Drilling Down Into Your Data

Doug Hennig

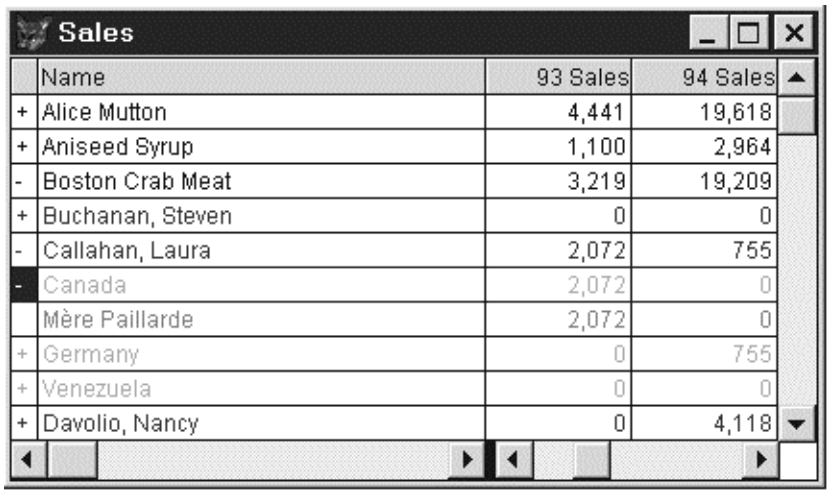
Do your users want to drill down into their data to see increasing levels of detail? You might think the TreeView control discussed last month is perfect for this, but it doesn't provide the ability to show columns of information. This month's article describes a technique for drilling down into data using a grid, plus describes how to make your grids act more like a spreadsheet with frozen panes.

Last year, a client asked me to create an application to provide their marketing staff with product sales breakdowns in a more flexible way than their accounting system could. Since their accounting system uses the Btrieve database manager, accessing the data is easy: I use DBFtrieve to read the Btrieve files and write the data to a VFP table. However, this application has a complex requirement: the need to "drill down" into the data. Initially, the user wants to see sales by product. They then want to "expand" a product to show the sales for that product broken down by region. Then, they can drill down into a region to see sales by city. Finally, they can drill down into a city to see the sales by customer for the expanded product.

Sound familiar? It sure sounded like the Windows Explorer to me: opening a folder shows the folders inside that folder, opening one of those folders shows its contents, and so on. So, my first thought was to use the Outline control that came with VFP (this was before VFP 5 with its ActiveX TreeView control, which we discussed last month, was available). However, I quickly discarded that idea because of the need for many columns of sales data (between various month-to-date and year-to-date values and quantities for this year and last year, the user needs to see more than ten columns of figures). In addition, the user wants the first column "locked" so as they scrolled horizontally to see the different sales columns, the name of the product is always visible. While VFP's Grid control provides a Partition property to split the grid into two panels, the behavior of the two panels is far from what Excel users are used to; for example, either panel can be scrolled horizontally to display any set of fields. What I needed was a cross between a spreadsheet and an Outline or TreeView control. This article describes the technique I came up with.

To provide a concrete example to go along with this article, let's discuss a similar sales application using the test data that comes with VFP. The ORDERS table contains order information, while the ORDITEMS table contains the products sold for each order. PRODUCTS contains product information while CUSTOMER and EMPLOYEE contain customer and employee information, respectively. We'll create a drill down application that shows, at the top level, sales by product. These sales can then be drilled down to the employee level, then down to country, and finally down to the specific customer. You can download the source code files for this application. Figure 1 shows the drill down form with some records expanded so you can see how the application works.

Figure 1. Drill Down Form



How I Spent my Summer

The secret to creating a control that can drill down into data while displaying multiple columns is to use a Grid with a filter so only certain records are visible at one time and with certain properties and methods that provide a spreadsheet-like behavior.

