Database Events in Visual FoxPro 7.0

By Doug Hennig

Overview

Database (DBC) events are an exciting new addition in Visual FoxPro 7.0. Since these events are automatically fired when anything in a database container is accessed (opened, closed, modified, deleted, etc.), they give us the ability to control what happens before and after the access. This document discusses DBC events in detail and explores some of the uses for this new technology, such as implementing table-level security and powerful developer tools.

Introduction

Like a trigger, which automatically executes procedural code when a record is inserted, updated, or deleted in a table, a DBC event automatically executes a specific routine when something is done with a member of the DBC (such as a table or view) or the DBC itself. Events are available for almost any action you can take on anything in a DBC, such as opening or closing a table or view, creating a connection, dropping a relation, validating the DBC, and so forth.

It doesn't matter whether you do something visually (for example, selecting a view in the Database Designer and pressing the Delete key) or programmatically (executing the DROP VIEW command), in a run-time or development environment, through VFP code or through ADO; the appropriate DBC event will fire.

By default, DBC events are turned off. There are two ways you can turn on DBC events for a given DBC. One is programmatically using something like:

```
dbsetprop(juststem(dbc()), 'Database', 'DBCEvents', .T.)
```

The other is with the Set Events On checkbox in the new Database Properties dialog (choose Properties from the Database menu or right-click in the Database Designer and choose Properties from the shortcut menu to display this dialog).

A DBC with events turned on is not backward-compatible with previous versions of VFP, nor can it be accessed through the VFP ODBC driver. Of course, you can always turn events back off again, but only from within VFP 7. If you want to maintain compatibility with older applications but make use of DBC events in your new versions, have your new applications explicitly turn DBC events on when they start. The Version property of a DBC is 11 if DBC events are turned on (normally it's 10).

Turning DBC events on doesn't mean anything will happen; you have to specify what events you want to handle by creating routines with the names VFP expects for DBC events. The procedure names all begin with "DBC_", followed by the name of the event (for example, "DBC_BeforeOpenTable" is the name of the procedure VFP will execute when a table is opened). The new ALANGUAGE(4) function fills an array with all possible DBC event names (although these names don't include the "DBC_" prefix); run DBCEventsList.prg (included in the sample files accompanying this document) to see the results.

DBC event code can be placed in the stored procedures of a DBC or in a separate PRG file. There are two ways you can specify a separate PRG file. One is programmatically using something like the following (where lcPRGName contains the name of an existing PRG):

```
dbsetprop(juststem(dbc()), 'Database', 'DBCEventFileName', lcPRGName)
```

The other way is using the Events File checkbox and associated Open File button in the Database Properties dialog. Doing this visually won't automatically create the PRG; you must select an existing file, even if it's empty.

Although you could type the DBC event names and parameter lists yourself, VFP can generate "template" code for you. Bring up the Database Properties dialog, check the Set Events On checkbox, then select the events you wish to handle in the Events listbox (any events that already have code will appear in bold) and click on the Edit Code button. (Neglecting the last step and clicking on OK instead simply closes the dialog without generating any template code.) An edit window for either the stored procedures of the DBC or the separate PRG file, depending on which you're using, appears with template code for each of the events you selected added. All you have to do is fill in the code to execute for each event. If you want to work on existing code for a single event or generate the template for it, double-clicking on the event will do the trick. Hold down the Ctrl and Shift keys while clicking on the Edit Code button to generate template code with a ? statement showing the event name and parameters passed.

Most actions have both "before" and "after" events. While the after events just receive notification that an action took place, the before events can actually prevent the action from occurring by returning .F. For example, if DBC_BeforeOpenTable returns .F. unconditionally, you won't be able to open any tables or views. Obviously, you'll want to be careful how you handle these events or you can make your database unusable!

How much overhead do DBC events add? Overhead.prg shows the difference in time to open a table with and without DBC_BeforeOpenTable and DBC_BeforeCloseTable event code. On my system, the difference is 5 to 6%, which isn't too bad. Obviously, putting extensive code into events will have an impact on performance.

