Extending the Visual FoxPro Database Container with EDC

By Whil Hentzen

Trying to write about EDC in a page or two presents me with the same challenge that folk who will be doing reviews of Visual FoxPro in the same space. I recall one recent article that spanned three pages and barely got past VFP's Wizards. So where do we start - and where do we end?

As described in the May issue of FoxTalk, VFP presents Fox developers with their first integrated, active data dictionary. However, it's not a complete data dictionary, and has purposely been architected with the intent that third party developers would create extensions to handle special needs. Two methodologies have been under development during Visual FoxPro's beta cycle, one being the DBCX approach that's a result of the Neon/Micromega/Flash collaboration and that was described briefly in the May article. The other approach, EDC, has been developed by Tom Rettig and it's this month's Cool Tool.

EDC is an acronym for "Extended Database Container" and is one of the "TRUE" (Tom Rettig's Utility Extensions) utilities that Tom has developed and put in the public domain (We'll cover other TRUE utilities, as well as an in-depth look at DBCX, in future columns.). As many of you are aware, a data dictionary is one of those mechanisms that draws developers like light attracts moths, and many will be tempted to write their own data dictionary extensions. By placing EDC in the public domain, Tom hopes to gain critical mass for the tool and further development by providing a common foundation shared amongst many developers. And lest you are thinking that this puts DBCX out of the picture, it's still very much in the plans for both approaches to co-exist peacefully and be able to talk to each other.

Getting Started with a Class Library

Lets talk about what EDC is. Many of you are still just getting into VFP and your head is probably spinning with all the new terminology and tools and so on. We'll start slow and get you used to the idea of using a class library at the same time that we introduce EDC. Those of you who are comfortable with VFP's object model and syntax can read really fast for the next few paragraphs.

The EDC utility consists primarily of one file, EDCLIB.PRG, that is used to create a class library that contains methods we can access to extend Visual FoxPro's DBC. And already, some of you are getting a funny feeling in the pit of your stomach. Think of it this way - a class library is similar in capability to a function library (it's actually a lot more, but this is a start.)

Note: In all of the commands that follow, I've included the extension of the filename for clarity, so you can see when I'm referring to the PRG or the VCX or whatever. You don't need to include the extension in real life.

With a procedure library, you issue the command

set procedure to MYPROCS.PRG

and then you can call a procedure named SomeFunc that's in MYPROCS:

m.nX = SomeFunc("some_parm")

Same idea holds with EDC. However, instead of just opening a procedure file, we create an object that contains methods (procedures) and variables (properties.)

```
set procedure to EDCLIB.PRG
oEDC = createobject("EDC")
```

Just as you can call functions from an open proc file, you can call methods from a class library. We do this via the syntax:

m.nX = oEDC.xxx()

where xxx is an EDC method that returns the value m.nX. There are a bunch of methods in EDC, and some of them take parameters. So if we wanted to execute the Herman method in EDC, we'd go

oEDC.herman()

and if we wanted to execute the Herman method with a parameter of "TODAY", we'd go

oEDC.herman("today")

Now, a method doesn't have to return a value, but all the methods in EDC do, so if we wanted to see what the Herman method returned, we'd use the syntax

m.uX = oEDC.herman("today')

I used a "u" because, as of this writing, we don't know what type of value the Herman method returns. And, by the way, you *do* know that I'm making the Herman method up, right?

Creating an EDC Class Library

As we've seen, the EDC class definitions are stored in EDCLIB.PRG. These class definitions allow us to instantiate an EDC object and name an object reference like oEDC. We can then access the EDC methods and properties using the object reference.

However, there's another format in which class definitions can reside besides the EDCLIB.PRG program. This is called a class library and it's structure, like virtually every other metadata container, is a Visual FoxPro table. Class libraries have the extension .VCX, and each class definition (such as EDC) is contained in one record in the .VCX table. You can use another TRUE utility, PRGTOVCX, to convert EDCLIB.PRG to a .VCX, and then instantiate an EDC object with the class library. Here's what you'd do:

First, you'll use the PRGTOVCX program to create a class library from the PRG file:

```
set procedure to PRGTOVCX.PRG
oPtoV = createobject("prgtovcx")
oPtoV.convert("edclib")
```

This works very similarly to the EDC object and methods we've experimented with - once we've created the oPtoV object, we can access it's methods. And the one we want is the Convert method. This one takes the name of a PRG file and creates a class library from it. The result of the third command above is a class library named EDCLIB.VCX. Then, in order to instantiate an EDC object, we'll issue the following commands:

```
set classlib to EDCLIB.VCX
oEDC = createobject("EDC")
m.uX = oEDC.herman("today')
```

What's the difference between a class library created from a VCX and a class library created from a procedure file? None. It's up to you to decide whether you prefer the programmatic definition or the visual definition.

