vsFlex 2.0

VideoSoft Custom Control Library

To learn how to use help, press F1

Introduction
Find out about:
Installation Product support Licensing New Features
Registration Other VideoSoft products.

vsFlexArray
A new way to display and operate on tabular data. FlexArray gives you total flexibility to display, sort, merge, and format tables containing strings and pictures.
Introduction Tutorial Reference Summary

vsFlexString
A powerful regular expression engine. With FlexString, you can find and replace patterns in strings. Use it to provide regular expression search-and-replace capabilities or to parse input strings.

Introduction Tutorial Reference Summary

Introduction

Welcome to **vsFlex2**, a VideoSoft custom control library. VideoSoft custom controls are innovative, flexible, and powerful. If you like them, make sure you check out our other award-winning products, **vsOcx** and **vsView2**.

Our distribution policy is almost as innovative as the controls. We want every Visual Basic programmer to get copies of our products and try them for as long as they want. Those who like the tools and find them useful (almost everybody, we hope) can buy licenses at reasonable prices. The only restriction is that unlicensed copies of the software will display a VideoSoft banner whenever they are loaded, as a friendly reminder that we do expect people to pay for the software if they use it.

We hope you'll like vsFlex2. If you have suggestions and ideas for new features or new tools, call us or write.

VideoSoft
5900 Hollis Street, Suite T
Emeryville, CA 94608
Phone 510/595-2400
Fax 510/595-2424

Control Summary

Icon	Object	Description
	**vsFlexArray**	A new way to display and operate on tabular data. FlexArray gives you total flexibility to display, sort, merge, and format tables containing strings and pictures.
	**vsFlexString**	A powerful regular expression engine. With FlexString, you can find and replace patterns in strings. Use it to provide regular expression search-and-replace capabilities or to parse input strings.

Installation

To install vsFlex2, use the INSTALL utility provided on the distribution diskette.

If you are upgrading from the original vsFlex, you will need to use a utility supplied with the distribution diskette to upgrade your existing projects. Optionally, you may still use the original vsFlex for legacy projects and vsFlex2 for new projects. Both versions may coexist on the same computer.

Distribution

vsFlex2 is royalty-free. You may include copies of the OCX and HLP files with as many copies of as many applications you ship.

You may not distribute the license file VSFLEX2.LIC.
And you dont have to: as long as you have the license file installed on your machine, vsFlex2 will stamp every application you compile so the banner will not appear when your users run the applications.

If you work with other developers, you may be interested in VideoSofts site licenses. Call us for details.

If you havent yet registered your copy of vsFlex2 and would like to do it now, click [**HERE**] to get a Registration/Order Form.

Product Support

Product support for vsFlex2 is available to licensed users through the following channels:

Internet	http://www.videosoft.com
CompuServe	CIS 74774,420 or join our forum by typing "GO VIDEOSOFT"
Mail	VideoSoft 5900 Hollis Street, Suite T Emeryville, CA 94608
Phone	510/595-2400
FAX	510/595-2424

Before calling for technical support, please make sure you know what version of vsFlex2 you are using. The version number appears in the About box that pops up when you double-click the (**About**) property in any vsFlex2 control.

New Features in Version 2

This section summarizes the new features in vsFlex2. If you are familiar with vsFlex, this section will get you up to speed quickly. For details on each new feature, check the main body of the documentation.

[] Data Binding

This is something a lot of people were waiting for. FlexArray is data bound, so you can read in entire tables very quickly, and then manipulate them at will. For details, check the following items in the reference section: **DataSource**, **VirtualData**, and **AutoResize**.

[] In-Cell Editing

FlexArray is now fully editable. You can implement data entry with text boxes or drop-down lists just by setting a couple of properties, and you can validate the data by responding to events. For details, check the following items in the reference section: **Editable**, **ComboList**, **EditCell**, **BeforeEdit**, **AfterEdit**, and **Validate**.

[] Subtotals and Outlining

Think of this as a cross between the Excel outliner and the Explorer. You can dynamically add subtotals to summarize data, then expand and collapse entire groups with a click of the mouse, and no code. It's fast, easy, and extremely powerful. For details, check the following items in the reference section: **Subtotal**, **Outline**, **OutlineBar**, **IsSubtotal**, and **IsCollapsed**.

[] AutoSize Columns

You can tell FlexArray to adjust the width of each column to fit the widest entry. For details, check the **AutoSize** method. This also works with data binding: just set the **AutoResize** property to **True** and all your data will fit comfortably within their columns.

[] ListBox-Style Selection

FlexArray now suports selection of non-adjacent rows, just like an extended-selection ListBox. Just set **SelectionMode** to **flexSelectionListBox**, and extended selection will be enabled using the mouse or keyboard. To control selection through code, use the **IsSelected**() property.

[] Formatted, International Sorting

FlexArray now recognizes numbers with embedded formatting characters such as thousand separators and currency signs, and it can also extract their values for you. (This works with international formats as well.) For details, check the following items in the reference section: **Sort**, **Value**, **ValueMatrix**.

[] Cell Flooding

Each cell may now be partially flooded to indicate the magnitude of the value it contains. This allows you to set up tables that also work as charts. It's great for business and scientific applications. For details, check the following items in the reference section: **CellFloodPercent**, **CellFloodColor**, and **FloodColor**.

[] More Compatibility with MSGrid

We have added six new properties to make FlexArray more compatible with Microsoft's basic grid control: you can now use **RowIsVisible** and **ColIsVisible**, **RowPos**and **ColPos**, **GridLineWidth**, and **FixedAlignment**. Porting code is easier than ever.

[] More Convenience

FlexArray has a new **Select** method that allows you to select a cell or a range with a single command. It's faster and convenient, and make your code a lot shorter and more readable. Also, the **Clear** method now takes optional parameters that allow you to specify what to clear and where.

[] More Control

You now have more control over the appearance of your FlexArray controls, with lots of new settings for the **GridLines** property, and more control over row and column resizing, with the

UserResized event.

[] More Speed

Try it out! We have optimized our painting code even more. Scrolling and selecting is even faster than in version 1.0.

[] Printing

vsFlex2 knows how to communicate with **vsView2**. This means you can print and preview your FlexArray with a single command, even if it spans multiple pages. Plus, you can embed FlexArray controls into **vsView2** reports.

Assuming you have a vsPrinter control named ***vp***, all it takes to print a FlexArray control named ***fa*** is this single line of code: ***vp***.**RenderControl** = ***fa***.**hWnd**.

Most of these enhancements were added in response to user requests, and we are grateful for their input.

Using the FlexArray Control

The FlexArray control allows you to display and operate on tabular data in new ways. On the surface, FlexArray is similar to a spreadsheet, but it allows total flexibility to display, sort, merge, and format tables containing strings and pictures.

The FlexArray control was designed to be compatible with Microsofts Grid control (GRID.VBX, GRID32.OCX). FlexArray implements most of the Grid controls properties, so it is easy to modify older projects to take advantage of FlexArrays extra functionality, which includes:

[] Data Binding

Bind FlexArray to data controls for easy acess to your data using the **DataSource** property.

[] Cell Editing

Just set the **Editable** property to **True** and you have basic cell editing. Trap the **BeforeEdit**, **AfterEdit**, and **Validate** events to control user input. And use the **ComboList** property to allow users to pick from lists instead of typing.
Alternatively, you may want to use FlexArrays **CellTop**, **CellLeft**, **CellWidth**, and **CellHeight** properties to place other controls directly over the current cell.

[] Individual cell formatting

Use FlexArrays **CellBackColor**, **CellForeColor**, **CellFont**, **CellAlignment**, **CellFloodPercent**, **CellPicture**, **CellPictureAlignment**, and **CellFloodColor** properties to control the appearance of individual cells.

[] Sorting

Use FlexArrays **Sort** property to sort information with speed and complete flexibility.

[] Row and Column moving

Use FlexArrays **RowPosition**() and **ColPosition**() properties to modify the layout of the information in the control at run time.

[] Cell Merging

Use FlexArrays exclusive **MergeCells** property to create clear, concise, and attractive data summaries without programming.

[] Design-time layout development

Use FlexArrays **FormatString** property to define column and row headers, widths, and alignment at design-time.

[] Miscellaneous

Other FlexArray advantages are: long strings in cells (up to 32k), more storage capacity, invisible columns and rows, better font control, more options for customizing colors and grid styles, for aligning text and pictures, less flicker, more control over cursor and selection appearance, and more.

Best of all, the FlexArray is small and does not require separate custom DLLs, so installation is quick and easy.

FlexArray Tutorial

This section of the manual takes you step-by-step through the creation of three Visual Basic projects using the FlexArray control:

Edit Demo
A data-entry tool with editable fields, drop-down lists, check boxes, and custom controls.

Data Demo
Merge, sort, subtotal, and rearrange data.

Outline Demo
Structure data with subtotals; collapse and expand details.

These are simple programs that focus on using the FlexArray control. We tried to reduce the amount of coding to a minimum, just enough to show how common tasks can be easily accomplished with the FlexArray. For more realistic (and ambitious) projects, please check out the samples on the distribution disks.

Edit Demo

This sample projects starts with a basic data-entry grid, then adds features such as clipboard support, drop-down lists, data validation, automatic formatting, and check boxes.

1) Create the Control

Start a new Visual Basic project including vsFlex2 (if you dont know how to add OCX files to a project, consult the Visual Basic manual). The vsFlex2 control icons will be added to the Visual Basic toolbox.

Create a FlexArray object on the form by clicking the FlexArray icon on the toolbox (), then clicking and dragging on the form.

Next, set the following control properties:

```
Cols = 5
Editable = True
ExtendLastCol = True
FillStyle = 1 ' Repeat
FixedCols = 0
FormatString = "=Product|Region|Sales Person|>Amount Sold|Bonus"
Name = fa
```

That's it. If you run the project now, you can already enter data into each column simply by typing it in. You may also activate the cell-editing mode by pressing the space bar or the F2 function key.

If you select a range and enter some data, the entry will be applied to the whole range. This is a real time-saver in some situations. If you don't like this behavior, set the **FillStyle** property to **flexFillSingle** (0).

2) Add Mouse Support

Some users use predominantly the mouse, others prefer the keyboard. By default, the FlexArray will enter cell-edit mode when the user starts typing something into a cell. It is often desirable to let the user initiate editing with the mouse, either by double-clicking on a cell or by right-clicking on it. This is accomplished with the following code:

```
Private Sub fa_MouseDown(Button%, Shift%, X!, Y!)
    If Button <> 2 Then Exit Sub
    If fa.MouseRow < fa.FixedRows Then Exit Sub
    If fa.Row <> fa.MouseRow Then fa.Row = fa.MouseRow
    If fa.Col <> fa.MouseCol Then fa.Col = fa.MouseCol
    fa.EditCell
End Sub
```

The first two lines make sure that the right mouse button was clicked over a non-fixed row. In that case, the next three lines select the cell that was right-clicked and put the control in cell-editing mode with the **EditCell** method.

Note that we only set the **Row** and **Col** and properties if the mouse is clicked outside the current cell. This is done this way because setting the **Row** and **Col** properties also resets **RowSel** and **ColSel**. If the user right-clicks on the current cell, then we do not set the properties and therefore do not reset the selection. This is a subtle touch, but it will be very noticeable to some users.

If we had chosen to implement cell-editing in response to the double-click, we would not even have to set the selection, since the control would have done it for us.

3) Add Clipboard Support

The Windows clipboard is a really great invention. You can use it to move data around, replicate it, and even to import and export data between applications.

To add clipboard support to our project, all it takes is the following code:

```
Private Sub fa_KeyDown(KeyCode%, Shift%)
    Dim Cpy%, Pst%
```

```
    ' copy: ctrl-C, ctrl-X, ctrl-ins
    If KeyCode = 67 And Shift = 2 Then Cpy = True
    If KeyCode = 88 And Shift = 2 Then Cpy = True
    If KeyCode = 45 And Shift = 2 Then Cpy = True

    ' paste: ctrl-V, shift-ins
    If KeyCode = 86 And Shift = 2 Then Pst = True
    If KeyCode = 45 And Shift = 1 Then Pst = True

    ' do it
    If Cpy Then
        Clipboard.Clear
        Clipboard.SetText fa.Clip
    ElseIf Pst Then
        fa.Clip = Clipboard.GetText
    End If
End Sub
```

The routine handles all standard keyboard commands related to the clipboard: CTRL-X, CTRL-C, or CTRL-Ins to copy, and CTRL-V or SHIFT-Ins to paste. The real work is done by the **Clip** property, which takes care of copying and pasting the clipboard text over the current range.

If you run the project now, you can move data around by selecting it, pressing CTRL-Ins, moving the selection, and pressing SHIFT-Ins. You can also start editing a cell by right-clicking on it.

4) Add Drop-Down Lists

Entering data is a tedious and error-prone process. Drop-down lists are great because they minimize the amount of typing you must do, reduce the chance of errors, and increase the consistency of the data.

Let's assume that our sample project only involves sales of three products (Applets, Widgets, and Gadgets), in four regions (North, South, East, and West), and that there are three sales persons (Mary, Sarah, and Paula).

In this case, asking the user to type in all this data would be a very mean thing to do. A much better approcah would be to use drop-down lists, which can be done easily using the **BeforeEdit** event and the **ComboList** property. The code below shows how to do it:

```
Private Sub fa_BeforeEdit(ByVal Row%, ByVal Col%, Cancel%)
    Select Case Col
        Case 0 ' product
            fa.ComboList = "Applets|Wahoos|Gadgets"
        Case 1 ' region
            fa.ComboList = "North|South|East|West"
        Case 2 ' sales person
            fa.ComboList = "Mary|Paula|Sarah"
        Case Else ' amount
            fa.ComboList = ""
    End Select
End Sub
```

As you can see, all we have to do is supply a list of choices, separated by pipes, according to the type of data being edited. The last two columns (**Sales Amount** and **Bonus**) do not use drop-down lists, so we set **ComboList** to an empty string.

To see the code in action, run the project and type the first character of an entry into a cell (or F2, or the space bar). You will get a regular drop-down list box, where you can pick the appropriate value easily. After making the selection, press ENTER to quit cell-editing mode so you can move the cursor freely.

5) Add Data Validation

If you pick data from a list, there's little need for field-level data validation (although you may still need record-level validation). If you type, on the other hand, then data-validation is essential.

In our example, we would like to prevent users from typing text or negative values in the **Sales Amount** field. The best way to do this is to use the **Validate** event, as shown below:

```
Private Sub fa_Validate(Row%, Col%, Value$, Cancel%)
    Select Case Col
        Case 3 ' sales amount
            If Val(Value) <= 0 Then
                Cancel = True
                MsgBox "Please enter a positive value"
            End If
    End Select
End Sub
```

The ***Value$*** parameter contains the entry to be checked. If it contains a an invalid string, the code sets the ***Cancel%*** parameter to **True**. This causes the original value to be restored and the cursor to be positioned over the offending cell, so the user can try again. Note that modifying the ***Value$*** parameter has no effect.

6) Add Automatic Formatting

Now let's say we want to format the numberic values entered so they look consistent. The easiest way to do this is to use the **AfterEdit** event, as shown below:

```
Private Sub fa_AfterEdit(ByVal Row%, ByVal Col%)
    Select Case Col
        Case 3 ' sales amount
            fa = Format(Val(fa.TextMatrix(Row, Col)), "#,###.00")
    End Select
End Sub
```

This routine takes the value just entered in column 3 (a dollar amount), formats it using VB's **Format** function, and assigns the result back to the control. Note that the assignment will copy the formatted value to all cells in the selected range.

7) Add CheckBoxes

We are almost done. The last thing we need to do is handle the situation when the user wants to modify the **Bonus** column, which contains a boolean value.

For this, we could use a drop-down list with "Yes|No" or "True|False" fields, but that would be too plain for some people. We could also use pictures, but for this tutorial we'll use a different (and often overlooked) approach: symbols.

