Data Handling Issues, Part II

Doug Hennig

While field and table validation rules protect your tables from invalid data, they also make data entry forms harder to use. In this second of a two-part article, Doug looks at a solution to this problem. He also examines the use of multi-purpose lookup tables, and discusses how to take advantage of new data dictionary features in VFP 5.0.

In last month's column, we looked at two aspects of data handling for real world situations: working with multiple data sets and primary key issues. This month, we'll look at more aspects, including validation issues, lookup tables, and how some new data dictionary features in VFP 5.0 affect your applications.

Hassles With Validation

In FoxPro 2.x, most developers used a data buffering scheme involving memory variables. The idea was to use SCATTER MEMVAR to create a set of memory variables with the same names as the fields in a table, use GETs to edit those memory variables in a screen, and then save the changes using GATHER MEMVAR only if the user chose the Save function. VFP's built-in data buffering means we can now create forms that edit the fields in a table directly, and either save the changes using TABLEUPDATE() or cancel them using TABLEREVERT(). Unfortunately, there's one flaw in this new scheme: validation rules.

Field and table validation rules act like soldiers guarding your data. VFP will not allow any data that disobey these rules to be stored in your tables. However, these soldiers seem to be a little overzealous: there's no way to control when they fire, nor is there a way to trap the error that occurs when the rules fail. Even with table buffering turned on, VFP checks a field validation rule when the user tries to exit a control bound to that field and a table validation rule when the user tries to move to another record. If the rule fails, VFP displays either the error message you defined in the database for the rule or a generic error message. In either case, you don't have any control over the appearance of the error dialog box. You also cannot defer validation checking to a time when it's more convenient. For example, if a user enters invalid information in a field, then clicks on the Cancel button in a form, they're still going to get an error message because there's no way to suppress the validation checking.

A less restrictive problem is system-assigned primary key values. Since VFP doesn't allow you to assign a primary key value in a trigger (which is the logical place to do it), most developers do it in the Default property for the primary key field by specifying a function name. For example, the routine called NEXT_ID.PRG on the source code disk assigns the next ID for the specified table by incrementing the ID field in NEXTID.DBF. The Default property for the CUST_ID field in the CUSTOMER table is NEXT_ID('CUSTOMER'). When a new record is added to the CUSTOMER table, NEXT_ID is called by VFP to increment NEXTID.ID and assign the new value to CUST_ID. The only problem with this mechanism is that if the user decides not to add the record after all, NEXTID.ID has already been incremented. In the case of surrogate keys (keys that have no meaning other than to provide a unique value), this isn't a

problem, but if the key represents the next invoice or check number, we'll have a "hole" in the numbering scheme.

To see an example of how these problems can be a pain, DO the CUSTOMER1 form on the source code disk. This form uses row buffering with controls directly bound against the CUSTOMER table. Try to enter "NJ" into the Region field; the error dialog that appears is controlled by VFP, not the form. Try clicking on the Cancel button. You'll continue to get the same error message until you first enter a valid value (blank, SK, CA, or NY) into the field. Click on the Add button then the Cancel button. Do this several times. Notice that even though no new records have been added to the table, the next available value for CUST_ID keeps rising.

The solution to these problems is to edit an updatable view rather than the table directly in a form. Since the view won't have field validation rules (unless you explicitly define them, of course), the user can enter whatever they like into any field. However, when they click on the Save button, TABLEUPDATE() tries to write the view record to the table, at which time field and table rules are checked. If any rules fail, VFP doesn't display an error message; instead, TABLEUPDATE() returns .F., in which case you can use AERROR() to determine which rule failed and handle it appropriately.

To see an example of this, DO the CUSTOMER2 form. You'll find that you can enter NJ into the Region field and won't get an error message until you click on the Save button. The error message that's displayed is defined by the form, not VFP. You'll also find that if you add and cancel several times, the next available CUST_ID is not incremented because NEXT_ID.PRG isn't called until a new CUSTOMER record is inserted when you click on the Save button.

