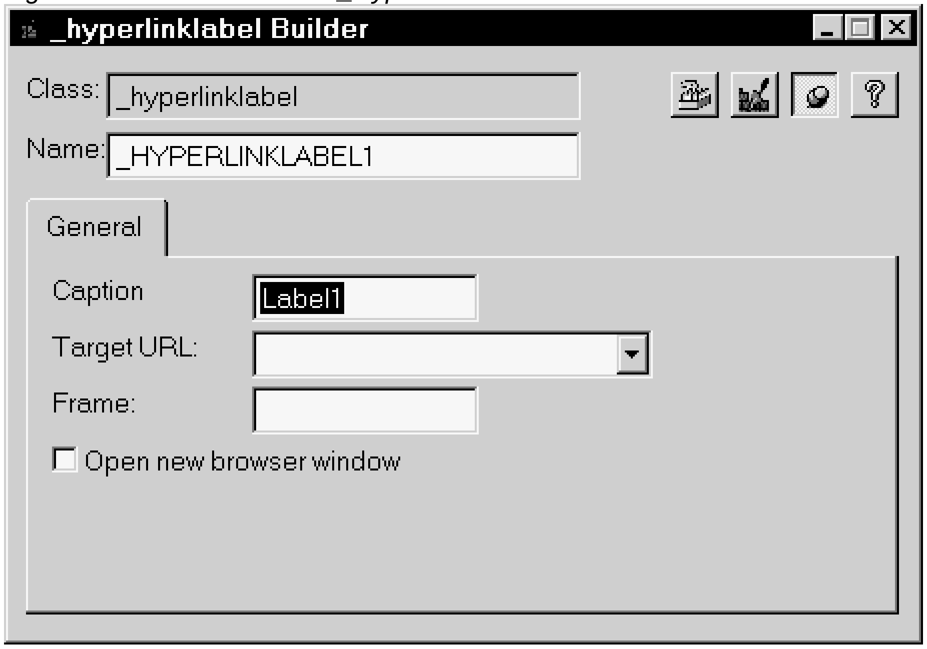
Building Builders with BuilderD

Doug Hennig

BuilderD is a new data-driven builder technology that comes with VFP 6. This month, Doug takes a look at how BuilderD works and how you can create builders with it in record time.

Over the past several months, we looked at the FoxPro Foundation Classes (FFC) that come with VFP 6. One of the things I noticed while I was looking at these classes is that they all have custom properties called Builder and BuilderX, and BuilderX is set to = HOME() + "Wizards\BuilderD,BuilderDForm" in each class. Digging a little deeper, I found that it's actually the classes in _BASE.VCX (subclasses of VFP base classes from which all FFC classes are subclassed) that have these properties added. I knew what these properties are for (they tell VFP the name of the builder for the class), but why does every class specify the same builder? Even more interesting, invoking the builder for each class shows a similar builder form but with different options for each one (Figure 1 shows the builder for the _HyperLinkLabel class, for example). How the heck does that happen when the same class is being used?

Figure 1. The builder for the _HyperLinkLabel class.



Class: _hyperlinklabel
Name: _HYPERLINKLABEL1
Caption: Label1
Target URL:
Frame:
Open new browser window

First a little background. As you're probably aware, VFP builders can be called a variety of ways, but the most common is probably by right-clicking on an object and choosing Builder from the context menu. This causes the program specified in the _BUILDER system variable (which by default contains BUILDER.APP in the VFP home directory) to be executed. BUILDER.APP checks to see if the selected object (we'll call this the "target object") has a BuilderX property, and if so, runs the program or instantiates the class specified in that property (if the builder is a class, it should be specified as the class library, a comma, and the class name). If it doesn't have a BuilderX property but has a Builder property, the builder runs the program or instantiates the class specified in that property (we'll see why there are two properties to specify the builder later). If neither of these properties exist, it looks in BUILDER.DBF (located in the WIZARDS subdirectory of the VFP home directory) for the base class of the target object and runs the builder registered in that table or displays a list of builders if more than one is registered for a particular class. If no builder can be found for the target object, an message to that effect is displayed.

