Understanding DBCX 2

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Albert Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web Site: www.stonefield.com

Overview

DBCX is a public domain data dictionary extension manager for Visual FoxPro. A colloboration between leading industry vendors Stonefield Systems Group, F1 Technologies, Flash Creative Management, and Micromega Systems, DBCX provides a core set of functions that allows third party products to effectively share extended data dictionary information. DBCX was recently rewritten to add speed and flexibility. This session concentrates on the design and implementation of DBCX so you can understand what goes on "under the hood" in products such as Visual FoxExpress and Stonefield Database Toolkit.

What is a Data Dictionary and Why is it Needed?

Until Visual FoxPro (VFP), xBase languages, including FoxPro 2.x, did not include a data dictionary. A data dictionary is a repository of information about databases. At a minimum, a data dictionary provides the information about tables and their indexes: field name, size, type, tag name, expression, filter, etc. In xBase, this information is stored in the headers of DBF and CDX files, so if these headers become corrupted, the only source of information about their structures is lost.

VFP implements a data dictionary by adding a database container (DBC) to store additional attributes of tables, such as field captions, field and table rules, default values, etc. However, surprising as this may seem, the DBC still doesn't contain structural information, which means you cannot use information in the DBC to create or update tables, nor recreate their indexes. In addition, many third party developers and consultants see a need to add their own attributes (such as the caption for grid column headers) to the standard ones supported by VFP.

One of the main advantages of a data dictionary is to use the values of its properties at run-time. This type of data dictionary is referred to as "active", and it allows you to create data-driven applications.

What's the big deal about creating a data-driven application? The main reason is maintainability. A data-driven application uses the data dictionary as its source of information about how to do its job, rather than hard-coding the necessary tasks. When the data dictionary changes, the application changes the way it works automatically without having to rewrite any code.

Here's an example. Say you want to create a table export module that allows the user to select which table to export. Of course, you don't want them to choose just any table: system tables (like next available key values) are hidden from the user and others (such as lookups tables) wouldn't make sense to export. You could have a hard-coded method that added just specific tables to the list displayed to the user, but then you couldn't reuse the export module in other applications. Instead, using a data dictionary, you could have an "Export" property for each table, and only put those tables for which this property is True into the list. This module is now generic and data-driven; it reads the information from the data dictionary at run-time.

By making your applications data-driven, you make them much more maintainable because you only need to change meta data in one place, the data dictionary.

DBCX

In the FoxPro 2.x world, each third party vendor created their own data dictionary to store additional properties about their data. This lead to duplicated meta data so integrating products from several vendors required extra work. To prevent this problem from occurring with VFP, several leading third party developers collaborated to created a standard data dictionary extension scheme. The following companies were involved in this effort:

- Stonefield Systems Group Inc.
- F1 Technologies
- Flash Creative Management
- MicroMega Systems

This standard scheme is called DBCX. DBCX was placed into the public domain so any third party developer can use it.

DBCX allows multiple products to enhance the VFP DBC without "stepping on each other's toes". It allows each developer to decide where and how the extended attributes will be stored by making extensive use of Visual FoxPro's object oriented capabilities and polymorphism.

In addition to these companies, other companies can use DBCX to add extensions to the VFP data dictionary for their own tools. The advantage of using DBCX is that these products can all work together without having to maintain separate data dictionary extensions.

The DBCX Model

The model for DBCX starts with the idea that every application has a registry table which contains a list of third party products that require their own extended data dictionary attributes and are registered for use with this application. Of course, multiple applications can use the same registry table if they all use the same suite of third party products. An unlimited number of third party products can be added to the registry table and can work together.

A DBCX manager class is used as a common interface to all registered third party classes. Anything requiring access to an extended attribute will access the attribute through the manager class, which automatically accesses the proper third party class managing that attribute. Thus, the DBCX manager is the only item that will directly access the extended attributes. The developer need not care what product manages the extended attributes, or even how they're stored.

The Registry Table

The registry table is called DBCXREG.DBF (although since this name is stored as a property of the DBCX manager class, it could have a different name if necessary). There is one copy of this table per application, although it could also be shared between applications if necessary. DBCXREG contains one record for each manager that manages the database extensions for an application. You can have as many copies of DBCXREG as you need, but can only use one for any one instance of the DBCX manager class.

The structure of DBCXREG.DBF is shown in Table 1, and its indexes are shown in Table 2.

Table 1. Structure of DBCXREG.DBF.

