Click & Retrieve Source CODE!

VB3 Forms on the Fly

BY MICHAEL C. AMUNDSEN

Bring real code reuse into your database apps. Use control tables to instantiate a generic form and build the specific forms you need.

After years of coding and recoding clients' custom data-entry forms, I've finally given up on the whole thing. In the process I've halved development time, improved data-entry screen consistency and usability, and slashed application training and support costs.

I rarely do custom forms anymore, because I provide a forms generator in all my apps. This way, one data-entry form is used by almost every data-entry screen in the application. If clients need a special form, I modify the generic one with control tables—that is, control information stored in INI files. Clients appreciate the benefits of the single-form approach, especially because the control table options give them nearly all the customization they ask for.

If you take the time to design and code a forms generator for your data-entry screens, you can call just one function that reads the data source, builds the form, and handles all the database update functions with a minimum of code. With a forms engine in place, a single line of code can create a fully functional data-entry screen (see Figure 1):

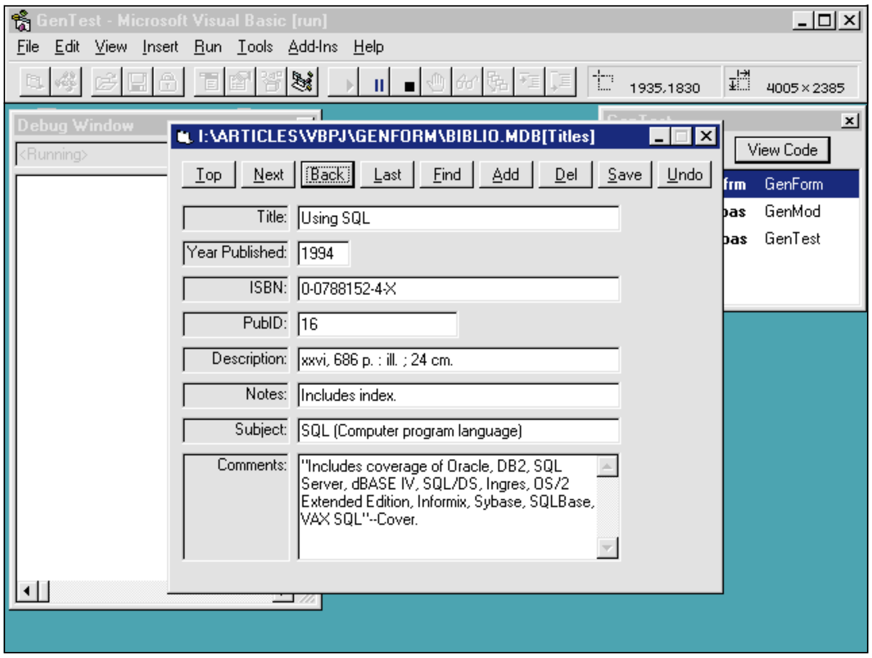
```
nResult = GenForm("biblio.mdb", _
   "Titles",0)
```

And when you add a new field to the data table it pops up on all the necessary data-entry screens without any changes to your VB code. You can string together calls to GenForm to create a multilevel data-entry application, or even incorporate the function into the menus of an MDI form that lets users open more than one data-entry form at once.

On the down side, boilerplate data-entry forms may not seem very "user-centric" to you. In his book *About Face*, Alan Cooper makes a strong case for programming and designing apps that do exactly what users want—that's the idea of the GUI. My GenForm system falls short of that ideal, but users do appreciate the consistency and predictability it gives them.

I've used the GenForm engine on site

Senior consultant Mike Amundsen specializes in system design and integration for the transportation industry. He cowrote the SAMS book Teach Yourself Database Programming with Visual Basic 4 in 21 Days, *contributed to SAMS'* Visual Basic 4 Unleashed, *and is a contributing editor on Cobb's* "Inside Visual Basic for Windows" *newsletter. You can reach him on CompuServe at 102461,1267.*


GenTest - Microsoft Visual Basic [run]
File Edit View Insert Run Tools Add-Ins Help
1935,1830
4005 x 2385
Debug Window
<Running>
I:\ARTICLES\VBPJ\GENFORM\BIBLIO.MDB[Titles]
Top Next Back Last Find Add Del Save Undo
View Code
frm GenForm
bas GenMod
bas GenTest

Title: Using SQL
Year Published: 1994
ISBN: 0-0788152-4-X
PubID: 16
Description: xxvi, 686 p. : ill. ; 24 cm.
Notes: Includes index.
Subject: SQL (Computer program language)
Comments: "Includes coverage of Oracle, DB2, SQL Server, dBASE IV, SQL/DS, Ingres, OS/2 Extended Edition, Informix, Sybase, SQLBase, VAX SQL"--Cover.

FIGURE 1 ***One Line of Code Built This Form.*** *A single call to the GenForm routine opens a database, loads a record set, populates a form with all the needed controls, enables the command buttons, and waits for user input. You get practical code reuse without having to master the intricacies of object-oriented programming.*

to quickly build fully functional data-entry screens for clients to test after initial data designs are developed. Using the forms engine this way offers a great opportunity for users and developers to refine their thinking about the data that is needed, and the order that fields should appear in, before approving the final database design. Usually my clients are very impressed with the screens and appreciate being able to test the data model before committing to a major development project.