OK, so what does all this mean? It means you can specify which builder to use for a particular class in one of three ways:

- You can change _BUILDER to the name of your builder program. This isn't a great way to do it, because you have to change _BUILDER constantly, depending on the selected object.
- You can register your own builders in BUILDER.DBF. One hassle with this approach is that you have to register the builder with the base class of the target object, so whenever you invoke a builder for any object of that base class, you'll get a dialog asking you to select which builder you want, even though some of them are specific for a certain class rather than the base class.
- You can create custom Builder and BuilderX properties in your classes (even in your base classes as Microsoft did with _BASE.VCX) and then fill them in with the name of the appropriate builder for each specific. This is the best approach, because it allows you to easily specify a custom builder for each class. The reason for having two properties is that BuilderX specifies a custom builder for the specific class, while Builder is intended for a builder for a set of common classes such as all comboboxes or grids. As we'll see later, we can click on a button in the builder specified in the BuilderX property to bring up the builder specified in the Builder property.

So we've figured out that specifying our builders in Builder and BuilderX properties is the way to go. But isn't it a lot of work to create your own builders, especially for classes that may not be used much?

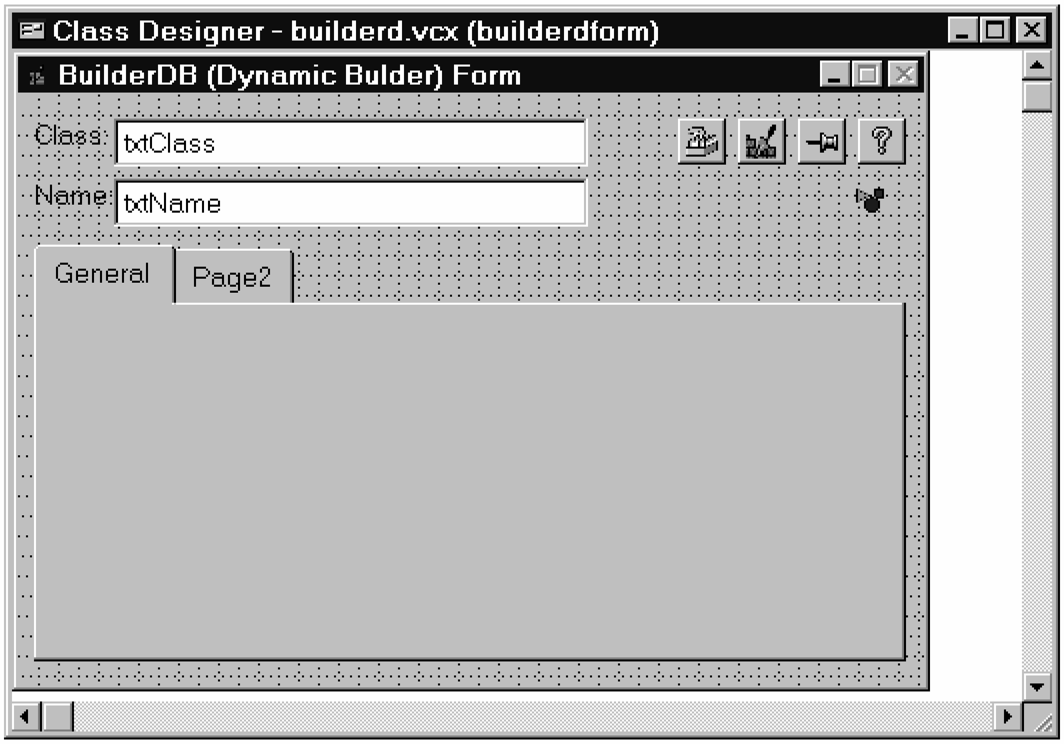
BuilderD

In the December 1997 issue of FoxTalk, Yuanitta Morhart and I looked at the BuilderB technology created by Ken Levy to make creating builders faster and easier. BuilderB is a set of classes you can subclass to create your own builders. You add a control to the builder subclass for each property of the target object you want the builder to maintain. Although BuilderB makes creating builders much easier, it's kind of a drag to have to create a new builder subclass for each class you want a builder for. Fortunately for us lazy types, Ken enhanced BuilderB by making it data-driven. The new technology, called BuilderD (the "D" stands for "Dynamic"), is available from Ken's Web site (www.classx.com) and is also included with VFP 6 (BUILDERD.VCX in the WIZARDS subdirectory of the VFP home directory).