Windows provides a font called **Wingdings** that contains lots of useful symbols. Among these are a hollow box (**Chr**(158)) and a checked box (**Chr**(254)). To use them, we need to assign the **Wingdings** font to all cells in the **Bonus** column, initialize the bonuses with the **Chr**(158) character, and then switch between **Chr**(158) and **Chr**(254) when we want to toggle someone's bonus. Here's the code to do it:

```
Private Sub Form_Load()
    With fa
        .ColWidth(4) = .ColWidth(4) / 2
        .Select 1, 4, .Rows - 1, 4
        .CellFontName = "Wingdings"
        .CellFontSize = 11
        .Text = Chr(168)
        .Select 1, 0
    End With
End Sub
```

When the form loads, we set up the **Bonus** column with the **Wingdings** font. To toggle the **Bonus** value, we simply modified the **BeforeEdit** handler:

```
Private Sub fa_BeforeEdit(ByVal Row%, ByVal Col%, Cancel%)
    Select Case Col
        Case 0 ' product
            fa.ComboList = "Applets|Wahoos|Gadgets"
        Case 1 ' region
            fa.ComboList = "North|South|East|West"
        Case 2 ' sales person
            fa.ComboList = "Mary|Paula|Sarah"
```

```
            Case 3 ' amount
                fa.ComboList = ""
            Case 4 ' bonus
                If fa = Chr(168) Then fa = Chr(254) Else fa = Chr(168)
                Cancel = True
        End Select
End Sub
```

If the user tries to edit the last column, the code toggles the value and sets ***Cancel%*** to **True**, preventing the user from typing into the cell.

That's it. You may want to play a little with the project and maybe add some features to it. Check out the distribution disk for samples that use modal forms for entering other types of data such as dates.

Data Demo

This sample projects starts with a basic grid full of randomly-generated data, then adds cell merging, automatic sorting, dynamic data layout (by allowing users to drag columns around and having the control automatically rearrage the data), and subtotals.

1) Create the Control

Start a new Visual Basic project including vsFlex2 (if you dont know how to add OCX files to a project, consult the Visual Basic manual). The vsFlex2 control icons will be added to the Visual Basic toolbox.

Create a FlexArray object on the form by clicking the FlexArray icon on the toolbox (), then clicking and dragging on the form.

Next, set the following control properties:

```
Cols = 4
ExtendLastCol = True
FixedCols = 0
MergeCells = flexMergeRestrictColumns
FormatString = "=Product|Region|Sales Person|>Amount Sold"
Name = fa
```

2) Fill it up with Random Data

Normally, you would perform this step simply by connecting the FlexArray to a database through the **DataSource** property. For this demo, however, we'll generate some random data instead.

You will need this code at the form's Load event:

```
Private Sub Form_Load()
    Dim r%, c%
    Dim product$(3), Dim region$(3), Dim person$(3)

    ' move the data into vectors
    InitVector product, "Applets|Wahoos|Widgets|Gadgets"
    InitVector person, "Mary|Sarah|Paula|Annie"
    InitVector region, "North|South|East|West"

    ' now copy random elements into the FlexArray
    With fa
        For r = 1 To .Rows - 1
            .TextMatrix(r, 0) = product$(Rnd() * 3)
            .TextMatrix(r, 1) = region$(Rnd() * 3)
            .TextMatrix(r, 2) = person$(Rnd() * 3)
            .TextMatrix(r, 3) = Format(Rnd() * 20000, "#,###.00")
        Next
    End With

    ' since we're already here, let's turn column merging on
    For c = 0 To 2
      fa.MergeCol(c) = True
    Next
    SortData
End Sub
```

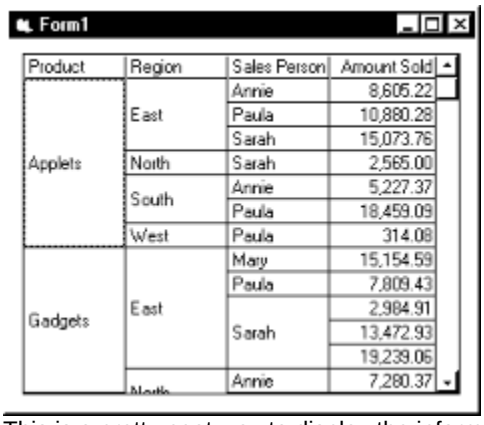
This routine uses two little helper functions: **InitVector** splits a string into several elements of an array, and **SortData** does exactly what it says:

```
Sub InitVector(v$(), ByVal s$)
    Dim i%, p%
    Do
        p = InStr(s, "|")
        If p = 0 Then Exit Do
        v(i) = Left(s, p - 1)
        i = i + 1
        s = Mid(s, p + 1)
    Loop
```

```
    v(i) = s
End Sub

Sub SortData()
    fa.Select 1, 0, fa.rows-1, fa.cols-1
    fa.Sort = flexSortStringAscending
End Sub
```

The **SortData** routine starts by selecting the first non-fixed row in the control using the **Select** method. Next, it sorts the entire control in ascending order using the **Sort** property.

At this point, you may want to run the project and see the results of the work so far:



This is a pretty neat way to display the information by product. But what if we were interested in analyzing sales person performance, or viewing sales by region? That's easy to, and is our next step.

3) Dynamic layout

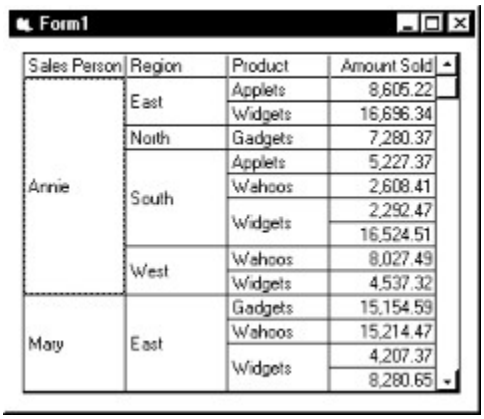
It would be nice if we could double-click on a column heading and have that column move to the left. That's what the code below does:

```
Private Sub fa_DblClick()
    Dim c%

    ' make sure we clicked on a valid row/column
    If fa.MouseCol > 0 Then Exit Sub
    c = fa.MouseCol
    If c = fa.Cols - 1 Then Exit Sub

    ' move the column to the left
    fa.Redraw = False
    fa.ColPosition(c) = 0
    SortData
    fa.Redraw = True
End Sub
```

That's it. The routine first checks to make sure the user clicked on the heading row, and not on the last column (the last column contains the sales totals, and we don't want to sort on that.) It uses the **MouseCol** and **MouseRow** properties for that.

If everything is OK, then the routine sets the **Redraw** property to **False** to avoid flicker, then moves the column that was clicked to the left using the **ColPosition**() property and resorts the data. Finally, it sets **Redraw** back on.

If the user double-clicked on the **Sales Person** column, here's what would happen:



There we go. You can see at a glance who sold what and where. But there's still something missing. If would be really nice to show subtotals next to the dtailed data, wouldn't it?

4) Subtotals

Adding subtotals is really easy. FlexArray has a **Subtotal** method that makes it almost trivial. To make things interesting, let's say we want an overall sales total plus subtotals for the first data column (in the example above, that would be **Sales Person**).

All we need to do is modify the **SortData** routine slightly:

```
Sub SortData()

    ' clear old subtotals
    fa.SubtotalPosition = flexSTAbove
    fa.Subtotal flexSTClear

    ' sort the data
    fa.Select 1, 0, 1, fa.Cols - 1
    fa.Sort = flexSortGenericAscending

    ' insert new subtotals
    fa.Subtotal flexSTSum, 0, 3, "#,###.00", 1, fa.BackColor
    fa.Subtotal flexSTSum, -1, 3, "#,###.00", 1, fa.BackColor
End Sub
```

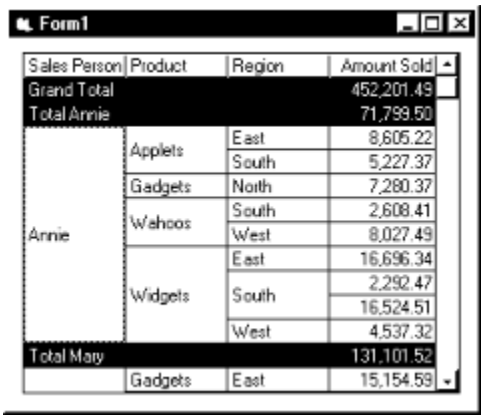
The routine starts by using the **Subtotal** method to clear any old subtotals. If also sets the **SubtotalPosition** property so the subtotals are inserted before the group of rows they refer to, rather than after.

The routine then performs the sorting , using the original code, and finally inserts the new subtotals.

The first **Subtotal** statement starts by specifying the function to be used, in this case **flexSTSum**, a simple sum. Next comes the number of the column to be totalled, in this case column 0. This means that every time the contents of column 0 change, a subtotal will be inserted. Next comes the column with the numbers to be added together, in this case column3, **Amount Sold**. The other parameters are optional: a format for the subtotals and the back and fore colors to be used for the new subtotal rows.

The second **Subtotal** statement is almost identical, except it tells the contorl to total on column -1. By specifying a negative column number, we are effectively asking the control to insert a grand total.

Here's the result:



That's very informative, don't you agree? We could easily add more levels of subtotals, or calculate average sales, maxima and minima, or standard deviations.

We could also add outlining, the ability to expand and collapse detail rows under a subtotal row. To do this, you would simply add a fixed column to the FlexArray, then set the **OutlineBar** property to a non-zero value. The fixed column would diplay an outline tree with buttons for collapsing and expanding the outline. Try it.

The next section shows how you can create custom outlines without using subtotals.

Outline Demo

This sample projects shows how you can structure data using FlexArray's outlining features. If you have used the Windows Explorer, you already know how useful it is to be able to open and close (or expand and collapse) entire branches of an outline so you can see what you want and not clutter the display with unnecessary details. Why not apply the same principle to your data?

This sample project reads several INI files and presents each one as a branch, with sections underneath wchich are also branches, that can be collapsed or expanded to show its tokens and settings.

1) Create the Control

Start a new Visual Basic project including vsFlex2 (if you dont know how to add OCX files to a project, consult the Visual Basic manual). The vsFlex2 control icons will be added to the Visual Basic toolbox.

Create a FlexArray object on the form by clicking the FlexArray icon on the toolbox (), then clicking and dragging on the form.

Next, set the following control properties:

```
Cols = 3
ExtendLastCol = True
FormatString = "   |Token                    |Setting"
Name = fa
OutlineBar = flexOutlineBarComplete
Rows = 1
SubtotalPosition = flexSTAbove
```

2) Add code to read the data

Double-click the form and add the following code to the Form_Load event:

```
Private Sub Form_Load()
    AddOutline "Win.ini"
    AddOutline "System.ini"
    AddOutline "Progman.ini"
    fa.Outline 0
End Sub
```

The code calls another routine to read in three INI files (win, system, and progman) and then sets the outline level to zero using the **Outline** method. Of course, the real work is done by the **AddOutline** routine, listed below:

```
Sub AddOutline(inifile$)
    Dim ln$, s$, p%
    With fa
        ' create master branch
        .AddItem Chr(9) & inifile
        .IsSubtotal(fa.Rows - 1) = True
        .Select fa.Rows - 1, 1
        .CellFontBold = True
        ' read ini file
        Open "c:\windows\" & inifile For Input As #1
        While Not EOF(1)
            Line Input #1, ln
            ' it's a section
            If Left(ln, 1) = "[" Then
                .AddItem Chr(9) & " " & Mid(ln, 2, Len(ln) - 2)
                .IsSubtotal(fa.Rows - 1) = True
                .RowData(fa.Rows - 1) = 1 ' outline level
                .Select fa.Rows - 1, 1
                .CellFontBold = True
            ' it's a regular line
            ElseIf InStr(ln, "=") > 0 Then
                p = InStr(ln, "=")
                s = Chr(9) & "   " & Left(ln, p - 1)
                s = s & Chr(9) & Mid(ln, p + 1)
```

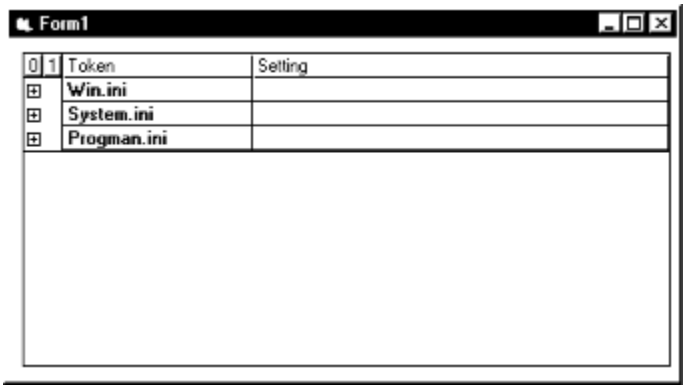
```
                .AddItem s
            End If
        Wend
        Close #1
    End With
End Sub
```

This routine a longer than most we're using. Sorry about that. But it is pretty simple: It starts by adding a row to the FlexArray control containing the name of the INI file being read. It marks the row as a subtotal using the **IsSubtotal**() property so the FlexArray control will know how to build the outline tree. It also sets the **CellFontBold** property to True to make the outline look better.

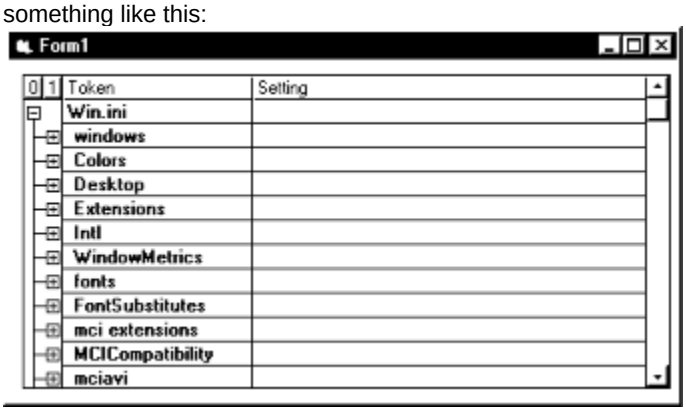
Next, the routine reads the INI file line by line. Section names are enclosed in square brackets. The code adds them to the control and then marks them as subtotals much the same way it marked the file name. The difference is that here it also sets the **RowData**() property to 1, indicating this is a level-1 branch. This subordinates all these branches to the file name, which is a level-0 branch.

Regular lines are parsed into token and setting and then added to the control. They are not marked as subtotals.

The project is pretty much done at this point. If you run it now, you should get a display more or less like this:



If you click on the plus signs, you'll expand the branch and will be able to see the contents of the each INI file. If you click the buttons at the top of the outline bar, you'll set the outline level for the control. For example, the display above shows only the level-0 branches. If you clicked on the level-1 button on the outline bar, the display would change to something like this:



Pretty easy, huh? You may expand the whole outline by shift-clicking the buttons at the top of the outline bar, or use the plus and minus buttons to collapse and expand individual branches.
The FlexArray provides the expanding and collapsing for you, but you ma extend and customize its behavior. Every time a branch is expanded or collapsed, FlexArray fires the **Collapsed** event so you may take actions in response to that. Furthermore, you may use the **IsCollapsed**() property to get and set the collapsed state of each branch in code.

2) Add mouse an keyboard handling

Let's say we wanted to allow users to expand and collapse outline branches by double-clicking on a row itself, rather than on the outlie bar. Here's the code to do that:

```
Private Sub fa_DblClick()
    Dim r%
    With fa
        r = .Row
        If .IsCollapsed(r) = flexOutlineCollapsed Then
            .IsCollapsed(r) = flexOutlineExpanded
        Else
            .IsCollapsed(r) = flexOutlineCollapsed
        End If
    End With
End Sub
```

The code checks the current row. If it is collapsed, then it expands it. Otherwise, it collapses it. Collapsing a detail row actually collapses its entire branch.

We can use the same code to implement the keyboard interface. We just call the **DblClick** event handler from the **KeyPress** handler:

```
Private Sub fa_KeyPress(KeyAscii As Integer)
    If KeyAscii = 13 Then fa_DblClick
End Sub
```

That's it. If you run the project, you'll see that the outline is easy to use and makes the data in the control easy to understand.

If you followed all the samples, you have a good understanding of the main FlexArray features. For more and more sophisticated sample projects, check out the distribution disks.

FlexArray Reference

Description A FlexArray control displays a series of rows and columns. The intersection of a row and column is a cell. You can read and set the contents of each cell programmatically.

