Data-Driving Applications with DBCX

By Whil Hentzen

Introduction

The Visual FoxPro Database Container (DBC) is an excellent starting point for storing meta data for applications, but it's not 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 doesn't do, what we would like (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 is a normal FoxPro table, containing a row for every table, field, index, view, and connection (as well as a few rows for other specialized objects).

The table below lists the structure of the DBC.

Field	Field Name	Type	Width
1	OBJECTID	Integer	4
2	PARENTID	Integer	4
3	OBJECTTYPE	Character	10
4	OBJECTNAM E	Character	128
5	PROPERTY	Memo (binary)	4
6	CODE	Memo (binary)	4
7	RIINFO	Character	6

8	USER	Memo	4

The Object ID field contains a unique id for each record contained in the DBC. Unfortunately, when a table's structure is modified, all of the records associated with the table in the DBC are deleted and new records are added. In other words, with each modification, all Object IDs may change.

The Parent ID field is also an integer. Its value reflects the parent of the current record in the DBC. For example, the parentid of a field is the objectid of the table the field is contained in.

The Object Type field indicates what type of object the current record holds information about. Possible object types include: Connection, Database, Field, Relationship, Table and View.

The Object Name field stores the name of a database, field, relationship, table or view. For fields, the name corresponds with the field name itself. For tables, the name is the physical file name. The object names for indexes correspond to their tag names.

The Property field is used to store the values of various properties, like captions, error messages and valid expressions that can be set in the DBC through the Modify Structure dialog.

The Code field is used to store procedures directly into the data base container.

The RIInfo field store information that is used by the RI Builder to generate stored procedures for relational integrity.

By default, no information is stored in the user memo field and this field can be used to store additional information in any manner we see fit.

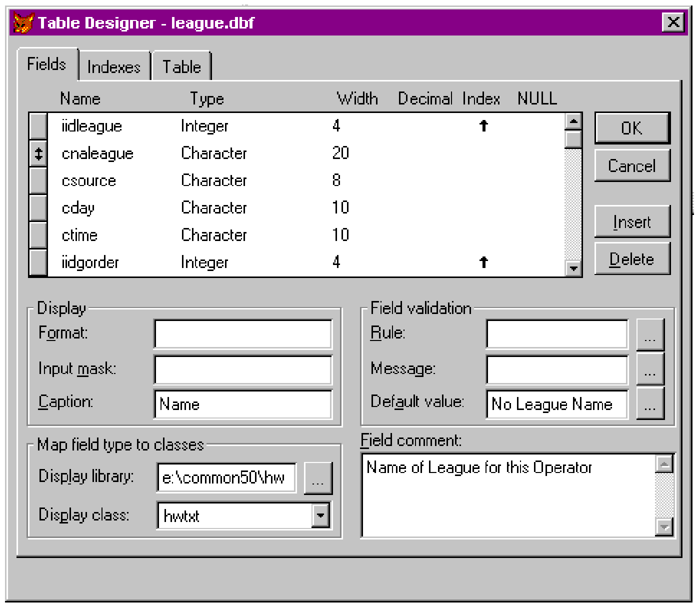
Limitations with DBC information

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

Take a look at the Field Mappings tab in the Tools, Options dialog. In the bottom of the page, there are four check boxes that specify what will happen when a field is dragged from the DE 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 Data Environment to a form.

Similarly, if the Copy Field Comment, Input Mask, and/or Format check boxes in the Field Mappings tab are selected, the values in those controls in the Modify Structure dialog 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 that big a deal, but other properties are - such as Format or Input Mask.



