images into the control, right-click on it and choose ImageListCtrl Properties from the shortcut menu, then choose the Images page in the ImageListCtrl Properties dialog and click on Insert Picture to add an image. Images can be referenced by the image number, but it's easier to assign a key value to them in this dialog and refer to them by the key (such as "Table" for the image used for table nodes).

One slight gotcha: you can't associate a TreeView control with an ImageList control visually; you have to do it in code. Here's an example (the ".Object" is required to make this work):

```
Thisform.oTree.ImageList = Thisform.oImageList.Object
```

Let's look at LoadTree method of the form. This method, which is called from Init, is responsible for loading the TreeView with the various objects in the DBC.

```
local laTables[1], ;
  lnTables, ;
  lcKey, ;
  lnI, ;
  lcTable, ;
  laViews[1], ;
  lnViews, ;
  lcView, ;
  laConnections[1], ;
  lnConnections, ;
  lcConnection
with This

* Add nodes for the database and its properties.

  .oTree.Nodes.Add(, 1, 'D', lower(set('DATABASE')), ;
    'Database')
  .LoadProperties('D', 'Database')

* Load tables into the TreeView.

  lnTables = adbobjects(laTables, 'Table')
  if lnTables > 0
    asort(laTables)
    lcKey = sys(2015)
    .oTree.Nodes.Add('D', 4, lcKey, 'Tables', 'Table')
    for lnI = 1 to lnTables
      lcTable = laTables[lnI]
      .LoadTable(lcTable, 'Table', lcKey)
    next lnI
  endif lnTables > 0
```

```
* Load views into the TreeView.

  lnViews = adbobjects(laViews, 'View')
  if lnViews > 0
    asort(laViews)
    lcKey = sys(2015)
    .oTree.Nodes.Add('D', 4, lcKey, 'Views', 'View')
    for lnI = 1 to lnViews
      lcView = laViews[lnI]
      .LoadTable(lcView, 'View', lcKey)
    next lnI
  endif lnViews > 0

* Load connections into the TreeView.

  lnConnections = adbobjects(laConnections, 'Connection')
  if lnConnections > 0
    asort(laConnections)
    lcKey = sys(2015)
    .oTree.Nodes.Add('D', 4, lcKey, 'Connections', ;
      'Connection')
    for lnI = 1 to lnConnections
      lcConnection = laConnections[lnI]
      .LoadConnection(lcConnection, lcKey)
    next lnI
  endif lnConnections > 0
endwith
```

This method adds nodes to the TreeView for the database itself, “Tables” (the “header” node for tables), “Views”, and “Connections”, and calls several helper methods to add nodes for each table, view, and connection. ADBOBJECTS() is used to determine the tables, views, and connections in the database. We won’t look at LoadTable (which loads nodes for either a table or a view) or LoadConnection, but we will look at LoadProperties, which adds a node for each property of the current item.

LoadProperties expects to be passed the key for the TreeView node for the item the properties are for, the type of item (such as “Field” or “Table”), and optionally the type of the item’s parent (only used for fields, in which case it’ll be either “Table” or “View”). It goes through the PROPS table and adds one node for each property that’s applicable to the type of item.

```
lparameters tcParentKey, ;
  tcType, ;
  tcParentType
local lnSelect, ;
  lcPropsKey, ;
  lcProperty, ;
  lcKey
with This
  lnSelect = select()
  if tcType $ 'Table,Database,View'
    lcPropsKey = sys(2015)
    .oTree.Nodes.Add(tcParentKey, 4, lcPropsKey, ;
      'Properties', 'Property')
  else
    lcPropsKey = tcParentKey
  endif tcType $ 'Table,Database,View'
  select PROPS
  scan for TYPE = tcTYPE and (empty(SUBTYPE) or ;
    (not empty(tcParentType) and SUBTYPE = tcParentType))
    lcProperty = trim(PROPERTY)
    lcKey      = 'P' + tcType + '/' + ;
      iif(tcType = 'Database', tcParentKey, ;
      substr(tcParentKey, 2)) + '/' + lcProperty
    .oTree.Nodes.Add(lcPropsKey, 4, lcKey, lcProperty, ;
      'Property')
  endscan for TYPE = tcTYPE ...
  select (lnSelect)
endwith
```