Remarks You can put text, a picture, or both in any cell of a FlexArray. The **Row** and **Col** properties specify the current cell in a FlexArray. You can specify the current cell in code, or the user can change it at run time using the mouse or the arrow keys. The **Text** property references the contents of the current cell.

If a cell's text is too long to be displayed in the cell, and the **WordWrap** property is set to **True**, the text wraps to the next line within the same cell. To display the wrapped text, you may need to increase the cell's column width or row height (using the **ColWidth**() or **RowHeight**() properties).

Use the **Rows** and **Cols** properties to determine the number of columns and rows in a FlexArray control.

When a new element of a control array is loaded at run time, the new element does not inherit the original control's run-time properties.

File Name VSFLEX2.OCX (32-bit version), or
VSFLEX2-.OCX (16-bit version)

Object Type vsFlexArray

Note Before you can use a FlexArray control in your application, you must add vsFlex2 to your project (see the Visual Basic manual for details). To automatically include vsFlex2 in new projects, put it in an AUTOLOAD file. When distributing your application, you should follow the instructions in the **Distribution** section of the vsFlex2 documentation.

FlexArray Summary

Properties (default: Text)

* **AllowBigSelection**	* **AllowUserResizing**	Appearance
* **AutoResize**	* **BackColor**	* **BackColorBkg**
* **BackColorFixed**	* **BackColorSel**	BorderStyle
* **BottomRow**	* **CellAlignment**	* **CellBackColor**
* **CellFloodColor**	* **CellFloodPercent**	* **CellFontBold**
* **CellFontItalic**	* **CellFontName**	* **CellFontSize**
* **CellFontStrikethru**	* **CellFontUnderline**	* **CellFontWidth**
* **CellForeColor**	* **CellHeight**	* **CellLeft**
* **CellPicture**	* **CellPictureAlignment**	* **CellTextStyle**
* **CellTop**	* **CellWidth**	* **ClientHeight**
* **ClientWidth**	* **Clip**	* **Col**
* **ColAlignment**	* **ColData**	* **ColIsVisible**
* **ColPos**	* **ColPosition**	* **Cols**
* **ColSel**	* **ColWidth**	* **ComboList**
* **DataSource**	* **Editable**	Enabled
* **ExtendLastCol**	* **FillStyle**	* **FixedAlignment**
* **FixedCols**	* **FixedRows**	* **FloodColor**
* **FocusRect**	Font	FontBold
FontItalic	FontName	FontSize
FontStrikethru	FontUnderline	FontWidth
* **ForeColor**	* **ForeColorFixed**	* **ForeColorSel**
* **FormatString**	* **GridColor**	* **GridColorFixed**
* **GridLines**	* **GridLinesFixed**	* **GridLineWidth**
* **HighLight**	hWnd	* **IsCollapsed**
* **IsSelected**	* **IsSubtotal**	* **LeftCol**
* **MergeCells**	* **MergeCol**	* **MergeRow**
* **MouseCol**	MouseIcon	MousePointer
* **MouseRow**	* **OutlineBar**	* **Picture**
* **PictureType**	* **Redraw**	* **RightCol**
* **Row**	* **RowData**	* **RowHeight**
* **RowHeightMin**	* **RowIsVisible**	* **RowPos**
* **RowPosition**	* **Rows**	* **RowSel**
* **ScrollBars**	* **ScrollTrack**	* **SelectionMode**
* **Sort**	* **SubtotalPosition**	* **Text**
* **TextArray**	* **TextMatrix**	* **TextStyle**
* **TextStyleFixed**	* **TopRow**	* **TreeColor**
* **Value**	* **ValueMatrix**	* **Version**
* **VirtualData**	* **WordWrap**	

Methods

* **AddItem**	* **RemoveItem**	* **Clear**
* **EditCell**	* **Subtotal**	Refresh
* **Outline**	* **Select**	* **AutoSize**

Events

* **SelChange**	* **RowColChange**	* **EnterCell**
* **LeaveCell**	* **Scroll**	* **Compare**
Click	DblClick	KeyDown
KeyPress	KeyUp	MouseDown
MouseMove	MouseUp	* **UserResized**
* **Collapsed**	* **Validate**	* **AfterEdit**
* **BeforeEdit**		

AddItem Method

Description Adds a row to the control.

Usage [***form!***]vsFlexArray.**AddItem** ***item$*** [, ***row%***]

Remarks

The **AddItem** method has these parts:

item$
String expression to add to the control. Use the tab character (**Chr$**(9)) to separate the contents of each cell in the row being added.

row%
Integer representing the position within the control where the new row is to be inserted (row 0 is the first row). If this part is omitted, the new row is appended at the bottom of the control.

Example

```
' create a row string
s$ = "Qtr 1" & Chr$(9) & "2312.32"

' insert it at the top
vsFlexArray.AddItem s$, 0

' insert it at the bottom
vsFlexArray.AddItem s$
```

AfterEdit Event [2]

Description Fired after the control exits cell-editing mode.

Usage Sub vsFlexArray_**AfterEdit**(ByVal ***row%***, ByVal ***col%***)

Remarks This event gets fired after the contents of a cell have been changed by the user.

The **AfterEdit** event is useful for performing actions such as re-sorting the data or calculating subtotals.

To perform validation, use the **Validate** event instead.

See also the the **Editable** and **ComboList** properties and the **EditCell** method .

AllowBigSelection Property

Description Sets/returns whether clicking on the fixed area should select entire columns and rows.

Usage [***form!***]vsFlexArray.**AllowBigSelection**[= ***setting%***]

Remarks This property is **True** by default.

Clicking on the top left fixed cell selects the entire control.

Data Type Boolean

AllowUserResizing Property

Description Sets/returns whether the user should be allowed to resize rows and columns with the mouse.

Usage [***form!***]vsFlexArray.**AllowUserResizing**[= ***setting%***]

Settings The **AllowUserResizing** property settings are:

flexResizeNone
flexResizeColumns
flexResizeRows
flexResizeBoth

Remarks If this property is set to a value other than **flexResizeNone**, the user can resize rows or columns at run time by using the mouse, as with the Microsoft Grid control.

To resize rows or columns, the mouse must be over the fixed area of the control, and close to a border between rows or columns. The mouse pointer will then change into an appropriate sizing pointer and the user can drag the row or column to change the row height or column width.

Rows with zero height and columns with zero width cannot be resized by the user. If you want to make them very small but still resizable, set their height or width to one, not to zero.

After the user resizes a row or column, the **UserResized** event is fired.

Data Type Integer

AutoResize Property [2]

Description Sets/returns whether column widths should be automatically adjusted when data is loaded from the database.

Usage [***form!***]vsFlexArray.**AutoResize**[= ***setting%***]

Remarks If this property is set to **True**, the control automatically resizes its columns to fit the widest entry every time new data is read in from the data base. This occurs by default when the control is loaded and every time the Refresh method is invoked for the data source control.

For this property to work, the **DataSource** property must be set to a valid data control.

See also the **AutoSize** method.

Data Type Boolean

AutoSize Method [2]

Description Resizes column widths to fit widest entry.

Usage [***form!***]vsFlexArray.**AutoSize** ***col1%***[, ***col2%***, ***equal%***]

Remarks The **AutoSize** method has three parts:

col1%, ***col2%***
Specify the first and last columns to be resized so their widths fit the widest entry in each column. The valid range for these parameters is between 0 and **Cols**-1. If ***col2%*** is omitted, then only ***col1%*** is resized.

equal%
If **True**, all columns between ***col1%*** and ***col2%*** are set to the same width. If **False**, then each column is resized independently. If this parameter is omitted, then it is assumed to be **False**.

BackColor* Properties

Description Sets/returns the background color of various elements of the control.

Usage
[***form!***]vsFlexArray.**BackColor**[= ***setting&***]
[***form!***]vsFlexArray.**BackColorBkg**[= ***setting&***]
[***form!***]vsFlexArray.**BackColorFixed**[= ***setting&***]
[***form!***]vsFlexArray.**BackColorSel**[= ***setting&***]

Remarks The picture below shows what part of the control each property refers to:

To set the background color of individual cells, use the **CellBackColor** property.

Data Type Color

BeforeEdit Event [2]

Description Fired before the control enters cell editing mode.

Usage Sub vsFlexArray_**BeforeEdit**(ByVal ***row%***, ByVal ***col%***, ***cancel%***)

Remarks This event is fired immediately before the control enters cell editing mode, and gives you a chance to prevent editing or to supply a list of choices for a drop-down list.

The ***row%*** and ***col%*** parameters specify which cell is about to be edited.

The ***cancel%*** parameter is **False** by default. If you set it to **True**, then the control prevents the built-in cell editor from being activated, and the cell retains its value.

If you do not cancel the editing process, you may supply a list of choices for a drop-down list through the **ComboList** property. If you set **ComboList** to an empty string (""), a regular text editor is used.

For details and examples, see the **Editable** property.

BottomRow Property [2]

Description Returns the last visible row in the control.

Usage ***setting%*** = [***form!***]vsFlexArray.**BottomRow**

Remarks The bottom row returned may be only partially visible.

You cannot set this property. To scroll the contents o fthe control through code, set the **TopRow** and **LeftCol** properties instead.

Data Type Integer

CellAlignment Property

Description Sets/returns the alignment of data in a cell or range.

Usage [***form!***]vsFlexArray.**CellAlignment**[= ***setting%***]

Settings The **CellAlignment** property settings are:

flexAlignLeftTop
flexAlignLeftCenter
flexAlignLeftBottom
flexAlignCenterTop
flexAlignCenterCenter
flexAlignCenterBottom
flexAlignRightTop
flexAlignRightCenter
flexAlignRightBottom
flexAlignGeneral(strings to left, numbers to right)

Remarks Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

Data Type Integer

CellBackColor, CellForeColor Properties

Description Sets/returns the background and foreground colors of individual cells or ranges.

Usage [***form!***]vsFlexArray.**CellBackColor**[= ***setting&***]
[***form!***]vsFlexArray.**CellForeColor**[= ***setting&***]

Remarks Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

Setting either of these properties to zero (black color) causes the control to paint the cell using the standard colors (**BackColor** and **ForeColor**). Thus, to set either of these properties to black, set them to **RGB**(1,0,0,) instead of **RGB**(0,0,0).

Data Type Color

CellFloodColor Property [2]

Description Sets/returns the color to be used for flooding a cell.

Usage [***form!***]vsFlexArray.**CellFloodColor**[= ***setting&***]

Remarks This property overrides the **FloodColor** property to determine the color to be used for flooding individual cells.

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

For details and examples, see the **CellFloodPercent** property.

Data Type Color

CellFloodPercent Property [2]

Description Sets/returns the percentage of flooding for a cell.

Usage [***form!***]vsFlexArray.**CellFloodPercent**[= ***setting%***]

Remarks This property allows you to fill up a portion of a cell so it can be used as a progress indicator or a bar in a bar chart.

Setting this property to a value between -100 and 100 causes the cell to be filled with the color specified by the **FloodColor** or **CellFloodColor** properties.

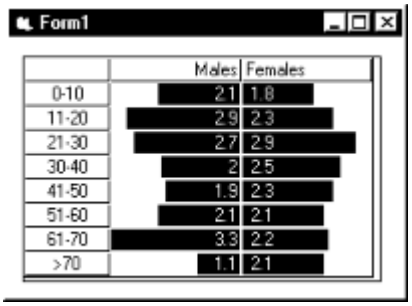
Positive values fill the cell from left to right. Negative values fill it from right to left.

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

The example below illustrates the use of this property.

Example

```
' example to demonstrate cell flooding
' assumes there's a global array "count(2,8)"
Private Sub Form_Load()
   Dim i%
   Dim max!

   ' place text in cells, keep track of maximum
   For i = 0 To 7
        fa.TextMatrix(i + 1, 1) = Str(count(0, i))
        fa.TextMatrix(i + 1, 2) = Str(count(1, i))
        If count(0, i) > max Then max = count(0, i)
        If count(1, i) > max Then max = count(1, i)
   Next

   ' set CellFloodPercent, using max to scale from
   ' 0 to -100 for column 1 and from 0 to 100 for
   ' column 2:
   For i = 0 To 7
        fa.row = i + 1
        fa.col = 1
        fa.CellFloodPercent = -100 * count(0, i) / max
        fa.col = 2
        fa.CellFloodPercent = 100 * count(1, i) / max
   Next
End Sub
```



Data Type Integer

CellFont* Properties

Description Sets/returns the font to be used for individual cells or ranges of cells.

Usage
[***form!***]vsFlexArray.**CellFontBold**[= ***setting%***]
[***form!***]vsFlexArray.**CellFontItalic**[= ***setting%***]
[***form!***]vsFlexArray.**CellFontName**[= ***setting$***]
[***form!***]vsFlexArray.**CellFontSize**[= setting!]
[***form!***]vsFlexArray.**CellFontStrikethru**[= ***setting%***]
[***form!***]vsFlexArray.**CellFontUnderline**[= ***setting%***]
[***form!***]vsFlexArray.**CellFontWidth**[= setting!]

Remarks Changing these properties affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

Setting **CellFontSize** to zero or **CellFontName** to an empty string resets the cell formatting and causes the default font to be used.

Setting **CellFontWidth** to zero causes the default width to be used.

Data Type
String (CellFontName)
Single (CellFontSize, CellFontWidth)
Boolean (CellFontBold, CellFontItalic, CellFontUnderline, CellFontStrikeThru)

CellHeight, CellLeft, CellTop, CellWidth Properties

Description Returns the position of the current cell, in Twips. Also brings the current cell into view, scrolling if necessary.

Usage

variable& = [***form!***]vsFlexArray.**CellHeight**
variable& = [***form!***]vsFlexArray.**CellLeft**
variable& = [***form!***]vsFlexArray.**CellTop**
variable& = [***form!***]vsFlexArray.**CellWidth**

Remarks These properties are useful for placing other controls over or near a specific cell. Whenever you read any of these properties, the control assumes that you want to work on the current cell and it automatically brings it into view, scrolling if necessary.

vsFlexArray version 2.0 offers in-cell editing of strings and combol lists, so these properties are not needed for most common editing needs (see the **Editable** and **ComboList** properties for details).

Still, you may want to link controls to cells for editing dates, pictures, or other types of data, in which case these propeties become useful. See the samples on the distribution disk for examples of custom cell-editing.

Data Type Long

CellPicture Property

Description Sets/returns a picture to be displayed in a cell or range.

Usage [***form!***]vsFlexArray.**CellPicture**[= ***picture***]

Remarks You can set this property at run time using Visual Basic's LoadPicture function on a bitmap, icon, or metafile, or by assigning to it another controls **Picture** property.

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

Each cell may contain text and a picture. The relative position of the text and picture is determined by the **CellAlignment** and **CellPictureAlignment** properties.

Data Type Picture

CellPictureAlignment Property

Description Sets/returns the alignment of pictures in a cell or range.

Usage [***form!***]vsFlexArray.**CellPictureAlignment**[= ***setting%***]

Settings The **CellPictureAlignment** property settings are:

flexAlignLeftTop
flexAlignLeftCenter
flexAlignLeftBottom
flexAlignCenterTop
flexAlignCenterCenter
flexAlignCenterBottom
flexAlignRightTop
flexAlignRightCenter
flexAlignRightBottom
flexAlignStretch
flexAlignTile

Remarks Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

See also the **CellPicture** property.

Data Type Integer

CellTextStyle Property

Description Sets/returns 3D effects for text in a cell or range.

Usage [***form!***]vsFlexArray.**CellTextStyle**[= ***setting%***]

Settings The **CellTextStyle** property settings are:

flexTextFlat
flexTextRaised
flexTextInset
flexTextRaisedLight
flexTextInsetLight

Remarks Settings **flexTextRaised** and **flexTextInset** work best for large and bold fonts. Settings **flexTextRaised**Light and **flexTextInsetLight** work best for small regular fonts.

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property.

See also the **TextStyle** and **TextStyleFixed** properties.

Data Type Integer

Clear Method [2]

Description Clears the contents of the control. Optional parameters specify what to clear and where.