First, drill down is handled using a "smoke and mirrors" technique. Rather than displaying data directly from the source tables, a "drill down" table is created that contains all the information we need in one place, with sales totals already calculated. This is a technique used by data warehousing applications, where data from a relational database is pulled into a denormalized structure for the simplest and fastest access. A performance hit is taken when the structure is created, but after that, the application doesn't have to worry about performing table joins or calculating totals, so performance is much better for the user.

The drill down table is filtered so only those records the user should see are in the current filter set. I control this using a field called VISIBLE and a filter on this field. Initially, this field is only set to .T. for product records. When the user drills down into a product, the VISIBLE field for all employee records for that product are set to .T. Now, all product records *and* those employee records for a specific product are visible. Similarly, when the user drills down into an employee, the VISIBLE field for all country records for that product and employee are set to .T. Now, all product records *and* those employee records for a specific product *and* those country records for a specific product and employee are visible. Collapsing a row does just the opposite: the VISIBLE field of all applicable records is set to .F., making those records invisible once again. Another field, EXPANDED, saves the current state of a row. This allows us to display a "+" or "-" to indicate if a row can be expanded or has already been expanded, just like an Outline or TreeView control.

Getting a VFP Grid control to act like a spreadsheet with a frozen pane takes some doing. The first thing is to set the Partition property to split the grid into two panels. Partition contains the number of pixels in the left panel, so you'll set it so the left panel only displays the first two columns (one for the "+" or "-" and one for the name). However, the first thing you'll notice when you do this is that both panels have a vertical scroll bar, either of which can be used to scroll the grid. This is confusing to the user and

takes up precious real estate, so the next job is to eliminate the scroll bar for the left panel. The solution is to unlink the two panels so they act like two grids in one, eliminate the scroll bar for the left panel, then relink the two panels again. Unfortunately, this can't be done at design time, but is relatively easy to do (once you know how <g>) in code at runtime. The Init() method of the grid, which we'll look at later, does this.

Next, you have to prevent the user from scrolling unwanted columns into either panel (column 3 or higher for the left panel or columns 1 or 2 for the right panel). While you can't prevent the user from scrolling, you can clean things up afterward. The Grid's Scrolled() event fires when the user scrolls, so by checking the LeftColumn property of each panel, you can determine whether unwanted columns appear in each panel, and use the DoScroll() method to put things back properly. Unfortunately, this causes a brief flicker as unwanted columns move into view and then disappear again, and using LockScreen doesn't help, but this is a minor glitch.

The Drill Down Table

Let's look at the drill down table in more detail. It must contain the entire set of records the user will see in the drill down form. This means there will be several "levels" of records. One level of records will be the topmost layer of records the user sees; we'll call these level 1. Each level 1 record will have a set of records which represents the first layer of breakdown; these records will be level 2. Each level 2 record will have a set of records called level 3, and so on.

The drill down table can consist of any fields you wish, with the following minimum fields:

- LEVEL: This numeric field will contain the level number.
- NAME: The text to display for each record. The source of the text may change from one level to the next. In our sales scenario, for example, NAME for level 1 records comes from PRODUCT.PROD_NAME while it consists of the employee's first and last names from the EMPLOYEE table for level 2.
- DESCRIP1: This field contains the sort value (such as a name) for level 1 records. All records which belong to a given level 1 record will contain the same value. The name of this field would probably be the name of the field from the original data source; we're calling it DESCRIP1 here as a place holder. In the case of our sales example, this field is called PROD_NAME.
- DESCRIP<n>: Each level except the lowest one will have a sort value field as described for DESCRIP1.
- VALUE<n>: These fields will contain the rolled up sales values at each level. Our sales example has the following field: SALES93, SALES94, SALES95, SALES96 (sales values for 1993 to 1996), QTY93, QTY94, QTY95, and QTY96 (quantity sold for 1993 to 1996). The values in these fields for each level's records represent the total values for that level. For example, since level 1 in our sales scenario is product, these fields contain the total sales and quantity sold for each product by year. Level 2 is employee, so for a given level 2 record, these fields contain the sales and quantity sold of the appropriate level 1 record (the product) by the employee.