DBC Event Descriptions

DBC events are available for the DBC itself as well as for tables, views, relations, and connections.

DBC Events for Databases

Almost everything you can do with a database has a DBC event associated with it; the only one missing is a DBC_CompileDatabase event (not that I think we necessarily need one; I'm just mentioning it for completeness).

A couple of new database-related features in VFP 7 that have nothing to do with DBC events: VALIDATE DATABASE RECOVER is now allowed in code, so you can use this dangerous but sometimes useful command in a runtime environment, and you can now use ISREADONLY() to determine if a database is read-only.

To see these events in action, run DatabaseEvents.prg.

DBC_OpenData(cDatabaseName, lExclusive, lNoupdate, lValidate)

Called after a DBC is opened. After it fires, if another DBC was current, DBC_Deactivate fires for that database (since it won't be the current database anymore). DBC_Activate for the newly opened database then fires (since that database becomes the current one). Only after DBC_Activate does DBC() return the name of the database.

If DBC_OpenData returns .F., the error "file access is denied" is triggered and the database can't be opened. Of course, be sure that you do this only under certain circumstances (for example, if the current

user isn't authorized to access the database; see the "Using DBC Events" section of this document for further discussion) or you'll be somewhat hosed since you won't be able to reopen the database even to turn that feature off! (I say "somewhat" because you can always USE the DBC as a table and modify the code directly in the Code memo of the StoredProceduresSource record, close it, and the use COMPILE DATABASE to compile the changes.)

DBC_CloseData(cDatabaseName, lAll)

Called before a DBC is closed; return .F. to prevent the DBC from being closed. DBC_Deactivate also fires (after DBC_CloseData), since the database is no longer the current database.

DBC_Activate(cDatabaseName)

Called when the DBC is made current via such means as SET DATABASE TO, after OPEN DATABASE completes, clicking on an open Database Designer window, and so on. If another DBC was current, DBC_Deactivate fires for that database first (if that database has events enabled).

DBC_Deactivate(cDatabaseName)

Called when the DBC is made non-current; return .F. to prevent this from happening (this also prevents the DBC from being closed).

DBC_BeforeAppendProc(cFileName, nCodePage, lOverwrite)

Called before the APPEND PROCEDURES command starts; return .F. to prevent the procedures from being updated.

DBC_AfterAppendProc(cFileName, nCodePage, lOverwrite)

Called after the APPEND PROCEDURES command has completed.

DBC_BeforeCopyProc(cFileName, nCodePage, lAdditive)

Called before the COPY PROCEDURES operation starts; return .F. to prevent procedures from being copied.

DBC_AfterCopyProc(cFileName, nCodePage, lAdditive)

Called after the COPY PROCEDURES operation has completed.

DBC_BeforeModifyProc()

Called before stored procedures are modified; return .F. to prevent the editor window from appearing.

DBC_AfterModifyProc(lChanged)

Called after stored procedures have been modified.

DBC_BeforeDBGetProp(cName, cType, cProperty)

Called before DBGETPROP() executes. Returning .F. prevents the property's value from being read, in which case DBGETPROP() returns .NULL.

DBC_AfterDBGetProp(cName, cType, cProperty)

Called after DBGETPROP() completes but before the value is actually returned.

DBC_BeforeDBSetProp(cName, cType, cProperty, ePropertyValue)

Called before DBSETPROP() executes; return .F. to prevent the property value from being changed.

DBC_AfterDBSetProp(cName, cType, cProperty, ePropertyValue)

Called after DBSETPROP() completes.

DBC_BeforeValidateData(lRecover, lNoConsole, lPrint, lFile, cFileName)

Called before VALIDATE DATABASE executes; return .F. to prevent the DBC from being validated. Note that the cFileName parameter isn't passed if lFile is .F.

DBC_AfterValidateData(lRecover, lNoConsole, lPrint, lFile, cFileName)

Called after VALIDATE DATABASE has completed.

DBC_ModifyData(cDatabaseName, lNoWait, lNoEdit)

Called after MODIFY DATABASE is issued; return False to prevent database modifications.