BuilderD consists of several classes, but the main one is BuilderDForm; this is the data-driven builder form (notice this is the class specified in BuilderX for the classes in _BASE.VCX). As you can see in Figure 2, this form has textboxes for the class and the name of the target object, buttons that provide functionality like bringing up the Class Browser and displaying help, and a pageframe with a couple of pages, but no controls to manage the values of properties. Here's how this form is populated with the appropriate controls when it's instantiated:

- The Init method calls the SetObject method, which calls the AddObjects method (this code is actually in BuilderBaseForm, the parent class of BuilderDForm).
- The AddObjects method calls the AddObjects method of the oBuilderDB object on the form, which comes from the BuilderDB class.
- BuilderDB.AddObjects is a complex method, but the basics are that it opens the builder definition table (by default, BUILDERD.DBF located in the WIZARDS subdirectory of the VFP home directory, but another table can be specified by changing the cBuilderTable property), finds a record for the class of the target object, finds all the records linked to that record, and uses information in those records to create controls on one or more pages of the pageframe. These controls are based on classes in BUILDERD.VCX, such as BuilderCheckBox and BuilderTextBox, that know how to be bound to properties of the target object.

Figure 2. BuilderDForm.


Class Designer - builderd.vcx (builderdform)
BuilderDB (Dynamic Bulder) Form
Class: txtClass
Name: txtName
General
Page2

The result of these steps is a builder that can manage one or more properties of the target object. A specific builder can actually do a lot more than that, such as putting code into methods of the target object or even the object's containers, but we'll take the simple approach for now.

BuilderD Table

Let's take a closer look at the builder definition table, BUILDERD.DBF, because understanding its structure is the key to creating your own builders. Table 1 shows the structure of BUILDERD.DBF; in this table, "property control" means the control in the builder that maintains a specific property of the target object.

Field	Purpose
TYPE	Defines the type of record. This contains either "CLASS" if this is a class record or "PROPERTY" if it's a property record.
ID	The identifier of the record, frequently the name of the class or property the record is for.
LINKS	A list of the ID values of other records (separated by carriage returns) linked to this record. This field is discussed in more detail below.
TEXT	The caption for the builder form if this is a "CLASS" record or the caption for the property control if it's a "PROPERTY" record.
DESC	The status bar text for the property control.
CLASSNAME	For a "CLASS" record, the name of the class this builder is for. BuilderDB looks for a record containing the name of the target object's class in this field. For a "PROPERTY" record, the class to instantiate for the property control. If this field is blank, a default class from BUILDERD.VCX will be used (BuilderCheckBox for logical properties and BuilderTextBox for all others). Any class you specify should be a subclass of a BuilderD class because those classes have special properties and methods used by BuilderDForm.
CLASSLIB	The class library containing the class specified in CLASSNAME. This property can contain an expression (such as HOME() + "WIZARDS\BUILDERD.VCX") rather than a constant; in that case, put parentheses around the expression. For a "PROPERTY" record, if this field is blank and CLASSNAME is specified, BUILDERD.VCX is assumed. For a "CLASS" record, BuilderDB looks for a record containing the same value in this field (after evaluating it if necessary) as the ClassLibrary property of the target object; leave this blank to create a builder for a class without worrying about what class library it's in.
MEMBER	Blank for "CLASS" records. For "PROPERTY" records, the name of the target object property maintained by the property control this record defines. If blank, ID must contain the name of the property.

HELPFILE	The name of the CHM file containing help for this class. If blank, the current help file is used.
HELPID	The ID for the help topic.
TOP	The Top setting for the property control. If 0, BuilderDForm will place the control below the previous one (the first control on a page is placed at the value specified in BuilderDB.nTop).
LEFT	The Left setting for the property control. If 0, BuilderDForm will place the control near the left edge of the page it's on (specified by BuilderDB.nLeft).
HEIGHT	The Height setting for the property control. If 0, the default Height for the control is used.
WIDTH	The Width setting for the property control. If 0, the default Width for the control is used.
ROWSRCTYPE	If the class to use for the property control (specified in CLASSNAME) is a combobox, the RowSourceType setting for the combobox.
ROWSOURCE	If the property control is a combobox, the RowSource setting for the combobox. For example, if ROWSRCTYPE is 1 (Value), ROWSOURCE will contain a comma-delimited list of values for the combobox.
STYLE	If the property control is a combobox, the Style setting for the combobox.
VALIDEXPR	An expression used to validate the value of the property.
READONLY	.T. if the property control is read-only.
UPDONCHNG	.T. if the property control's value is written to the target object's property as it's changed (that is, from the InteractiveChange method).
UPDATED	The datetime the record was last modified (not used by BuilderD but for information only).
COMMENT	Comments about the record (not used by BuilderD but for information only).
USER	User information for the record (not used by BuilderD but for information only).