Usage [***form!***]vsFlexArray.**Clear** [***where%***] [, ***what%***]

Remarks The **Clear** method has two optional parts:

where%
Specifies what part of the control should be cleared. Valid settings are:
flexClearEverywhere (default)
flexClearScrollable
flexClearSelection

what%
Specifies what should be cleared. Valid settings are:
flexClearEverything (default)
flexClearText
flexClearFormatting including pictures

The **Clear** method does not affect the number of rows and columns on the control.

ClientHeight, ClientWidth Properties [2]

Description Return the size of the control, in Twips, excluding its borders.

Usage
variable& = [***form!***]vsFlexArray.**ClientHeight**
variable& = [***form!***]vsFlexArray.**ClientWidth**

Remarks These properties may be useful for setting column widths and row heights.

The example shows how to make a control with equal-width columns that extend across the entire control. Note that the **ExtendLastCol** property is set to **True** to eliminate round-off errors.

Example

```
' ColWidth(-1) means all columns
fa.ColWidth(-1) = fa.ClientWidth / fa.Cols

' make last column extend to fix round-off errors
fa.ExtendLastCol = True
```

Data Type Long

Clip Property

Description Sets/returns the contents of a range.

Usage [***form!***]vsFlexArray.**Clip**[= ***setting$***]

Remarks The string assigned to **Clip** may contain the contents of multiple rows and columns. Tab characters (**Chr$**(9)) indicate column breaks, and carriage return characters (**Chr$**(13)) indicate row breaks.

When a string is assigned to **Clip**, only the selected cells are affected. If there are more cells in the selected region than are described in the clip string, the remaining cells are left alone. If there are more cells described in the clip string than in the selected region, the unused portion of the clip string is ignored.

The example below puts text into a selected area two rows high and two columns wide.

Example

```
' build clip string
s$ = "1st" & Chr$(9) & "a" & Chr$(13)
s$ = s$ & "2nd" & Chr$(9) & "b"

' paste it over current selection
vsFlexArray.Clip = s$
```

Data Type String

Col, Row Properties

Description Sets/returns the active row and column.

Usage [***form!***]vsFlexArray.**Col**[= ***setting%***]
[***form!***]vsFlexArray.**Row**[= ***setting%***]

Remarks Use these properties to make a cell current or to find out which row or column contains the current cell. Columns and rows are numbered from zero, beginning at the top for rows and at the left for columns.

Note that the **Col**, **Row** properties are not the same as the **Cols**, **Rows** properties.

Setting these properties automatically resets **RowSel** and **ColSel**, so the selection becomes the current cell. Therefore, to specify a block selection, you must set **Row** and **Col** first, then set **RowSel** and **ColSel** (or use the **Select** method to do it all with a single statement).

Data Type Integer

ColAlignment Property

Description Sets/returns the column alignment.

Usage [***form!***]vsFlexArray.**ColAlignment**(***col%***)[= ***setting%***]

Settings The **ColAlignment** property settings are:

flexAlignLeftTop
flexAlignLeftCenter
flexAlignLeftBottom
flexAlignCenterTop
flexAlignCenterCenter
flexAlignCenterBottom
flexAlignRightTop
flexAlignRightCenter
flexAlignRightBottom
flexAlignGeneral(strings to left, numbers to right)

Remarks Any column may have an alignment that is different from other columns. This property affects all cells in the specified column, including those in fixed rows (unless you override this setting with the **FixedAlignment**() property).

If ***row%*** is -1, then the control assumes you want to set the alignment of all columns at once.

To set the alignment of the fixed parts of a column, use the **FixedAlignment**() property.

To set individual cell alignments, use the **CellAlignment** property.

To set column alignments at design time, use the **FormatString** property.

Data Type Integer

ColData, RowData Properties

Description Sets/returns a long value with user-defined information.

Usage [***form!***]vsFlexArray.**ColData**(***col%***)[= ***setting&***]
[***form!***]vsFlexArray.**RowData**(***row%***)[= ***setting&***]

Remarks Use **RowData**() and **ColData**() to associate a specific number with each row or column on a vsFlexArray. You can then use these numbers in code to identify the items.

A typical use for **RowData**() is to keep indices into an array of data structures associated with each row.

If you use vsFlexArray's built-in subtotalling and outlining features, **RowData**() is used to store the subtotal level (see the **Outline** and **SubTotal** methods).

Data Type Long

ColIsVisible, RowIsVisible Property [2]

Description Returns whether a given row or column is currently within view.

Usage
variable% = [***form!***]vsFlexArray.**ColIsVisible**(***col%***)
variable% = [***form!***]vsFlexArray.**RowIsVisible**(***row%***)

Remarks These properties only determine whether the specified column or row are within the visible area of the control or whether they have been scrolled off the visible part of the control. If a row has zero height or is collapsed but is within view, **RowIsVisible**() will return **True**.

These properties are provided for backward compatibility with the Microsoft Grid control.

Data Type Boolean

Collapsed Event [2]

Description Fired after the user expands or collapses row groups.

Usage Sub vsFlexArray_**Collapsed**()

Remarks This event gets fired after the collapsed state of a row or group of rows changes, as a result of a call to the **Outline** method, setting the **IsCollapsed**() property, or by user interaction with the **OutlineBar**.

For details about outlining and an example, see the **OutlineBar** property.

ColPos, RowPos Property [2]

Description Returns the top of a row or left of a column relative to the edge of the control, in Twips.

Usage
variable& = [***form!***]vsFlexArray.**ColPos**(***col%***)
variable& = [***form!***]vsFlexArray.**RowPos**(***row%***)

Remarks These properties are provided for backward compatibility with the Microsoft Grid control.

Data Type Long

ColPosition, RowPosition Properties

Description Sets a new position for a row or a column.

Usage
[***form!***]vsFlexArray.**ColPosition**(***col%***)[= ***setting%***]
[***form!***]vsFlexArray.**RowPosition**(***row%***)[= ***setting%***]

Remarks The index and setting must correspond to valid row or column numbers (in the range 0 to **Rows** - 1 or **Cols** - 1) or an error will be generated.

When a column or row is moved with **ColPosition**() or **RowPosition**(), all formatting information moves with it, including width, height, alignment, colors, fonts, etc. To move text only, use the **Clip** property instead.

The example below shows how to make a column the leftmost column when the user clicks on it.

Example

```
Sub vsFlexArray_Click ()
   Dim col%

   ' find out which column was clicked
   col% = vsFlexArray.MouseCol

   ' move it all the way to the left
   vsFlexArray.ColPosition(col%) = 0
End Sub
```

Data Type Integer

Cols, Rows Properties

Description Sets/returns the total number of columns or rows.

Usage [***form!***]vsFlexArray.**Cols**[= ***setting%***]
[***form!***]vsFlexArray.**Rows**[= ***setting%***]

Remarks You can use these properties to expand and shrink a vsFlexArray dynamically at run time.

The minimum number of rows and columns is 0. The maximum number is limited by the memory available on your computer, as long as the total number of cells (rows times columns) is smaller than 250,000.

If FlexArray runs out of memory while trying to add rows, columns, or cell contents, it will trigger a VB error. To make sure your code works properly when dealing with large arrays, you should add error-trapping code to your programs.

Note that the **Cols**, **Rows** properties are not the same as the **Col**, **Row** properties.

Data Type Integer

ColSel, RowSel Properties

Description Sets/returns the limits of a range.

Usage [***form!***]vsFlexArray.**ColSel**[= ***setting%***]
[***form!***]vsFlexArray.**RowSel**[= ***setting%***]

Remarks Use these properties to select a specific region of the control from code, or to determine the dimensions of an area that the user has selected.

The cursor it the cell at **Row**, **Col**. The selection is the region between rows **Row** and **RowSel** and columns **Col** and **ColSel**. Note that **RowSel** may be above or below **Row**, and **ColSel** may be to the left or to the right of **Col**.

Whenever you set the **Row** and **Col** properties, **RowSel** and **ColSel** are automatically reset so the cursor becomes the current selection. If you want to select a block of cells from code, you must set the **Row** and **Col** properties first, then set **RowSel** and **ColSel** (or use the **Select** method do do it all with a single statement).

Data Type Integer

ColWidth Property

Description Sets/returns the width of the specified column in Twips.

Usage [***form!***]vsFlexArray.**ColWidth**(***col%***)[= ***setting&***]

Remarks Use this property to set the width of a column at run time. For instructions on setting column widths at design-time, see the **FormatString** property.

You can set **ColWidth**() to zero to create invisible columns, or to -1 to reset the column width to its default value, which depends on the size of the current font.

If ***col%*** is -1, then the control assumes you want to set the width of all columns to the specified value.

Data Type Long

ComboList Property [2]

Description Sets/returns the contents of the cell editor's combo list.

Usage [***form!***]vsFlexArray.**ComboList**[= ***setting$***]

Remarks This property allows the user to change a cell's contents by picking values from a combo list.

To use it, the **Editable** property must be set to **True**, and you must respond to the **BeforeEdit** event by setting the **ComboList** property to a string containing the list of options, separated by pipe characters ("|"). If you want the user to edit the cell using a regular text box, set the **ComboList** property to an empty string ("").

The example below shows how this works. The first column contains a list of names, which are picked from combo lists, and the second column contains values, which are entered using a regular text box.

Example

```
Private Sub fa_BeforeEdit(ByVal row%, ByVal col%, cancel%)
  Dim s$

  ' build choice list
  s$ = "Robert|John Paul|Jimmy|Bonzo"

  Select Case col

      ' use combo list for col 0
      Case 0: fa.ComboList = s$

      ' use text box for other columns
      Case Else: fa.ComboList = ""
  End Select
End Sub
```

Data Type String

Compare Event

Description Fired when the **Sort** property is set to **flexSortCustom**, to compare rows.

Usage Sub vsFlexArray_**Compare**(***row1%***, ***row2%***, ***compare%***);

Remarks When the **Sort** property is set to **flexSortCustom**, this event is fired several times, to compare row pairs. The event handler should compare rows ***row1%*** and ***row2%*** and return the result in the ***compare%*** parameter:

-1	if ***row1%*** should appear before ***row2%***
0	if the rows are equal
+1	if ***row1%*** should appear after ***row2%***.

Note that custom sorts are orders of magnitude slower than the built in sorts, so you should avoid using them unless you data sets are small.

See also the **Sort** property.

DataSource Property [2]

Description Sets/returns the data source.

Usage [***form!***]vsFlexArray.**DataSource**[= ***dataControl***]

Remarks To bind a vsFlexArray to a table in a database, you must specify a Data control in the **DataSource** property at design time using the Properties window. This property is not available at run time.

The vsFlexArray data binding is read-only: changes made to the control are not written back automatically to the database. If you wish to update the database, you must write code to do that.

The contents of a data-bound vsFlexArray are updated whenever the associated Data control is refreshed.

See also the **VirtualData** and **AutoResize** properties for more data-binding options.

Data Type Data Source

Editable Property [2]

Description Sets/returns whether the control should provide in-cell editing.

Usage [***form!***]vsFlexArray.**Editable**[= ***setting%***]

Remarks If the **Editable** property is set to **True**, the control provides in-cell editing using a regular text editor or a drop-down list box.

By default, the control goes into editing mode when the user presses the edit key (F2), the space bar, or any printable character. You may force the control into cell editing mode by using the **EditCell** method, or prevent it from entering edit mode by trapping the **BeforeEdit** event and setting the ***cancel%*** parameter to **True**.

The editor used is a text editor or a drop-down list, depending on the setting of the **ComboList** property. You may set this property in response to the **BeforeEdit** event.

You may perform data validation in response to the **Validate** event, and perfrom post-editing work such as re-sorting the control in response to the **AfterEdit** event.

See the **ComboList** property and **EditCell** method for examples of cell-editing features.

Data Type Boolean

EditCell Method [2]

Description Puts the control in cell-editing mode.

Usage [***form!***]vsFlexArray.**EditCell**

Remarks If the **Editable** property is set to **True**, the control goes into editing mode automatically when the user presses the edit key (F2), the space bar, or any printable character. You may use the **EditCell** method to force the control into cell editing mode.

Note that **EditCell** will force the control into editing mode even if the **Editable** property is set to **False**.

A typical use for this method is shown in the example below. The code traps the right mouse button to initiate editing.

Example

```
Sub fa_MouseDown(Button%, Shift%, X!, Y!)
  If Button = 2 Then
      fa.Select fa.MouseRow, fa.MouseCol
      fa.EditCell
  End If
End Sub
```

EnterCell Event

Description Fired when a cell becomes active.

Usage Sub vsFlexArray_**EnterCell**()

Remarks This event occurs whenever the user clicks a cell other than the selected cell or when you programmatically change the active cell within a selection.

See also the **LeaveCell** event.

ExtendLastCol Property [2]

Description Sets/returns whether the last column should be adjusted to fit the control width.

Usage [***form!***]vsFlexArray.**ExtendLastCol**[= ***setting%***]

Remarks This property only affects painting. It does not modify the **ColWidth**() property for the last column.

Data Type Boolean

FillStyle Property

Description Sets/returns whether changes to the **Text** or Cell formatting properties apply to the current sell or to the selection.

Usage [***form!***]vsFlexArray.**FillStyle**[= ***setting%***]

Settings The **FillStyle** property settings are:

flexFillSingle
flexFillRepeat

Remarks If **FillStyle** is set to **flexFillSingle**, then setting the **Text** property or any of the Cell formatting properties affects the current cell only.

If **FillStyle** is set to **flexFillRepeat**, then setting the **Text** property or any of the Cell formatting properties affects the whole selected range.

The **FillStyle** property also determines whether changes caused by in-cell editing should apply to the current cell only or to the entire selection.

Data Type Integer

FixedAlignment Property [2]

Description Sets/returns the alignment for the fixed rows in a column.

Usage [***form!***]vsFlexArray.**FixedAlignment**(***col%***)[= ***setting%***]

Settings The **FixedAlignment** property settings are:

flexAlignLeftTop
flexAlignLeftCenter
flexAlignLeftBottom
flexAlignCenterTop
flexAlignCenterCenter
flexAlignCenterBottom
flexAlignRightTop
flexAlignRightCenter
flexAlignRightBottom
flexAlignGeneral(strings to left, numbers to right)

Remarks The **FixedAlignment**() property behaves like the **ColAlignment**() property except that it only affects the alignment of fixed cells. You can use **FixedAlignment**() to align headings differently from the rest of the columns.

You can also use the **CellAlignment** property to control the alignment of individual cells.

This property is provided for backward compatibility with the Microsoft Grid control.

Data Type Integer

FixedCols, FixedRows Properties

Description Sets/returns the total number of fixed (non-scrollable) columns or rows.

Usage [***form!***]vsFlexArray.**FixedCols**[= ***setting%***]
[***form!***]vsFlexArray.**FixedRows**[= ***setting%***]

Remarks A fixed column is a stationary column on the left side of the control. A fixed row is a stationary row along the top of the control. You can have zero or more fixed columns and zero or more fixed rows.

Fixed columns and rows do not move when the other columns or rows in the control are scrolled. You can select the colors, font, grid and text style use for the fixed columns and rows.

Fixed columns and rows are typically used in spreadsheet applications to display row numbers and column letters or in database applications to show field names.

Data Type Integer

FloodColor Property [2]

Description Sets/returns the color to be used for flooding cells.

Usage [***form!***]vsFlexArray.**FloodColor**[= ***setting&***]

Remarks The color specified is used for painting the flooded portion of cells which have the **CellFloodPercent** property set to a non-zero value.

To control the flooding color of individual cells, set the **CellFloodColor** property.

For details and an example, see the **CellFloodPercent** property.

Data Type Color

FocusRect Property

Description Sets/returns the type of focus rectangle that should be drawn around the current cell.

Usage [***form!***]vsFlexArray.**FocusRect**[= ***setting%***]

Settings The **FocusRect** property settings are:

flexFocusNone
flexFocusLight
flexFocusHeavy

Remarks If a focus rectangle is drawn, then the current cell is painted in the background color, as in most spreadsheets and grids. Otherwise, the current cell is painted in the selection color, so you can see which cell is selected even without the focus rectangle.