But ultimately you have to please the one who is paying for the project. And at the end of the day a forms engine lets you do decent data-entry forms quicker, more accurately, and cheaper—especially at three points in the life of a typical project:

• Initial development stage: Generate a quick series of screens to show clients how things will work. Rapid application development at its finest!
• Final acceptance stage: Customers want you to modify the standard screens a bit. Being able to do this from control tables may mean making a completion date that would otherwise slip.
• Maintenance stage: A year later you need to add a new field to a table. Just update some control tables and all the screens that call the new data table get the new field automatically.

To get all these benefits, you just need to learn how to use data table information to generate the input controls needed to populate a data-entry form. The code that does this automatically adds check boxes, date fields, formatted currency fields, and free-form memo fields as well as simple text and number fields to data-entry screens. You also need the know-how to write routines that automatically size and space these controls as needed. In addition, such routines let you add customizing features to the generic form, including methods for handling hidden, required, or display-only fields. Finally, you need the ability to store the layout data in control tables for future use, and to use these tables to create customized data-entry screens for specific users—or for an entire installation site.

DESIGNING A RUNTIME FORMS ENGINE

Before you start coding a runtime forms engine, look at some basic design parameters, along with the major operations that a forms engine handles. Design parameters determine how the forms engine will interact with your VB programs, as well as how the engine generally behaves when it creates data-entry forms and processes user-form interactions.

CONTINUED ON PAGE 64.

VB3

```
Option Explicit                                 ' global vars
Global db As Database                           ' single db object
Global ds() As Dynaset                          ' data set object array
Global frmList() As Form                        ' display form array
Global cDynaset() As String                     ' data set name array
Global nFrmCount As Integer                     ' for forms array
Global nDBOpen As Integer                       ' open db flag
Global nErr As Integer                          ' error flag
Global Const DB_FIXEDFIELD = &H1                ' Field Attributes
Global Const DB_VARIABLEFIELD = &H2
Global Const DB_AUTOINCRFIELD = &H10
Global Const DB_UPDATABLEFIELD = &H20
Global Const DB_BOOLEAN = 1                     ' Field Data Types
Global Const DB_BYTE = 2
Global Const DB_INTEGER = 3
Global Const DB_LONG = 4
Global Const DB_CURRENCY = 5
Global Const DB_SINGLE = 6
Global Const DB_DOUBLE = 7
Global Const DB_DATE = 8
Global Const DB_TEXT = 10
Global Const DB_LONGBINARY = 11
Global Const DB_MEMO = 12
Function GenForm (cDB As String, cRS As String, nMode As Integer) As Integer
  On Error GoTo GenFormErr
  nErr = False
  GenForm = -1                                  ' assume an error occurs
  If nDBOpen = False Then
    OpenDB cDB                                  ' open database
  End If
  If nErr = False Then
    GenForm = LoadForm(cRS, nMode)              ' load recordset
  End If
  GoTo GenFormExit
GenFormErr:
  MsgBox "Err:" & Str(Err) & "[" & Error(Err) & "]", 0, "GenMain Error"
  nErr = True
  nDBOpen = False
  GenForm = -1
  Resume Next
GenFormExit:
End Function
Function LoadForm (cRecordSource As String, nMode As Integer) As Integer
  On Error GoTo LoadFormErr
  ' get recordsource and start a new form
  nFrmCount = nFrmCount + 1
  ReDim Preserve frmList(nFrmCount) As Form
  ReDim Preserve cDynaset(nFrmCount) As String
  ReDim Preserve ds(nFrmCount) As Dynaset
  cDynaset(nFrmCount) = cRecordSource
  Set frmList(nFrmCount) = New frmGenForm
  Load frmList(nFrmCount)
  frmList(nFrmCount).Show nMode
  LoadForm = nFrmCount
  GoTo LoadFormExit
LoadFormErr:
  MsgBox "Err:" & Str(Err) & "[" & Error(Err) & "]", 0, "LoadForm Error"
  nErr = True
  LoadForm = -1
  Resume Next
LoadFormExit:
End Function
Sub OpenDB (cDBF As String)
  On Error GoTo OpenDBErr
Set db = OpenDatabase(cDBF, False, False)  ' open new db
  If nErr = False Then
    nDBOpen = True
  Else
    nDBOpen = False
  End If
  GoTo OpenDBExit
OpenDBErr:
  MsgBox "Err:" & Str(Err) & "[" & Error(Err) & "]", 0, "OpenDB Error"
  nErr = True
  Resume Next
OpenDBExit:
End Sub
```

LISTING 1 ***Declare Your Array.*** *GENMOD.BAS contains the global variables, constants, and all three high-level routines needed to invoke the runtime forms engine.*

CONTINUED FROM PAGE 61.

Once you understand the basic design specs you can elaborate on them if you need to.