The table needs to be indexed so all records for a given level appear together. Typically, the index expression will be DESCRIP1 + DESCRIP2 + … + STR(LEVEL, 1) + NAME. Take care to ensure the index expression doesn't exceed the maximum key size (usually 254 characters, but it's less if you use a collate sequence other than MACHINE). For our sales scenario, the index expression is UPPER(PROD_NAME + EMP_NAME + COUNTRY + STR(LEVEL, 1) + NAME); UPPER() is used so the display order is case-insensitive.

Table 1 shows a brief set of the drill down data for our sales example. Level 1 represents products, level 2 is employees for a given product, level 3 is the country a given product was sold to by a certain employee, and level 4 is the specific customer a product was sold to by an employee in a country.

Table 1. Example of a Drill Down Table.

LEV	NAME	PROD_NAME	EMP_NAME	COUNTRY	SALES93
1	Alice Mutton	Alice Mutton			70
2	Buchanan, Steven	Alice Mutton	Buchanan, Steven		25
3	Brazil	Alice Mutton	Buchanan, Steven	Brazil	10
4	Hanari Carnes	Alice Mutton	Buchanan, Steven	Brazil	10
3	USA	Alice Mutton	Buchanan, Steven	USA	15
4	Save-a-lot Markets	Alice Mutton	Buchanan, Steven	USA	15
2	Callahan, Laura	Alice Mutton	Callahan, Laura		45
3	UK	Alice Mutton	Callahan, Laura	UK	10
4	Island Trading	Alice Mutton	Callahan, Laura	UK	10
3	USA	Alice Mutton	Callahan, Laura	USA	35
4	Rattlesnake Canyon Grocery	Alice Mutton	Callahan, Laura	USA	20
4	White Clover Markets	Alice Mutton	Callahan, Laura	USA	15
1	Aniseed Syrup	Aniseed Syrup			44
2	Callahan, Laura	Aniseed Syrup	Callahan, Laura		10
3	Sweden	Aniseed Syrup	Callahan, Laura	Sweden	10
4	Berglunds snabbköp	Aniseed Syrup	Callahan, Laura	Sweden	10
2	Davolio, Nancy	Aniseed Syrup	Davolio, Nancy		34
3	Denmark	Aniseed Syrup	Davolio, Nancy	Denmark	5
4	Vaffeljernet	Aniseed Syrup	Davolio, Nancy	Denmark	5
3	USA	Aniseed Syrup	Davolio, Nancy	USA	29
4	Rattlesnake Canyon Grocery	Aniseed Syrup	Davolio, Nancy	USA	12
4	Save-a-lot Markets	Aniseed Syrup	Davolio, Nancy	USA	17

Notice the following:

- NAME contains the product name for level 1, the employee name for level 2, the country for level 3, and the customer for level 4.
- PROD_NAME, EMP_NAME, and COUNTRY contain the same values for all records for a given product, employee, and country.
- EMP_NAME is empty for a level 1 record because that is the rollup record for a given product. COUNTRY is empty for a level 2 record because that is the rollup record for a given employee.
- For each level, SALES93 contains the sum of values for records at the next lower level. For example, the Alice Mutton product has 70 for SALES93, which is the sum of 25 and 45 (from the two level 2 records for Alice Mutton) while the Steven Buchanan record for the Alice Mutton product has 25 for SALES93, which

is the sum of 10 and 15 (from the two level 3 records for this employee and this product).

Because both the original sources of the data and the format for the drill down table may vary greatly from application to application, a different program must be used to create each drill down table. However, the basic idea for such as program is as follows:

- Use a SQL SELECT statement to create a cursor of level 1 records, grouping on the level 1 field.
- Use a SQL SELECT statement to create a cursor of level 2 records, grouping on the level 1 and level 2 fields.
- Use a SQL SELECT statement to create a cursor of level 3 records, grouping on the level 1, level 2, and level 3 fields.
- Create as many cursors as necessary, depending on how many drill down levels are desired.
- Concatenate all the cursors into a single table.
- Add a record for grand totals; use level 0.
- Index the table so all records for a given level appear together.