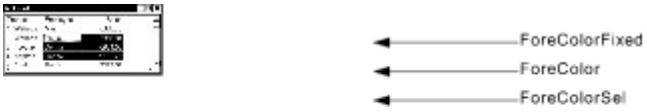
Data Type Integer

ForeColor* Properties

Description Sets/returns the colors used to draw text on each part of the control.

Usage
[***form!***]vsFlexArray.**ForeColor**[= ***setting&***]
[***form!***]vsFlexArray.**ForeColorFixed**[= ***setting&***]
[***form!***]vsFlexArray.**ForeColorSel**[= ***setting&***]

Remarks The picture below shows what part of the control each property refers to:



To set the text color of individual cells, use the **CellForeColor** property.

Data Type Color

FormatString Property

Description Sets up the control's column widths, alignments, and fixed row and column text at design time.

Usage [***form!***]vsFlexArray.**FormatString**[= ***setting&***]

Remarks Use **FormatString** at design time to define the following elements of the control: number of rows and columns, text for row and column headings, column width, and column alignment.

The **FormatString** is made up of segments separated by pipe characters ("|"). The text between pipes defines a column, and it may contain the special alignment characters "<", "^", or ">", to align the entire column to the left, center, or right. The text is assigned to row zero, and its width defines the width of each column.

The **FormatString** may also contain a semi-colon (";"), which causes the remaining of the string to be interpreted as row heading and width information. The text is assigned to column zero, and the longest string defines the width of column zero.

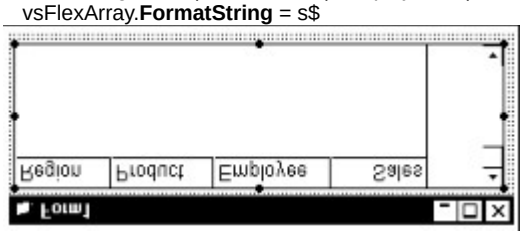
If the first character in the **FormatString** is an equals sign ("="), then all non-fixed rows will have the same width.

vsFlexArray will create additional rows and columns to accommodate all fields defined by the **FormatString**, but it will not delete rows or columns if only a few fields are specified.

The examples below illustrate how the **FormatString** property works.

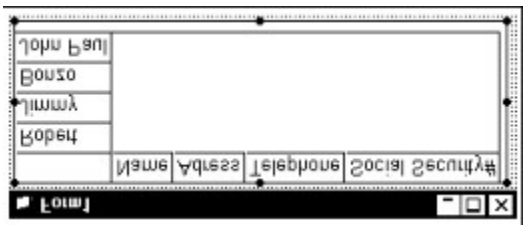
Example

```
' set column headers
s$ = "<Region      |<Product      |<Employee     |>Sales          "
vsFlexArray.FormatString = s$
```



```
' set row headers (note semicolon at start)
s$ = ";Name|Adress|Telephone|Social Security#"
vsFlexArray.FormatString = s$
```

```
'   set column and row headers
s$ = "|Name|Adress|Telephone|Social Security#"
s$ = s$ & ";|Robert|Jimmy|Bonzo|John Paul"
vsFlexArray.FormatString = s$
```



Data Type String

GridColor, GridColorFixed Properties

Description Sets/returns the color used to draw the grid lines.

Usage [***form!***]vsFlexArray.**GridColor**[= ***setting&***]
[***form!***]vsFlexArray.**GridColorFixed**[= ***setting&***]

Remarks The **GridColor** property is ignored when **GridLines** is set to one of the 3D styles. Raised and inset grid lines are always drawn in black and white.

Data Type Color

GridLines, GridLinesFixed Properties

Description Sets/returns what type of lines should be draw between cells.

Usage [***form!***]vsFlexArray.**GridLines**[= ***setting%***]
[***form!***]vsFlexArray.**GridLinesFixed**[= ***setting%***]

Settings The **GridLines, GridLinesFixed** property settings are:

flexGridNone
flexGridFlat
flexGridInset
flexGridRaised
flexGridFlatHorz
flexGridInsetHorz
flexGridRaisedHorz
flexGridSkipHorz
flexGridFlatVert
flexGridInsetVert
flexGridRaisedVert
flexGridSkipVert

Remarks **GridColor** determines the color of the grid lines when the **GridLines** property is set to one of the flat styles (**flexGridFlat**, **flexGridFlatHorz**, flexGridVert). Raised and inset grid lines are always drawn in black and white.

See also the **GridLineWidth** property.

Data Type Integer

GridLineWidth Property [2]

Description Sets/returns the width of the grid lines, in Pixels.

Usage [***form!***]vsFlexArray.**GridLineWidth**[= ***setting%***]

Remarks **GridLineWidth** determines the thickness, in pixels, of the grid lines when the **GridLines** property is set to one of the flat styles (**flexGridFlat**, **flexGridFlatHorz**, **flexGridFlatVert**). Raised and inset grid lines are always drawn with a width of 1 pixel.

This property is provided for backward compatibility with the Microsoft Grid control.

Data Type Integer

HighLight Property

Description Sets/returns whether selected cells should be highlighted.

Usage [***form!***]vsFlexArray.**HighLight**[= ***setting%***]

Settings The **HighLight** property settings are:

flexHighlightNever
flexHighlightAlways
flexHighlightWithFocus

Remarks When this property is set to **flexHighlightNever** and the user selects a range of cells, there is no visual cue to show which cells are selected.

Data Type Integer

IsCollapsed Property [2]

Description Sets/returns the collapsed state of a row.

Usage [***form!***]vsFlexArray.**IsCollapsed**(***row%***)[= ***setting%***]

Settings The **IsCollapsed** property settings are:

flexOutlineExpanded	show everything
flexOutlineSubtotals	show subordinate subtotals, hide data
flexOutlineCollapsed	hide subordinate subtotals

Remarks Read this property to determine whether a row is visible or has been collapsed and is therefore hidden from view.

Set this property to expand or collapse an outline programmatically.

When collapsing or expanding, the setting applies to the entire group of rows under the same subtotal row. When you set this property, the control fires the **Collapsed** event.

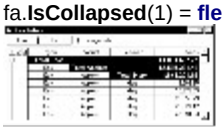
If you set this property and there are no subtotal rows in the control, a runtime error will occur.

The example below shows the effect of each setting.

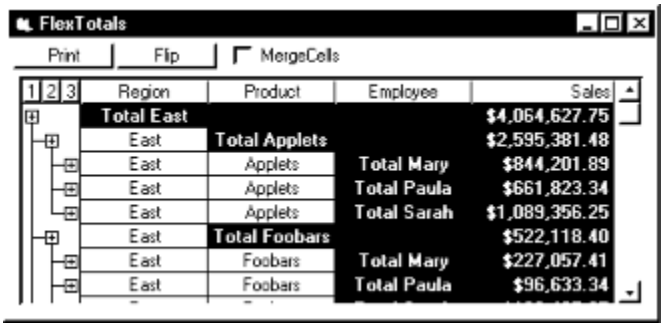
See also the **Outline** and **Subtotal** methods.

Example

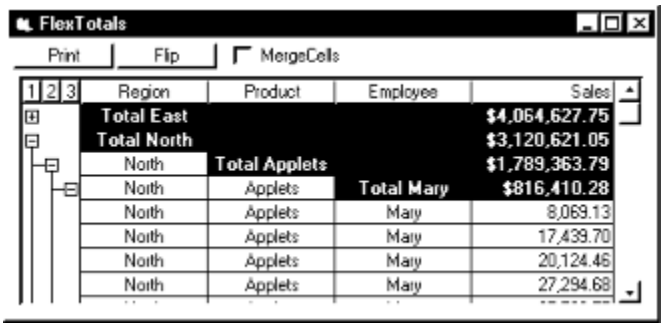
```
' show all sales in East region
fa.IsCollapsed(1) = flexOutlineExpanded
```



```
' show all subtotals for East region
fa.IsCollapsed(3) = flexOutlineSubtotals
```




```
' show only the total for East region
fa.IsCollapsed(3) = flexOutlineCollapsed
```



Data Type Integer

IsSelected Property [2]

Description Sets/returns whether a row is selected (for ListBox-type selections).

Usage [***form!***]vsFlexArray.**IsSelected**(***row%***)[= ***setting%***]

Remarks This property allows you to select individual rows, not necessarily adjacent, independently of the **ColSel** and **RowSel** properties.

To implement this type of row selection, you will normally set the **SelectionMode** property to **flexSelectionListBox**, which allows the user to select individual rows using the mouse or the keyboard, and to toggle the selection for a row by ctrl-clicking on it.

If you set **SelectionMode** to something other than **flexSelectionListBox**, you may still select and de-select rows using the **IsSelected**() property, but the user will not be able to alter the selection with the mouse or keyboard (unless, of course, you write the code to do it).

Data Type Boolean

IsSubtotal Property [2]

Description Sets/returns whether a row contains subtotals (as opposed to data).

Usage [***form!***]vsFlexArray.**IsSubtotal**(***row%***)[= ***setting%***]

Remarks This property allows you to determine whether a given row is a regular or subtotal row. The difference between the two is that subtotal rows are used to provide outlining and that they may be added and removed automatically with the **Subtotal** method.

You may set up your own custom subtotal rows by setting this property to **True** and setting the **RowData**() property to indicate the outline levels. vsFlexArray will take of collapsing and expanding the outline automatically.

See also the **Outline** and **Subtotal** methods.

Data Type Boolean

LeaveCell Event

Description Fired before the currently active cell changes to a different cell.

Usage Sub vsFlexArray_**LeaveCell**()

Remarks This event is useful if you want to implement custom cell-editing capabilities. In this case, you can trap this event to validate and apply changes to a cell before the user activates another cell.

The code below shows how this can be done. It assumes that there is a PictureBox control that is used to edit the contents of the current cell.

See also the **EnterCell** event.

Example

```
Sub FlexArray_LeaveCell ()

  ' if the picture box is up, copy its contents
  If PictureBox.Visible Then
    vsFlexArray.CellPicture = PictureBox
    PictureBox.Visible = False
  End If
End Sub
```

LeftCol Property

Description Sets/returns the leftmost visible column (other than a fixed column) in the control.

Usage [***form!***]vsFlexArray.**LeftCol**[= ***setting%***]

Remarks Use this property to scroll a vsFlexArray programmatically. Use the **TopRow** property to determine the topmost visible row.

When setting this property, the largest possible column number is the total number of columns minus the number of columns that will fit the display. Attempting to set **LeftCol** to a greater value will cause the control to set it to the largest possible value.

If you need to ensure that a certain cell is visible, do not use this property. Simply make the cell current by setting the **Row** and **Col** properties, then bring it into view by reading the **CellTop** property.

Data Type Integer

MergeCells Property

Description Sets/returns whether cells with the same contents should be merged in a single cell.

Usage [***form!***]vsFlexArray.**MergeCells**[= ***setting%***]

Settings The **MergeCells** property settings are:

flexMergeNever
flexMergeFree
flexMergeRestrictRows
flexMergeRestrictColumns
flexMergeRestrictAll

Remarks The vsFlexArray cell merging technology allows you to present data in a clear, appealing way.

To use it, you must set the **MergeCells** property to a value other than **flexMergeNever**, and then set the **MergeRow**() and **MergeCol**() array properties to **True** for the rows and columns you wish to merge.

The control will then merge cells with the same contents, and will update the merging automatically to reflect changes to the contents of any cells.

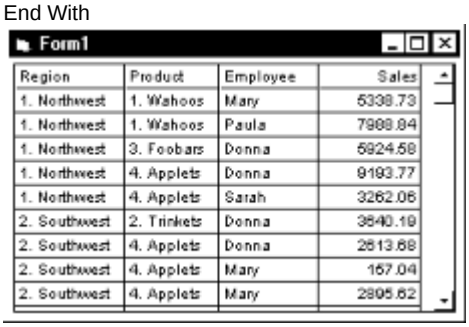
When **MergeCells** is set to a value other than **flexMergeNever**, the control does not highlight the current selection. This is done because merged cells can be part within the selection, and part outside.

The difference between the Free and Restricted settings is whether cells with the same contents should always be merged (Free settings) or only when adjacent cells to the left or to the top are also merged.

The examples below illustrate the difference.

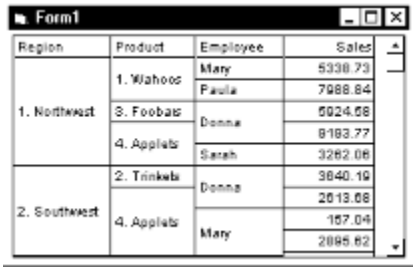
Example

```
' regular spreadsheet view
With vsFlexArray
   .MergeCells = flexMergeNever
   .MergeRow(0) = True
   .MergeRow(1) = True
   .MergeRow(2) = True
   .MergeRow(3) = False
End With
```

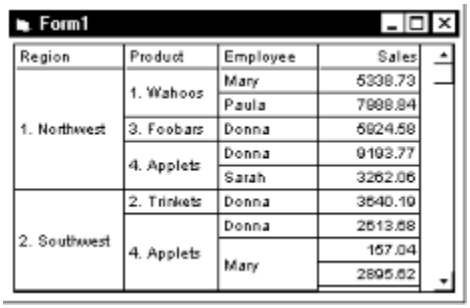



```
' free merging
' notice how the third employee cell
' (Donna) merges across products to
' its left and across sales to its right.
With vsFlexArray
   .MergeCells = flexMergeFree
   .MergeRow(0) = True
   .MergeRow(1) = True
   .MergeRow(2) = True
   .MergeRow(3) = False
```

End With




```
' restricted merging
' notice how the third employee cell
' (Donna) no longer merges across
' sales.
With vsFlexArray
   .MergeCells = flexMergeRestrictAll
   .MergeRow(0) = True
   .MergeRow(1) = True
   .MergeRow(2) = True
   .MergeRow(3) = False
End With
```



Data Type Integer

MergeCol, MergeRow Properties

Description Sets/returns whether a row (or column) should have its cells merged (see also the **MergeCells** property).

Usage
[***form!***]vsFlexArray.**MergeCol**(***col%***)[= ***setting%***]
[***form!***]vsFlexArray.**MergeRow**(***row%***)[= ***setting%***]

Remarks If the **MergeCells** property is set to a non-zero value, adjacent cells with identical values are merged if they are in a row whose **MergeRow**() property is set to **True** or in a column whose **MergeCol**() property is set to **True**.

For details and examples, see the **MergeCells** property.

Data Type Boolean

MouseCol, MouseRow Properties

Description Sets/returns the row (or column) over which the mouse pointer is.

Usage [***form!***]vsFlexArray.**MouseCol**[= ***setting%***]
[***form!***]vsFlexArray.**MouseRow**[= ***setting%***]

Remarks These properties return the mouse pointer coordinates in terms of rows and columns.

You may trap the MouseMove event and use these properties to display context-sensitive help for individual cells, or to test whether the user has clicked on a fixed row or column.

Data Type Integer

Outline Method [2]

Description Sets an outline level for displaying subtotals.

Usage [***form!***]vsFlexArray.**Outline** ***level%***

Remarks This method collapses or expands an outline so that only subtotals of level ***level%*** or lower are displayed.

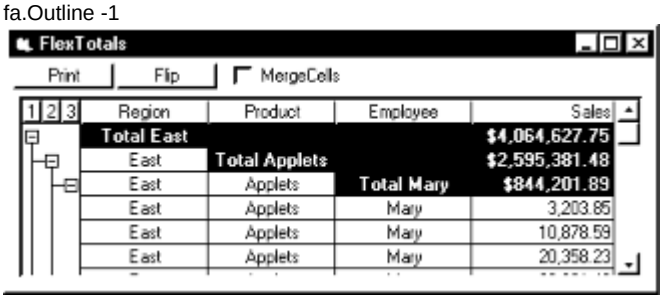
If ***level%*** is negative, then the outline is totally expanded.

To set up an outline structure using automatic subtotals, see the **Subtotal** method. To set up a custom outline structure, see he **IsSubtotal**() property.

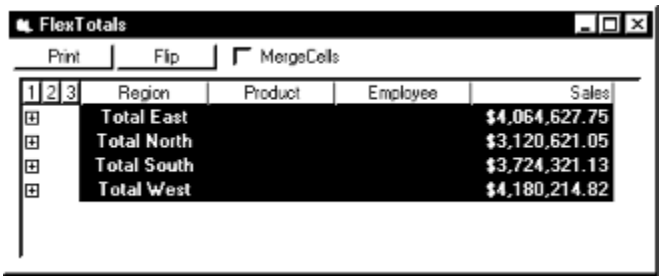
See also the **OutlineBar** property.