With this method, one call to one routine generates fully functional forms. The routine accepts two parameters: the name of the database to open and the record source to load. I use the Microsoft Jet database, but you could extend this forms engine to work with any valid data source, including ODBC or other desktop DBMSs.

The record source can have any valid Microsoft Jet table name, querydef, or Microsoft Jet SQL SELECT statement that creates a data set. I'll use Microsoft Jet dynaset objects for all data sets, but you could enhance the routine to allow the creation of snapshot or table objects.

Our basic engine design will produce only one-page data-entry forms—no tabbed dialogs or "More>>" buttons. A single set of command buttons (Top, Next, Back, and Last) will let users navigate the data set. And a simple Find button will let users enter any valid SQL WHERE clause (again, you can elabo-

VB3

```
Option Explicit
Dim InpFld() As Control   ' form/field stuff
Dim InpLbl() As Control
Dim btnText(9) As String
Dim nFlds As Integer
Dim nTop As Integer
Dim nAdd As Integer
Dim nForm As Integer
Const nLblLeft = 120      ' constants for form
Const nLblHigh = 300
Const nLblWide = 1200
Const nTxtLeft = 1400
Const nTxtWide = 3600
Const nTxtHigh = 300
Const nMmoWide = 3600
Const nMmoHigh = 1200
Const nMmoLeft = 1400
Const nBtnWide = 600
Const nBtnHigh = 300
Const nBtnSpace = 60
Sub Form_Activate ()
  If nErr = True Then
    Unload Me
  End If
End Sub
Sub Form_Load ()
  On Error GoTo FormLoadErr
  nErr = False            ' create a dynaset
  nForm = nFrmCount
  nTop = 600
  MakeData                ' load dataset
  MakeFields              ' load input controls
  MakeForm                ' finish off form
  LayoutLoad              ' get old layout
  RecRead                 ' read first record
  GoTo FormLoadExit
FormLoadErr:
  MsgBox "Err:" & Str(Err) & " [" & Error(Err) & "]", _
    0, "FormLoad Error"
  nErr = True
  Resume Next
FormLoadExit:
End Sub
Sub Form_Unload (Cancel As Integer)
  On Error Resume Next
  Me.WindowState = 0
  LayoutSave
  ds(nForm).Close
  Unload Me
End Sub
```

LISTING 2 ***Create a Jet Dynaset Object.*** *The form-level declarations and Form_ events for GENFORM.FRM provide a code-reuse project for forms. The Form_Load event shows the main operations required to build a form at run time.*

rate on this if you need to). Finally, to update the data set contents, the form will contain buttons such as Add, Delete, Save, and Undo.

This forms engine can also store input-form details such as the location and type of controls used and the data fields bound to those controls. And the engine can capture details about each field; for example, it can spot enabled controls and whether they should be visible on the input form. Store all this form information in a modifiable ASCII text file so you can customize data-entry form contents and behavior.

Obviously you need to invest some programmer-hours to build your library of data-entry screen-generating routines. But you can amortize this effort across multiple VB projects. To make it easy, organize your project into two members: the GENFORM.FRM form and the GENMOD.BAS module. The form holds all form-level code and variables needed to create data-entry forms. The module holds a handful of global-level constants and variables and the high-level routines needed to make calls to the form.

GENMOD.BAS needs to declare an array of forms and dynasets at the global level. Then you can run multiple forms at once. The engine also needs to determine the data field types in each field of the record source. So you also need a set of declared constants that define all possible data field types (see Listing 1).

GENMOD.BAS also contains three high-level routines for invoking the forms engine. The first, GenForm, invokes the runtime engine and calls two other routines: OpenDB, which uses the first parameter passed to GenForm as the database file to open, and LoadForm. LoadForm adds new members to several arrays, then creates, loads, and shows a new data-entry form.

Making all these calls work requires a generic VB form object that contains all the required input controls and command buttons, VB code for responding to user actions on the form, and routines for determining types of input controls needed and for positioning these controls properly on the form.

USING CONTROL ARRAYS AT RUN TIME

To create the generic form, start by adding a set of controls to the form object at design time. You'll need at least one control for each data type on the data-entry form, including text, numeric, date, currency, memo, and Boolean (True/False) data fields. Because you're building a runtime engine, you won't know just how many and what type of input controls you need until the program is already running. Fortunately, you can use the VB control array—a perfect built-in programming paradigm for just such a situation.

VB can create control arrays at run time. Just add one instance of each required control to the form at design time. Then, when the program starts, you can use VB code to add additional copies (instances) of any control already placed on the form.