DBC_PackData()

Called before PACK DATABASE executes; returning .F. from this event prevents the database from being packed, but also triggers a "file access denied" error, so be prepared to trap for this.

DBC Events for Tables

There's a DBC event associated with everything you can do structurally with a table. One thing that's a little goofy: the REMOVE TABLE and DROP TABLE commands, which do the same thing, have a different set of events. If you want to trap the removal of a table, you have to be sure to handle both sets of events. Another issue is that if you open a table with a different alias, that alias is passed for the cTableName parameter in all table events rather than the name of the table. The workaround is to use CURSORGETPROP('SourceName') to determine the real name of the table. Also, after a table has been renamed, a bug currently causes subsequent events to receive the former table name as the cTableName parameter.

To see these events in action, run TableEvents.prg.

DBC_BeforeAddTable(cTableName, cLongTableName)

Called before a free table is added to the DBC; return .F. to prevent the table from being added.

DBC_AfterAddTable(cTableName, cLongTableName)

Called after a free table is added to the DBC.

DBC_BeforeCreateTable(cTableName, cLongTableName)

Called before a table is created; return .F. to prevent table creation. Currently, a bug causes the cLongTableName parameter to receive the same value as cTableName when the table is created in the Table Designer; it correctly receives the long name when CREATE TABLE is used.

DBC_AfterCreateTable(cTableName, cLongTableName)

Called after a table is created. This event has the same bug as the Before event.

DBC_BeforeDropTable(cTableName, lRecycle)

Called before a table is dropped; return .F. to prevent the table from being dropped.

DBC_AfterDropTable(cTableName, lRecycle)

Called after a table is dropped.

DBC_BeforeRemoveTable(cTableName, lDelete, lRecycle)

Called before a table is removed from the DBC; return .F. to prevent the table from being removed.

DBC_AfterRemoveTable(cTableName, lDelete, lRecycle)

Called after a table has been removed.

DBC_BeforeModifyTable(cTableName)

Called before a table structure is modified; return .F. to prevent modification.

DBC_AfterModifyTable(cTableName, lChanged)

Called after a table structure has been modified. You can't tell what changes are made unless you save the structural information somewhere in DBC_BeforeModifyTable (or use meta data), then compare the current structure with the saved information in DBC_AfterModifyTable.

DBC_BeforeRenameTable(cPreviousName, cNewName)

Called before a table is renamed; return .F. to prevent the table from being renamed. A bug currently prevents the method of an object called from code for this event from being executed when the table is renamed in the Table Designer (in the debugger, you can see the statement is skipped). It works correctly when the table is renamed programmatically with RENAME TABLE.

DBC_AfterRenameTable(cPreviousName, cNewName)

Called after a table has been renamed. This event has the same bug as the Before event.

DBC_BeforeOpenTable(cTableName)

Called before a table is opened; returning .F. prevents the table from being opened and triggers a "file access denied" error, which you should be prepared to trap.

DBC_AfterOpenTable(cTableName)

Called after a table is opened.

DBC_BeforeCloseTable(cTableName)

Called before a table is closed; return .F. to prevent the table from being closed (no error is triggered).

DBC_AfterCloseTable(cTableName)

Called after a table is closed.

DBC Events for Views

As with tables, anything you can do with a view has a DBC event associated with it. As you may expect, some actions with views, such as creating or opening, cause the source tables to be opened, so the DBC_BeforeOpenTable and DBC_AfterOpenTable events fire for these tables. Interestingly, these events fire even if the tables are already open, giving us a glimpse of some internal processes involved in opening the view (such as opening a second copy of the tables).

As with tables, if a view is opened with a different alias, that alias is passed for the cTableName parameter in the Open and Close events rather than the name of the view.

To see these events in action, run ViewEvents.prg.

DBC_BeforeCreateView(cViewName)

Called before a view is created; return .F. to prevent view creation.

DBC_AfterCreateView(cViewName, lRemote)

Called after a view is created.