The sample MAKESALE.PRG creates a drill down table called SALES.DBF from the VFP TESTDATA database for our sales scenario. This code isn't shown here due to its length, but can be downloaded from the Source Code Download site.

The Drill Down Form

The drill down form is a class called DrillDownForm in DRILLDWN.VCX. It consists of a form, a grid, and some custom properties and methods. This class was created to be as generic as possible. This means it makes few assumptions about the structure of the drill down table and must have certain properties set after it's instantiated but before it's displayed to the user. We'll look at the details of the class first, then look at how to use it.

The assumption this class makes about the structure of the drill down table is that there's a column called LEVEL containing the level for each record and a column called NAME containing the display value the user should see for each level. If you use different names, you'll need to change the SetupData() and Expand() methods of the class.

Form Properties

BaseClass	Form (use your own class here)
Caption	Sales
DataSession	2-Private Data Session
Height	360
Name	frmSales
Width	600

Form Custom Properties

aColorLevel[1]	An array of colors to use for each level
aLevelKey[1]	An array containing the key field for each level
cOrder	The order to use so the data is properly sorted for drill downs

cTable	The name (including path if necessary) the data will come from
cTextBoxClass	The class to create the text boxes in the grid from (default = DrillDownGridTextBox)
nMaxLevels	The numbers of levels in the cursor

Form Methods

Load() simply sets up some environmental things:

```
set talk off
set deleted off
set notify off
set bell off
```

Resize() resizes the grid when the form is resized so the grid always fills the form:

```
with This
  .LockScreen      = .T.
  .grdSales.Width  = .Width
  .grdSales.Height = .Height
  .LockScreen      = .F.
endwith
```

Show() sets the DynamicForeColor for each column (we'll see how Thisform.aColorLevel is initialized later) and displays the form:

```
LPARAMETERS nStyle
This.grdSales.SetAll('DynamicForeColor', ;
  'Thisform.aColorLevel[LEVEL + 1]', 'Column')
```

Form Custom Methods

SetupData(), which is called before the form is displayed, creates a cursor from the source drill down table (to keep the class generic, the name of the drill down table is stored in the cTable property of the class rather than hard-coded). A cursor is needed rather than using the SALES table directly because the values of two fields are changed as the user expands and collapses rows. In order for this to work in a multi-user environment, the SALES table itself can't be modified but a cursor based on it can be. The cursor, called LV_SALES, consists of all fields from the source table plus two new fields, VISIBLE and EXPANDED, which indicate, respectively, whether the record is visible to the user and whether levels under this record are visible. Initially, VISIBLE is .T. only for level 0 (grand total) and level 1 (top level) records.

```
local lcAlias, ;
  lcCDX, ;
  lnTag, ;
  lcKey
with This

* Ensure properties we need are set (these could
* be ASSERT statements in VFP 5).

  if empty(.cTable)
    wait window 'cTable not specified'
    return .F.
  endif empty(.cTable)
  if empty(.cOrder)
    wait window 'cOrder not specified'
    return .F.
  endif empty(.cOrder)
```

```
* Open the table the data will come from.

  select 0
  use (.cTable)
  lcAlias = alias()

* Ensure the specified tag is a valid one (this could
* be an ASSERT statement in VFP 5).

  lcCDX = strtran(dbf(lcAlias), '.DBF', '.CDX')
  lnTag = tagno(.cOrder, lcCDX, lcAlias)
  if lnTag = 0
    wait window 'cOrder not valid'
    return .F.
  endif lnTag = 0

* Create a cursor for sales with additional fields
* we'll need: EXPANDED to indicate if the current
* record is expanded or not and VISIBLE to flag if
* the current record can be seen.

  select *, ;
      .F. as EXPANDED, ;
      LEVEL < 2 as VISIBLE ;
    from (lcAlias) ;
    into cursor TEMPSALES

* Now open the cursor again so it'll be read-write
* and we can create indexes on it. We can then close
* the original copy.

  use dbf('TEMPSALES') alias LV_SALES again in 0
  select LV_SALES
  use in TEMPSALES

* Create the index we'll need for the cursor from the
* specified index in the SALES table.

  lcKey = key(lnTag, lcAlias)
  index on &lcKey tag (.cOrder)

* Create an index for VISIBLE and set a filter on it
* so we can only see visible records.

  index on VISIBLE tag VISIBLE
  set filter to VISIBLE

* Use the specified order and move to the first record.

  set order to (.cOrder)
  go top

* Set data source properties for the grid now that the
* cursor exists.

  with .grdSales
    .RecordSource = 'LV_SALES'
    .Column1.ControlSource = ;
      "iif(LEVEL > Thisform.nMaxLevels or " + ;
      "LEVEL = 0, ' ', iif(EXPANDED, '-', '+'))"
    .Column1.Header1.Caption = ''
    .Column2.ControlSource = 'LV_SALES.NAME'
  endwith

* We can close the original table since we don't need
* it anymore.

  use in (lcAlias)
endwith
```