Name	Type	Width	Decimal	Index	NULL
iidleague	Integer	4		↑	
cnaleague	Character	20			
csource	Character	8			
cday	Character	10			
ctime	Character	10			
iidgorder	Integer	4		↑	

Display
Format:
Input mask:
Caption: Name

Field validation
Rule:
Message:
Default value: No League Name

Map field type to classes
Display library: e:\common50\hw
Display class: hwtxt

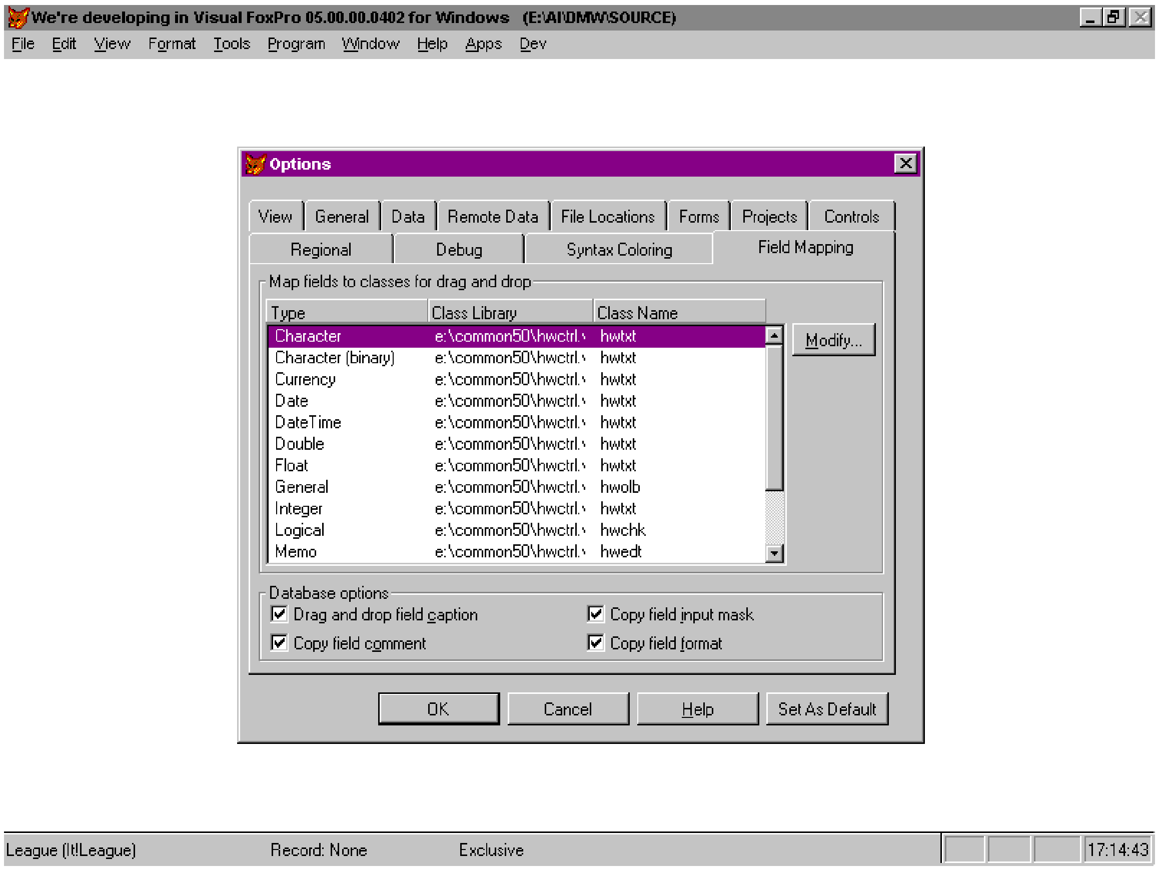
Field comment:
Name of League for this Operator

(## Production Note: This is a BMP)

Much of the information in the Modify Structure dialog is stored in the Properties field in the DBC.

Next, let's take a look at the Validation rules. Although they are entered in the Modify Structure dialog, and are not 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 run time to evaluate the values in the DBC that have to do with validation.

Finally, let's examine the Map Field Types to Classes controls. In the Field Mapping tab of the Tools, Options dialog, 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.