Field	Type	Size	Purpose
cProdName	C	40	The name of the product the extension manager is associated with.
mLibPath	M	4	The path to the extension manager's class library, relative to the location of DBCXREG.DBF.
cLibName	C	12	The name of the extension manager's class library.
cClassName	C	30	The name of the class in the class library.
cObjName	C	30	The name of the object to create when the extension manager's class is instantiated. If this field is empty, the default is "o" + cClassName.
lLastID	I	4	The last DBCX ID issued (only used in the system record; see below).
tLastUpdt	T	8	The date and time the record was last updated.

Table 2. Indexes for DBCXREG.DBF.

Tag Name	Index Expression	Type
Deleted	DELETED()	Regular
cProdName	UPPER(cProdName)	Candidate

There are two types of records in DBCXREG:

- The first record is the "system record" (the cProdName field contains "SYSTEM RECORD"). Its purpose is to contain the next available ID number for database objects (the iLastID field). This value is not specific to a particular DBC; all DBCs managed by a particular set of meta data tables share this value, so each record in each DBC has a unique DBCX ID. Note: DBCX 2 doesn't use DBCX IDs; this is provided for backward compatibility with DBCX 1 managers.
- The Core Manager record manages the common "Core" properties. It is automatically registered by the DBCX manager class.
- Other managers each have their own records in DBCXREG if you are using the products that provide these managers. The following table shows an example of the contents of DBCXREG when the Core Properties, Stonefield Database Toolkit, and Visual FoxExpress managers are registered.

Table 3. Example of DBCXREG Contents.

Field Name	Record #1	Record #2	Record #3	Record #4
cProdName	SYSTEM RECORD	Core Manager	FoxExpress	Stonefield Database Toolkit
mLibPath		..\..\VFEFRAME\LIBS\	..\..\VFEFRAME\LIBS\	\STONEFIELD\SDT\SOURCE\
cLibName		DBCXMGR.VCX	CVFEMGR.VCX	SDT.VCX
cClassName		CoreMgr	cVFEManager	SDTMgr
cObjName			oVFEManager	
iLastId	157	0	0	0
tLastUpdt	09/22/98 11:29:34 AM	08/22/98 11:29:34 AM	08/22/98 11:29:34 AM	08/22/98 11:29:34 AM

When you instantiate the DBCX manager class, you can specify that it should create a copy of DBCXREG.DBF (if it doesn't already exist for the current application) and add a record for the Core Properties manager.

The DBCX Manager Class

DBCXMgr is the name of the DBCX manager class. This class is defined in DBCXMGR.VCX. You instantiate it like any other class. Here's an example:

```
set classlib to DBCXMGR
oMeta = createobject('DBCXMgr')
```

The easiest way to think of DBCXMgr is as a conductor. Many of the methods in DBCXMgr merely call methods in the "real" extension managers.

When DBCXMgr is instantiated, its Init() method opens DBCXREG.DBF (actually, the name of the table stored in the cRegistryName property, which is DBCXREG.DBF by default). It looks in DBCXREG for all installed managers, and uses AddObject() to add an instance of each manager to itself. The Init() of each extension manager is fired as it's instantiated and it generally opens its own meta data tables (DBCXMgr has no tables of its own other than DBCXREG). DBCXMgr.Init() also calls a method in each manager that adds all of the properties maintained by that manager (basically, a list of the fields in its extension table preceded by a unique prefix used by the manager) to a cursor called DBCXPROPS. Thus, after its Init() is done, oMeta will contain one object for each registered manager, and the following tables will be open: the DBCX registry table, a cursor containing all the properties

maintained by all the managers, and one table (at least) for each manager. DBCXMgr use a private datasession, so these tables aren't directly visible to other forms or objects.

DBCXMgr can be instantiated with three parameters passed to its Init() method. The first should either be True or False, depending on whether you want "debug" mode turned on. Debug mode displays useful debugging messages when something goes wrong, but isn't really suitable for a production environment. The second parameter is the path to the DBCXREG table. This is required if DBCXREG isn't in your VFP path or current directory. The third parameter should be True if you want DBCXMgr to automatically create the DBCXREG table if it doesn't exist and add the CoreMgr record to it. Here's an example of instantiating DBCXMgr with debug mode turned off and telling it to look in the METADATA subdirectory for the meta data tables:

```
oMeta = createobject('DBCXMgr', .F., 'METADATA')
```

See the DBCX documentation file, DBCXREF.DOC, for the complete list of methods and properties in DBCXMgr.

