## Version icon: 5.0
## Platform icon: Windows

## Figures: 07HENTZ1.TIF, 07HENTZ2.TIF, 07HENTZ3.TIF
## File for Companion Disk: 07HENTZE.EXE Whil: Please insert name if indeed there's a download file. JF

Using DBCx to Data-Drive Your Applications

Whil Hentzen

"Been there, done that"—getting tired of this late '90s bromide? Nearly as tired as doing something for the 50th time, I'll bet. While the VFP 5.0 database container has a number of improvements over previous versions, it isn't complete. In this article, I'll examine a number of places in your applications where data dictionary extensions are still needed, and show you how to use a publicly available data dictionary extender, DBCx, to provide that functionality.

The Visual FoxPro Database Container (DBC) is an excellent starting point for storing meta data for applications, but as I've described in previous *FoxTalk* articles ("Extending the Visual FoxPro Data Dictionary," May 1995; "Extend the Visual FoxPro Database Container with EDC," August 1995), it isn't complete. Two solutions were proposed during the VFP 3.0 beta cycle, Tom Rettig's EDC and DBCx, the result of a collaborative effort between Flash Creative Management, Micromega Systems, Neon Software, and Stonefield Systems Group.

Two years later, VFP has matured and the database container has received some enhancements, but the fundamental requirements for extending the data dictionary remain. Let's revisit what the database container does in VFP 5.0, what it still won't do, what we would like it to do (or, at least, what we need), and how we can get that functionality from DBCx.

What we can get from 5.0's DBC

Let's first review what we're starting out with. The DBC has a row for every table, field, index, view, and connection (as well as a few rows for other specialized objects). Each row contains fields for a unique ID, an ID that points to the parent object of that row's object (for example, the parent of a table is the database; the parent of a field is the table), a description of the object, an empty, unused memo field named *User*, and most importantly, not much else.

Well, okay, there is a memo field called Properties that looks promising. However, upon further examination, it isn't all that helpful. It contains the data that you can enter into various controls in the Modify Structure dialog box, such as Display Format and Validation Rule. Why isn't this helpful?

Four basic types of information can be entered in the Modify Structure dialog box. Field characteristics such as size and type are stored in the header of the DBF itself. Display, Field validation, and Map field information is all stored in the Properties field, but to see why this isn't useful, let's examine how this information is used.

Look at the Field Mappings tab in the Tools/Options dialog box (See Figure 1.) At the bottom of the page, four check boxes specify what will happen when a field is dragged from the Data Environment to a form during design. If the Drag and Drop Field Caption check box is selected, the value in the Modify Structure Caption field will be used as the caption for the adjoining label created when the field is dragged from the DE to a form.

Similarly, if the Copy Field Comment, Input Mask, or Format check boxes in the Field Mappings tab are selected, the values in those controls in the Modify Structure dialog box will be placed in the respective properties for that field when the field is dragged from the DE to the form.

However, try this: Make a change to the comment, and then drag the field from the DE onto the form again. The new comment will be reflected in the new field, but the first copy of the field will still have the old field comment. Maybe comments aren't a big deal, but other properties are—such as Format and Input Mask.

## Production: Insert 07HENTZ1.TIF
## Whil: You did not send me this TIF. Please send right away. Thanks. JF

***Figure 1**. Much of the information in the Modify Structure dialog box is stored in the properties field in the DBC.*

## Whil: We need a reference to Figure 1 somewhere in the body text. Will you pick a spot and insert it? Thanks JF

Next, let's look at the Validation rules. Although they're entered in the Modify Structure dialog box and aren't stored in the Properties field, changes do propagate during the life of the application. The reason is that the VFP database engine actively reads from the DBC during runtime to evaluate the values in the DBC that involve validation.

Finally, let's examine the Map Field Types to Classes controls. In the Field Mapping tab of the Tools/Options dialog box (see Figure 2), you can specify which of your classes you want to use as the default class when dragging fields from a DE to a form, instead of using VFP's base classes. Each data type can be defined differently—typically, you might specify a check box class for the logical data type and an edit box class for memo fields.

## Production: Insert 07HENTZ2.TIF
## Whil: I do not have this figure. Please send. JF
## Also, we need a reference to Figure 2 within the text. Please insert.