Structure of the EDC

OK, you should now be comfortable with what a class library is and how we can, in general terms, use it. Now let's talk about EDC and what it can do. Remember that this is a data dictionary extension. What extensions might a data dictionary need? (Those of you who are going to cheat and use the answers from the May article, to the corner with you.) Well, an

extension is any capability or functionality that is missing from the DBC and Database Designer. The rest of this article will examine three typical examples of data dictionary extensions: Tracking Last ID Used, Input Masks, and Reindexing Routines. But first, we need to understand how the EDC is constructed. Let's use the LastID property as an example.

You'll often have the need to store the "last ID used" for purposes of incrementing check numbers, invoice numbers, ID tags, and even primary keys. What we might do is keep a list of fields that have "last ID" requirements, and store the last ID value along with that field. And we'd need a function call whereby we can pass the name of a field, and have the next ID returned to us (as well as incrementing the last ID value in the data dictionary.)

EDC can do this! It has a "uIncProp" method. We pass it the name of the thing to increment, and the method returns the next value. We know how to do the method call, right?

m.nNextID = oEDC.uIncProp(<thing to increment>)

Not too bad. But we've glossed over some details. Like, how does it know what "thing to increment" is...And if we're getting a value back, it must be keeping that last value somewhere. Where is that "somewhere?"

We all know that the DBC is Visual FoxPro's version of the data dictionary, right? And that the DBC is really just a DBF file, right? Here's what EDC does: It creates a second .DBF that is linked to the DBC via the USER field in the DBC. When the EDC is created, a key value is placed in the DBC USER field, and that key is the primary key in the EDC table.

What about the stuff that's missing in the DBC? It goes into the EDC! Let's take, for example, the "last ID" value. This is what we call a "property" in Visual FoxPro - a property is just a variable. We're going to store the "last id" value in the EDC table, and then when we use the uIncProp method of oEDC, it will get that value from the EDC, get the value, bump it by one, store the new value back into the EDC, and then return the incremented value. The important thing to understand is that the EDC table holds the values, such as a last id used, and the EDC class library contains methods that allow us to painlessly access and manipulate those values.

Now let's take a look at what's happening under the hood. It's pretty obvious that we're just going to add fields to the EDC to hold each of these properties, right? We'll add a field for the "Last ID" property, another field for the "InputMask" property, a third for the "IndexExpression" property, and so on. At first blush, this sounds like a great idea, but it's got a small problem and a serious problem. The small problem is that we're now limited to 253 properties, since Visual FoxPro can only have 255 fields in a table, and the EDC requires two for it's own use. The serious problem is one of tunnel vision.

Suppose we had created a bunch of properties for our own application, and stuck them into the EDC as separate fields. Sometime later, we decide to hook a third party product into ours. Since one of the purposes of EDC is to eliminate multiple data dictionaries and extensions, it's could be possible that this third party tool is also using EDC to handle it's extended properties. And now we have the potential for collisions - what if their extended properties were named the same as ours? Crunch! And now this 253 property limit becomes a little more serious, doesn't it, since it doesn't take a real stretch of the imagination to see the number of combined properties for three or four third party products going past 253.

So what we're going to do, or rather, what Tom has done, is use a single field in the EDC for each set of extensions. In other words, an "extension" is a collection of one or more properties. If we decide to use several third party products, we'd simply add a field to the EDC for each third party product extension. For the rest of this discussion, we're going to use an EDC with multiple extensions for various products, but only access one of them.

So now we have this EDC table, and it's got the following fields - cUniqueID, mEDCObject, CAPCON, FOXEXPRESS, FOXFIRE, FOXZEN, STONEFIELD and TRO,. The first two belong to the EDC, while the rest are extensions for specific products. (We'll discuss how those fields get into the EDC table in a minute.)

Since the EDC is a table, we can have records in it, right? Now let's say that we've got a LastID property for one of the extensions. Since it's pretty likely that we could have more than one field in our system (and thus, in our data dictionary) that requires a LastID property, we'll need to store multiple LastID values. Since each of these values would map to a specific field in the database (each field in the database is in a separate record in the DBC), we'll use separate records in the EDC to hold each LastID value.