Example

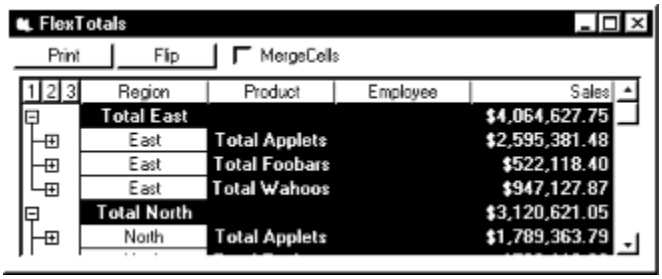
```
' expand the outline
fa.Outline -1
```



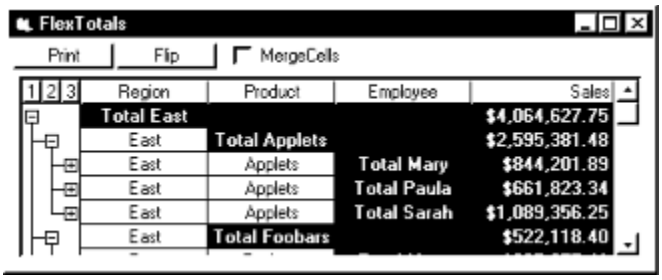

```
' show level 1 subtotals only
fa.Outline 1
```




```
' show levels 1 and 2 subtotals only
fa.Outline 2
```




```
' show levels 1, 2, and 3 subtotals only
fa.Outline 3
```


FlexTotals
Print
Flip
MergeCells
1
2
3
Region
Product
Employee
Sales
Total East
$4,064,627.75
East
Total Applets
$2,595,381.48
East
Applets
Total Mary
$844,201.89
East
Applets
Total Paula
$661,823.34
East
Applets
Total Sarah
$1,089,356.25
East
Total Foobars
$522,118.40

OutlineBar Property [2]

Description Sets/returns the type of outline bar that should be displayed.

Usage [***form!***]vsFlexArray.**OutlineBar**[= ***setting%***]

Settings The **OutlineBar** property settings are:

flexOutlineBarNone
flexOutlineBarComplete
flexOutlineBarSimple

Remarks This property determines whether the control should reserve the first fixed column for displaying an outline bar. The column's width is then set automatically by the control.

The outline bar contains a tree showing the outline structure and buttons that can be used to collapse and expand portions of the outline (similar to a Visual Basic TreeView control or to the Explorer).

Clicking on a collapsed branch expands it, clicking on an expanded branch collapses it, and shift-clicking on a branch collapses it but shows the subordinate subtotals.

The **flexOutlineBarSimple** setting shows only the outline tree. The **flexOutlineBarComplete** setting shows also a row of buttons at the top which can be used to set the outline level for the entire control with a single click (clicking collapses and shift-clicking expands to the indicated level).

After the user expands or collapses the outline using the outline bar, the controls fires the **Collapsed** event.

The color used to draw the outline tree is specified by the **TreeColor** property.

See also the **Outline** and **Subtotal** methods.

Data Type Integer

Picture Property

Description Returns a picture of the entire control (could be huge).

Usage ***picture*** = [***form!***]vsFlexArray.**Picture**

Remarks This property returns a picture (bitmap) representation of the entire control, including rows and columns that are not visible on the screen. If you have a vsFlexArray control with 1000 rows, for example, the bitmap will include all of them, and the picture will be huge.

To create a picture of a part of the control, write a routine to hide all the elements you don't want to show, get the picture, and then restore the control.

To reduce memory requirements for the bitmap and increase speed, you may consider setting the **PictureType** property to **flexPictureMonochrome**. The picture will not look as nice, but it will require less memory. The code below shows how you can trap out-of-memory errors and automatically switch to monochrome mode.

The example below shows a routine that creates a picture of the the current selection.

Example

```
Sub CopySelection(fa As Control)
  Dim i%, tr%, lc%, hl%

  ' get ready to operate
  fa.Redraw = False     ' to eliminate flicker
  hl = fa.HighLight     ' save current settings
  tr = fa.TopRow
  lc = fa.LeftCol
  fa.HighLight = 0      ' no highlight on picture

  ' hide non-selected rows and columns
  ' (saving original sizes)
  For i = fa.FixedRows To fa.Rows - 1
      If i < fa.Row Or i > fa.RowSel Then
          fa.RowData(i) = fa.RowHeight(i)
          fa.RowHeight(i) = 0
      End If
  Next
  For i = fa.FixedCols To fa.Cols - 1
      If i < fa.Col Or i > fa.ColSel Then
          fa.ColData(i) = fa.ColWidth(i)
          fa.ColWidth(i) = 0
      End If
  Next

  ' scroll to top left corner
  fa.TopRow = fa.FixedRows
  fa.LeftCol = fa.FixedCols


  ' copy picture (with error-trapping)
  Clipboard.Clear
  On Error Resume Next
  fa.PictureType = flexPictureColor
  Clipboard.SetData fa.Picture
  If Error <> 0 Then
     fa.PictureType = flexPictureMonochrome
     Clipboard.SetData fa.Picture
  Endif

  ' restore control
  For i = fa.FixedRows To fa.Rows - 1
      If i < fa.Row Or i > fa.RowSel Then
          fa.RowHeight(i) = fa.RowData(i)
      End If
  Next
  For i = fa.FixedCols To fa.Cols - 1
```

```
        If i < fa.Col Or i > fa.ColSel Then
            fa.ColWidth(i) = fa.ColData(i)
        End If
    Next
    fa.TopRow = tr
    fa.LeftCol = lc
    fa.HighLight = hl
    fa.Redraw = True
End Sub
```

Data Type Picture

PictureType Property

Description Sets/returns the type of picture that should be generated by the **Picture** property.

Usage [***form!***]vsFlexArray.**PictureType**[= ***setting%***]

Settings The **PictureType** property settings are:

flexPictureColor
flexPictureMonochrome

Remarks Set this property to **flexPictureColor** to obtain a high-quality image. Set it to **flexPictureMonochrome** to obtain a lower-quality image that consumes less memory and is faster to print and render.

For an example, see the **Picture** property.

Data Type Integer

Redraw Property

Description Enables or disables redrawing of the FlexArray control.

Usage [***form!***]vsFlexArray.**Redraw**[= ***setting%***]

Remarks Use this property to reduce flicker while making extensive updates to the contents of the control.

For example, the code below turns repainting off, makes several changes to the contents of the control, and then turns repainting back on to show the results.

Example

```
' freeze control to avoid flicker
fa.Redraw = False

' update vsFlexArray contents
For i% = fa.FixedRows To fa.Rows - 1
   fa.TextMatrix(i%, 1) = GetName(i%, 1)
   fa.TextMatrix(i%, 2) = GetName(i%, 2)
Next

' show results
fa.Redraw = True
```

Data Type Boolean

RemoveItem Method

Description Removes a row from the control.

Usage [***form!***]vsFlexArray.**RemoveItem** ***row%***

Remarks The **RemoveItem** method has these parts:

row%
Integer representing the row to remove. To remove the first row, use index = 0.

RightCol Property [2]

Description Returns the last visible column in the control.

Usage ***setting%*** = [***form!***]vsFlexArray.**RightCol**

Remarks The column returned may be only partially visible.

You cannot set this property. To scroll the contents o fthe control through code, set the **TopRow** and **LeftCol** properties instead.

Data Type Integer

RowColChange Event

Description Fired when the currently active cell changes to a different cell.

Usage Sub vsFlexArray_**RowColChange**()

Remarks This event occurs whenever the user clicks a cell other than the selected cell or when you programmatically change the active cell within a selection.

You can trigger this event in code by changing the current cell using the **Col** and **Row** properties.

The **RowColChange** event does not occur when the selected range changes but the active cell remains the same.

RowHeight Property

Description Sets/returns the height of the specified row in Twips.

Usage [***form!***]vsFlexArray.**RowHeight**(***row%***)[= ***setting&***]

Remarks You can set **RowHeight** to zero to create invisible rows, or to -1 to reset the row height to its default value, which depends on the size of the current font.

If ***row%*** is -1, then the control assumes you want to set the height of all rows at once.

Data Type Long

RowHeightMin Property

Description Sets/returns a minimum row height for the entire control, in Twips.

Usage [***form!***]vsFlexArray.**RowHeightMin**[= ***setting&***]

Remarks Use this property if you wish to use small fonts but want the rows to be tall. Setting this property is often easier than setting individual row heights with the **RowHeight**() property.

Data Type Long

Scroll Event

Description Fired after the control scrolls.

Usage Sub vsFlexArray_**Scroll**()

Remarks Use this event to synchronize the scrolling of multiple controls.

ScrollBars Property

Description Sets/returns whether the control should display horizontal or vertical scroll bars.

Usage [***form!***]vsFlexArray.**ScrollBars**[= ***setting%***]

Settings The **ScrollBars** property settings are:

flexScrollBarNone
flexScrollBarHorizontal
flexScrollBarVertical
flexScrollBarBoth

Remarks Scroll bars appear on a vsFlexArray only if its contents extend beyond the controls borders. For example, a vertical scroll bar appears when the vsFlexArray can't display all of its rows.

If **ScrollBars** is set to **flexScrollBarNone**, the control will not have scroll bars, regardless of its contents.

Note that if the control has no scroll bars in either direction, it will not allow any scrolling in that direction, even if the user uses the keyboard to select a cell that is outside the visible area of the control. (However, you may still scroll the control through code by setting the **TopRow** and **LeftCol** properties.)

Data Type Integer

ScrollTrack Property

Description Sets/returns scrolling should occur while the user moves the scroll thumb.

Usage [***form!***]vsFlexArray.**ScrollTrack**[= ***setting%***]

Remarks This property should normally be set to **False** to avoid excessive scrolling and flickering. Set it to **True** if you want to emulate other controls that have this behavior.

Data Type Boolean

SelChange Event

Description Fired after the selected range changes.

Usage Sub vsFlexArray_**SelChange**()

Remarks The **SelChange** event occurs whenever the user clicks a cell other than the selected cell or after the user selects a new range of cells with the mouse or keyboard.

The **SelChange** event does not occur while the user extends the selection with the mouse.

You can trigger this event in code by changing the selected region using the **Row**, **Col**, **RowSel**, or **ColSel** properties.

Select Method [2]

Description Selects a range with a single command.

Usage [***form!***]vsFlexArray.**Select** ByVal ***row1%***, ByVal ***col1%***, ByVal ***row2%***, ByVal ***col2%***

Remarks This method allows you to select ranges or cells (by omitting the last two paramters) with a single command. It is shorter and more efficient than setting the **Row**, **Col**, **RowSel**, and **ColSel** properties separately.

SelectionMode Property

Description Sets/returns whether to use regular, row, column, or ListBox selection.

Usage [***form!***]vsFlexArray.**SelectionMode**[= ***setting%***]

Settings The **SelectionMode** property settings are:

flexSelectionFree
flexSelectionByRow
flexSelectionByColumn
flexSelectionListBox

Remarks Setting **flexSelectionFree** allows selections to be made normally, spreadsheet-style.

Setting **flexSelectionByRow** forces selections to span entire rows, as in a record-based display.

Setting **flexSelectionByColumn** forces selections to span entire columns, as if selecting ranges for a chart or fields for sorting.

Setting **flexSelectionListBox** forces selections to span entire rows and allows for extended selections spanning non-adjacent rows. Control-clicking with the mouse toggles the selection for an individual row. The **IsSelected**() property allows programmatic control over extended selections.

Data Type Integer

Sort Property

Description Sets a sorting order for the selected rows using the selected columns as keys.

Usage [***form!***]vsFlexArray.**Sort** = ***setting%***

Settings The **Sort** property settings are:

flexSortNone
flexSortGenericAscending
flexSortGenericDescending
flexSortNumericAscending
flexSortNumericDescending
flexSortStringNoCaseAscending
flexSortStringNoCaseDescending
flexSortStringAscending
flexSortStringDescending
flexSortCustom

Remarks The **Sort** property always sorts entire rows.

The range of rows to be sorted is specified by setting the **Row** and **RowSel** properties. If **Row** and **RowSel** are the same, the control assumes that you want to sort all non-fixed rows.

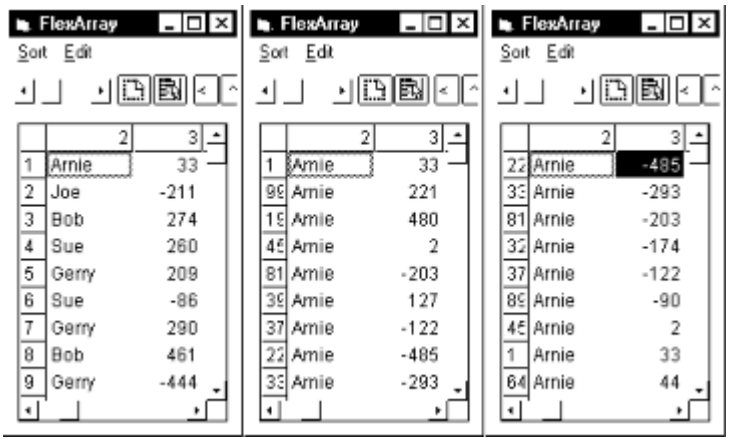
They keys used for sorting are determined by the **Col** and **ColSel** properties, always from the left to the right. For example, if **Col** = 3 and **ColSel** = 1, the sort would be done according to the contents of columns 1, then 2, then 3.

The method used to compare the rows is determined by the setting, as explained above. The **flexSortCustom** setting is the most flexible, since it fires a **Compare** event that allows you to compare rows in any way you want, using any columns in any order (see the **Compare** event for details). However, this method is also much slower that the others, typically by a factor of ten, so it should be used only when really necessary.

An alternative to using the **flexSortCustom** setting is to create an invisible column, fill it with the keys, then sort based on it with one of the other settings. This is a very good approach for sorting based on dates, for example.

Example

```
' fill control with random data (left picture)
For i% = fa.FixedRows to fa.Rows - 1
   fa.TextMatrix(i%, 1) = RandomName()
   fa.TextMatrix(i%, 2) = RandomNumber()
Next

' sort by name (center picture)
fa.Row = 1
fa.Col = 1
fa.Sort = flexSortGenericAscending

' sort by name and number (right picture)
fa.Row = 1
fa.Col = 1
fa.ColSel = 2
fa.Sort = flexSortGenericAscending
```



Data Type Integer

Subtotal Method [2]

Description Calculates subtotals.

Usage [***form!***]vsFlexArray.**Subtotal** ***function%*** [, ***groupOn%***] [, ***totalOn%***] [, ***format$***] [, ***backColor&***] [, ***foreColor&***] [, ***fontBold%***]

Remarks The **Subtotal** method has the following parts:

function%
Specifies the type of function to be used for the subtotals. Valid settings are:

flexSTClear	removes all existing subtotals
flexSTSum	sum
flexSTPercent	percent of total sum
flexSTCount	record count
flexSTAverage	average
flexSTMax	maximum
flexSTMin	minimum
flexSTStd	standard deviation
flexSTVar	variance

groupOn%
Specifies the column that contains the categories that should be grouped together. A subtotal row will be inserted every time the value in this column changes. To calculate grand totals, set ***groupOn%*** to -1, as the example below shows.

totalOn%
Specifies the column that contains the values to be calculated.

format$
Specifies the format to be used for displaying the results. The syntax for the format string is similar but not identical to the syntax used with Visual Basic's Format command. The rules used to format values are as follows:
If the format string contains a:
"$" (dollar sign) then a locale-dependent currency sign is prepended to the output.
"," (comma) then locale-dependent thousand separators are added to the output.
"(" (parentheses) then negative values are displayed within parentheses.
"." (decimal point) then the number of decimals is determined by the number of zero or pound ("0" or "#") characters after the decimal point.

[***backColor&***][, ***foreColor&***]
Specify the colors to be used for the cells in the subtotal rows.