***Figure 2**. The Field Mapping tab of the Tools/Options dialog box allows you to specify which class is used when dragging fields from a Data Environment to a form.*

The default mappings that you selected can be overridden on a field by field basis for specific needs with the Map Field Types to Classes controls in the Modify Structure dialog box. The Display Library value points to the VCX while the Display Class is the actual class itself. These settings override, for a specific field, which class will be used when dragging that field to any form. This information is also stored in the Properties field in the DBC, and, like the Display properties, is read only once during initial creation of the control. Changes made to the Map Field Types controls aren't propagated through the application as changes are continually made to them.

So, the bottom line is that much of the information entered into the Modify Structure dialog box and stored in the Properties field in the DBC is used only once. We'd like that information to persist through the application through its lifetime—not just upon initial application.

What's missing from the DBC?

*\\\ the structural information isn't stored in the DBC. Did I get that right, Jeana?

One requirement of a complete data dictionary is that the developer be able to rebuild the data structures from the information in the data dictionary. Because structural information such as field type and size and index tag expressions isn't stored in the DBC, this is impossible. This means that we can't build a reindex routine, a rebuild routine, or other similar maintenance routines that rely solely on information in the DBC.

But there's more missing than that. It doesn't take a lot of imagination to think of other information about the data structures that we'd like to keep around. Starting with the tables themselves, useful information might include long descriptive names, a flag for whether the table should be opened upon application startup, an alias to use during opening, and perhaps a default or specialized location.

With?? ## Whil: I don't understand "going next to" here. JF That's cuz you're not a geek… <g> The table is one level. The field is the next level, and the index is the third level. We often go "from databases, to tables, to fields, to indexes"## fields, we need several types of long descriptive names. The first name would be used for a persistent caption for an associated label. The next could be used in grids and lists that use the field, and a third for report headings. Obviously we need to store length, type, and whether nulls are allowed. Other useful things include persistent formats and input masks, as well as tool tips, help text, status bar text, and allowed data ranges.

Indexes can use a long descriptive name, a flag indicating whether they're selectable by the user, and a comment field.

A few more minutes of thought might reveal a variety of other information that you might like to keep around, such as security-related settings that allow read/write access or even visibility of selected fields. Or how about combined fields for output, such as the concatenation of first, middle, and last name fields—there is no place for this type of meta data in the DBC.

How do we get there? The DBCx concept

Obviously, we need to store this additional meta data somewhere—let's review quickly what the options are. The first choice of many developers might be that unused empty memo field in the DBC named User. Microsoft left it there for the express purpose of DBC extensions. Why not just store information in this memo field?

For example, we could store the following data in the DBC record for the League Name field, cNaLeague:

```
Caption: Name
List: Lg Name
Report1: League
Report2: Name
Format:
Help: The League Name identifies the
       specific group that plays on a . . .
StatusBar: Enter the League Name
```

This approach has many problems. Two obvious issues involve performance and conflict. Parsing out data from memo fields is time-consuming compared to simple SELECTS from an optimized table. And what happens when more than one type of meta data is being stored in the User field? If you're the only person using the User field, you're safe, but your troubles begin the minute you decide to take advantage of a third-party tool that uses the User field to store more information.

The originators of DBCx started out with the same concept as Tom Rettig did with EDC: store a single key value in the User field that points to another table. But that's pretty much where the similarity ended. Because one requirement of DBCx was to allow multiple third-party vendors to play in the same sandbox without fighting, a single table to store additional meta data was going to become difficult to administer.

Instead, they used the concept of a *registry*—anyone who wanted to share in the DBCx plan could include his or her own record without stomping on anyone else's turf. Each of these records would point to the data dictionary extensions for that third party. Because many developers had common needs, the collaborators agreed to use Flash's Codebook as the location for a set of basic extensions, and then each would write his or her own extensions as needed after that (see **Listing 1**).

```
DBCXREG.DBF
mDBCPath
cDBCName
cProdName      Codebook
cVersion       V3.2
mDBCXPath
cDBCXName      CDBKMETA.DBF
cDBCXAlias     CODEBOOK
mLibPath       ..\..\..\COMMON50
cLibName       CDBKMGR.VCX
cClassName     CdbkMgr
iLstID
tLastUpdt
lDefault
cObjName
```