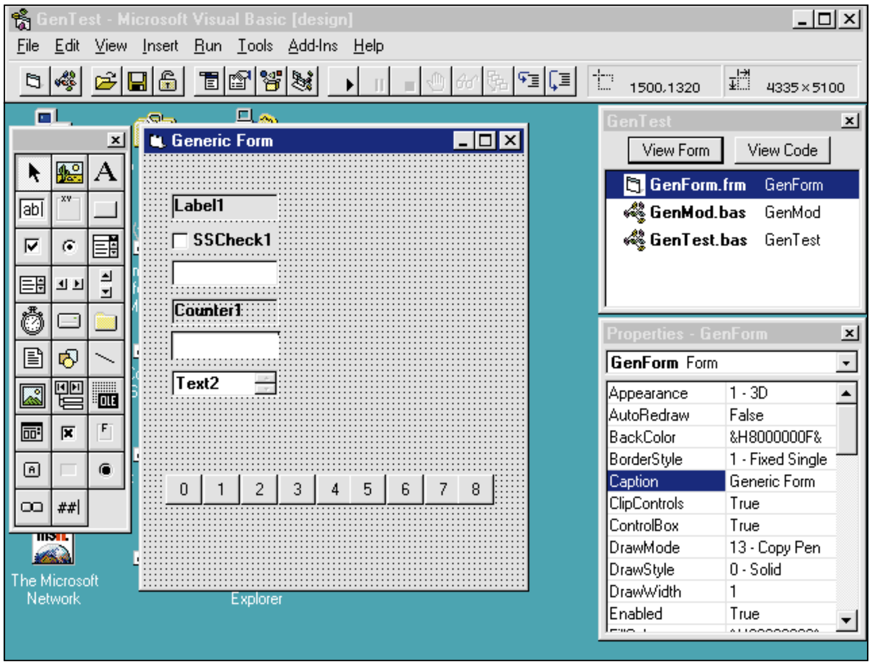
I need to warn you that things can get tricky here. You can add a control at run time only if at least one instance was added to the form at design time. And that instance must already belong to a control array.

To create a control array, start VB and/or load a new project. Add a text box control to your project by double-clicking on the text-box icon in the VB toolbox window. Select the text box on your form by clicking on it once to give it focus. Note the default name that VB assigns to the control ("Text1"). Select the Properties box, set the Index to zero, and you've got a control array.

With a set of control arrays for the form you can load new instances of any needed control at run time, safely generating data-entry forms that contain as many copies of any control you need. GENFORM can handle text, numeric, date, currency, memo, and Boolean fields. You need a control array for each unique field type—except for date and currency fields. Use the masked edit control for those. I've provided a list of data-field types matched to their corresponding input controls (see Table 1). You'll notice one special case on the list: the COUNTER data type. Jet lets users define an auto-incrementing field. You can use this as a unique, nonupdatable key field. The generic form can display the counter data while preventing users from editing that field.

You need one last control array for the generic input form: command buttons. This gives the form navigation and data update capability. Add a single array of nine command buttons to the form at design time, because the size of this array won't change.

Without control arrays you can certainly add nine command buttons to the project and place any needed code behind each button. But using control arrays here produces VB code you can read and maintain



GenForm.frm	GenForm
GenMod.bas	GenMod
GenTest.bas	GenTest

Appearance	1 - 3D
AutoRedraw	False
BackColor	&H8000000F&
BorderStyle	1 - Fixed Single
Caption	Generic Form
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen
DrawStyle	0 - Solid
DrawWidth	1
Enabled	True

FIGURE 2 ***Copying Controls.*** *GENFORM.FRM handles adding all required control arrays at design time. By creating a set of control arrays for the form, you can now load new instances of any control at run time. This form generated the data-entry screen shown in Figure 1.*

VB3

```
Sub MakeFields ()
  Dim x As Integer
  Dim lDbType As Long
  On Error GoTo MakeFieldsErr
  ReDim InpFld(nFlds) As Control
  ReDim InpLbl(nFlds) As Control
  For x = 0 To nFlds
    lDbType = ds(nForm).Fields(x).Type
    Select Case lDbType
      Case Is = DB_BOOLEAN
        FldBoolean x
      Case Is = DB_MEMO
        FldMemo x
      Case Is = DB_SINGLE
        FldNumber x
      Case Is = DB_DOUBLE
        FldNumber x
      Case Is = DB_LONG
        FldNumber x
      Case Is = DB_INTEGER
        FldNumber x
      Case Is = DB_CURRENCY
        FldCurrency x
      Case Is = DB_BYTE
        FldNumber x
      Case Is = DB_DATE
        FldDate x
      Case Else
        FldText x
    End Select
    InpFld(x).Visible = True
    InpFld(x).TabIndex = x
    If ds(nForm).Fields(x).Attributes And _
      DB_UPDATABLEFIELD Then
      InpFld(x).Enabled = True
    Else
      InpFld(x).Enabled = False
    End If
  Next x
  GoTo MakeFieldsExit
MakeFieldsErr:
  MsgBox "Err:" & Str(Err) & "[" & Error$ & "]", 0, _
    "MakeFields Error"
  nErr = True
  Resume Next
MakeFieldsExit:
End Sub
Sub MakeLabels (x As Integer, nHigh As Integer)
  If x <> 0 Then                        ' set up
label for field
    Load Label1(x)
  End If
  Set InpLbl(x) = Label1(x)
  InpLbl(x).Top = nTop
  InpLbl(x).Height = nHigh
  InpLbl(x).Left = nLblLeft
  InpLbl(x).Width = nLblWide
  InpLbl(x).Alignment = 1
  InpLbl(x).Visible = True
  InpLbl(x).FontBold = False
  InpLbl(x).BackStyle = 0
  InpLbl(x).Caption = ds(nForm).Fields(x).Name & ":"
End Sub
Sub FldText (x As Integer)
  If x <> 0 Then
    Load Text1(x)
  End If
  Set InpFld(x) = Text1(x)
  InpFld(x).Top = nTop
  InpFld(x).Height = nTxtHigh
  InpFld(x).Left = nTxtLeft
  InpFld(x).Width = nTxtWide
  InpFld(x).FontBold = False
  InpFld(x).MaxLength = ds(nForm).Fields(x).Size
  InpFld(x).Tag = ds(nForm).Fields(x).Name
  MakeLabels x, nTxtHigh
  nTop = nTop + nTxtHigh + 90
End Sub
```