Then, all we need to do is pass the uIncProp method the name of the table/field combination (and the type of object that our property is attached to, and the name of the property), and this uniquely identifies the record in the DBC. With the DBC record in hand, we'll look in the USER field to find the foreign key that links the DBC to the EDC, and voila! We have the record in the EDC that contains all of the properties for this table/field combination. And since we also passed the name of the property in our uIncProp method, we can select the appropriate property value from the field.

m.nNextID = oEDC.uIncProp("customer.nNoCust", "field", "LASTID")

Rather elegant, wouldn't you say? And the nice thing is that the EDC class library contains a boatload of methods that make all this either transparent or very easy. (As an added point of interest, we can even pass the name of the DBC if we're working with multiple DBCs, in order to uniquely identify a table/field combo that may be repeated in multiple DBCs. I'm just glossing over some of the nuances in order to get us started.)

[Screen shot:COOL01.TIF The DBC user field contains a key to tie the record to the EDC table. The EDC table contains a field for each extension. The extension field contains one or more properties. The EDC contains a record for each instance of a property.

Our last question deals with the issue of "How did the extension field get added to the EDC in the first place?" The EDC class has a method, lOpen, that is used to do this. You call lOpen with the name of an extension, and if the extension doesn't already exist, you will be prompted to whether or not you want to add the extension. We'll also call the lOpen method in order to open the extended field for use. This is demonstrated in the next section.

Getting Started with EDC: Last ID Used

OK, we're ready to take it from the top. You can follow along if you like:

```
set procedure to EDCLIB.PRG
oEDC = createobject("EDC")
oEDC.lOpen("FOXZEN")
oEDC.uSetProp("customer.nNoCust", "field", "LastID", 100001)
m.nNextID = oEDC.uIncProp("customer.cNoCust", "field", "LastID")
```

Let's talk through the entire process. The first two give us access to EDC and it's methods and properties. The lOpen method opens up the FOXZEN extension.

The fourth line places the value 100001 into the LastID property in a record for the customer.nNoCust field. If we wanted to initialize a LastID value for another field (either in the same table or a different table), we could use the command

oEDC.uSetProp("invoice.nNoInvoice", "field", "LastID", 1234)

The last line (in the code above a few paragraphs) increments the LastID property to 10002 in EDC, and returns that value to m.nNextID. If you open up EDC, you'll see a memo field called FOXZEN. The contents say something like :

```
~
lastid ~~ N.....95.05.16 19"49:21    10002
~
```

The 100002 is in there because there are two records in the Customer table, one with a value of 100001 and the other with a value of 100002. This memo field gets updated each you make a call to a method like uSetProp or uIncProp.

There's an awful lot of magic going on in EDC, and I won't even try to explain any of it. But it's worthwhile to go through another example so you can start to make the jump to figuring out how you might want to use it yourself. I mean, we're already trashed the need for you to write your own ID incrementing function inVisual FoxPro, right?

Input Masks

Let's take a second example of how we might extend the data dictionary. This will be the famous "Input Mask" requirement. The InputMask is a character string that goes into the Properties sheet for a textbox control. For example, an input mask for a North American phone number would be "(999) 999-9999".

Let's suppose we have a database called TRUEBORG, and it contains a table called Customer. And let's suppose our customer table has a field called cVoice that holds a phone number. In the olden days, we'd have to create a GET field on the screen, and then open up that GET to place the format of "R" and "(999) 999-9999 in the GET. And we'd either copy the GET from screen to screen, or manually enter the format and input mask into each GET that we create.

We're going to create a pair of properties in the EDC called InputMask and Format:

```
oEDC = createobject("EDC")
oEDC.lOpen("FOXZEN")
m.lx = oEDC.uSetProp("customer.cVoice", "field", "InputMask", "(999) 999-9999")
m.lx = oEDC.uSetProp("customer.cVoice", "field", "Format", "R")
```

If you open up the EDC, you'll see something like the following in the FOXZEN memo field for the customer.cVoice record:

```
~
InputMask ~~ C.....95.05.16 19"49:21   (999) 999-9999
~
Format ~~ C.....95.05.16 19"49:21   R
~
```

Now you see how we can store multiple properties for a single element in the DBC. Now we're going to create a form with the Voice field on it. We'll create the form, open the Data Environment, and add the Customer table. Then we'll drag the Voice field from the Data Environment to the form. Visual FoxPro automatically creates a textbox control. Next, open up the Properties window. You'll notice that the ControlSource for the textbox is customer.cVoice, but the Format and InputMask are both empty.

