Expanding and Contracting Forms

Rick Jones and Stephanie Hudson

2.0 95 97 2000 DOWNLOAD

Building an effective user interface can be challenging, but it's also extremely rewarding when a few lines of code enhance the intelligence of the application. In this article, Stephanie Hudson and Rick Jones show you how a utility lets your forms expand or contract at the click of a button.

THIS article is written for anyone that continually receives Help desk feedback on the difficulty of working with a particular application area or form. As an application becomes more widely used, the importance of its user interface increases. Your attitude toward your UI often shifts from "as long as it gets me there" to "get me there in style." As developers, our creativity isn't only useful in tackling technical challenges, but more often, it plays a crucial part in packaging those solutions. Adding style to your interface does have an opportunity cost: While you're enhancing the UI, there many other things that aren't getting done. However, most investments in your UI can pay for themselves in reduced support, reduced training costs, and more accurate data entry. We devised a utility to make the process of providing one kind of enhancement as efficient as possible.

In any interface there are an infinite number of ways to present functionality to the user. In many cases, an application's functionality is straight forward. Typical examples are forms that allow a user to add or search for a record, or filter a report. However, there are other situations where the interface might need to combine distinctly different but related tasks. An example is a form that allows the user to filter a report while building and managing their own custom filters for that report. Another example is a system that allows users to both enter new information and maintain historical information for a group of entities.

With these kinds of forms, presenting all of the functionality on the form at the same time can be just plain confusing for users. In these cases, you might be showing too much of your form (as you know, they say less is more). Switching between forms might just increase an application's complexity and make using the two functions more difficult for the user. A useful solution is to initially present a form that shows the base or minimum functionality. A user can then expand the form to display more extensive functionality, usually by clicking on a button labeled "Options," "Advanced," "More," or just ">>."

Managing your form window

The method that you can use to create expanding and contracting forms is determined by the border type of your form. If your application uses a dialog border style then you're in luck, since that technique is the easiest to implement. In that scenario, simply place the "expanded view" controls at the bottom of your form, disable them by setting their Enabled property to False, and size the form window so that the controls are hidden. To reveal the expanded controls, all you need to do is change the form's window size, enable your controls and, optionally, disable the controls that you don't want the user to use in expanded mode. To return the form to its contracted state, you disable the expanded controls and shrink the form's window size to hide them.

However, if your application border is sizeable, then you'll need to use a different solution since your users can resize the form and view those pesky disabled controls. That's just not a marketable solution. When we ran into this very problem, we developed a technique that uses a control's tag property to hide controls that aren't required to be visible.

The following Expand _Contract utility has two purposes: to determine what mode the user is in and use that mode to set the form and controls' size, location, visibility, and Enabled property. A Boolean parameter passed to the routine signals in which mode the form should be displayed. A value of False indicates that the form should be in "Base" mode while True indicates "Expand" mode. Once called, the utility will size the window, hide and move unused "Expanded" controls, disable the "Base" controls (to target the user's attention) and, finally, reposition and enable the "Expanded" controls. The visual effect of the "Base" mode is a small window with all of the unused controls set to invisible and relocated to position 0,0 (see **Figure 1**). The visual effect of the "Expanded" mode is a larger window with disabled "Base" controls and visible "Expanded" controls

(see Figure 2). All the user has to do is press a command button to change the form's mode.

Mark your territory

The only preparation required when using the Expand _Contract utility is to store all size and location information (in twips) in the Tag property for the form and all of the controls. First, you need to manually store the "Expanded" form size in the form's tag property. The data is stored as a series of values separated by semi-colons and preceded by the word "Expanded." Since an Access form has several sections, you must store the height for each section, though you only need to store the width for the form as a whole. The format for the form's tag is as shown:

```
Expanded;expanded width;header height;
      detail height;footer height
```

A typical example for a form 2,000 twips wide with a header 400 twips high, a detail section 1,000 twips high, and an invisible footer section would look like this:

```
Expanded; 2000;400;1000;0
```

For each control, you only need to put "BasePane" or "ExpandedPane" in the Tag property to indicate whether or not they're to appear in the Base or Expanded view. A control that appears without change in both views doesn't require any setting in its Tag property.