DBC_BeforeDropView(cViewName)

Called before a view is dropped; return .F. to prevent the view from being dropped.

DBC_AfterDropView(cViewName)

Called after a view is dropped.

DBC_BeforeModifyView(cViewName)

Called before a view is modified; return .F. to prevent modification.

DBC_AfterModifyView(cViewName, lChanged)

Called after a view has been modified.

DBC_BeforeRenameView(cPreviousName, cNewName)

Called before a view is renamed; return .F. to prevent the view from being renamed.

DBC_AfterRenameView(cPreviousName, cNewName)

Called after a view has been renamed.

DBC_BeforeCreateOffline(cViewName, cPath)

Called before a view is taken offline; return .F. to prevent this from happening.

DBC_AfterCreateOffline(cViewName, cPath)

Called after a view is taken offline.

DBC_BeforeDropOffline(cViewName)

Called before a view is taken back online; return .F. to leave the view offline. A bug causes the template code to have two parameters (the second being "cPath") when in fact only one is passed.

DBC_AfterDropOffline(cViewName)

Called after a view is taken back online. As with DBC_BeforeDropOffline, the template code incorrectly has two parameters instead of one.

DBC_BeforeOpenTable(cTableName)

Called before a view is opened; returning .F. prevents the view from being opened and triggers a "file access denied" error, which you should be prepared to trap.

DBC_AfterOpenTable(cTableName)

Called after a view is opened.

DBC_BeforeCloseTable(cTableName)

Called before a view is closed; return .F. to prevent the view from being closed (no error is triggered).

DBC_AfterCloseTable(cTableName)

Called after a view is closed.

DBC Events for Relations

There are only two types of events for relations: adding and removing. When a relation is modified, it's first removed then re-added, so both the remove and add events fire.

All of these methods receive five main parameters. cRelationID is the name of the relation (which is often something like "Relation 1"), cTableName is the name of the child table, cRelatedChild is the tag for the child table (this name isn't clear and will hopefully be changed to something like cTableTag in a later version), cRelatedTable is the name of the parent table, and cRelatedTag is the tag for the parent table.

To see these events in action, run RelationEvents.prg; it modifies an existing relation and shows each of these events firing.

DBC_BeforeAddRelation(cRelationID, cTableName, cRelatedChild, cRelatedTable, cRelatedTag)

Called before a relation is added; return .F. to prevent the relation from being added.

DBC_AfterAddRelation(cRelationID, cTableName, cRelatedChild, cRelatedTable, cRelatedTag)

Called after a relation is added.

DBC_BeforeDropRelation(cRelationID, cTableName, cRelatedChild, cRelatedTable, cRelatedTag)

Called before a relation is dropped; return .F. to prevent the relation from being dropped.

DBC_AfterDropRelation(cRelationID, cTableName, cRelatedChild, cRelatedTable, cRelatedTag)

Called after a relation is dropped.

DBC Events for Connections

There are four types of events for connections: adding, removing, renaming, and modifying.

A bug prevents the method of an object called from code for all of these events from being executed when the action is done visually (that is, from functions in the Database Designer); in the debugger, you can see the statement is skipped. It works correctly when the actions are performed programmatically.

To see these events in action, first create an ODBC datasource called TASTRADE that points to the VFP sample TASTRADE database, and then run ConnectionEvents.prg.

DBC_BeforeCreateConnection()

Called before a connection is created; return .F. to prevent the connection from being created. Note that no name is passed, even if the CREATE CONNECTION command is used with a connection name.

DBC_AfterCreateConnection(cConnectionName, cDataSourceName, cUserID, cPassword, cConnectionString)

Called after a connection is created.

DBC_BeforeDeleteConnection(cConnectionName)

Called before a connection is deleted; return .F. to prevent the connection from being deleted.

DBC_AfterDeleteConnection(cConnectionName)

Called after a connection is deleted.

DBC_BeforeModifyConnection(cConnectionName)

Called before a connection is modified; return .F. to prevent modification.