LISTING 3 ***What Kind of Control Do You Need?*** *Make and Fld routines read the Dyanset field collection and generate input and label controls for your data-entry form. GENFORM.FRM's MakeFields routine uses the Fields.Type value to decide what kind of input control to add to the form.*

Field Type	Control Type
BOOLEAN	Checkbox control
INTEGER	Text control
LONG	Text control
CURRENCY	Masked Edit control
SINGLE	Text control
DOUBLE	Text control
DATE	Masked Edit Control
TEXT	Text control
MEMO	Text control (multiline with scroll bars)
COUNTER	Label control

TABLE 1 ***What's Your Control Type?*** *Here's each data-field type and the corresponding input control that you'll use on your form in the GENFORM code reuse project.*

more easily. Instead of needing nine separate routines, all button clicks of a control array arrive at a single routine. Now you can code all button actions in a Select Case…End Select structure (see Figure 2).

DYNASET OBJECT FIELDS COLLECTIONS

Once you've populated the form with controls, add code to use the record source parameter passed in the GenForm function to create a Microsoft Jet dynaset object. Then you can load the field information and build the form for data entry (see Listing 2).

The Form_Load event shows the main operations needed to build a form at run time. For now, note that the MakeData routine performs the steps needed to create a new Dynaset object. Once created, the Dynaset lets you get information about the number, types, and sizes of the data fields. You'll use this information to create the input controls needed for the data-entry form.

You can get detailed information about fields in a dynaset with its fields collection. This contains information about every field in the data object, including field name, type, size, and any special data field attributes, such as whether it's auto-incrementing or nonupdatable.

GENFORM.FRM's MakeFields routine uses the Fields.Type value to pick the input control to add to the form. This routine also calls the MakeLabels routine to create an onscreen prompt for the input control (see Listing 3). The MakeData, MakeFields, and MakeLabels routines call various routines that you use to size and position the input controls.

While you're working with this set of routines, note the use of the Type property of the fields collection in the Select Case…End Case loop of the MakeFields routine. The Type property returns an integer value that's compared (using the "Case Is =" construct) to global-level constants defined in GENMOD.BAS.

The results determine the type of field needed for the input form. The same routine uses the Attributes property of the field collection to see if a field is updatable. Some data sets that result from SQL JOINS will have one or more nonupdatable fields. This usually happens when you build a data set that contains fields related to other fields in other physical tables. You'll also get one or more nonupdatable fields when you perform SQL SELECTS on a subset of a table located on the "many" side of a defined one-to-many relationship. GENFORM disables nonupdatable fields on the input form.

The code for the FldText routine shows what you need to do to add a new control to the form at run time. First the routine checks to see if this is the first control on the form (x=0). If not, the routine loads a new instance of the control before setting the local control array (InpFld). Then it establishes the control's size and location, based on predefined form-level constants.

The Top property is based on the current location of the control on the form. Controls are added down the left side of the form in a single row. As you add more controls, the routine updates the nTop variable to reflect the next available location on the form. Finally, it updates the control's MaxLength and Tag properties using values directly from the dynaset's fields collection. Before exiting the routine, MakeLabels is called to create an on-screen label to the left of the input control.

Once the controls are added to the form, GENFORM knows the data-entry screen's final size. Then GENFORM adds captions to the command buttons and positions them on the on form, using the MakeBtn and MakeForm routines, then displays the form for the user (see Listing 4).

To make it all work you need the set of routines that handles reading and writing the data set and responding to user clicks on the button bar. Three routines move data between the controls and the database: RecInit (to initialize controls), RecRead (to move data from data set to controls), and RecWrite (to move data from controls to data set). You use these three routines in the Select Case…End Select loop in the cmdBtn_Click event. The cmdBtn_Click event code handles all user button-bar activity (see Listing 5).

You need one more record-handling routine to take care of user Find operations: RecFind. It's pretty basic but it works. It asks users for any valid SQL Where clause to apply against the data set. You can enhance this if you need a more powerful search tool.

USING GENFORM IN YOUR PROGRAMS

Now that the nuts and bolts are in place, take the GENFORM engine for a quick spin. Start a new project. Make sure you have the Masked Edit control and the Sheridan 3D controls in your VB Pro toolbox window. With VB4 make sure you've referenced the correct data-access object library as well. Then load GENMOD.BAS and GENFORM.FRM into your current project and you're set.