Table 1. The structure of BUILDERD.DBF.

Tables 2 and 3 show the records that make up a couple of builders, the ones for the FFC _HyperLinkBase and _HyperLinkLabel classes. I haven't shown all fields in BUILDERD.DBF because of space considerations; only those fields pertinent to the discussion are shown.

TYPE	ID	LINKS	CLASSNAME	CLASSLIB
CLASS	_HyperLinkBase	cTarget cFrame lNewWindow	_HyperLinkBase	(HOME()+"FFC_Hyperlink.vcx")
CLASS	_HyperLinkLabel	Caption _HyperLinkBase	_HyperLinkLabel	(HOME()+"FFC_Hyperlink.vcx")

Table 2. The record for the builders for the _HyperLinkBase and _HyperLinkLabel classes.

TYPE	ID	TEXT	CLASSNAME	ROWSRCTYPE	ROWSOURCE
PROPERTY	cTarget	Target URL:	BuilderComboBox	1	www.microsoft.com/vfoxpro
PROPERTY	cFrame	Frame:			
PROPERTY	lNewWindow	Open new browser window			
PROPERTY	Caption	Caption			

Table 3. Records defining the properties managed by the _HyperLinkBase and _HyperLinkLabel builders.

In the record in Table 2 for _HyperLinkBase, we see that CLASSNAME and CLASSLIB specify which class this is the builder for (notice that CLASSLIB contains an expression that'll be evaluated at runtime rather than a hard-coded value), and LINKS lists the records that specify the properties of this class the builder will manage. The cTarget PROPERTY record shown in Table 3 indicates that this property will be managed by a BuilderComboBox control with a RowSource containing the former URL of the Microsoft VFP Web site (it's now msdn.microsoft.com/vfoxpro). cFrame will be managed by a BuilderTextBox object (because it's a character property and the class isn't defined) and lNewWindow will have a BuilderCheckBox object (because it's a logical property).

Seems simple so far, right? Well, the LINKS field can actually get more complicated. First, if an ID specified in the LINKS field for a CLASS record doesn't have a matching record, that ID is assumed to be the name of a property. Thus, you could actually create a builder in a single record by simply specifying the properties it manages in the LINKS field of the CLASS record. Of course, you'd have to live with captions for the properties being the same as the property name, no status bar text, and default classes and sizes for the properties, but that's not too bad for a quick and dirty builder. A second complication is that an ID specified in the LINKS fields for a CLASS can point to another CLASS record rather than to a PROPERTY record. In that case, this class “inherits” all of the links of the specified class. You can see this in the _HyperLinkLabel record in Table 2; one of its links is _HyperLinkBase, so not only does the builder for this class manage the Caption property (specifically listed in its LINKS field), but also the cTarget, cFrame, and lNewWindow properties because those are specified in the LINKS field for _HyperLinkBase. Thirdly, a PROPERTY record can be linked to another PROPERTY record. In that case, the record “inherits” all non-blank fields from the linked record. Finally, LINKS can contain the caption for the page in the builder pageframe if you specify it as @<caption> (for example, @Properties to use “Properties” as the page caption).