Extension Manager Classes

In addition to DBCXMgr, DBCXMGR.VCX contains the BaseMgr class. BaseMgr is never instantiated directly, but is instead subclassed to create extension managers. BaseMgr contains the minimum properties and methods necessary for an extension manager. A manager subclass will probably add properties and methods to the base set, and may even override some of the base methods if necessary. We'll look at the Core Properties manager class, which is subclassed from BaseMgr, in a moment.

See the DBCX documentation file, DBCXREF.DOC, for the complete list of methods and properties in BaseMgr.

How DBCX Extends a Database

DBCXMgr itself has no extended attributes for a DBC. Instead, each extension manager maintains a set of attributes, each being stored in a field in the manager's meta data table. Each record in the meta data table is linked to the appropriate record in the DBC through the database name, record type, and object name.

You can find the value of a given extended attribute for any database object by calling the DBCXGetProp() method in DBCXMgr. You specify the name and type of the object (similar to how you specify it to the VFP DBGETPROP() function) and the name of the property you wish to obtain. The property name can either be the "long" name of the property or the prefix for a given manager and the name of the field the property is stored in. For example, to get the Caption property, which is stored in the cCaption field of the Core Properties manager's table (which has a prefix of CB), for the CUSTOMER table, use either:

```
? oMeta.DBCXGetProp('customer', 'Table', 'Caption')
```

or:

```
? oMeta.DBCXGetProp('customer', 'Table', 'CBcCaption')
```

DBCXGetProp() looks in the DBCXPROPS cursor to see which of its manager objects maintains that property, then calls the DBCXGetProp() method of that manager to do the actual work. Typically, a specific manager won't override the BaseMgr's DBCXGetProp() method, because it's pretty simple:

- Find the record in the meta data table for the object type and name.
- Strip the prefix off the property name to obtain the name of the field containing the desired attribute.
- Return the contents of that field.

Once a DBCXMgr method has been called specifying an object name and type, you can call other methods for the same object without having to specify the object name and type each time. For example, to change the Caption property for the same table, use:

```
oMeta.DBCXSetProp('Caption', 'Customer Table')
```

To create a new property, use the DBCXCreateProp() method. This method requires the name of the property (including the manager prefix), the long name, the name of the manager object to add the property to (the manager object name is usually "o" plus the name of the manager class defined in the registry, but this name can be overridden by the cObjName field in DBCXREG), and optionally the property type (the default is Memo), size (the default is 10), and number of decimals (the default is 0). Here are a couple of examples:

```
oMeta.DBCXCreateProp('CBMyNewProp', 'oCoreMgr',  'MyNewPropertyName')
oMeta.DBCXCreateProp('SDTMyTestProp', 'oSDTMgr', 'MyTestPropertyName', 'C', 2)
```

The first example creates a Memo field called MyNewProp in the Core Properties manager's meta data table, and the second creates a 2 byte Character field called MyTestProp that the SDT manager handles.

The Core Properties Manager

The Core Properties manager (the class CoreMgr contained in DBCXMGR.VCX) provides the "common" set of extended attributes. It defines extensions for the structural information of tables and indexes, as well as some other attributes. It stores its extensions in a table called COREMETA.DBF. Since CoreMgr is considered the "common" manager, third party developers should not duplicate the attributes it maintains, but should maintain their own.

The Core Properties manager uses "CB" as its prefix, so when you want to use DBCXGetProp() or DBCXSetProp() to get or change the value of a CoreMgr property, you can optionally specify the field name with "CB" as the prefix. For example, use "CBcCaption" to get the caption for a table.

Table 4 shows the structure of COREMETA.DBF and Table 5 shows its indexes.

Table 4. COREMETA.DBF Structure.

Field	Type	Size	Purpose	Used For
iID	I	4	This will be an incremented value. The last value used will be stored in the DBCX registry table.	All
cDBCName	C	119	The name of the database container (without a path or extension) associated with a data item. This field will be empty if the data item is not associated with a database container, such as a free table.	All
cRecType	C	1	A letter that describes what the record represents. The following are the possible values: C = Connection D = Database F = Field I = Index R = Relation T = Table V = View U = User-defined object, such as a virtual field	All
cObjectNam	C	120	The name of the data object.	All
mPath	M	4	The path to the object if it isn't associated with a database container.	All
nCodePage	N	5	The code page for the table.	Table