Save this project (for future GENFORM work) as GENFORM.VBP. Next time you need to start a new project using GENFORM, copy this project to a new name and start from there.

You can write a simple GENFORM project with fewer than a dozen lines of code, using your new project template. First, add a BAS module to the project. Create a Sub Main routine and add this code to the Main routine:

```
Public Sub Main()
   Dim x as Integer
   Dim vbModelessForm as Integer
   Mid vbModalForm as Integer
   vbModelessForm = 0
   vbModalForm = 1
   ' be sure to point to the
   ' biblio.mdb on your system
   x = GenForm("biblio.mdb", _
      "Authors",vbModelessForm)
End Sub
```

Change the Startup Form to Sub Main and save the project with a new name (GENTEST). Now run the project—you've got a data-entry screen ready to use. Adding more lines to the Sub Main routine will let you open several data sets at once.

The GenForm function returns a pointer into the frmList forms array. If you want to, you could use this pointer to get other information from the form or its controls:

```
Public Sub Main()
Dim x as Integer
   Dim vbModelessForm as Integer
   Mid vbModalForm as Integer
   vbModelessForm = 0
   vbModalForm = 1
   ' be sure to point to the
   ' biblio.mdb on your system
   x = GenForm("biblio.mdb", _
      "Authors",vbModelessForm)
   x = GenForm("biblio.mdb", _
      "Publishers", vbModelessForm)
   x = GenForm("biblio.mdb", _
      "Titles", vbModelessForm)
End Sub
```

In this example, GENFORM loads each data set and shows you a data-entry form for each call. You can move between these forms, performing data-entry as needed.

The version of GENFORM described here treats all forms as modeless dialogs. You can have several forms up at the same time and switch between them whenever you want. But on occasion you may want to put up a series of modal forms. This code example shows you how:

```
Public Sub Main()
   Dim x as Integer
   Dim vbModelessForm as Integer
   Mid vbModalForm as Integer
   vbModelessForm = 0
   vbModalForm = 1
   ' be sure to point to the
   ' biblio.mdb on your system
   x = GenForm("biblio.mdb", _
      "Authors",vbModalForm)
   x = GenForm("biblio.mdb", _
      "Publishers", vbModalForm)
   x = GenForm("biblio.mdb", _
      "Titles", vbModalForm)
End Sub
```

This minor change can help you build data-entry applications that use cascading data-entry forms. GENFORM can also make itself useful as a child of an MDI form. You can build a simple MDI form containing a menu that lets users call up other forms for data entry. Each menu option can invoke a call to the GENFORM engine. To accomplish this you need to make one change to the GENFORM.FRM file: set the MDIChild property to True.

Unfortunately, because this property is read-only at run time, you can't set it using VB code. But you can make a file copy of GENFORM.FRM, adjust it for MDI use, then save it under the name GENMDI.FRM. Then if your project calls for an MDI version of GENFORM, you can load the GENMDI.FRM. Use the original GENFORM.FRM for projects that won't run under an MDI form. My GENTMDI.VBP uses the GENMDI.FRM version of the forms engine as a child for an MDI form (see Figure 3).

When you use GENFORM in this way, it enables you to simultaneously call up more than one instance of the same form and dynaset—excellent for users who need to look up one master record while updating a second one in the same table.

Now that you can use GENFORM in VB projects in at least three different ways (modeless dialogs, modal cascading forms, and MDI child forms), add a few bells and whistles.

SOME BELLS AND WHISTLES

Using Microsoft Jet to create SQL SELECT statements, you can customize displays by creating dynasets with only the fields your users really need. For example, perhaps a user wants to enter and review data in the Publishers table of BIBLIO.MDB. The user would need two input screens: one for name and address information and another for editing the Comments field. Instead of creating two separate data forms in your project, you could simply use two different SQL statements as the RecordSource parameters in your calls to the GENFORM engine. In effect, you are using SQL statements to design your data-entry forms:

```
' When using SQL to design forms be
' sure to point to the biblio.mdb on
' your system.
   x = GenForm("biblio.mdb", " _
      Publishers",0)
' full screen
cSelect="SELECT PubID, Name,Comments _
   FROM Publishers"
x = GenForm("biblio.mdb", cSelect, 0)
' comment screen
```

You can also make a forms engine pay off by adding the ability to store form layouts for modification and later use. GENFORM has two routines to do just that. LayoutSave saves the layout details of a data-entry form upon exit, and LayoutLoad can load these details when the form is first created. The ASCII text file format lets you use Notepad or any other ASCII editor to inspect and change settings. As a result, you can create custom screens without adding VB code. This version of GENFORM saves and restores some key information for each form:

- Form caption, size, and location.
- Input control location and size.
- Label captions.
- Label control location and size.

The routine saves data in a file that's named for the RecordSource and that ends with an FRL file extension (see Listing 6). You can use the file to change the location of any control and even change the prompt captions. Once you've done this, all users who call up the form will see the modifications. Now you can have customers all over the country, each with the same source code but distinct data-entry forms—all with a simple ASCII-control file.