[***fontBold%***]
Specifies whether text in the subtotal rows should be boldfaced.

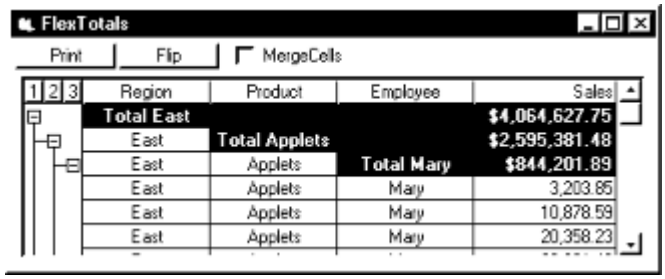
All parts except ***function%*** are optional, but you must specify at least ***groupOn%***, ***totalOn%***, and ***format$*** unless the ***function%*** is **flexSTClear**, in which case they are unnecessary.

The example below shows how to use the **Subtotal** method.

Example

```
' remove old totals, calculate new totals
Sub DoSubtotals()
   Dim fmt$, fn%

   ' set format for calculated totals
   fmt$ = "$(#,###.00)"

   ' set function to use for totaling
   fn = flexSTSum

   ' remove any existing junk
   fa.Subtotal flexSTClear
```

```
  ' calculate fresh subtotals (the order doesn't matter)
  ' (sales values are in column 4)
  fa.Subtotal fn%, 1, 4, fmt$, RED    ' col 1: region
  fa.Subtotal fn%, 2, 4, fmt$, GREEN ' col 2: product
  fa.Subtotal fn%, 3, 4, fmt$, BLUE ' col3: employee

  ' total on a negative column to get a grand total
  fa.Subtotal fn%, -1, 4, fmt$, BLUE
End Sub
```



SubtotalPosition Property [2]

Description Sets/returns whether subtotals should be inserted above or below the data.

Usage [***form!***]vsFlexArray.**SubtotalPosition**[= ***setting%***]

Settings The **SubtotalPosition** property settings are:

flexSTBelow
flexSTAbove

Remarks If you modify this property at run time, any existing subtotals are cleared, and you must use the **Subtotal** method to regenerate them.

For more details on subtotals and an example, see the **SubTotal** method.

Data Type Integer

Text Property

Description Sets/returns the text contents of a cell or range of cells.

Usage [***form!***]vsFlexArray[.**Text**][= ***setting$***]

Remarks When retrieving, the **Text** property always retrieves the contents of the current cell, defined by the **Row** and **Col** properties.

When setting, the **Text** property sets the contents of the current cell or of the current selection, depending on the setting of the **FillStyle** property.

You may read or set the contents of an arbitrary cell using the **TextMatrix**() property.

You may read the value of a cell formatted with thousand separators using the **Value** or **ValueMatrix**() properties.

Data Type String

TextArray Property

Description Sets/returns the contents of an arbitrary cell (see also the **TextMatrix** property).

Usage [***form!***]vsFlexArray.**TextArray**(***index&***)[= ***setting$***]

Remarks This property is obsolete and is provided for backward compatibility only. (it was introduced for use with the VBX version of vsFlexArray version 1). You should use the TexMatrix property instead.

Use this property to set or retrieve the contents of a cell without changing the **Row** and **Col** properties.

The index& parameter determines which cell to use. It is calculated by multiplying the desired row by the **Cols** property and adding the desired column. The clearest and most convenient way to calculate the index is to define a function to do it, as shown below.

Example

```
' calculate index for use with TextArray property
Function faIndex(fa As Control, row%, col%) As Long
   faIndex = row * fa.Cols + col
End Function
```

Data Type String

TextMatrix Property

Description Sets/returns the contents of an arbitrary cell (row/col subscripts).

Usage [***form!***]vsFlexArray.**TextMatrix**(***row%***, ***col%***)[= ***setting$***]

Remarks This property allows you to set or retrieve the contents of a cell without changing the **Row** and **Col** properties.

The ***row%*** and ***col%*** parameters determine which cell to use.

Data Type String

TextStyle, TextStyleFixed Properties

Description Sets/returns 3D effects for displaying text.

Usage [***form!***]vsFlexArray.**TextStyle**[= ***setting%***]
[***form!***]vsFlexArray.**TextStyleFixed**[= ***setting%***]

Settings The **TextStyle, TextStyleFixed** property settings are:

flexTextFlat
flexTextRaised
flexTextInset
flexTextRaisedLight
flexTextInsetLight

Remarks Settings **flexTextRaised** and **flexTextInset** work best for large and bold fonts. Settings **flexTextRaisedLight** and **flexTextInsetLight** work best for small regular fonts.

See also the **CellTextStyle** property.

Data Type Integer

TopRow Property

Description Sets/returns the topmost row displayed in the control.

Usage [***form!***]vsFlexArray.**TopRow**[= ***setting%***]

Remarks Use this property to read or set the top visible row of the control, causing it to scroll if necessary. Use the **LeftCol** property to determine the leftmost visible column.

When setting this property, the largest possible value is the total number of rows minus the number of rows that will fit the display. Attempting to set **TopRow** to a greater row number will cause the control to set it to the largest possible value.

If you need to ensure that a certain cell is visible, do not use this property. Simply make the cell current by setting the **Row** and **Col** properties, then bring it into view by reading the **CellTop** property.

Data Type Integer

TreeColor Property [2]

Description Sets/returns the color used to draw the outline tree.

Usage [***form!***]vsFlexArray.**TreeColor**[= ***setting&***]

Remarks The outline tree is drawn only if the **OutlineBar** property is set to a non-zero value. It allows users to collapse and expand the outline.

For details on outlines and an example, see the **Outline** method.

Data Type Color

UserResized Event [2]

Description Fired after the user resizes a row or a column.

Usage Sub vsFlexArray_**UserResized**(ByVal ***row%***, ByVal ***col%***)

Remarks If the user resized a row, ***row%*** contains the number of the row that was resized and ***col%*** is contains -1.

If the user resized a column, ***col%*** contains the number of the column that was resized and ***row%*** contains -1.

Validate Event [2]

Description Fired before the control exits cell-editing mode.

Usage Sub vsFlexArray_**Validate**(ByVal ***row%***, ByVal ***col%***, ByVal ***value$***, ***cancel%***)

Remarks The **Validate** event is fired before any changes made by the user are committed to the cell.

You may trap this event to check whether value$ contains an entry that is valid for cell (***row%***, ***col%***). If the entry is valid, just return and the cell contents will be automatically updated. If not, set the ***cancel%*** parameter to **True** and the changes will be discarded. In this case, you will probably want to display some sort of error message and set the active cell back to (***row%***, ***col%***), since the user may have moved it with the mouse.

For more details on in-cell editing, see the **Editable** and **ComboList** properties.

The example below shows a typical use of the **Validate** event. Column 1 only accepts strings, and column 2 only accepts numbers greater than zero.

Example

```
Sub fa_Validate(ByVal row%, ByVal col%, ByVal Value$, cancel%)
  Dim c$

  ' different validation for each column
  Select Case col

    ' column 1 only accepts strings
    Case 1
      c = Left(Value)
      If UCase(c) < "A" And UCase(c) > "Z" Then
        MsgBox "Please enter a string..."
        fa.Select row, col
        Cancel = True
      End If

    ' column 2 only accepts numbers > 0
    Case 2
      If Val(Value) <= 0 Then
        MsgBox "Please enter a number > 0..."
        fa.Select row, col
        Cancel = True
      End If

  End Select
End Sub
```

Value Property [2]

Description Returns the numeric value of the active cell.

Usage ***variable#*** = [***form!***]vsFlexArray.**Value**

Remarks This property is similar to Visual Basic's Val function, except it interprets thousand separators, currency signs, and parenthesized negative values.

To retrieve the value of an arbitrary cell without selecting it first, use the **ValueMatrix**() property.

Data Type Double

ValueMatrix Property [2]

Description Returns the numeric value of an arbitrary cell.

Usage ***variable#*** = [***form!***]vsFlexArray.**ValueMatrix**(***row%***, ***col%***)

Remarks This property is similar to the **Value** property, except it allows you to specify the cell whose value is to be retrieved.

Data Type Double

Version Property

Description Returns the version of vsFlex currently loaded in memory.

Usage ***variable%*** = [***form!***]vsFlexArray.**Version**

Remarks You may want to check this value at the Form_Load event, to make sure the version that is executing is at least as current as the version used to develop your application.

The version number is a three digit integer where the first digit represents the major version number and the last two represent the minor version number. For example, version 2.00 would return 200.

This property is read-only.

Data Type Integer

VirtualData Property [2]

Description Sets/returns whether data should be loaded at once or as needed.

Usage [***form!***]vsFlexArray.**VirtualData**[= ***setting%***]

Remarks The defaul behavior for a data-bound vsFlexArray is to retrieve the entire table from the **DataSource** into memory as soon as it is loaded, and every time the source Data control is refreshed. If the data source is very large (over a thousand or so records), this process may become slow.

By setting the **VirtualData** property to **True**, data is retrieved only when is is needed (for displaying or reading its value, for example). In this case, the control loads much faster. The trade-off is that the control will probably have to retrieve the data some time later, so it may become slower until every record has been retrieved.

Data Type Boolean

WordWrap Property

Description Sets/returns whether text should wrap within a cell.

Usage [***form!***]vsFlexArray.**WordWrap**[= ***setting%***]

Remarks The vsFlexArray can display text slightly faster if you set **WordWrap** to **False**.

Data Type Boolean

Using the FlexString Control

The FlexString control allows you to incorporate regular-expression text matching into your VB programs. This allows you to parse complex text input easily or to offer regular expression search and replace features such as those found in professional packages such as Microsoft Word, Visual C++, and Visual Basic.

FlexString looks for text patterns on its **Text** property, and lets you inspect and change the matches it found. The text patterns are specified through the **Pattern** property, using regular expressions.

The syntax for **regular expressions**is described below.

Regular expressions

FlexString has a string property called **Pattern** that holds a regular expression.

A regular expression is a notation for specifying strings. Like an arithmetic expression, a regular expression is a basic expression or one created by applying operators to simpler expressions. FlexString recognizes the following operators (special characters):

Char	Meaning
^	Matches the beginning of a string.
$	Matches the end of a string.
.	Matches any character.
[ ]	Character class, or complemented character class if the first character inside the brackets is a caret (^).
*	Repeat previous zero or more times.
+	Repeat previous one or more times.
?	Repeat previous zero or one time.
\	Escape next character.
{ }	Tagged match.

The following examples illustrate how these characters are used:

Pattern	Matches
^stuff	strings that start with "stuff".
stuff$	strings that end with "stuff".
^...$	any 3-character string.
[AEIOU]	any uppercase vowel.
[0-9]	any digit.
[A-Za-z][0-9]	any letter followed by any digit.
[^0-9]	any character except a digit.
[A-Z][0-9]*	any upper-case letter optionally followed by any number of digits.
[A-Z][0-9]+	any upper-case letter followed by at least one digit.
[A-Z][0-9]?	any upper-case letter optionally followed by one digit.
[+-]?[0-9]+	any integer optionally preceded by a sign.
[+-]?[0-9]+\.?[0-9]*	any real number.

Matching

As soon as you assign a string to the **Text** or **Pattern** properties, FlexString tries to find as many matches as it can, and returns the number of matches found in the **MatchCount** property. You can then scan through the matches by changing the **MatchIndex** property and reading the **MatchString** property.

For example, the following code would scan a string and print all phone numbers in the San Francisco area (the phone pattern used in the example is not very flexible, but its good enough to show how the control works):

Example

```
Dim i%
Dim PhonePat$

FlexString.Text = ClientList$
PhonePat = "(415)[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]"
FlexString.Pattern = PhonePat
Debug.Print FlexString.MatchCount " match(es) found."
For i% = 0 to FlexString.MatchCount - 1
   FlexString.MatchIndex = I
   Debug.Print FlexString.MatchString
Next
```

Replacing

You can also replace matches automatically, using the **Replace** property. For example, say you wanted to change all instances of the (415) area code with (510):

Example

```
Dim PhonePat$

FlexString.Text = ClientList$
PhonePat = "(415)"
FlexString.Pattern = PhonePat
Debug.Print FlexString.MatchCount " match(es) found."
FlexString.Replace = "(510)"
```

When a string is assigned to the **Replace** property, FlexString immediately replaces all matches with the new string.

This is convenient, but it could also be done in VB code. But FlexString goes way beyond simple search and replace. It allows you to to **tag matches**.

Tag Matches

The FlexString control allows you to tag matches using curly brackets. By tagging the matches, you can easily determine what parts of the string matched what parts of the pattern.

For example, say you wanted to make your letters more informal by replacing all occurrences of "Mr. John Doe", "Ms Jane Doe", and "Mrs. Penny Doe", with "John", "Jane", and "Penny". Heres the code to do it:

Example

```
FlexString.Text = ClientList$

' replace "Mr. Name" or "Mr Name" with "Name"
FlexString.Pattern = "Mr\.? {[A-Za-z]+} {[A-Za-z]+}"
FlexString.Replace = "{0}" ' tag 0 matched the first name

' replace "Ms. Name", "Ms Name", "Mrs. Name", "Mrs Name", with "Name"
FlexString.Pattern = "Mr?s\.? {[A-Za-z]+} {[A-Za-z]+}"
FlexString.Replace = "{0}"
```

The curly brackets mark the tagged parts of the pattern. In this example, there are two tags, {0} and {1}, that match the persons first and last names. The first tag is used in the replace string to retrieve the persons first name. The second is not used.

FlexString Tutorial

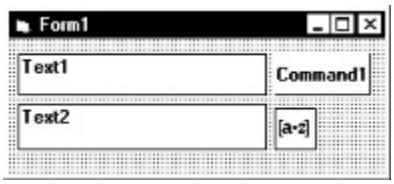
This section of the manual takes you step-by-step through the creation of a Visual Basic project using the FlexString control.

The sample project illustrates some of FlexStrings pattern matching capabilities. It shows how FlexString can be used to implement a mathematical expression evaluator. You can use this project as is, to allow users to enter expressions instead of numeric constants, or use it as a starting point for a more sophisticated evaluator with variables and custom functions.

Create Controls

Start a new Visual Basic project including vsFlex2 (if you dont know how to add OCXs to a project, consult the Visual Basic manual). The vsFlex2 control icons will be added to the Visual Basic toolbox.

Create a FlexString object on the form by clicking the FlexString icon on the toolbox, then clicking and dragging on the form. Also create two text boxes and a command button. Arrange the controls and resize the form so it looks like this:



Set Properties

Use the Properties window to set the following properties:

Object	Property
Form	Caption = "Evaluator"
	Caption = "Eval"
Command1	
Text1	Caption = "(5+3)*2"
Text2	Caption = ""
FlexString1	Name = "FS"

Evaluating expressions

Ok, now comes the fun part. Let's write the VB code to evaluate expressions and attach it to the command button.

First, double-click the command button and enter the following code:

```
        Sub Command1_Click ()
  Caption = "Evaluator"
  Text2 = Format(Evaluate(Text1), "0.00")
End Sub
```

We start by setting the form's caption to "Evaluator". We do this because the caption will also be used to show messages while we evaluate the expression.

Then, we call the Evaluate function, which we haven't written yet. The function takes as a parameter a mathematical expression and returns the result. If there are any syntax errors in the expression, the function shows the error in the form's caption.