Field	Type	Size	Purpose	Used For
nBlockSize	N	5	The block size for the memo file for the table.	Table
cCaption	C	128	The caption for the object.	All
mTagFilter	M	4	The expression used if the index tag is a filtered index.	Index
mTagExpr	M	4	The index expression.	Index
cTagType	C	1	The type of index. The values are as follows: C = Candidate P = Primary U = Unique R = Regular	Index
cCollate	C	10	The collate sequence for the index.	Index
lAscending	L	1	The default order of an index tag. This will be True for ascending and False for descending.	Index
mExpr	M	4	The expression for a virtual field.	Field
nField	N	3	The field number in a table.	Field
cType	C	1	The data type for a field. Valid values are: C = Character or Character Binary D = Date T = Datetime L = Logical M = Memo or Memo Binary G = General Y = Currency N = Numeric B = Double F = Float P = Picture I = Integer	Field
lBinary	L	1	If the field type is C or M, this will indicate if the data is character binary or memo binary data that you want to maintain without change across code pages.	Field
nSize	N	3	The size of a field.	Field
nDecimals	N	3	The number of decimal places in a numeric field.	Field
lNull	L	1	This will be True if a field can contain .NULL. values.	Field
mFormat	M	4	The Format for the field.	Field
mInputMask	M	4	The InputMask for the field.	Field
mNotes	M	4	Stores any additional notes that may be necessary.	All
mComment	M	4	Comments about the object.	All

Field	Type	Size	Purpose	Used For
tLastMod	T	8	The DateTime of the last modification.	All

Table 5. Indexes for COREMETA.DBF

Tag Name	Index Expression	Type
Deleted	DELETED()	Regular
iId	iId	Regular
ObjectName	UPPER(cDBCName + cRecType + cObjectNam)	Regular

Using DBCX

This section describes how to use DBCX from the Command window or programs and how VFE 98 uses DBCX.

Using DBCX From the Command Window or Programs

As we discussed earlier, DBCXMgr can be instantiated with up to three parameters. The first is whether "debug" mode should be used or not, the second is the directory where the meta data tables can be found, and the third is whether the meta data tables should be created if they're not found. Here's an example:

```
set classlib to DBCXMGR
oMeta = createobject('DBCXMgr', .T., '', .T.)
```

In this case, DBCXMgr will be in debug mode, the meta data tables will be located in the current directory, and DBCXMgr will create them if they're not found. If DBCXMgr has to create the meta data tables, it creates DBCXREG.DBF, automatically registers CoreMgr by adding a record for it to DBCXREG, then instantiates CoreMgr and calls its CreateDBCXMeta method to create COREMETA.DBF. Note that creating the meta data tables doesn't populate them with meta data about database objects; we'll get to that in a minute.

Since DBCXMgr is in a private datasession, it doesn't "see" the databases used by your application, so you must do one of two things: specify the database name every time you specify an object name to a DBCX method (which would be a pain) or use the SetDatabase() method to tell DBCX the default database to use. Here's an example that opens the TESTDATA database that comes with VFP and tells DBCXMgr we're using it:

```
open database (_SAMPLES + 'DATA\TESTDATA')
oMeta.SetDatabase(dbc())
```

To create meta data for the objects in the database, use the Validate() method. If you want to see the progress of Validate(), set the lShowStatus property to .T. Validate() can accept object name and type parameters to just validate a single object; passing no parameters tells it to validate the entire default database.

```
oMeta.lShowStatus = .T.
oMeta.Validate()
```

Since free tables don't belong to a database, they have to be added to the meta data separately. Here's an example that creates a free table and meta data for it. Notice the leading "!" in the call to Validate() so we indicate this table isn't part of the default database; use this syntax for all method calls for free tables.

```
create table TEST free (FIELD1 C(10), FIELD2 C(10))
index on FIELD1 tag FIELD1
index on FIELD2 tag FIELD2
oMeta.Validate('!test', 'Table')
```

Let's see how to set and get extended properties. Although a default caption is assigned to all non-field objects (field captions are stored in the DBC, so there's no need to store them in an extended property), a more appropriate caption may be desired. Also, since a free table doesn't have database properties, we'll want to assign field captions and comments as extended properties. If you use several calls in a row for the same object, you don't have to specify the object name or type each time; DBCX will use the same values as the previous call.

```
oMeta.DBCXSetProp('customer.company', 'Index', 'Caption', 'Company Name')
? oMeta.DBCXGetProp('Caption')
oMeta.DBCXSetProp('!test.field1', 'Field', 'Caption', 'First Field')
oMeta.DBCXSetProp('Comment', 'First Field Comment')
oMeta.DBCXSetProp('!test.field2', 'Field', 'Caption', 'Second Field')
oMeta.DBCXSetProp('Comment', 'Second Field Comment')
```