Expand() is called when the user expands or collapses the current row by clicking on the + column for the row.

```
local llExpanded, ;
  lnLevel, ;
  lcKey, ;
  lnI, ;
  luKey, ;
  lnRecno

* If we've drilled down below the maximum expandable
* level or we're sitting on the "totals" record, don't
* go on.

if LEVEL > This.nMaxLevels or LEVEL = 0
  return
endif LEVEL > This.nMaxLevels ...

* Toggle the expanded status for the current record,
* and save the new status. Increment the level so we
* know which one to work with.

This.LockScreen = .T.
replace EXPANDED with not EXPANDED
llExpanded = EXPANDED
lnLevel    = LEVEL + 1

* Create the key expression we'll use to know which
* records to toggle the "visible" status for.

lcKey = ''
for lnI = 1 to LEVEL
  luKey = evaluate(This.aLevelKey[lnI])
  lcKey = lcKey + iif(empty(lcKey), '', ' and ') + ;
    This.aLevelKey[lnI] + '=='
  do case
    case type('luKey') = 'C'
      lcKey = lcKey + '"' + luKey + '"'
    case type('luKey') = 'N'
      lcKey = lcKey + ltrim(str(luKey))
    case type('luKey') = 'D'
      lcKey = lcKey + '{' + dtoc(luKey) + '}'
  endcase
next lnI

* Turn off the filter so we can see all records, save
* the current record pointer, and move to the next
* record (the first one under the current one).

set filter to
lnRecno = recno()
skip

* If we're collapsing a level, collapse all levels
* under it. Otherwise, toggle the "visible" flag for
* all records under this one.

if not llExpanded
  replace VISIBLE with .F., EXPANDED with .F. ;
    while &lcKey
else
  replace VISIBLE with not VISIBLE while &lcKey ;
    for LEVEL = lnLevel
endif not llExpanded

* Reposition the record pointer to the record we were
* drilling down and set the filter so we only see
```

```
* "visible" records, then refresh the grid.

go lnRecno
set filter to VISIBLE
This.grdSales.Refresh()
This.LockScreen = .F.
```