Here's the code from the Click() method of the Save button; this code is simpler than you'd use in a "real" form but shows how field validation rule failure can be handled.

```
local lcAlias, ;
  laError[1], ;
  lcField, ;
  lcDBCField, ;
  lcMessage, ;
  lnI
lcAlias = alias()
if tableupdate()
  if cursorgetprop('SourceType') <> 3
       = requery()
  endif cursorgetprop('SourceType') <> 3
  Thisform.Refresh()
else
  select (lcAlias)
  = aerror(laError)
  lnError = laError[1]
  do case

* If the error was caused by a field validation
* rule being violated, get the error message for
* the field from the DBC (or use a generic message
* if there isn't one) and display it. Find the
* control for the field and set focus to it.

        case lnError = 1582
               lcField    = alias() + '.' + laError[3]
               lcDBCField = iif(cursorgetprop('SourceType') = 3, ;
                      lcField, dbgetprop(lcField, 'Field', 'UpdateName'))
               lcDBCField = substr(lcDBCField, at('!', lcDBCField) + 1)
               lcMessage  = dbgetprop(lcDBCField, 'Field', 'RuleText')
```

```
                lcMessage  = iif(empty(lcMessage), ;
                        'Improper value entered into ' + laError[3] + '.', ;
                        evaluate(lcMessage))
                = messagebox(lcMessage, 0, Thisform.Caption)
                for lnI = 1 to Thisform.ControlCount
                        if type('Thisform.Controls[lnI].ControlSource') <> 'U' and ;
                                upper(Thisform.Controls[lnI].ControlSource) = lcField
                                Thisform.Controls[lnI].SetFocus()
                                exit
                        endif type('Thisform.Controls[lnI].ControlSource') ...
                next lnI

* Display VFP's error message.

        otherwise
                = messagebox('Error #' + ltrim(str(laError[1])) + ;
                chr(13) + laError[2], 0, Thisform.Caption)
  endcase
endif tableupdate()
```

There's one thing about updatable views that frequently catches developers: if the table the view is defined from is buffered, the TABLEUPDATE() for the view will write the view record into the table's buffer, not directly to the disk. If you forget to also issue TABLEUPDATE() for the table, you'll get an error message when VFP tries to close to table because there are uncommitted changes in the table's buffer.

Lookup Tables

All but the simplest applications use lookup tables. Since most lookup tables have the same structure (code and description), some developers like to use a "multi-lookup" table. A multi-lookup table combines many small lookup tables into one larger table. This table usually has the same structure as an individual lookup table, but with the addition of a TYPE field. This field contains a value indicating what type of lookup each record is for. For example, A might represent customer type lookups, B customer sales regions, C employee types, etc. The primary key for this table is TYPE + CODE so within each type, the CODE must be unique. Here are some sample records:

TYPE	CODE	DESCRIPTION
A	PRO	Prospect
A	REG	Regular
A	SPE	Special
B	NE	Northeast
B	SW	Southwest
C	FTP	Full-time permanent
C	FTT	Full-time temporary
C	PTP	Part-time permanent
C	PTT	Part-time temporary

The advantage of having a multi-lookup table is that there are fewer tables to open and maintain than when a lot of smaller lookup tables are used. The disadvantage is that setting up the relationships is a little more complex because the type must be part of the relationship and a filter has to be set on the lookup table so the user can only see codes matching the desired type. Here's an example that opens CUSTOMER.DBF and two copies of LOOKUPS.DBF and sets relationships up for the customer type and customer sales region lookups:

```
select 0
```

```
use LOOKUPS again alias CUST_TYPE order TYPECODE
set filter to TYPE = 'A'
select 0
use LOOKUPS again alias CUST_REGION order TYPECODE
set filter to TYPE = 'B'
select 0
use CUSTOMER
set relation to 'A' + TYPE into CUST_TYPE, ;
  'B' + REGION into CUST_REGION
```

Working with multi-lookup tables is much easier in VFP because you can use views. You define one view for each type of lookup needed and only include records of the desired type in the view. For example:

```
create sql view CUST_TYPE_LOOKUP as ;
  select CODE, DESCRIP ;
  from LOOKUPS ;
  where TYPE = 'A'
create sql view CUST_REGION_LOOKUP as ;
  select CODE, DESCRIP ;
  from LOOKUPS ;
  where TYPE = 'B'
```