DBC_AfterModifyConnection(cConnectionName, lChanged)

Called after a connection has been modified.

DBC_BeforeRenameConnection(cPreviousName, cNewName)

Called before a connection is renamed; return .F. to prevent the connection from being renamed.

DBC_AfterRenameConnection(cPreviousName, cNewName)

Called after a connection has been renamed.

Using DBC Events

OK, enough theory; what can we actually use DBC events for? There are two different kinds of things you can use them for: development tools and run-time behavior.

Development Tools

DBC events can be used in a number of tools that can make development easier and more productive. Here are some examples.

Enforcing Standards

Your organization might have standards for naming tables, views, fields, and so forth (for example, perhaps all tables in the Accounts Receivable database should start with "AR", the first character of all field names must represent the data type or be a four-letter abbreviation for the table, and so on). You may also require that the Comment property of every table, view, and field be filled in. Maybe every table should have a tag on DELETED() and a primary key on a field called ID with NEWID() as its DefaultValue property.

Rather than writing an auditing tool to check for these things after the fact, you could create DBC events that either warn if the standard isn't followed or, in the case of the standard tags, automatically add them. DBC_AfterAddTable, DBC_AfterCreateTable, DBC_AfterModifyTable, DBC_BeforeRenameTable, DBC_AfterCreateView, DBC_AfterModifyView, DBC_BeforeRenameView, DBC_AfterCreateConnection, DBC_AfterModifyConnection, and DBC_BeforeRenameConnection are all candidates for this.

Handling Renamed Objects

While changing the name of a table is easy (RENAME TABLE will do it with a single command), tracking down every place the former name was used isn't. Stored procedures, forms, reports, and PRG files can all reference the former table name. You could put code in DBC_AfterRenameTable (as well as DBC_AfterRenameView and DBC_AfterRenameConnection) to open a project, go through all files in the project, look for the former name (the cPreviousName parameter contains this), and either replace it with the new name (contained in cNewName) or at least print the location so a developer can make the changes manually. Because a database might be used by more than one project, you might have to provide a way to track which projects to process.

Handling renamed fields and indexes is trickier; because DBC_AfterModifyTable and DBC_AfterModifyView don't tell you what changes were made, you have to store the previous structure somewhere (such as in a cursor in DBC_BeforeModify events or in meta data), and then in the DBC_AfterModify events try to figure out what happened. It's not easy; a renamed field looks no different than if you delete one field and add another.

Handling New or Removed Fields

When a field is added or removed from a table, you often have to add or remove controls bound to that field in forms and reports. Similar to the previous point, DBC_AfterModifyTable and DBC_AfterModifyView could print a list of all forms and reports using that table or view so you can then modify them as required.

Team Development

When anything in the database changes (such as table or views being added, removed, or altered), DBC event code could automatically send an email to every member of the development team informing them of the changes. You could also log who changed something and when, and even prompt the user to enter a short comment about the change.

Automate FoxAudit

TakeNote Technologies (www.takenote.com) sells a great tool called FoxAudit that automatically logs all changes made to records in tables. It works by hooking into the insert, update, and delete triggers for those tables you want to audit, and recording the changes made in a log table. Although FoxAudit comes with a wizard to hook into the triggers and generate the stored procedures code to perform auditing, you have to remember to run the wizard after you create or add a table. DBC_AfterAddTable and DBC_AfterCreateTable could do this for you automatically.

Automatically Update Meta Data

DBCX is a public domain set of classes that maintain a data dictionary, or meta data, for VFP data (a copy of DBCX, including documentation, is available from the Technical Papers page of Stonefield's Web site, www.stonefield.com). Several commercial tools, including Visual FoxExpress and Stonefield Database Toolkit, use DBCX as their data dictionary manager. The main DBCX class, DBCXMgr, has a Validate method that creates or updates the meta data for the specified data object. When you add, modify, or delete a data object, you must either call this method or use the visual interface that came with your commercial tool to do this for you. Failing to update the meta data means that DBCX (and therefore the commercial tool) can't perform its job correctly.