Instead of manually typing "R" and "(999) 999-9999" into the properties, we'll use calls to oEDC to find out what the InputMask and Format properties are for the customer.cVoice field.

```
- input mask:
=oEDC.uGetProp("customer.cVoice", "field", "InputMask")
- format:
=oEDC.uGetProp("customer.cVoice", "field", "Format")
```

[ss: COOL02.TIF The Input Mask property is just a reference to an EDC method.

Note that we need the equals signs in order to tell Visual FoxPro that these are function calls, not character strings. Now run the form, and see that the Voice field is formatted properly. Not bad, eh? This means that we only have to maintain the Format and InputMask properties in one place - in the EDC - instead of maintaining them all over everywhere.

This works fine. However, what if you forget to instantiate the oEDC object before you run the form? Try doing a

```
clear all
close all
do form CUSTOMER
```

and then run the form. You'll see that it bombs on you. It can't find the object reference oEDC and thus can't fill the InputMask property. This is probably bad. We should practice safe computing and provide a test in the expression. We'll use the test:

type("oEDC") = "O"

to determine whether oEDC exists or not (if it doesn't exist, then the comparison will evaluate to .F.) We can use this as the expression in an IIF function, like so:

- input mask:
= iif(type("oEDC") = "O", oEDC.uGetProp("customer.cVoice", "field", "InputMask"), "")
- format:
=iif(type("oEDC") = "O", oEDC.uGetProp("customer.cVoice", "field", "Format"), "")

[ss: COOL03.TIF We can check to see whether or not oEDC is instantiated right in the property expression.

Then, if we run the form with oEDC instantiated, we will get the phone number formatted, but if we run it without oEDC instantiated, we'll still get the phone number and the data, but it will not be formatted.

Ss: COOL04.TIF - The form runs when oEDC is instantiated...

ss: COOL05.TIF: - as well as when it is not.

But we're not done yet. Imagine that we need to place the Voice field on another form. Well, it's going to be pretty easy to drag that field from the DE onto a new form, and then change the properties for Format and InputMask right? Yes, but it could be easier.

Imagine if we used the expressions

= iif(type("oEDC") = "O", oEDC.uGetProp(this.ControlSource, "field", "InputMask"), "")
= iif(type("oEDC") = "O", oEDC.uGetProp(this.ControlSource, "field", "Format"), "")

as the Format and InputMask properties. Note that "this.ControlSource" does not have quotes around it in the Properties window - it's an expression to be evaluated when it is passed to the EDC uGetProp method, not a character string that uGetProp can directly use.

[ss: COOL06.TIF Referencing the current control's ControlSource property as the table/field value to search for in the EDC makes our work just that much easier.

We can then use this same expression to grab the Format and InputMask properties of different fields. All we'd have to do is cut and paste the expression from one Property window to another- no changing of code necessary. This is really great. But wait! (Are you getting tired of being led into false climaxes?) There's those nasty words "Cut and Paste." This has a bad sound to it, doesn't it?

Let's try this. We'll create a textbox class that has those expressions in it, and then when we create a textbox, we'll use our own textbox class and just populate the Control Source from the table in the Data Environment.

First, create a class with the CREATE CLASS command, and name it txtBaseEDC. Base it on either Visual FoxPro's textbox base class or your own textbox base class if you've already created on. Save it in a class library (the .VCX file, for those of you coming up the curve) named BASECTRL if you don't have a class library for controls yet, or use the same class library as your own textbox base class is in.

Next, after you've created your form, create a textbox from your textbox base class. There are several ways you can do this. If you've got your class library in a project, open up the Class hierarchy until the txtBaseEDC class is displayed, and then drag that item from the project to your form. You can also drag your class from the Form Control toolbar. First you need to change the Form Control toolbar to display your classes instead of Visual FoxPro's base classes (do this through the Add menu option that appears when you select the View Classes icon in the Form Control toolbar, or use the Add command button in the Controls tab of the Options dialog box. Once the toolbar has your txtBaseEDC class on it, click on the icon and then click on your form to create the control.

The third step is to tie this new textbox to the Voice field in the Customer table. Open up the Data Environment and add the Customer table if you haven't already. Then, select the Voice textbox control in the form, and bring the Properties window forward. Select the ControlSource property (on the Data tab) and then open up the combo box. All of the fields in all of the tables in the Data Environment will be available for you to select from, and you can select the cVoice field instead of trying to remember the spelling and syntax.