Using views gives you the best of both worlds: one physical table to manage but many easy-to-use logical representations. You don't need to worry about the lookup type when setting up relations, doing SEEKs, or displaying a pick list. Here's code that sets up the same type of relations as the previous code did, but using views:

```
select 0
use CUST_TYPE_LOOKUP
index on CODE tag CODE
select 0
use CUST_REGION_LOOKUP
index on CODE tag CODE
select 0
use CUSTOMER
set relation to TYPE into CUST_TYPE_LOOKUP, ;
  REGION into CUST_REGION_LOOKUP
```

This code shows the one downside of using views: because views don't have predefined indexes, you can't set up the relationships in the DataEnvironment of a form or report because you have to recreate the indexes every time the view is opened. Other than that, the code shown above could either represent opening separate lookup tables or different views of the same lookup table.

VFP 5.0

Version 5.0 of VFP should be released by the time you read this. Although there are a lot of new features in 5.0, not many changes were made to the database container. However, the ones that were made will make a big difference in your productivity. Let's take a look at these.

One change that will simplify things is having a multi-user database container. This means you no longer have to ensure no one else on your network has the database open before adding new tables, making table structure changes, or rebuilding indexes. Since it's possible another user added or removed tables or views from the database container while you had it open, a new Refresh function in the Database menu re-reads the database

from disk. Other new functions in the Database menu, such as Find Object and Arrange, make it easier to work with databases containing a lot of objects.

The Table Designer has a slightly new appearance. Instead of a Table button to display a dialog of table properties (such as rule and message), there's a Table page. This page also includes the name of the DBF and database and statistical information about the table (number of records, number of fields, and record size). The biggest change, though, is on the Fields page. Four new properties appear: InputMask, Format, DisplayLibrary, and DisplayClass (these are the names of the properties as the DBGETPROP() and DBSETPROP() functions expect them). These new properties together with a new VFP feature called Intellidrop will greatly improve your productivity in creating forms. Here's how it works.

In VFP 3, when you drag a field from the DataEnvironment, Project Manager, or Database Designer to a form, you get a control with the following attributes:

- The control is a VFP base class: Checkbox for Logical fields, Editbox for Memo fields, OLEBoundControl for General fields, and Textbox for all other data types.
- Textbox controls are sized to hold about 10 characters, regardless of the actual field size, forcing you to resize the control.
- The name of the control is the name of the class followed by an instance number (for example, the first Textbox control is Textbox1, the second is Textbox2, etc.). Usually, you'd rename the control to something more meaningful like txtCompany.
- No label is automatically created as a caption for the field.

Developers complained long and hard about the shortcomings of dropping a field on a form, and Microsoft responded with the Intellidrop feature in VFP 5. When you drop a field on a form, Intellidrop:

- creates a control of the class defined in the DisplayClass property for the field defined in the database (the DisplayLibrary property tells VFP where this class is stored);
- sizes the control appropriately for the field size;
- copies the InputMask, Format, and Comment properties of the field to the same properties of the control; and
- creates a Label object to the left of the control whose Caption property is set to the Caption of the field.

You can turn off some or all of these features by bringing up the Tools Options dialog, selecting the Field Mapping page, and unchecking the appropriate checkbox. You can also define which class to use by default for each data type; this class is used whenever a field with "<default>" as the DisplayClass is dropped on a form.

Like many things, Intellidrop isn't perfect. Its shortcomings are:

- The name of the control is still the class name followed by an instance number.

- The controls aren't always sized perfectly; you might still need to tweak the width a bit.
- The label it creates is of the Label base class; there's no way to define a different class to use.
- Only classes stored in visual class libraries (VCXs) can be used (this isn't so much a complaint as an observation).
- If you change the field properties in the database, you have to delete the label and field control and redrop the field on the form, since these properties aren't dynamically tied to the control.

In addition to the benefits provided via Intellidrop, having the InputMask and Format for a field now stored in the database means we can create controls that dynamically set their InputMask and Format properties as appropriate for their ControlSource at runtime. The advantage of doing this is reduced maintenance: if you decide a field should only contain upper-case data, you'd have to edit every form and report displaying the field if the format is hard-coded into its control. If the control asks the database at runtime for the format for the field, no editing is required at all. This is also especially useful for applications where the user has some ability to customize the application, such as an accounting system where the user can define the format for account codes or inventory part numbers.