When you click on a node in the TreeView, the TreeView's NodeClick method fires. This method calls the GetCurrentProperty method of the form, passing it an object reference to the selected node. GetCurrentProperty determines which item the node is for and which property was selected by pulling that information from the Key property of the node. When LoadProperties adds a property node to the TreeView, it assigns a key such as "PTable/customer/Path". In this example, "P" indicates that this is a property node (GetCurrentProperty disables the editing controls if it isn't), "Table" indicates this is a table (as opposed to a field, view, database, or connection), "customer" is the name of the object, and "Path" is the name of the property. The code then calls the FindObjectInDBC method to position the DBC (which is opened as a table in Init) to the record for this item, locates the record for the property in PROPS, and calls GetProperty (which we won't look at) to read the value for the specified property from the PROPERTY memo. It then updates the controls to display the value.

```
lparameters toNode
local lcKey, ;
  lnPos, ;
  lcType, ;
  lcObject, ;
  lcProperty, ;
  lcValue
with This
  lcKey = toNode.Key

* If this is a property node, figure out which property
* for which object.

  if left(lcKey, 1) = 'P'
    lcKey      = substr(lcKey, 2)
    lnPos      = at('/', lcKey, 1)
    lcType     = left(lcKey, lnPos - 1)
    lcKey      = substr(lcKey, lnPos + 1)
    lnPos      = at('/', lcKey)
    lcObject   = left(lcKey, lnPos - 1)
    lcProperty = substr(lcKey, lnPos + 1)

* Find the record for the object in the DBC, then get
* the value of the specified property.

    .FindObjectInDBC(lcObject, lcType)
    = seek(padr(lcType, len(PROPS.TYPE)) + lcProperty, ;
      'PROPS', 'PROPERTY')
    lcValue = .GetProperty(PROPS.ID, PROPS.BINARY)

* Update the controls.

    .edtValue.Enabled   = .T.
    .edtValue.Value     = lcValue
    .lblComment.Caption = PROPS.COMMENT
    .cmdUpdate.Enabled  = .T.
    .cmdCancel.Enabled  = .T.
  else
    .edtValue.Enabled   = .F.
    .edtValue.Value     = ''
    .lblComment.Caption = ''
    .cmdUpdate.Enabled  = .F.
    .cmdCancel.Enabled  = .F.
  endif left(lcKey, 1) = 'P'
endwith
```

The Click method of the Save button calls the form's SetCurrentProperty method, which in turn calls SetProperty to write the property value to the PROPERTY memo in the DBC. Feel free to examine this code yourself.

Check it out

To see FIXDBC in action, make a copy of a database container and its tables in a test directory. USE the DBC as a table, browse it, find a record with OBJECTNAME = "Table", and change the path of the table in the PROPERTY memo (for example, add "..\" in front of the existing path). USE to close the DBC as a table, then open it as a database using OPEN DATABASE. To confirm that the table's record in now damaged, type VALIDATE DATABASE. DO FORM FIXDBC, navigate through the TreeView to the Path property for that table (you'll get a warning message that the property's actual length doesn't match the stored length), enter the correct path in the editbox (for example, remove the "..\"), and click on Save. You should no longer get an error when you use VALIDATE DATABASE.