We're developing in Visual FoxPro 05.00.00.0402 for Windows (E:\AI\DMW\SOURCE)
File Edit View Format Tools Program Window Help Apps Dev
Options
View General Data Remote Data File Locations Forms Projects Controls
Regional Debug Syntax Coloring Field Mapping
Modify...
OK Cancel Help Set As Default
League (It!League) Record: None Exclusive 17:14:43

Map fields to classes for drag and drop

Type	Class Library	Class Name
Character	e:\common50\hwctrl.\	hwtxt
Character (binary)	e:\common50\hwctrl.\	hwtxt
Currency	e:\common50\hwctrl.\	hwtxt
Date	e:\common50\hwctrl.\	hwtxt
DateTime	e:\common50\hwctrl.\	hwtxt
Double	e:\common50\hwctrl.\	hwtxt
Float	e:\common50\hwctrl.\	hwtxt
General	e:\common50\hwctrl.\	hwolb
Integer	e:\common50\hwctrl.\	hwtxt
Logical	e:\common50\hwctrl.\	hwchk
Memo	e:\common50\hwctrl.\	hwedt

Database options

- ✓ Drag and drop field caption
- ✓ Copy field input mask
- ✓ Copy field comment
- ✓ Copy field format

Production Note: This is a BMP

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

These 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. 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 are not 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 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. We'd like it to be active and dynamic.

What's missing from the DBC?

One of the requirements of a complete data dictionary is that the developer be able to rebuild the data structures from the information in the data dictionary. Given this requirement, there are two types of information missing from the DBC. The first type is a fair amount of basic structural information. For example, while there is a record for each field, that record doesn't contain the size or type of the field. That information is still kept in the DBF header.

Similarly, while there is a record for an index, the information needed to rebuild that index is missing – the expression, whether it's ascending or descending, and so on.

Finally, there isn't any place to hold your own meta data. Upon first examination, the Properties field looks promising, but 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, such as Display Format, and Validation Rule, but you can't add to it yourself, and there are limitations to what you can do with the existing information.

The second option is the User field, but, again, this has problems associated with it.

Since structural information such as field type and size and index tag expressions are not stored in the DBC, rebuilding the data structures 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, information that would be useful include long descriptive names, a flag for whether or not the table should be opened upon application start up, an alias to use during opening, and perhaps a default or specialized location.

Going next to fields, we have a need for 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 or not they're selectable by the user, and a comment field.

A few more minutes of thought come up with a variety of other information that you might like to keep around. Security-related settings that allow read/write access or even visibility of selected fields is one. Combined fields for output, such as the concatenation of first, middle and last name fields, is another - there is no place for this type of meta data in the DBC.

How do we get there?

There are basically three approaches you can take to extend the DBC.

1. Modify the structure of the DBC to include additional fields

2. Store additional information in the user memo field of the DBC

3. Store additional information in a separate table.

Adding Fields to the DBC

The first and most obvious approach is to simply add fields directly to the DBC. You can add fields to the DBC by using the DBC as a table and issuing a MODIFY STRUCTURE command. You can also add index tags to the DBC, but the indices are not stored in the DCX (the CDX file attached to the DBC table file.)

While this approach is the most obvious, it's probably the poorest solution for several reasons.

- If the DBC becomes corrupt, you'll not only lose the DBC, you'll also lose your extensions.
- If a new version of Visual FoxPro is released with DBC enhancements, what will happen to your additional fields when you open the DBC in the new version? As has been oft heard before: Unexpected results may occur.
- You cannot add records to the DBC. This means that you can't store information that doesn't relate directly to an object already in the DBC. For example, neither a calculated field for a screen or a concatenated field for a report can be mapped directly to a row in the DBC.
- The DBC already has 8 fields in it and because Visual FoxPro still has a limit of 255 fields per table, you have a limited number of extensions you can add.

For these reasons, adding fields to the DBC is probably not a very good approach to extending the DBC, except in a very limited fashion.

Storing Additional Information in the User Memo of the DBC

Because the User memo field of the DBC is a regular FoxPro memo field, the amount of information that can be stored in it is limited only by the amount of disk space on your computers. Storing additional properties in the user memo is another possibility for extending the database container.