Here's code you can put in the Init() method of a control (or better yet, in the Init() method of a class) to set the InputMask and Format properties at runtime (the SFTextbox class in the CONTROLS.VCX class library on the source code disk has this code):

```
if not empty(This.ControlSource) and ;
  not empty(dbc()) and ;
  indbc(This.ControlSource, 'Field')
  local lcInputMask, lcFormat
  lcInputMask = dbgetprop(This.ControlSource, ;
        'Field', 'InputMask')
  lcFormat = dbgetprop(This.ControlSource, ;
        'Field', 'Format')
  This.InputMask = iif(empty(lcInputMask), ;
        This.InputMask, lcInputMask)
  This.Format = iif(empty(lcFormat), ;
        This.Format, lcFormat)
endif not empty(This.ControlSource) ...
```

An example of using this scheme is shown in the EMPLOYEE.SCX form on the source code disk. If you examine the controls on this form, you'll find no InputMask or Format properties set. However, when you DO the form, the Phone field is formatted as 999-999-9999 and the Category field is forced to upper-case because that's how those fields are defined in the database.

I tested the performance hit this code gives by including or excluding the code shown above from the SFTextbox class used for the controls on the form, and found the form takes about 15% longer to instantiate with this code. However, that means the form takes 0.327 seconds to display rather than 0.283 seconds, so the user really won't notice the difference. Thus, perhaps with the exception of the busiest of forms, I suggest using this scheme to ensure your forms keep up to date with changes made to the database.

Another new feature you're going to love is being able to update fields in the current table in field and table validation rules. VFP 3 allowed you to update fields in another table in validation code, but attempting to change a field in the current table caused an "illegal recursion" error. This meant you couldn't define code to, for example, timestamp a record (put the date and time of the last change into a field), as a table rule, but instead had to do it in a form. The advantage of putting this code in a rule is that it automatically happens without the form developer having to code for it, and it also works in browses or programmatic changes to the table. In VFP 5, you can now use code similar to the following (this code is in the stored procedures of the DATA database on the source code disk and is called as the table validation rule for the EMPLOYEE table):

```
function EmployeeTableRule
local ltStamp

* Assign the EMP_ID field if it hasn't been already.

if empty(EMP_ID)
  replace EMP_ID with NEXT_ID('EMPLOYEE')
endif empty(EMP_ID)

* Timestamp the record.

ltStamp = datetime()
if LAST_UPDATE <> ltStamp
  replace LAST_UPDATE with ltStamp
endif LAST_UPDATE <> ltStamp
return .T.
```

This code does two things: assigns the primary key for the current record if it hasn't already been assigned (this is another way to prevent "holes" in sequential primary keys as I discussed earlier) and timestamp the record every time it's changed (you could also store the name of the user who made the change). You could use field level validation rules to ensure that the contents of a field are forced to upper case or partial entries are automatically filled in with a complete value (similar to the AutoCorrect feature in Word 95).

There are a few important things to know when writing rules that update the current table:

- Validation rules can modify the current table, but triggers cannot; they still give an "illegal recursion" error.
- The code that updates a field should only do so if the field's value is different from what should be stored. Notice the code above only updates the two fields if their values need to be updated. This prevents recursion from occurring (yes, recursion can still occur in validation rules if you don't do it properly).
- If it's possible that recursion may occur, you can prevent it from going past more than one level (and causing VFP to bomb) by checking the calling stack (using the PROGRAM() function) and not doing the update if the validation code called called itself. Here's an example:

```
lcProgram = ''
lnLevel   = 1
do while not empty(program(lnLevel))
```

```
  lcCaller  = lcProgram
  lcProgram = program(lnLevel)
  lnLevel   = lnLevel + 1
enddo while not empty(program(lnLevel))
if not lcCaller == program()
  * do the REPLACE here since we haven't
  * called ourselves
endif not lcCaller == program()
```

Thanks to Linda Teh, Andy Neil, and Jim Slater for exploring and clarifying these issues in the VFP 5 beta forum on CompuServe.

Conclusion

Next month's column will be "Christmas stocking stuffers", a potpourri of idea tidbits. We'll look at lots of little things, such as where to put utility functions, why you shouldn't use the GO command, and how to visually indicate that a control is read-only.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America. CompuServe 75156,2326.