Notice that the Format and InputMask properties are already filled in - and they refer to the ControlSource property in order to grab the appropriate property from the EDC. Pretty painless, wouldn't you say?

COOL07.TIF: the InputMask for our EDC textbox baseclass refers to the current object's ControlSource in order to make it completely generic.

Here's the best part. When we decide we need to add another field to this form - or any other form, we use our base textbox class when placing the textbox for that field on the form. Then, after populating the ControlSource property, we're all set. This new textbox already has InputMask and Format properties, so when you run the form, the property will look for the Input Mask or Format in the EDC. If you decide you need to change the input mask for a specific field, you change it once in the EDC, and that change will automatically ripple through every form that contains that field. This is what an active data dictionary is all about!

Using EDC with a Reindexing Utility

Let's look at one more example of how we might use the EDC. Another hole in the DBC is the inability to recreate the indexes for a table based solely on the information in the DBC. Visual FoxPro does come with a utility called GENDBC - in the TOOLS\GENDBC directory - that will create a program file that contains all the commands needed to recreate an entire database. However, it will create that database from scratch - wiping out any data that you might already have in your tables, so it isn't a viable tool for our reindexing requirement.

In order to create an index from scratch, we need to know the tag name, the index expression, whether it's ascending or descending, whether it's unique, whether there's a filter expression, and whether or not the index in on the primary key. The DBC contains each index, however, so we could create an "index expression" property in the EDC and then create a record in the EDC for every DBC index record. The index expression, by the way, will be the entire command we'd use, not just the parameters that differ. (This technique is what GENDBC uses as well, by the way.) In other words, we'll place the following string in our "TagString" property:

index on upper(cName) tag ucName

instead of creating five properties, one for each component of an index. As we've seen above, there are two pieces to using the EDC - we need a mechanism to fill the EDC to begin with, and another to use the EDC during our day-to-day work. In this reindexing situation, we'll need to be able to fill the EDC TagString property with the index expressions for the existing indexes. This will typically be a developer utility. We'll also need a mechanism that the user can use to reindex his files should the indexes become corrupted. The code for the EDC population tool would look like this (Please don't complain that the following code is not complete or doesn't work - it is fleshed-out pseudo-code intended to show how the EDC methods would be intertwined in an application, not ready-to-run code. Some of the gory details have been left out and no error checking has been done. These locations are marked with the "*\\\" symbol.)

```
* SETUP_TS.PRG
*
* Populate EDC with an index expression property
* Call this with syntax like
*
* do SETUP_TS.PRG with "trueborg.dbc"
*

lparameter lcNaDBC

*
* create a table that contains two fields
*  field 1: table
```

```
*  field 2: indextag
* for all tables/indexes in the dbc
*

*\\\ make sure we can open the DBC first

*
* note this "self-join" due to the DBC table structure
*
select ;
  db1.objectname as cNaTable, ;
  db2.objectname as cNaTag ;
 from &lcNaDBC.dbc db1, &lcNaDBC db2 ;
 where db1.objectid = db2.parentid ;
  and db2.objecttype = "Index" ;
 order by db1.objectname, db2.objectname ;
 into cursor XXX

*
* set up the EDC
*
open data &lcNaDBC
set procedure to EDCLIB
oEDC = createobject("EDC")
m.lx = oEDC.lOpen("FOXZEN")

*
* run through each record of the list of tables/tags
* and insert a property value into the EDC
*
scan
 *
 *\\\ code to determine the index or alter table command
 * goes here
 *
 * the resulting command will look like
 *   "index on upper(cName) tag ucName descending"
 *
 * m.lcTagString = ...
 *
 m.lcNaTabTag = allt(cNaTable) + "." + allt(cNaTag)

 *
 * place the property value into the EDC
 *
 m.lx = oEDC.uSetProp( m.lcNaTabTag, "index", "tagstring", m.lcTagString )

endscan
```