Creating a Builder

Let’s check it out. Start by creating a subclass of the VFP CheckBox class called TestCheck in TEST.VCX. Add a custom property called BuilderX to this class and set its value to = HOME() + “Wizards\BuilderD,BuilderDForm”. Then right-click on the class and choose Builder from the context menu. Oops, we get a “There are no registered builders of this type” error. That makes sense, since we haven’t defined one yet (although wouldn’t it be nice if it automatically created one for us—more on this later). Do the following:

- In the Command window, type USE HOME() + “WIZARDS\BUILDERS”, then BROWSE
- Choose “Append New Record” from the Table menu
- Enter “CLASS” for TYPE, “TestCheck” for ID, “Enabled”, “AutoSize”, and “Caption” (pressing Enter after each one) for LINKS, “My Test Builder” for TEXT, “TestCheck” for CLASSNAME, and “TEST.VCX” for CLASSLIB.
- Close the browse window and type USE in the Command window to close the table.
- Right-click on the TestCheck class and choose Builder.

Cool, huh? Your very own builder, created in just about one minute! Uncheck Enabled and enter a different Caption, and watch these properties change instantly. Notice that we didn’t bother creating records for the properties; Enabled and Caption records already existed in BUILDERD.DBF, so we just reused them, and since no AutoSize record existed, BuilderD just assumed we wanted to manage that property.

Let’s build another one and see how little we can enter and still get a working builder. Create a subclass of the VFP TextBox class called TestText in TEST.VCX. Again, add a BuilderX property and set it to = HOME() + “Wizards\BuilderD,BuilderDForm”. Create a record in BUILDERD.DBF and just specify TYPE (“CLASS”), ID (“TestText”), LINKS (“ReadOnly”), and CLASSNAME (“TestText”; we can skip CLASSLIB but CLASSNAME is required). Close the table, then bring up the builder for the class. Voila—a working builder. Check out the Builder button in the builder; it brings up another builder (one specified in Builder if that property existed and was filled in, or the default builder for the base class of this class, which is the VFP Text Box Builder in this case). Thus, even though we can have specific builders for a class, we can still access more generic builders as well.

Pre-Built Builders

I recently created a couple of builders using BuilderD, one for my SFGrid class and the other for SFPageFrame. The SFGrid builder maintains the DeleteMark and RecordMark properties (you could easily add other properties you might frequently change) but since they’re easily changed in the Property Sheet, they weren’t the real reason I created the builder. The real reason was because I like to use SFGridTextBox objects (a subclass of SFTextBox with a few properties set differently to improve their appearance in a grid) in the columns of a grid, and it’s a pain to replace the generic TextBox objects with SFGridTextBox objects. So I created a button (SFGridTextBoxButton in SFBUILDERS.VCX) that does this automatically. Here’s the code from the Click method of this button:

```
local loColumn, ;
  loControl, ;
  lcName
for each loColumn in Thisform.oObject.Columns
  for each loControl in loColumn.Controls
    if upper(loControl.Class) = 'TEXTBOX'
      lcName = loControl.Name
      loColumn.RemoveObject(lcName)
      loColumn.NewObject(lcName, 'SFGridTextBox', ;
        'SFCtrls.vcx')
    endif upper(loControl.Class) = 'TEXTBOX'
  next loControl
next loColumn
wait window 'SFGridTextBox added to each column' ;
  timeout 2
```

BuilderDForm has a reference to the target object stored in its oObject property, so any control on the builder form can reference or change things about the target object. In this code, Thisform.oObject.Columns references the Columns collection of the grid being affected by the builder.