After you've manually stored the forms as "Expanded" size and tagged each control with it's opening mode, you run the following code. This code will run through every control on the form and update each control's Tag property with the control's position on the form. To use the following code you need to download our Get Delimitated Value (GDV) utility which separates stored tag information. It's part of our sample database in this month's Source Code files (found at www.smartaccessnewsletter.com), or you can get it from our site at www.lunaconsulting.com—if you don't already use similar tag parsing code. The GDV code will take the control's tag information and separate it into readable chunks using a semi-colon to separate each value. A sample result is shown in Figure 3. Here's the routine:

```
Public Function Mark(frmMark As String)
Dim frm As Form
Dim ctl As Control
Set frm = Forms(frmMark)
For Each ctl In frm
  If GDV(ctl.Tag, 1) = "ExpandPane" Then
       ctl.Tag = "ExpandPane;" & ctl.Left & ";" & _
                 ctl.Width & ";" & ctl.Top & ";"
  End If
Next ctl
End Function
```

Expand your horizon

After you've marked your controls, you're ready to add the Expand_Contract functionality. The following function should be called from a command button Click event passing either True (Expand) or False (Contract), and the name of the form to process. The first lines of the routine set the expanded and base size for the form sections as constants, declare the routine's variables, and get a reference to the form whose name was passed to the routine in the strOpen variable:

Figure 1. The view of a "Base" form. All of the extended controls are invisible.

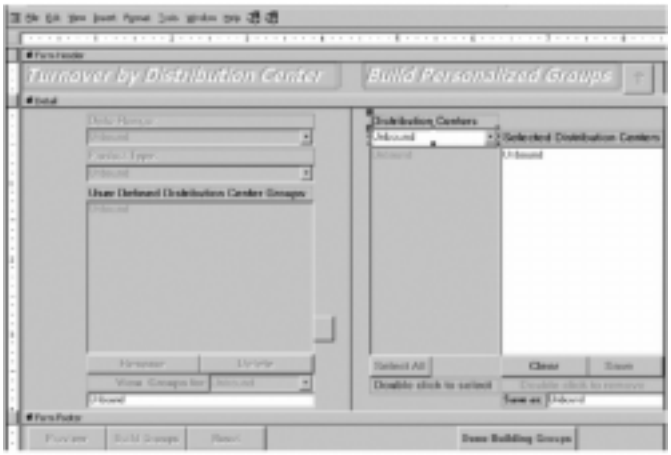

Figure 2. The view of an "Expanded" form. Note the disabled "Base" controls.

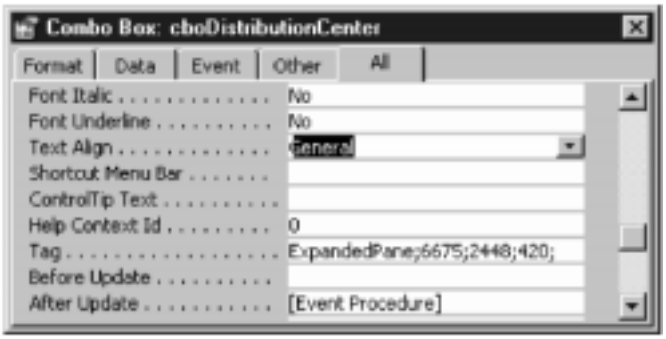

Figure 3. How an "'Expanded" control tag property should look.

```
Sub Expand_Collapse(bolExpandedPane As Boolean, _
        strOpen as String)
Const NarrowSize = 8.5
Const BaseSize = 4.6
Const BaseHeaderHeight = 1
Const BaseDetailHeight = 5
Const BaseFooterHeight = 2
Dim ctl As Control
Dim frm as Form
Set frm = [Forms](strOpen)
```

The routine then checks the first parameter passed to it (bolExpandedPane) to determine if the form is to be expanded or contracted. If the form is going to be contracted, the code gets the Expanded information from the form's Tag property (using the GDV utility) and sets the heights of the appropriate sections:

```
If bolExpandedPane Then
  If GDV(frm.Tag, 1) = "Expanded" Then
       frm.Width = GDV(frm.Tag, 2)
       frm.Section(acHeader).Height = GDV(frm.Tag, 3)
       frm.Section(acDetail).Height = GDV(frm.Tag, 4)
       frm.Section(acFooter).Height = GDV(frm.Tag, 5)
```

With the form resized, the code then loops through every control on the form. If the control is part of the Expanded view, the code pulls the information from the tag field to resize the control, and makes the control visible. If the control is flagged as part of the Base format, this code disables it. Any controls without a Tag setting are left alone:

```
       For Each ctl In frm
              If GDV(ctl.Tag, 1) = "ExpandPane" Then
                     ctl.Left = GDV(ctl.Tag, 2)
                     ctl.Width = GDV(ctl.Tag, 3)
                     ctl.Visible = True
              End If
              If GDV(ctl.Tag, 1) = "BasePane" Then
                     ctl.Enabled = False
              End If
       Next ctl
```