UpdateMeta.prg shows how you can automate the process of updating DBCX meta data. Every action with an impact on meta data has DBC event code that calls the appropriate DBCX method to ensure the meta data is kept in sync with the data objects.

Some commercial tools, such as Visual ProMatrix and FoxFire, have their own data dictionaries. You could use a similar mechanism to keep those data dictionaries up-to-date as well.

A bug prevents the method of an object called from DBC_AfterRenameTable from being executed when the table is renamed in the Table Designer, or for any connection actions taken in the Database Designer, so the meta data won't be updated in these cases.

Practical Jokes

All work and no play makes Doug a dull programmer! Imagine the fun you can have with your fellow developers when you create DBC_Before events that return False in their DBCs while they're on their lunch breaks. Sit back and watch them try to alter the structure of a table, remove a table, and so on. For even more fun, make the DBC events time-specific, like, say, only between 3 and 4 p.m. on Thursdays. April 1 is a particularly good day to wreak havoc!

Run-time Behavior

DBC events can also be used in a run-time environment to provide functionality VFP developers have requested for years.

Table and Database Security

Returning .F. from DBC_BeforeOpenTable prevents a table or view from being opened. Obviously, you don't want to do this for every table and view under all conditions (otherwise the database would be rendered useless), but having table opening based on user security may make sense for some applications. TestSecurity.prg demonstrates how this might work. It's unlikely you'll want to present a login dialog to the user when the DBC is being accessed from ADO or a COM object, so you'll have to use a more complex design in that case.

As with table security, an entire database can be secured by returning .F. from DBC_OpenData.

Hacker Prevention

Although it's not common, some developers have expressed the desire to prevent someone from altering the structure of a table or view. It's probably not a malicious hack they're worried about, but likely a user who typifies the expression "a little knowledge is a dangerous thing". This user could be using a development copy of VFP or ADO. To prevent this, return .F. in DBC_BeforeModifyTable and DBC_BeforeModifyView.

To prevent someone from changing or disabling the events for the DBC (which would allow them to get around everything DBC events is providing for you), return .F. in DBC_BeforeModifyProc, DBC_BeforeAppendProc, and DBC_BeforeDBSetProp if cType is "database" and cProperty is "DBCEvents" or "DBCEventFileName". This has only limited usefulness, since someone with a development copy of VFP can USE the DBC, modify the code in the Code memo of the StoredProceduresSource record, and then close and COMPILE DATABASE. To prevent this, use a separate PRG file for the DBC event code, and build that PRG into the EXE so it can't be viewed or altered. The downside is that now the DBC can only be opened when the EXE is running, preventing its access from ADO.

To prevent someone from seeing the stored procedures code, return .F. in DBC_BeforeModifyProc and DBC_BeforeCopyProc.

The one thing missing in this scheme is that the user can USE the DBC as a table and then bypass or eliminate DBC event code. I've sent an ER to Microsoft that would have the DBC_BeforeOpenTable event fire when a DBC is opened as a table so we can prevent this from occurring.

Automatically Index Views

Some developers like to index views so SEEK can be used to locate records. The problem is that you have to create the index every time the view is opened, be sure to set the buffer mode correctly (indexes can only be created when row buffering is used), change the index creation code if the view structure changes, and so forth. Wouldn't it be nice if VFP could automatically create an index for a view whenever you opened it? DBC_AfterOpenTable can do that for you; IndexViews.prg shows an example of how to do that. Of course, this sample program uses hard-coded names and expressions; you likely will want to use meta data (such as DBCX, which supports indexes for views) to store the index definitions.

Summary

DBC events allow us to implement tools that can make our development team more productive and provide runtime features such as table security. They're a welcome addition to our VFP toolkit!

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of the award-winning Stonefield Database Toolkit (SDT). He is co-author, along with Tamar Granor and Kevin McNeish, of "What's New in Visual FoxPro 7.0", available from Hentzenwerke Publishing (www.hentzenwerke.com). He writes the monthly "Reusable Tools" column for FoxTalk. Doug has spoken at Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP).

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