One complication: how do I get this button on the builder form? I could create a record for it in the BUILDERD table, but that would add the button on a page of the pageframe and I'd rather have the button appear with the other builder buttons. So, I created a "button loader" class (SFBuilderButtonLoader in SFBUILDERS.VCX) which adds a button to the builder form and then returns .F. so it doesn't actually get instantiated. SFBuilderButtonLoader looks in the USER memo field of the current record in the BUILDERD table (the record that caused the class to be instantiated) for the class name and library of the button to add to the form (note in the code below that BUILDERD is open with the alias "BUILDER"). For SFGridTextBoxButton, for example, I created a record in the BUILDERD table with "SFGridTextBoxButton" as the ID but "SFBuilderButtonLoader" as the class, and entered "SFBuilders, SFGridTextBoxButton" in the USER memo. This tells SFBuilderButtonLoader to add an SFGridTextBoxButton to the form. Here's the code from the Init method of SFBuilderButtonLoader:

```
local lcClass, ;
  lnPos, ;
  lcLibrary, ;
  lnTop, ;
  lnLeft, ;
  loControl

* If the BUILDERD table is open and positioned to the
* record for the button to be loaded (which it should
* be), we'll add the button by getting the class and
* library for the button from the USER memo.

if used('BUILDER') and ;
  lower(BUILDER.CLASSNAME) = lower(This.Class)
  with Thisform
    lcClass = BUILDER.USER
    lnPos   = at(',', lcClass)
    if lnPos > 0
      lcLibrary = left(lcClass, lnPos - 1)
      lcClass   = substr(lcClass, lnPos + 1)
    else
      lcLibrary = ''
    endif lnPos > 0

* Add the button. If we succeeded, find the first open
* "slot" in the button area on the form.

    .NewObject(lcClass, lcClass, lcLibrary)
    if type('.' + lcClass + '.Name') = 'C'
      lnTop  = .cmdClassBrowser.Top + ;
        .cmdClassBrowser.Height + 5
      lnLeft = .cmdClassBrowser.Left
      for each loControl in .Controls
        if loControl.Left = lnLeft and ;
          loControl.Top = lnTop
          lnLeft = lnLeft + ;
            .cmdClassBrowser.Width + 6
          if lnLeft + .cmdClassBrowser.Width > .Width
            lnLeft = .cmdClassBrowser.Left
            lnTop  = lnTop + .cmdClassBrowser.Height + 5
          endif lnLeft + .cmdClassBrowser.Width > .Width
        endif loControl.Left = lnLeft ...
      next loControl

* Set the button position to the located slot and make
* it visible.

      with .&lcClass
        .Top     = lnTop
        .Left    = lnLeft
        .Visible = .T.
      endwith
    endif type('.' + lcClass + '.Name') = 'C'
  endwith
endif used('BUILDER') ...
return .F.
```

The SFGrid record in the BUILDERD table has DeleteMark, RecordMark, and SFGridTextBox in its LINKS column, so this builder gets these controls. I can now bring up my BuilderD builder for any SFGrid object, use the VFP Grid Builder (by clicking on the Builder button in the BuilderD form) to quickly create the columns in the grid, and then change those columns to use SFGridTextBox objects by clicking on my Add SFGridTextBox button.

For SFPageFrame, I wanted a similar function: a button that adds code to the RightClick method of each page in the pageframe; this allows right-clicking on a page to provide a context menu for the pageframe or (more likely) entire form. As with the SFGrid builder, I created a record in the BuilderD table that instantiates an SFBuilderButtonLoader object that then adds an SFCodePageButton object to the builder form. Here's the code from the Click method of SFCodePageButton that adds the desired code to each page:

```
local lcCode, ;
  loPage, ;
  lcCurrentCode
lcCode = 'This.Parent.ShowMenu()' + chr(13)
for each loPage in Thisform.oObject.Pages
  lcCurrentCode = loPage.ReadMethod('RightClick')
  if not lcCode $ lcCurrentCode
    loPage.WriteMethod('RightClick', lcCurrentCode + ;
      iif(empty(lcCurrentCode), '', chr(13)) + lcCode)
  endif not lcCode $ lcCurrentCode
next loPage
wait window 'Code added to each page' timeout 2
```

To add these new builders to your system, copy SFBUILDERS.VCX and VCT (found with the download files for this month's article) to the WIZARDS subdirectory of the VFP directory or some other directory in your VFP path. Then open the BUILDERD table and APPEND FROM the NEWBUILDERS table, add Builder and BuilderX properties to your classes, and enter = HOME() + "Wizards\BuilderD,BuilderDForm" into BuilderX.

But Wait, It Gets Better

As easy as it is to create a builder using BuilderD, a phrase I've heard FoxPro guru Andy Griebel use occurs to me: "It's too hard". Wouldn't it be wonderful if there was a visual front-end to BUILDERD.DBF, along the same lines as the forms we provide to our users? After all, we don't just give them a browse window and tell them to start entering data (yeah, I know we're developers and don't need no stinkin' forms, but follow me on this). What I'm talking about is something that builds builders; yes folks, a builder builder. In fact, let's use BuilderD itself to build the builder builder (does that make it a builder builder builder?). That's what we'll do next month. Until then, have fun with BuilderD.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit and Stonefield Query. 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 the 1997 and 1998 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). He can be reached at dhennig@stonefield.com.