Listing 1. *The record in the registry, DBCXREG.DBF, that points to the Codebook extensions.*

It’s important to remember that the records in the DBCXREG.DBF registry are simply pointers to a second set of tables. This second set of tables actually contain the extensions to the data dictionary. Typically, they each take the form of a table where each field in the table represents a different extension (such as long descriptive table name, field length, index tag, or tool tip text). Each record in the table is usually mapped to a corresponding record in the DBC via the ID stored in the DBC’s User field. Each table is supplied by a different third party, such as Codebook, Micromega, Neon, or Stonefield.

Thus, if the DBC record for the League Name field had a User ID value of 422, there would be a matching record in the Codebook table that had a unique ID of 422, a matching record in the Micromega table that also had a unique ID of 422, and so on. See Figure 3 for an example of the relationships between the tables.

## Production: Insert 07HENTZ3.TIF
## Whil: Again, I don’t have this TIF, nor is there a reference to it within the text. Please add. Thanks.

Figure 3*. The User field in the DBC contains an ID that points to a record in one or more of the specific meta data tables. Here, the ID 422 points to the record for the League table in the CDBKMETA table.*

By now, you may think that we’re done. We’ve seen how all the data hooks together, but there’s a missing piece: the set of programs that gets all of these pieces to work together. It isn’t evident that the piece is missing because there is no corresponding piece for DBC. Well, actually, there is, but it’s invisible—it’s part of the VFP engine inside VFP.EXE. Because DBCx isn’t part of VFP, it has to be driven by a set of external programs.

DBCx consists of a class library, DBCXMGR.VCX, that contains pointers to class libraries for each extension, such as CDBKMGR.VCX, SDTMGR.VCX, and so on. In order to get things going, instantiate DBCXMGR like so:

```
set classlib to DBCXMGR additive
oMeta=createobject('MetaMgr',.f.)
```

These two lines of code create a “meta-manager” object onto which are hung additional objects for each registry entrant in DBCXREG.DBF. Each of these objects has its own custom methods and properties that allow access to its data dictionary extensions. Third-party products typically will provide a set of wrappers to shield you from having to delve deeply into the DBCX internals—after all, that’s their job, right? For example, the reindex routine in the Stonefield Database Toolkit can be called like this:

```
oMeta.oSDTMgr.lQuiet = .f.
m.llIndexWorked = oMeta.oSDTMgr.Reindex()
if m.llIndexWorked
  wait window nowait "Reindex successful"
else
 messagebox("Reindex was unsuccessful. ;
            Call for help.")
endif
```

## Production: Please try to fit second-to-last line of code, above, on one line. Thanks. JF I did it.

Let’s look behind the scenes at what’s happening, and what you would do to manually access meta data through DBCx. There are two basic steps. First, given a particular object, such as a table, field or index, find its User ID. Next, given that ID, find the specific value for a data dictionary extension of interest, such as a long descriptive name for an index or a tool tip for a field.

For example, suppose you want to get the long descriptive name and the status bar text for a field. Here’s how you’d get the ID:

```
m.liIDOfField = oMeta.DBGetDBCKey(dbc(), ;
               'Field', 'Database.Field')
```
## Whil: This line is too long – can it break before the 55-column mark?

In this line, “Field” is the actual literal parameter but “Database.Field” represents a string such as “Customer.Address.” Remember that m.liIDOfField is the value in the User field, such as 422 for League Name, as described earlier in this article.

Now that you have the ID, you can fetch one or more properties:

```
m.lcNaFieldLong = oMeta.DBCXGetProp('SDTmCaption', ;
                  m.liIDOfField)
m.lcToolTipText = oMeta.DBCXGetProp('CBmToolTip', ;
                  m.liIDOfField)
```
## Whil, again these lines are too long. Please shorten and wrap. Thanks.

Because you’re getting these values from the DBC and the DBC extended tables on the fly, you can create data-driven applications that always use the very latest data. Let’s look at a practical implementation.

How to extend through DBCx

One thing I always hate about generic “how-to” articles is that I still have to spend lots of time trying to figure out how to get the process to work with *my* stuff. In an attempt to avoid that, I’ll describe one possible real-life scenario, pointing out some of the issues you might run into as well as a possible solution.