Using control files also lets you make selected fields read only by setting their

VB3

```
Sub MakeBtns ()
  btnText(0) = "&Top"            ' load text for command buttons
  btnText(1) = "&Next"
  btnText(2) = "&Back"
  btnText(3) = "&Last"
  btnText(4) = "&Find"
  btnText(5) = "&Add"
  btnText(6) = "&Del"
  btnText(7) = "&Save"
  btnText(8) = "&Undo"
End Sub
Sub MakeForm ()
  Dim x As Integer
  Me.Width = (9 * nBtnWide) + (9 * nBtnSpace) + 240
  ' set up form
  Me.Height = nTop + 600
  Me.Caption = db.Name & "[" & ds(nForm).Name & "]"
  MakeBtns
  ' load button captions
  For x = 0 To 8
  ' place buttons on form
    cmdBtn(x).Top = 120
    cmdBtn(x).Width = nBtnWide
    cmdBtn(x).Height = 300
    cmdBtn(x).Left = 120 + (nBtnWide * x) + (nBtnSpace * x)
    cmdBtn(x).Caption = btnText(x)
    cmdBtn(x).TabIndex = x + nFlds + 1
  Next x
  Me.Top = (Screen.Height - Me.Height) / 2
' center form on screen
  Me.Left = (Screen.Width - Me.Width) / 2
End Sub
```

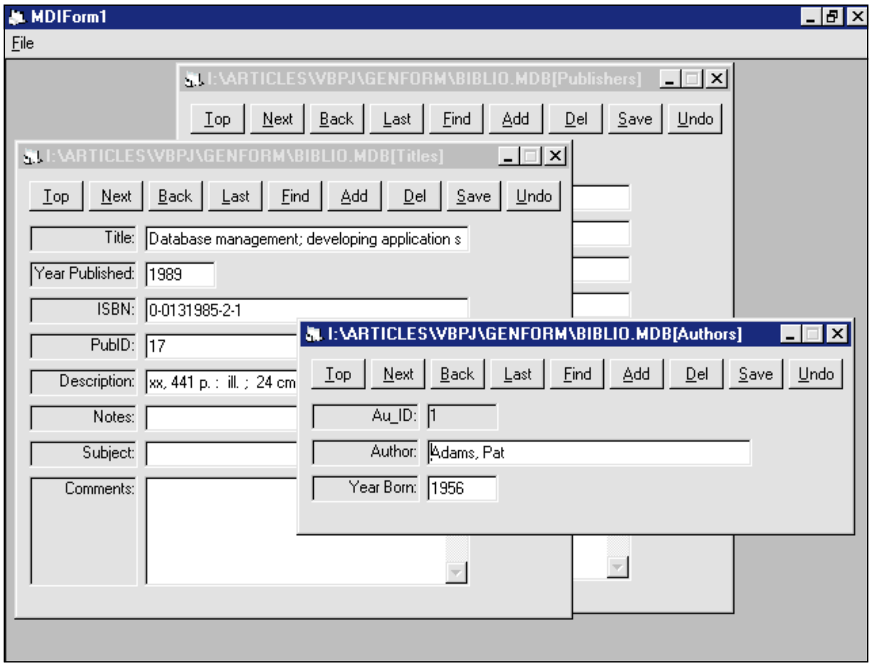
LISTING 4 ***Position Your Buttons.*** *The MakeBtns and MakeForm routines set captions for the button bar and for the final dimensions of the input form. They also position the command buttons on the form before displaying it for the user.*

Enabled property to False in the FRL control file. This helps with multiuser sites where many users view data, but only a handful have permission to modify it.

Later on you can add new fields to data tables, using a forms engine to cut down on the retro-programming of old screens. You just need to update physical tables with the new field information. Then the new fields will show up on screen the next time someone starts the form. You may need to modify some control tables to account for the new field, but at least you don't have to add a lot of new VB code to your old programs.

FUTURE ENHANCMENTS

This GENFORM engine can do useful work for you, but a number of enhancements could improve its overall usefulness and versatility. For example, even though GENFORM lets you modify form layouts and store them in the FRL files, you have to perform this modification using an ASCII text editor. It would be nice if you could load the form as if you were about to edit records, but instead switch to "Form Edit" mode, then use


MDIForm1
File
I:\ARTICLES\VBPJ\GENFORM\BIBLIO.MDB[Publishers]
Top
Next
Back
Last
Find
Add
Del
Save
Undo
I:\ARTICLES\VBPJ\GENFORM\BIBLIO.MDB[Titles]
I:\ARTICLES\VBPJ\GENFORM\BIBLIO.MDB[Authors]

Title: Database management; developing application s
Year Published: 1989
ISBN: 0-0131985-2-1
PubID: 17
Description: xx, 441 p. : ill. ; 24 cm
Notes:
Subject:
Comments:

Au_ID: 1
Author: Adams, Pat
Year Born: 1956

FIGURE 3 ***It Works with MDI, Too.*** *GENMDI.FRM uses the forms engine as a child for MDI forms. Using it, you can build a simple MDI form containing a menu that lets users call up other forms for data entry.*