Not surprisingly, the code for handling the base view is similar to the previous example. Any controls that are part of the Expanded view are made invisible and moved to position 0,0 on the form. Controls that are part of the Base view are enabled. Finally, the constants at the top of the page are used to return the form's sections to their original size:

```
  Else
        For Each ctl In frm
              If GDV(ctl.Tag, 1) = "ExpandPane" Then
                     ctl.Visible = False
                     ctl.Left = 0
                     ctl.Width = 0
              End If
              If GDV(ctl.Tag, 1) = "BasePane" Then
                     ctl.Enabled = True
              End If
        Next ctl
        frm.Width = NarrowSize
        frm.Section(acHeader).Height = _
                     BaseHeaderHeight * pTwip
        frm.Section(acDetail).Height = _
                     BaseDetailHeight * pTwip
        frm.Section(acFooter).Height = _
                     BaseFooterHeight * pTwip
  End If
End If
End Sub
```

Check out the design view

After you incorporate the Expand_ Contract code into your application, you might realize that continued form development can be difficult since all of your "Expanded" controls are now shoved into the upper-left hand corner of your form at position 0,0. The best line of defense is to add the Expand_Contract functionality just before deploying the application.

However, if you need to revisit the form for maintenance after incorporating this code, you'll need to create a form that allows you to open the form using the Expand_ Contract utility in "Expanded" mode. This code segment loads a combo box that lists all of the forms in your database (since it uses AllForms, this code only works in Access 2000):

```
Private Sub Form_Load()
Dim obj As AccessObject, dbs As Object
Dim strForms As String
Set dbs = Application.CurrentProject
  For Each obj In dbs.AllForms
        If obj.Name <> Me.Name Then
                strForms = obj.Name & ";" & strForms
        End If
  Next obj
  Me.cbobox.RowSource = strForms
End Sub
```

Now, all that you have to do is create a button whose click event procedure opens the form in design view and runs the Expand_Contract procedure with

Continues on page 22

Your Access Data . . .

Continued from page 13

Outlook items using the Exchange-Access Wizard in Access 97 or the built-in export feature of Access 2000. When you need to export contact, task, calendar, or other data from Access tables to the appropriate type of Outlook items, you need to write code to do the exporting. In this article, I've provided the code that you can use to get your application data into your Outlook personal information manager. ▲

DOWNLOAD **OUTLOOK.ZIP at www.smartaccessnewsletter.com**

Helen Feddema has been working with Access since the beta of v. 1.0, and with Outlook since the beta of Outlook 97, and has written or co-authored several books on Access and Outlook. Her most recent Access book is *DAO Object Model: The Definitive Reference* (O'Reilly, January 2000). Her most recent Outlook contribution is five chapters on Outlook programming for the Que book *Special Edition: Outlook 2000*. She's currently writing another O'Reilly book on the Outlook object model. Helen is the editor of the biweekly "Woody's Access Watch" ezine (http://www.woodyswatch.com/waw) and writes its Access Archon column. She has a Web site with Office code samples at http://www.ulster.net/~hfeddema, hfeddema@ulster.net.

Naming Names: The Major Outlook Items

Outlook items are referenced using different terminology in the interface and code, which can be confusing. **Table 1** lists the major Outlook items with the following information:

- The item's name from the Outlook interface.
- The relevant object in the Outlook object model.
- The message class to be used in VBA code.

Be particularly careful with mail messages and notes—a mail message has the message class of IPM.Note, while a note is called a NoteItem in the object model.

Table 1. Outlook Items in the Interface and in code.

Interface name	Object model name	Message class
Contact	ContactItem	IPM.Contact
Task	TaskItem	IPM.Task
Mail Message	MailItem	IPM.Note
Appointment	AppointmentItem	IPM.Appointment
Journal Entry	JournalItem	IPM.Activity
Note	NoteItem	IPM.StickyNote

Contracting Forms . . .

Continued from page 20

bolExpandedPane set to True to open your form in "Expanded" design view, and continue with the development.

Final notes: After expanding or contracting, you should set the forms focus to a visible control. The simplest way to change your form's size is to use the RunCommand acCmdSizeToFitForm to resize the window.

By giving the user more control over the interface, you not only empower your users, the application gains a logical division that makes your form more intuitive, minimizes training costs, and reduces Help desk inquiries. A little interface work goes a long way. Everyone involved with your software will benefit from a little focus. ▲

DOWNLOAD **EXPCONT.ZIP at www.smartaccessnewsletter.com**

Rick Jones is the owner of Luna Consulting, a firm specializing in Microsoft Access and SQL Server applications. He's also an Access instructor for AppDev, a nationally recognized training company. Voice: 206-935-2732, www.lunaconsulting.com, RJones@speakeasy.org.

Stephanie Hudson is a database developer working with VBA intensive Access applications for Luna Consulting. She was previously a business analyst specializing in project feasibility, data manipulation, and market analysis.