Note that the key command here is the oEDC.uSetProp method where the fully created TagString property is placed into the EDC. The m.lcNaTabTag parameter identifies this property as belonging to the appropriate record in the DBC. Once we have populated our EDC, and kept it current, of course, we'll need a mechanism for the user to reindex his tables. This will typically go on a menu option that is access-restricted to certain personnel. The code might look something like this:

```
* REINDEX.PRG
*
* Reindex all tables in the DBC
* Call this with syntax like
*
* do REINDEX
*

lparameter lcNaDBC

*
* create a table that contains two fields
*  field 1: table
*  field 2: indextag
* for all tables/indexes in the dbc
*
* we'll place this in an array so that we
* don't get caught up with switching between tables
```

```
*

*\\\ make sure we can open the DBC first

select ;
  db1.objectname as cNaTable, ;
  db2.objectname as cNaTag ;
 from &lcNaDBC.dbc db1, &lcNaDBC db2 ;
 where db1.objectid = db2.parentid ;
  and db2.objecttype = "Index" ;
 order by db1.objectname, db2.objectname ;
 into array aTagOPlenty

*
* set up the EDC
*
open data &lcNaDBC
set procedure to EDCLIB
oEDC = createobject("EDC")
m.lx = oEDC.lOpen("FOXZEN")

*
* run through each record of the list of tables/tags
* if new table, delete the existing tags first
* then create a tag for each record in EDC
*

m.cCurTable = aTagOPlenty[1,1]
m.cOldTable = m.cCurTable
*
* start up with the first table
*
select 0
use &cOldTable exclusive
delete tag all
*
* create each tag in the array
*
for i = 1 to alen(aTagOPlenty,1)
 if aTagOPlenty[i,1] != m.cOldTable
  *
  * if this tag is for a new table, open the new table
  * (on top of the old one, thus closing it) and rid the
  * tags from this new table
  *
  sele &cOldTable
  use &cCurTable exclusive
  delete tag all
 else
  select &cCurTable
 endif
 wait window nowait "Indexing " + uppe(allt(m.cCurTable)) + " (" + aTagOPlenty[i,2]) + ")"

 *
 * get the command from the EDC for this table/index
 *
 m.lcNaTabTag = aTagOPlenty[i,1] + ".” + aTagOPlenty[i,2]
 m.cTagString = oEDC.uGetProp( m.lcNaTabTag, "index", "tagstring", m.lcTagString )

 *
 * execute the command that creates the index or alters the table
 *
 &cTagString

 m.cOldTable = m.cCurTable
 if i < alen(aTagOPlenty,1)
  *
  * get the name of the table for the next tag in the array
  *
  m.cCurTable = aTagOPlenty[i+1,1]
 endif
next
use in &cCurTable
```

Well, my editor is making those slashing gestures across his throat, so we're going to do one more thing and then call it quits. In the version of TRUE enclosed on the Companion Disk, there are two small bugs. These are the fixes:

```
EDCLIB.PRG

1. Line 156: Change
                lcRelativePath = lcRelativePath + rcEdcFileExt
   To
                lcRelativePath = lcRelativePath + "." + rcEdcFileExt

2. Line 3,161: Change
           IF ATC(ALLTRIM(tcPropertyName), ccEDC_REG_LISTFILES) > 0
   To
           IF ATC(ALLTRIM(tcPropertyName), ccEDC_REG_LISTFILES) > 0;
                 AND NOT EMPTY(lcValueString)

And, just in case you get into PrgToVcx, here are 2 fixes for that.
PrgToVcx is useful for those who want EDC in a visual class format.

PRGTOVCX:

1. Comment out the IF at line 748 and the ENDIF at line 750.  Leave 749
alone.

2. Change line 807 from LOWER(mInit) to LOWER(mValue).
```

I didn't make the changes to the programs myself in order ensure the TRUE files integrity.

Conclusion

Whew! And to think we've just scratched the surface! EDC contains over 30 exposed methods - but we've only looked at a few. We haven't covered accessing the DBC properties, handling multiple DBCs, or the Registry, for example. And there are additional issues regarding free tables, multiple extensions that overlap, and so on. We could literally fill several issues of FoxTalk with examples of the different things you could do with EDC, but hopefully this will give you a start. The documentation that comes with EDC runs over 50 pages, and that doesn't include the demo program.

Before I go, I need to say thanks to Doug Hennig, Andy Neil, Tom Rettig, and Steve Sawyer who all helped out with parts of the May article and this one.

As of this writing, Visual FoxPro is still in beta testing, and thus EDC is not available on any of the public forums. It's included on this month's companion disk, of course, and will be in one of the FoxPro libraries as soon as they are reorganized to handle the release of VFP. I'm only guessing, but I would bet that a search under "EDC", "TRUE" or "RETTIG" will turn up EDC.