VB3

```
Sub cmdBtn_click (Index As Integer)
  Dim x As Integer
  Dim cMsg As String
  On Error GoTo cmdBtnErr
  Select Case Index              ' handle button
pushers
    Case Is = 0                  ' top
      RecWrite
      ds(nForm).MoveFirst
      RecInit
      RecRead
    Case Is = 1                  ' next
      RecWrite
      If ds(nForm).EOF Then
        ds(nForm).MoveLast
      Else
        ds(nForm).MoveNext
      End If
      RecInit
      RecRead
    Case Is = 2                  ' previous
      RecWrite
      If ds(nForm).BOF Then
        ds(nForm).MoveFirst
      Else
        ds(nForm).MovePrevious
      End If
      RecInit
      RecRead
    Case Is = 3                  ' last
      RecWrite
      ds(nForm).MoveLast
      RecInit
      RecRead
    Case Is = 4                  ' find
      RecFind
      RecInit
      RecRead
    Case Is = 5                  ' add new
      RecWrite
      nAdd = True
      ds(nForm).AddNew
      RecInit
    Case Is = 6                  ' delete
      ds(nForm).Delete
      If Not ds(nForm).EOF Then
        ds(nForm).MoveNext
      Else
        ds(nForm).MoveLast
      End If
      RecInit
      RecRead
    Case Is = 7                  ' update
      RecWrite
      RecInit
      RecRead
    Case Is = 8                  ' restore
      nAdd = False
      If Not ds(nForm).EOF And Not ds(nForm).BOF Then
        RecInit
        RecRead
      End If
  End Select
  Select Case Index              ' handle button
enable/disable stuff
    Case Is = 5
      For x = 0 To 6
        cmdBtn(x).Enabled = False
      Next x
      cmdBtn(7).Enabled = True
      cmdBtn(8).Enabled = True
    Case Else
      For x = 0 To 8
        cmdBtn(x).Enabled = True
      Next x
  End Select
  GoTo cmdBtnExit
cmdBtnErr:
  cMsg = "err:" & Str(Err) & "[" & Error$ & "]"
  MsgBox cMsg, 0, "cmdBtn Error"
  nErr = True
  Resume Next
cmdBtnExit:
End Sub
```

LISTING 5 ***Deal with Button-Bar Events.*** *You need a routine to handle user actions on the button bar, such as the cmdBtn_Click event code listed here. This routine handles all the actions that a user will need to add, edit, delete, or locate records in the table.*

your mouse to move fields to their desired locations and toggle the Visible and Enabled properties of a field using popup menus.

Once you have the screen as you like it, save the modified control values for future use. Mostly you'll set up context menus (right-mouse popups) that you invoke using some unlikely combination of control keys and mouse clicks over an unused portion of the form.

If you're working with data tables that build in a lot of referential integrity, MS Jet can handle most data validation for you. But if you need additional data-entry validation, you may want to consider building a set of validation routines into the GENFORM engine.

To add validation routines, establish a set of default validation requirements (such as numeric data only, capital letters only, and so on) and create a set of routines for the engine to call as needed. Set values in the Tag property of the control to indicate the validation routines you need to run, and add code to the forms that checks the Tag property and calls the routines as the user performs data entry.

VB3

```
Form.Title=D:\VB3\BIBLIO.MDB _
  [Authors]
Form.Top= 3975
Form.Left= 3375
Form.Height= 1980
Form.Width= 6180
Au_ID.Number= 0
Au_ID.FldLeft= 1400
Au_ID.FldTop= 600
Au_ID.FldHeight= 300
Au_ID.FldWidth= 800
Au_ID.FldVisible=-1
Au_ID.FldEnabled=-1
Au_ID.LblCaption=Au_ID:
Au_ID.LblLeft= 120
Au_ID.LblTop= 600
Au_ID.LblHeight= 300
Au_ID.LblWidth= 1200
Au_ID.LblVisible=-1
```

LISTING 6 ***Real Reuse Power.*** *FRL files such as AUTHORS.FRL, created by GENFORM, let you make a forms engine payoff. They add the ability to store form layouts for modification and later use.*

For more refinement, add flags for form-level validations as well, plus optional command buttons at the bottom of the form to call user-defined routines, such as those that call additional GENFORM screens. Do this by defining a global array that contains captions for the optional buttons. Load these buttons at run time, like the other input controls. Build a single global Select Case…End Case structure to hold all calls to the user-defined routines. This way you can generate quick custom screens and add calls to specialized routines right on the same form.

For more sophisticated storage and retrieval of forms data, you'll want to modify my simple GENFORM. In multiuser settings try storing separate FRL files for each user, possibly in their system directories. This way individual users can see different versions of the same screen, depending on their security levels or user preferences.

Also, the ASCII format I've specified, while handy, should give way to using a database if you need scalability. This would keep the data close to the app, yet still let you provide custom sets based on user ID. And you could modify individual screens without having to update the FRL files on each and every workstation.

As you can see, it takes a bit of planning to construct a forms generator, even a simple one. But you'll be glad you made the investment when you need to get a big project out in a hurry. ☒