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
```

The problems with this approach are many and varied.

- The limitation of not being able to add records to the DBC still exists.
- A large amount of code would be required to dig various sorts of data out of a single field.

- Performance problems may accrue due to the requirements of parsing data from multiple records in the memo field.
- Multiple applications or products each trying to use the User field for storing their own type of meta data will likely run into conflicts.

Storing Additional Information in a Separate Table

The last approach we'll look at is storing additional information in a separate table or tables. This approach has the disadvantage of the additional overhead of additional tables, but seems the most flexible and safest. In this approach, an id is added to the user memo of the DBC which is used as a pointer to a related record or records in an extension table.

Two public domain database container extensions, EDC by Tom Rettig and DBCX by Flash Creative Management, Micromega Systems, Neon Software, and Stonefield Systems Group that use this approach have been released to the public domain.

EDC and DBCX are very similar in terms of their functionality. Both of these extensions rely on storing an id in the user memo of the DBC and use this id to point to information in separate tables. Both use SetProp and GetProp methods similar to the Visual FoxPro DBSetProp and DBGetProp functions to read and write information from the extension.

How do we get there? The DBCx concept.

In terms of physical implementation, however, that's where the similarities ended. Since one of the requirements of DBCx was to allow multiple third party vendors play in the same sandbox without fighting, a single table to store additional meta data was going to become difficult administer.

Essentially, DBCX is a manager class. The manager class manages database container extensions. Each extension is also a manager class. Their purpose is to read and write database container extended properties.

Instead, the architects used the concept of a 'registry' - anyone who wanted to share in the DBCx plan could include their own record without stomping on anyone else's turf. Each of these records would point to the data dictionary extensions for that third party. Since 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 their own extensions as needed after that.

The Registry Table

DBCXREG.DBF, the Registry Table, is the cornerstone of DBCX. It tells DBCX, among other things, the tables that are being used as DBC Extensions and the objects used to manage those tables. It must be named DBCXREG.DBF. You can have as many of these as you need, but can only point to one within any given project. At Runtime, this table must be somewhere in your path, or "included" in your APP/EXE.

Table Structure

A description of the DBCXREG.DBF file follows:

Field	Purpose
MDbcPath	The path to a database container that may be used by the extension manager.
CDbcName	The name of the database container if one is used by the extension manager.
CProdName	The name of the registered product.
CVersion	The version number of the registered product.
MDbcxPath	The path to the product extension file.
CdbcxName	The name of the product extension file.