Here are the three things you have to know:

1. What files are needed.
2. Where they go.
3. Where the commands that run DBCx go in your application.

The files you need depend on which third-party products or tools you’re using. Let’s use Codebook as our starting point. You’ll need to put DBCXMGR.VCX and CDBKMGR.VCX in your path so that when you build your applications, they can be found. I put these in COMMON because only one copy is needed across all applications.

Each of your applications needs its own copy of DBCXREG.DBF and CDBKMETA.DBF. I put these into the APPFILES directory of an application—the place where application-specific data resides. (My user file, error log, custom reports, and so on, also go here. These files aren’t data specific, but rather common across all data sets, so I don’t keep multiple sets of the same files in each data directory.)

That’s it for the files!

Now let’s see how this stuff actually is handled in an application. You may have a startup program in which you instantiate your classes, such as oApp, oLibrary, and so on. I do, too, but I quickly found that this isn’t where oMeta should be created, because oMeta needs to know about a database, and early in our startup process, we hadn’t handled data yet. Thus, we simply initialize oMeta in the startup so that it’s scoped properly, and then actually instantiate it in the Init() of oApp’s instantiation:

```
* IT.PRG
* My startup program is always called IT.PRG.
*
<bunches of code here>

set classlib to HWAPP, HWLIB, HWCTRL, CUST addi

* Declare oMeta here even though we're going
* to instantiate it later - so it's scoped properly.
private oMeta
oMeta=.f.
wait window nowait "Setting up libraries..."
private oLib
oLib=createobject("LIB")

<more code here>

wait window nowait "Initializing..."
oApp.it()
```

Then, in oApp.init():

```
<bunches of code>
*
* data dictionary
*
set classlib to DBCXMGR additive
* Set debugging on if we're in development mode.
if this.cMethod = "DEV"
 oMeta=createobject('MetaMgr',.t.)
else
 oMeta=createobject('MetaMgr',.f.)
endif
if type('oMeta') <> "O" or isnull(oMeta)
 messagebox("Unable to instantiate data dictionary. ;
   Please call your developer. Shutting down.")
 this.lWeAreDone = .t.
endif
```

(The second parameter passed to createobject is a parameter that turns debugging messages on.)

Now we're ready to use DBCx to do some data driving. Let's use a simple form with a format and a tool tip for a text box as our first example. The database has a table named League, and the League table contains a field named cNaLeague (League Name—good thing we're going to put a tool tip on it, eh?).

In the Init of the form, we issue three commands. The first gets the ID for the League Name field, and the second and third get the format and tool tip values for that field:

```
m.liIDOfField = oMeta.DBGetDBCKey(dbc(), 'Field', ;
                'Database.Field')
```

## Whil: This line is too long.

Now that we have the ID, we can fetch one or more properties:

```
*\\\ JF: I need to verify the text of these two commands!
```

## You also need to shorten these lines! <g>

```
m.lcNaFieldFormat = oMeta.DBCXGetProp( ;
           'CBmInFormat', m.liIDOfField)
m.lcToolTip = oMeta.DBCXGetProp( ;
           'CBmToolTip', m.liIDOfField)
```

Finally, we can set the properties for the field:

```
thisform.hwTxtNaField.Format = m.lcNaFieldFormat
thisform.hwTxtNaField.ToolTip = m.lcToolTip
```

Let's examine a different way to do this. After all, it seems like a lot of code to write for a property that, once set, may not change a lot. It might be nice to have these properties *\\\ properties is correct set automatically, right? You could combine both lines of codepro in a single expression and place it in the property sheet. The following value in the Format property would automatically update the format for the field regardless of how many times it was changed:

## Whoa . . . way too long line here, Whil!

```
oMeta.DBCXGetProp('CBmInFormat', ;
  oMeta.DBGetDBCKey(dbc(), 'Field', 'Database.Field'))
```

In this article I've described the basic mechanism for using DBCx to extend the VFP 5.0 database container. Next month, I'll look at some of those routines that ought to be data driven, and explain how DBCx can help you.

Whil Hentzen is editor of FoxTalk*. whil@compuserve.com, whil@hentzenwerke.com.*