Calculated fields can be defined by putting the calculation expression into the mExpr field in CoreMeta (the long property name is Expression). Here's an example that defines a Total Price field for the ORDITEMS table with the calculation ORDITEM.UNIT_PRICE * ORDITEMS.QUANTITY (in other words, the total price for a line item in an order), then uses the expression in a browse window. Note the use of ISNULL() on the return value of DBCXGetProp() to see if an object exists in the meta data, and AddRow() to create a new meta data record.

```
if isnull(oMeta.DBCXGetProp('orditems.total_price', 'User', 'Expression'))
   oMeta.AddRow('orditems.total_price', 'User')
   oMeta.DBCXSetProp('Expression', 'orditems.unit_price * orditems.quantity')
   oMeta.DBCXSetProp('Caption', 'Total Price')
   oMeta.DBCXSetProp('Comment', 'A calculated field of the total price')
endif isnull(oMeta.DBCXGetProp(...
lcExpr = oMeta.DBCXGetProp('Expression')
use ORDITEMS
browse fields ORDER_ID, LINE_NO, UNIT_PRICE, QUANTITY, TOTAL_PRICE = &lcExpr
```

New properties can be created using DBCXCreateProp(). You specify the name of the property (manager prefix and field name), the name of the manager maintaining this property, the long name of the property, and its data type and size (optional for those data types that have a fixed size such as Date, Logical, and Currency). The following code creates a Select (can the index be selected by the user) extended property for indexes and set it to .T. for some but not all indexes.

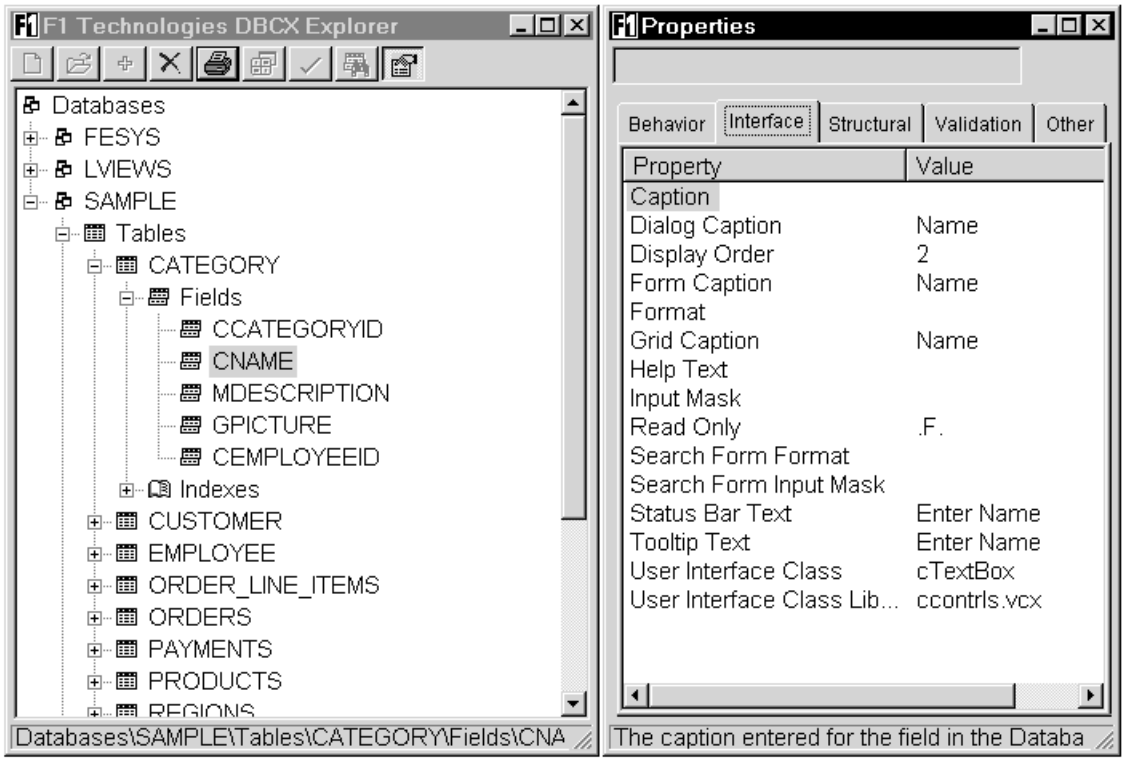
```
oMeta.DBCXCreateProp('CBlSelect', 'oCoreMgr', 'Select', 'L')
oMeta.DBCXSetProp('customer.company', 'Index', 'Select', .T.)
oMeta.DBCXSetProp('customer.contact', 'Index', 'Select', .T.)
oMeta.DBCXSetProp('customer.postalcode', 'Index', 'Select', .T.)
oMeta.DBCXSetProp('customer.country', 'Index', 'Select', .T.)
```