AddGridColumn() is a custom method that adds a column to the grid and sets various properties.

```
lparameters tcControlSource, ;
  tcCaption
local lnColumn
with This.grdSales
  .AddColumn()
  lnColumn = .ColumnCount
  .Columns[lnColumn].Name      = 'Column' + ;
    ltrim(str(lnColumn))
  .Columns[lnColumn].Width     = 75
  .Columns[lnColumn].ReadOnly  = .T.
  .Columns[lnColumn].InputMask = '999,999,999'
  .Columns[lnColumn].Header1.Alignment = 1
  .Columns[lnColumn].Header1.Caption   = tcCaption

* Create a textbox for the new column and make it the
* current control for the column.

  .Columns[lnColumn].AddObject('DrillText', ;
    This.cTextBoxClass)
  .Columns[lnColumn].DrillText.Visible = .T.
  .Columns[lnColumn].CurrentControl = 'DrillText'
  .Columns[lnColumn].ControlSource  = tcControlSource
endwith
```

Grid Properties

We can't set the RecordSource of the grid or ControlSource of the columns to the LV_SALES cursor at design time because that cursor won't exist until *after* the grid has been instantiated, which would cause an error. Thus, we'll leave these properties blank for now and set them programmatically at runtime in the SetupData() method of the form.

BaseClass	Grid (use your own class here)
ColumnCount	2
DeleteMark	.F.
Height	360
Name	grdSales
Partition	268 (only approximate; will be set at runtime to the proper value)
ReadOnly	.T.
RecordMark	.F.
RecordSource	None (it will be set at runtime)
Width	569
Column1.ControlSource	None (it will be set at runtime)
Column1.Width	13
Column1.Header1.Caption	None
Column2.ControlSource	None (it will be set at runtime)
Column2.Width	232
Column2.Header1.Caption	Name

Grid Methods

Init() sets the left panel to not have a scroll bar. This can't be done at design time, so the grid will appear to have two vertical scroll bars. At runtime, however, this eliminates the scroll bar for the left panel, making the grid look more like an Excel spreadsheet with a

frozen panel. Init() also adjusts the partition position so the left panel just includes the first two columns.

```
with This
  .PanelLink  = .F.
  .Panel      = 0
  .ScrollBars = 1
  .Panel      = 1
  .PanelLink  = .T.
  .Partition  = .Column1.Width + .Column2.Width + 2
endwith
```

Scrolled(), which is fired whenever the user scrolls the grid, ensures the appropriate columns are displayed in the appropriate panel by forcing the grid to scroll unwanted columns out of view if necessary.

```
LPARAMETERS nDirection
local lnI
with This
  do case

* If either of the first two columns is visible in the
* right panel, scroll until they're no longer there.

    case .Panel = 1 and .LeftColumn <= 3
      for lnI = .LeftColumn to 2
        .DoScroll(5)
      next lnI

* If any column but the first two is visible in the left
* panel, scroll until they're no longer there.

    case .Panel = 0 and .LeftColumn > 1
      for lnI = .LeftColumn to 2 step - 1
        .DoScroll(4)
      next lnI
  endcase
endwith
```

Column1.Text1.GotFocus() ensures the right panel doesn’t show the first two columns by setting focus to column 3 (the first column in the right panel) if necessary. This makes the grid act more like a spreadsheet.

```
with This.Parent.Parent
  if .Partition <> 0 and .Panel = 1
    .Column3.SetFocus()
  endif .Partition <> 0 ...
endwith
```

Column1.Text1.LostFocus() moves to the third column (the first column we want displayed in the right panel). This makes the grid act more like a spreadsheet.

```
with This.Parent.Parent
  if .Partition <> 0 and .Panel = 0
    .Panel = 1
    .Column3.SetFocus()
  endif .Partition <> 0 ...
endwith
```

Column1.Text1.KeyPress() accepts Enter as signal to expand or collapse the row, and ignores any keypress that would put a value into the column.

```
LPARAMETERS nKeyCode, nShiftAltCtrl
do case
  case nKeyCode = 13
    Thisform.Expand()
    nodefault
  case between(nKeyCode, 32, 127)
    nodefault
endcase
```

Column1.Text1.Click() expands or collapses the current row.

```
Thisform.Expand()
This.SelStart = 0
This.SelLength = 1
nodefault
```

Column2.Text1.When() prevents focus from being set to the Name column:

```
return .F.
```

DrillDownGridTextBox Class

This class is used to create textbox controls in the columns in the right panel of the grid at runtime.