Other uses for FIXDBC

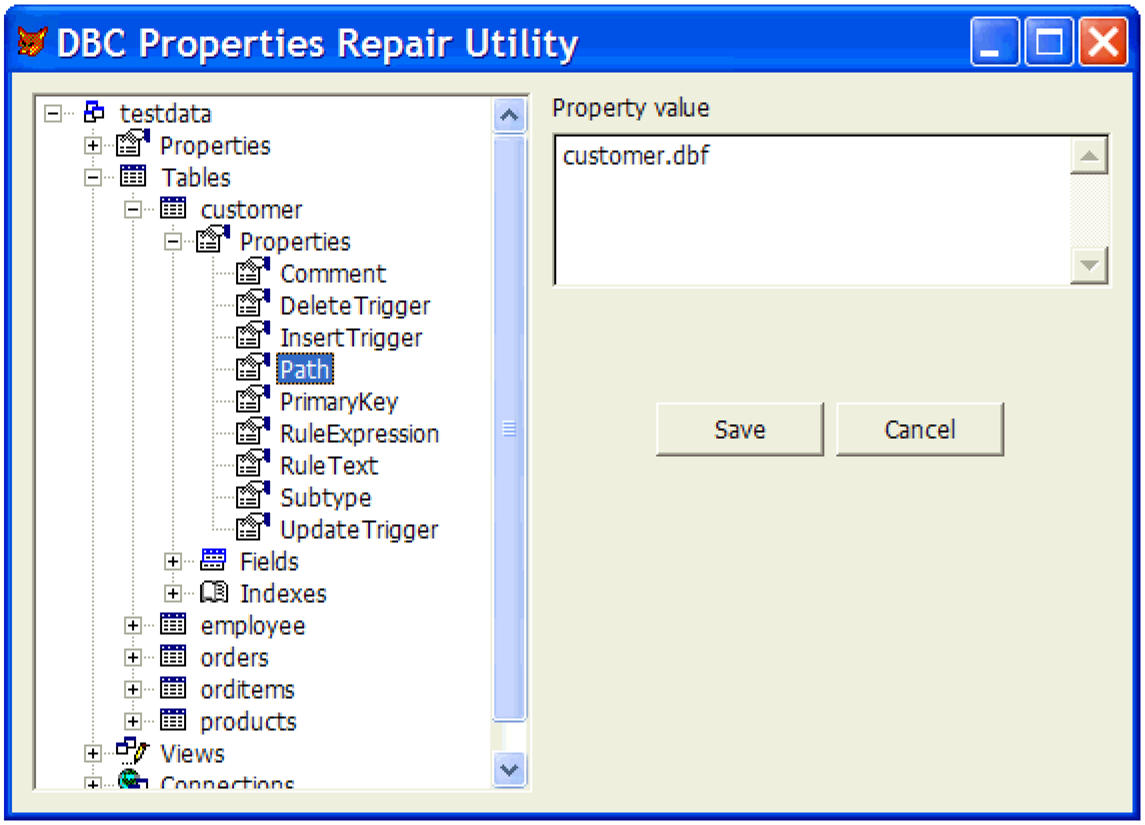
In addition to helping you repair corrupted properties in DBC records, FIXDBC also allows you to change the values of properties, even those that are supposedly read-only. For example, while DBGETPROP() can give you the value of the Path property for a table (the DBF name and location), DBSETPROP() cannot change this value, so you can't programmatically change the DBF name or location of a table. However, this is easily done in FIXDBC: simply navigate to the Path property for the table, enter the desired value, and click on Save. Of course, since the table header contains a backlink to the DBC, you have to adjust that as well, but that's easily done: open the table EXCLUSIVE, and when VFP complains that the backlink is invalid and asks if you want to locate the DBC, choose Yes and do so.

Even better, you can use FIXDBC to change the SQL SELECT statement for a view. This has a couple of possibilities:

- If you need to make a simple tweak to the SQL statement (not adding or removing fields, but perhaps changing the WHERE or ORDER BY clause), it may be easier to do that in FIXDBC than worrying about whether the VFP View Designer will mangle your view (not as much of a problem starting in VFP 8) or modifying and then running some code to redefine the view programmatically.
- It's easy to create a flexible WHERE clause for a view using FIXDBC. I don't mean having a parameter in the WHERE clause (such WHERE COUNTRY=?pcCountry) but having the entire clause be macro-expanded (such as WHERE &lcWhere) when the view is opened or requeried. This allows you to completely change the type of records that are retrieved simply by changing the value of the variable. You can even create a flexible ORDER BY clause (that is, ORDER BY &lcOrderBy). Figure 4 shows an example of a view with macros in both the WHERE and ORDER BY clauses. In this case, creating a cursor of German customers sorted by city is a simple as:

```
lcWhere   = 'COUNTRY = "Germany"'
lcOrderBy = 'CITY'
use CUST_VIEW
```

Figure 4. FIXDBC makes it easy to create views with flexible WHERE, ORDER BY, and other clauses.



Summary

While you likely won't need to use it often (at least hopefully), FIXDBC may be a life-saver when you do need it. In addition, being able to create more flexible views means that you may not need to create as many views for your application.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com