CDbcxAlias	The Alias name to be used when the product extension table is opened.
MLibPath	The path to the product specific extension manager class library.
CLibName	The name of the product specific extension manager class library. This may be .VCX or .PRG.
CclassName	The name of the class object for use when the object is instantiated (CREATEOBJECT).
ILastId	The last id used in the database container.
TLastUpdt	The entry was made (DateTime).

There is one index on DBCXREG.DBF:

Tag Name	Index Expression	Type
CProdName	UPPER(cProdName)	Candidate

Each registry table must have a record where the cProdName is SYSTEM RECORD. This is the only record that uses the iLastId field. The system record holds the last id used in the database container and is used to generate the next available id.

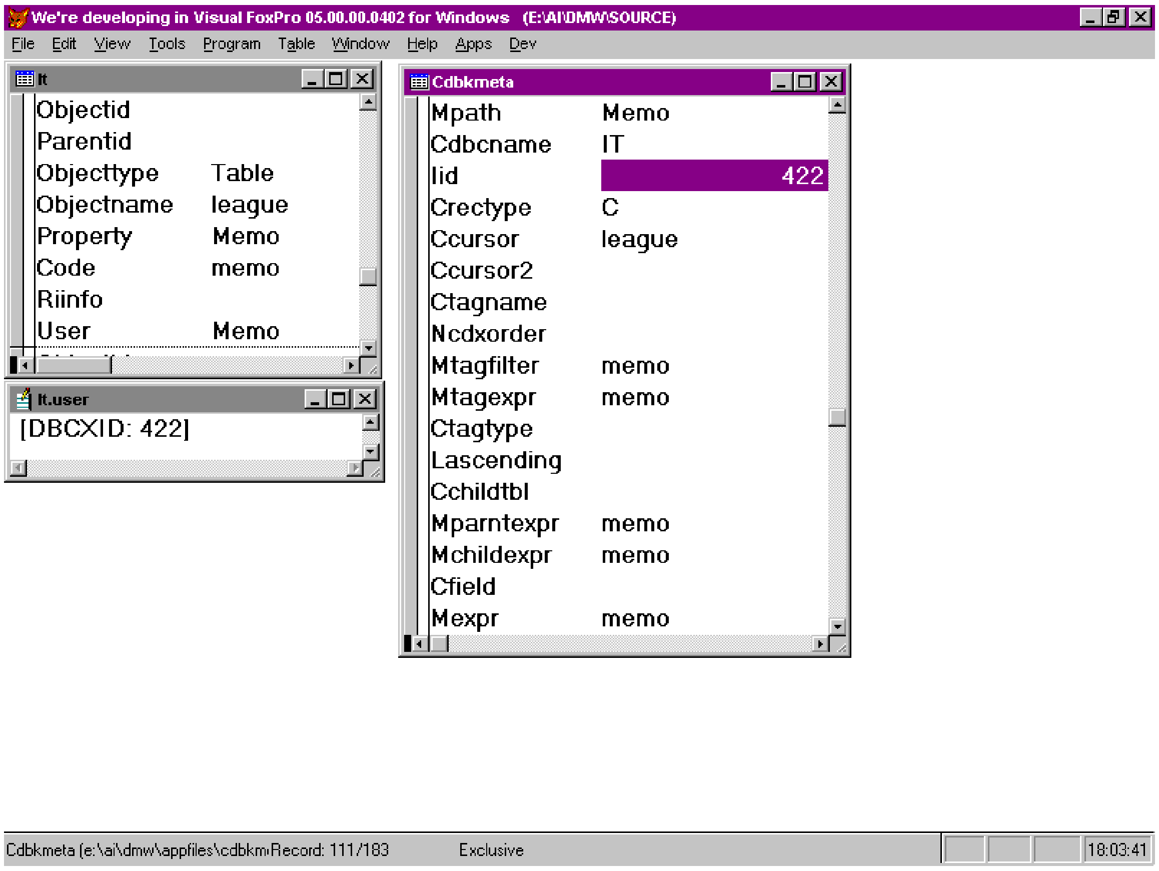
Example of DBCx Registry

Field Name	Record #1	Record #2	Record #3	Record #4
MDbcPath		Cdbk30\		
CDbcName		CdbkMgr.Dbc		

Field Name	Record #1	Record #2	Record #3	Record #4
CprodName	SYSTEM RECORD	CodeBook	FoxExpress	Foxfire!
CVersion		V3.2	V3.0	V3.01
mDBCXPath		MetaData\	MetaData\	MetaData\
cDBCXName		CdbkMeta.Dbf	FeMeta.Dbf	FfMeta.Dbf
cDBCXAlias		CodeBook	FoxExpress	Foxfire
mLibPath		libs\	fe\	ff\
cLibName		CdbkMgr.vcx	FeMgr.vcx	FfMgr.vcx
cClassName		CdbkMgr	FeMgr	FfMgr
iLastId	0			
TLastUpdt	09/22/95 11:29:34 AM	08/22/95 11:29:34 AM	08/22/95 11:29:34 AM	08/22/95 11:29:34 AM

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, and 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 has a unique ID of 422, a matching record in the Micromega table that also had a unique ID of 422, and so on.


We're developing in Visual FoxPro 05.00.00.0402 for Windows (E:\AI\DMW\SOURCE)
File Edit View Tools Program Table Window Help Apps Dev

It

Objectid	
Parentid	
Objecttype	Table
Objectname	league
Property	Memo
Code	memo
Riinfo	
User	Memo

It.user

[DBCXID: 422]

Cdbkmeta

Mpath	Memo
Cdbcname	IT
Iid	422
Crectype	C
Ccursor	league
Ccursor2	
Ctagname	
Ncdxorder	
Mtagfilter	memo
Mtagexpr	memo
Ctagtype	
Lascending	
Cchildtbl	
Mparntexpr	memo
Mchildexpr	memo
Cfield	
Mexpr	memo

Cdbkmeta (e:\ai\dmw\appfiles\cdbkm Record: 111/183 Exclusive 18:03:41

Production Note: This is a BMP

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. This missing piece is the set of programs that gets all of these pieces to work together. It isn’t evident that this is missing because there is no corresponding piece for DBC. Well, actually, there is, but it’s not visible - it’s part of the VFP engine inside VFP.EXE. Since DBCx isn’t part of VFP, it has to be driven by a set of external programs.

The DBCX Manager Class

DBCx consists of a class library, DBCXMGR.VCX, that contains the Manager Class, METAMGR. Instantiating an object from METAMGR, like so:

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

It then looks for DBCXREG.DBF, and creates member objects that point to each installed 3rd Party class. It compiles a list of properties from each of the classes and waits for any requests from the applications that are in use.

When the oMetaMgr is created, the registry table is read, each of the objects listed in the registry table are created and each of the tables associated with an extension are opened. Now your application is ready to take advantage of DBCX.

In other words, 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 so:

```
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
```

Let's look behind the scenes at what is being done, and what you would do in order manually access meta data through DBCx. There are two basic steps. The first is to, 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, let's suppose we want to get the long descriptive name and the status bar text for a field. We would get the ID like so:

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

where "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 we have the ID, we can fetch one or more properties:

```
m.lcNaFieldLong = oMeta.DBCXGetProp('SDTmCaption', m.liIDOfField)
m.lcToolTipText = oMeta.DBCXGetProp('CBmToolTip', m.liIDOfField)
```

Since we are getting these values from the DBC and the DBC extended tables on the fly, we 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’ discussions is that you still have to spend all this time trying to figure out how to get it to work with your stuff. In order to try 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.

The three things you have to know are

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 just 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 need their own copy of DBCXREG.DBF and CDBKMETA.DBF. I put these into the APPFILES directory of an application - the place where application-specific data goes. (This is the place where my user file, error log, custom reports, and so on go. 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 gets handled in an application. You may have a start up program in which you instantiate your classes, such as oApp, oLibrary, and so on. I do too, but I quickly found that this is not where oMeta should be created. The reason is that oMeta needs to know about a database, and early on in our startup, we haven’t handled data yet. Thus, we simply initialize oMeta in the start up 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 are going
* to instantiate it later - so it's scoped properly
private oMeta
oMeta=.f.
wait window nowait "Setting up libraries..."
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
```

Now we are 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 two 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. Here we get the ID:

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

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

```
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 look at a different way to do this. After all, this 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 set automatically, right? You could combine both of these 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:

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

So that's the mechanism for using DBCx to extend the VFP 5.0 database container.

What's Coming Up

The architecture you've seen was conceived several years ago. Since then, Visual FoxPro and the development environments it's been used in have changed. A number of real-world installations have also provided experience that pointed out assumptions in the original model that have proved to be inaccurate, shortcomings in the architecture, and new requirements that hadn't been forseen during development of the first version.

As of this writing, a new version of DBCX is under consideration. It is too early to commit to what enhancements or new features will be created, or even if a new version will be built, but by September, these issues should be available for, at least, limited public discussion.

Hentzenwerke Corporation

735 North Water Street

Milwaukee WI 53202

USA

414.224.7654

414.224.7650 fax

whil@compuserve.com

whil@hentzenwerke.com

www.hentzenwerke.com