A new method in DBCX 2 called DBCXGetAllObjects() creates an array of all objects with a certain property set to a specified value. By default, this method just puts the name of matching objects into a one-dimensional array, but you can specify another property (most commonly Caption) to put into the first column of a two-dimensional array (the second column is the object name). This method is ideal for creating an array used as the RowSource for a combobox or listbox in which the user can select an object from the list of matching object. The following code, for example, gets the name and caption of all indexes from the CUSTOMER table that have the Select property set to .T.; this could then be displayed to the user in a combobox so they can choose the sort order for a report or data entry form.

```
dimension laIndexes[1]
lnObjects = oMeta.DBCXGetAllObjects('index customer', @laIndexes, 'Caption', ;
   'Select', .T.)
? ltrim(str(lnObjects)) + ' indexes selectable for CUSTOMER'
display memory like laIndexes
```

How VFE 98 Uses DBCX

VFE provides a new tool called DBCX Explorer that provides an interface to the DBCX properties it uses.



The cGrid class (in CCONTRLS.VCX) has an AddFieldColumn method for adding a column to the grid for a particular field. This method uses DBCX meta data for the specified field in the following ways:

- The VFEcGrdCaptn property is used as the Caption for the column header.
- The CBnSize property is used to determine the width of the column.
- The VFEcObjType property specifies the class used for the ActiveControl of the column.
- The VFEcQFindTag property is used as the tag for the "quick find" function for the field.

The cAbstractDataItem class (in CDATA.VCX) has an abstract DBCXGetProp method that wraps DBCXMgr's DBCXGetProp method. This method is specified in each subclass (for example, cCursor and cField) to get the desired property for the specific data element. Calling DBCX methods allows these classes to be data-driven; changing the meta data for a table or field changes the behavior of the class without having to modify any code. Here are some of the ways these classes use DBCX meta data:

- Several of the property Access methods look up and return the appropriate DBCX value. For example, cField's DefinedSize_Access returns the CBnSize property for the field while Type_Access returns CBcType.
- The CheckRequired(), CheckMin(), CheckMax(),and CheckIsInList() methods ensure a value was entered for the data item if required (the VFElRequired property) and compare the value against defined minimum (VFEmRangeLo), maximum (VFEmRangeHi), and enumerated (VFEuListValues) values.
- The SetPropFromDBCX() method is interesting: it adds a new property (using AddProperty) to the object and sets its value to the specified DBCX property. This method is called from the CheckRequired(), CheckMin(), CheckMax(),and CheckIsInList() methods mentioned above so DBCXGetProp() is only called the first time it's used; after that, the desired value is assumed to be in newly created property. For example, CheckRequired() calls SetPropFromDBCX() to add a new lRequired property and set its value to the VFElRequired DBCX property the first time it's called. Subsequent calls to CheckRequired() just check the

value of the lRequired property. This provides a performance boost because testing the value of a property is obviously faster than querying DBCX each time the value must be checked.

- The DecorateObject() method sets the values of various properties of a specified object to the corresponding DBCX properties of the data item. This method is used to set the properties of a control bound to the data item to the appropriate values. For example, the cSecurityName property is set to the VFEcSecurity property, ToolTipText is set to VFEmToolTip, and ReadOnly is set to VFElReadOnly.

The cQueryForm class (in CQUERY.VCX) uses several DBCX properties to determine its behavior. The VFElSearch property is used to determine if a data item should appear or not, the VFEcDlgCaption is the caption to display for the item, and VFEnDispOrder is the order to display the items in.

Summary

DBCX helps us get away from the "minefield" of FoxPro 2.x: each third party product having its own incompatible data dictionary. DBCX allows multiple products to enhance the VFP DBC, and provides a mechanism for these products to work together without having to maintain separate data dictionary extensions. Several companies use DBCX to add extensions to the VFP data dictionary for their own tools: Visual FoxExpress and Stonefield Database Toolkit. It also provides a mechanism to add additional attributes to existing managers so you don't need to go to the effort of creating your own manager.