BaseClass	TextBox (use your own class here)
BorderStyle	0 - None
Margin	0

When() prevents the textbox from receiving focus if there isn't a partition or if focus is in the left panel:

```
with This.Parent.Parent
  return .Partition = 0 or .Panel = 1
endwith
```

Using the Drill Down Form

SALES.PRG is a sample program using the drill down form. This program does the following:

- Instantiates a DrillDownForm object.
- Sets the cTable and cOrder properties to the name of the drill down table and the tag that orders the data properly, then calls the SetupData() method to create the LV_SALES cursor and set the data source properties of the grid (they can't be set at design time since this cursor won't exist until after the grid has been instantiated).
- Sets the nMaxLevels property to the number of levels that can be expanded (one less than the total number of levels) and initializes the aLevelKey array to the names of the fields containing the "key" value for each level.

- Initializes the aColorLevel array to the color desired for each level. Using color to distinguish levels is very important; if each level is same color, it's difficult to visually tell what data you're looking at. Although the colors are hard-coded in SALES.PRG, you'll probably want to read these values from a user preferences table or from the Windows Registry. Row 1 of aColorLevel is for level 0, row 2 is for level 1, etc.
- Creates columns of the grid to display the sales values using the custom AddGridColumn() method.
- Shows the DrillDownForm object.

Here's the code for SALES.PRG:

```
#include FOXPRO.H
set talk off

* Create an instance of the DrillDownForm class into a public variable so it
* exists when this program is done.

public oSales
set classlib to DRILLDWN
oSales = createobject('DrillDownForm')
with oSales

* Initialize the properties containing the name and order of the data table and
* set up the data for the drill down.

  .cTable = 'SALES'
  .cOrder = 'SALES'
  .SetupData()

* Initialize the levels and level keys we need.

  .nMaxLevels = 3
  dimension .aLevelKey[.nMaxLevels]
  .aLevelKey[1] = 'PROD_NAME'
  .aLevelKey[2] = 'EMP_NAME'
  .aLevelKey[3] = 'COUNTRY'

* Initialize color settings.

  dimension .aColorLevel[.nMaxLevels + 2]
  .aColorLevel[1] = COLOR_RED
  .aColorLevel[2] = COLOR_BLACK
  .aColorLevel[3] = COLOR_BLUE
  .aColorLevel[4] = COLOR_GREEN
  .aColorLevel[5] = COLOR_MAGENTA

* Create grid columns we want to display. Pass the
* control source and header caption.

  .AddGridColumn('lv_sales.sales93', '93 Sales')
  .AddGridColumn('lv_sales.sales94', '94 Sales')
  .AddGridColumn('lv_sales.sales95', '95 Sales')
  .AddGridColumn('lv_sales.sales96', '96 Sales')
  .AddGridColumn('lv_sales.qty93',   '93 Qty')
  .AddGridColumn('lv_sales.qty94',   '94 Qty')
  .AddGridColumn('lv_sales.qty95',   '95 Qty')
  .AddGridColumn('lv_sales.qty96',   '96 Qty')
endwith

* Display the drill down form.

oSales.Show()
```

Conclusion

The DrillDownForm class described in this article can be made even more generic by using a subclass of the Grid control rather than a Form. The advantage of that approach is you could drop one of these objects on any form to provide drill down capabilities as well as other form controls. Another enhancement would be to use a property containing the number of columns in the left panel rather than hard-coding it two in various methods. Other changes you can make are to use a different class in place of DrillDownGridTextBox (if you need different behavior, for example); you just have to set the cTextBoxClass property of the DrillDownForm class to tell it which class to use for the grid text boxes. I hope you enjoy the techniques I outlined in this article!

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 Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. 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. He was a Microsoft Most Valuable Professional (MVP) for 1996. CompuServe 75156,2326.