Here's the evaluate function. Notice how it uses the FlexString's pattern matching capabilities to break the expression apart and interpret it:

```
        Function Evaluate (ByVal s$) As Double
  Dim s1$, s2$, s3$
  Dim v#
```

```
  ' get ready to parse
  fs = Trim(s)     ' set breakpoint on this line
  Debug.Print s    ' remove this line later

  ' interpret subexpressions enclosed in parentheses
  fs.Pattern = "{.*}({.*}){.*}"
  If fs.MatchCount > 0 Then
    fs.TagIndex = 0: s1 = fs.TagString          ' stuff to the left
    fs.TagIndex = 1: s2 = fs.TagString          ' stuff in brackets
    fs.TagIndex = 2: s3 = fs.TagString          ' stuff to the right
    v = Evaluate(s2)                           ' eval subexpression
    Evaluate = Evaluate(s1 + Format(v) + s3) ' eval other stuff
    Exit Function
  End If

  ' add and subtract (high-priority operators)
  fs.Pattern = "{.*}{[+-]}{.*}"
  If fs.MatchCount > 0 Then
    fs.TagIndex = 0: s1 = fs.TagString          ' operand 1
    fs.TagIndex = 2: s2 = fs.TagString          ' operand 2
    fs.TagIndex = 1                             ' operator
    Select Case fs.TagString
      Case "+": Evaluate = Evaluate(s1) + Evaluate(s2)
      Case "-": Evaluate = Evaluate(s1) - Evaluate(s2)
    End Select
    Exit Function
  End If

  ' multiply and divide (lower-priority operators)
  fs.Pattern = "{.*}{[*/]}{.*}"
  If fs.MatchCount > 0 Then
    fs.TagIndex = 0: s1 = fs.TagString          ' operand 1
    fs.TagIndex = 2: s2 = fs.TagString          ' operand 2
    fs.TagIndex = 1                             ' operator
    Select Case fs.TagString
      Case "*": Evaluate = Evaluate(s1) * Evaluate(s2)
      Case "/": Evaluate = Evaluate(s1) / Evaluate(s2)
    End Select
    Exit Function
  End If

  ' power (lowest-priority operator)
  fs.Pattern = "{.*}^{.*}"
  If fs.MatchCount > 0 Then
    fs.TagIndex = 0: s1 = fs.TagString
    fs.TagIndex = 1: s2 = fs.TagString
    Evaluate = Evaluate(s1) ^ Evaluate(s2)
    Exit Function
  End If

  ' number (nothing else matched, so this should be a number)
  fs.Pattern = "[0-9]+\.?[0-9]*"
  If fs.MatchCount > 0 Then
    Evaluate = Val(s)
  Else
    Caption = "error: Syntax": Beep
  End If
End Function
```

If you understand how FlexString works, the Evaluate function is pretty simple. To see it in action, run the project and step through the evaluation of the default expression, "(5+3)*2" by setting a breakpoint on the line that reads "fs = Trim(s)".

When the program stops, press **F8** until the first **If** statement. The current expression contains a subexpression enclosed in parentheses, so there will be a match against the pattern "{.*}({.*}){.*}". Press **F8** 6 times to extract the subexpression using the **TagIndex** property. Here's what we have at this point:

```
s = "(5+3)*2"
```

fs.**Pattern** = "{.*}({.*}){.*}"
s1 = ""
s2 = "5+3" **' parentheses removed**
s3 = "*2"

Press **F8** again. This is a recursive call, so we are back at the start of the function, but the expression being evaluated is now "5+3". There are no subexpressions here, so execution will skip over the first block of code and will match the second pattern, "{.*}{[+-]}{.*}". Again, the **TagIndex** property is used to break the expression into components, and a recursive call is used to evaluate the left and right portions of the expression.

Press **F8** to follow execution a bit further. We are back at the start of the function, but now the expression is "5". There are no subexpressions or operators here, so execution will flow until the last block, and will match the pattern "[0-9]+\.?[0-9]*". Now we can call VB's **Val** function and return the value 5.

If you keep pressing **F8**, you will see that the function will evaluate the 3, then add 5 and 3 together to get 8, and will them multiply that by 2 to get the final result, which is 16. If you added the Debug.Print statement to the code, the debug window will show the subexpressions that were evaluated:

```
            (5+3)*2
5+3
5
3
8*2
8
2
```

If you want, try adding support for variables and functions such as **Sin**, **Cos**, etc. It is easy, all you have to do is add the appropriate patterns and corresponding blocks of code.

FlexString Reference

Description	The FlexString control is a powerful regular expression engine. With FlexString, you can define, find and replace patterns in strings.
Remarks	Use FlexString it to provide regular expression search-and-replace capabilities similar to those available in professional packages such as Word, Visual C++, or Visual Basic. Or use it to parse input strings in complex formats.
File Name	VSFLEX.VBX for the VBX version, VSFLEX32.OCX for the 32-bit OCX version, or VSFLEX16.OCX for the 16-bit OCX version
Object Type	FlexString
Note	Before you can use a FlexString control in your application, you must add vsFlex2 to your project (see the Visual Basic manual for details). To automatically include vsFlex2 in new projects, put it in an AUTOLOAD file. When distributing your application, you should follow the instructions in the **Distribution** section of the vsFlex2 documentation.

FlexString Summary

Properties (default: Text)

* **CaseSensitive**	* **Error**	* **MatchCount**
* **MatchIndex**	* **MatchLength**	* **MatchStart**
* **MatchString**	* **Pattern**	* **Replace**
* **Soundex**	* **TagCount**	* **TagIndex**
* **TagLength**	* **TagStart**	* **TagString**
* **Text**	* **Version**	

CaseSensitive Property

Description Sets/returns whether matching should be case-sensitive.

Usage [***form!***]vsFlexString.**CaseSensitive**[= ***setting%***]

Remarks Setting **CaseSensitive** to **True** will in some cases allow you to use simpler regular expressions. Setting it to **True** gives more control over the matching process.

Data Type Boolean

Error Property

Description Fired after the control exits cell-editing mode.

Usage ***variable%*** = [***form!***]vsFlexString.**Error**

Settings The **Error** property settings are:

flexErrNone
flexErrOutOfMemory
flexErrSquareB
flexErrCurlyB
flexErrBadPattern
flexErrBadTagIndex
flexErrNoMatch
flexErrInvalidMatchIndex

Remarks You should always check the **Error** property when a match fails.

flexErrOutOfMemory
will occur if you assign a string that is too long to the **Text** property or a pattern that is to complex to the **Pattern** property.

flexErrSquareB, **flexErrCurlyB**
will occur when you assign a pattern with unbalanced squre or culry brackets ([,], {, }) to the **Pattern** property. If you want to locate brackets within the search string, remember to escape them with the backslash character (i.e. use "\{" instead of "{").

flexErrBadPattern
will occur when you try to retrieve the results of a match and the **Pattern** or **Text** properties are empty.

flexErrBadTagIndex
will occur when you use a tag in a replacement string for which there is no match (i.e. **Pattern** = "{[a-z]*} ", **Replace** = "{0} {1}").

flexErrNoMatch
will occur when you try to retrieve the results of a match and the match failed.

flexErrInvalidMatchIndex
will occur when you try to select a match greater then or equal to the number of matches (**MatchCount**).

Data Type Integer

MatchCount Property

Description Returns the number of matches found after setting the **Pattern** or **Text** properties.

Usage ***variable%*** = [***form!***]vsFlexString.**MatchCount**

Remarks You can retrieve information about each match by setting the **MatchIndex** property to a value between 0 and **MatchCount** - 1 and then reading the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type Integer

MatchIndex Property

Description Sets/returns the index of the current match when there are multiple matches.

Usage [***form!***]vsFlexString.**MatchIndex**[= ***setting%***]

Remarks You can retrieve information about each match by setting the **MatchIndex** property to a value between 0 and **MatchCount** - 1 and then reading the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type Integer

MatchLength Property

Description Returns the length of the current match, in characters.

Usage ***variable%*** = [***form!***]vsFlexString.**MatchLength**

Remarks You can retrieve information about each match by setting the **MatchIndex** property to a value between 0 and **MatchCount** - 1 and then reading the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type Integer

MatchStart Property

Description Returns the position of the current match within the **Text** string, starting from zero.

Usage ***variable%*** = [***form!***]vsFlexString.**MatchStart**

Remarks You can retrieve information about each match by setting the **MatchIndex** property to a value between 0 and **MatchCount** - 1 and then reading the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type Integer

MatchString Property

Description Sets/returns the string corresponding to the current match.

Usage [***form!***]vsFlexString.**MatchString**[= ***setting$***]

Remarks You can retrieve information about each match by setting the **MatchIndex** property to a value between 0 and **MatchCount** - 1 and then reading the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type String

Pattern Property

Description Sets/returns the regular expression being used for matching against the **Text** string.

Usage [***form!***]vsFlexString.**Pattern**[= ***setting$***]

Remarks The regular expression syntax recognized by vsFlexString is based o the following special characters:

Char	Meaning
^	Beginning of a string.
$	End of a string.
.	Any character.
[***list***]	Any character in ***list***.
[^***list***]	Any character not in ***list***.
?	Repeat previous zero or one time.
*	Repeat previous zero or more times.
+	Repeat previous one or more times.
\	Escape next character.
{***pat***}	Tag this part of the match.

Example

```
fs.Pattern = "^stuff" ' string starting with "stuff"
fs.Pattern = "stuff$" ' string ending with "stuff"
fs.Pattern = "o.d"       ' "old", "odd", etc
fs.Pattern = "o[ld]d" ' "old" or "odd" only
fs.Pattern = "o[^l]d" ' not "old"
fs.Pattern = "od?"       ' "o" or "od"
fs.Pattern = "od*"       ' "o", "od", "odd"
fs.Pattern = "od+"       ' "od", "odd", etc
fs.Pattern = "\."       ' decimal point
```

Data Type String

Replace Property

Description Sets a string to replace all matches.

Usage [***form!***]vsFlexString.**Replace** = ***setting$***

Remarks The replacement occurs as soon as you assign the new text to the **Replace** property. To perform the replacement on several strings, you must set both the **Text** and **Replace** properties for each original string.

The **Replace** string may contain tags, specified using curly brackets with the tag number between them, e.g. "{***n***}". The tags expand into the portions of the original **Text** string that were matched to the corresponding tags in the search **Pattern**. The example below illustrates this:

Example

```
' set up a pattern to search for a filename and
' extension: tag them both (using curly brackets)
' (note how the period is escaped with a backslash)
fs.Pattern = "{[A-Za-z]+}\.{...}"

' assign a string to be matched agains the pattern
' tag 0 will match the filename and
' tag 1 will match the extension
fs.Text = "AUTOEXEC.BAT"

' expand the string (note that each tag may be used
' several times)
fs.Replace = "File {0}.{1}, Name: {0}, Ext: {1}"
Debug.? fs.Text
```

File AUTOEXEC.BAT, Name: AUTOEXEC, Ext: BAT

Data Type String

Soundex Property

Description Returns a phonetic code representing the current **Text** string.

Usage ***variable$*** = [***form!***]vsFlexString.**Soundex**

Remarks This property allows you to search a database for strings even if you don't know the exact spelling. The database must include a **Soundex** field that encodes another field such as last name. When doing the search, look for the **Soundex** code instead of looking for the name.

The **Soundex** code consists of an uppercase letter followed by up to three digits. It is built by assigning codes to each character of the input string, then discarding vowels and repeated codes. The table below shows a few strings and their **Soundex** codes:

Andersen, Anderson, Anders: **A536**
Agassis, Agassi, Agaci: **A2**
Nixon, Nickson: **N25**
Johnson, Jonson: **J525**
Johnston: **J523**
Rumpelstiltskin, Runpilztiskin, Rumpel: **R514**

The advantages of this system are that the code is short, that it will rarely miss a match, and that the system is widely known and already implemented in many databases (the **Soundex** method was developed in 1918 by M.K. Odell and R.C. Russel). The disadvantage is that it will often find spurious matches that are only vaguely similar to the search string.

Data Type String

TagCount Property

Description Returns the number of tags found after setting the **Pattern**, **Text**, or **MatchIndex** properties.

Usage ***variable%*** = [***form!***]vsFlexString.**TagCount**

Remarks You can retrieve information about each tag by setting the **TagIndex** property to a value between 0 and **TagCount** - 1 and reading the **TagLength**, **TagStart**, and **TagString** properties.

Tags are defined by enclosing parts of the regular expression string in the **Pattern** property between curly brackets.

Example

```
fs.Text = "Mary had a little lamb"
fs.Pattern = "Mary had {.*}"
Debug.? fs.TagCount; fs.TagIndex; "[" + fs.TagString + "]"
  1   0 [a little lamb]
```

Data Type Integer

TagIndex Property

Description Sets/returns the index of the current tag when there are multiple tags in the **Pattern** string.

Usage [***form!***]vsFlexString.**TagIndex**[= ***setting%***]

Remarks You can retrieve information about the current tag by reading the **TagLength**, **TagStart**, and **TagString** properties.

The **TagIndex** property can range from 0 to **TagCount** - 1.

Example

```
fs.Text = "Mary had a little lamb"
fs.Pattern = "{[^ ]*} had {.*}"
fs.TagIndex = 0
Debug.? "[" + fs.TagString + "]"
   [Mary]
fs.TagIndex = 1
Debug.? "[" + fs.TagString + "]"
   [a little lamb]
```

Data Type Integer

TagLength Property

Description Returns the length of the current tag, in characters.

Usage ***variable%*** = [***form!***]vsFlexString.**TagLength**

Remarks You can retrieve information about the current tag by reading the **TagLength**, **TagStart**, and **TagString** properties.

Data Type Integer

TagStart Property

Description Returns the position of the current tag within the **Text** string, starting from zero.

Usage ***variable%*** = [***form!***]vsFlexString.**TagStart**

Remarks You can retrieve information about the current tag by reading the **TagLength**, **TagStart**, and **TagString** properties.

Data Type Integer

TagString Property

Description Sets/returns the string corresponding to the current tag.

Usage [***form!***]vsFlexString.**TagString**[= ***setting$***]

Remarks You can retrieve information about the current tag by reading the **TagLength**, **TagStart**, and **TagString** properties.

If you assign a new string to the **TagString** property, vsFlexString will modify the string in the **Text** property and will attempt a new match.

Data Type String

Text Property

Description Sets/returns the text to be scanned searching for the **Pattern** string.

Usage [***form!***]vsFlexString[.**Text**][= ***setting$***]

Remarks vsFlexString will attempt a match as soon as you assign a string to the **Text** or **Pattern** properties.

To find out how many matches were found, read the **MatchCount** property.

To retrieve information about each match, set the **MatchIndex** property to a value between 0 and **MatchCount** - 1, then read the **MatchLength**, **MatchStart**, and **MatchString** properties.

Data Type String

Version Property

Description Returns the version of vsFlex currently loaded in memory.

Usage ***variable% =*** [***form!***]vsFlexString.**Version**

Remarks You may want to check this value at the Form_Load event, to make sure the version that is executing is at least as current as the version used to develop your application.

The version number is a three digit integer where the first digit represents the major version number and the last two represent the minor version number. For example, version 2.00 would return 200.

This property is read-only.

Data Type Integer

VideoSoft Products

To get a registration/order form, click **HERE**.

vsOCX/vsVBX

A set of three custom controls for interface design and text parsing.

Icon	Name	Description
	**vsElastic**	Smart containers that resize themselves and their child controls, automatically create labels and 3-D frames for its child controls, and can also be used as progress indicators and labels.
	**vsIndexTab**	Allows you to group controls by subject, using the familiar notebook metaphor that has become a Windows standard.
	**vsAwk**	Parsing engine named and patterned after the popular Unix utility, plus a powerful expression evaluator.

vsView2

A set of four custom controls for creating, viewing, and printing text and graphics.

Icon	Name	Description
	**vsPrinter**	A much improved printer object with word wrap, headers and footers, multi-column printing, graphics, zooming and panning, and multi-page Print Preview capability.
	**vsViewPort**	A control that gives you a scrollable virtual area so you can fit more controls in your windows. Use it to implement custom Print Preview, fill-out forms, and programs with scrollable pictures or control lists.
	**vsDraw**	A versatile drawing control that lets you create complex images, view them on the screen, copy them to the clipboard, or print them. Use it to create technical drawings, maps, and diagrams.
	**vsInForm**	A control that you can drop into any container to customize its title bar, frame, resizing behavior, and frame buttons. InForm also allows you to monitor the clipboard, drag and drop files from File manager, and more.

vsFlex2

A set of two custom controls for analyzing, formatting, and displaying information.

Icon	Name	Description
	**vsFlexArray**	A new way to display and operate on tabular data. FlexArray gives you total flexibility to display, sort, merge, and format tables containing strings and pictures.
	**vsFlexString**	A powerful regular expression engine. With FlexString, you can find and replace patterns in strings. Use it to provide regular expression search-and-replace capabilities or to parse input strings.

Order/Registration Form

(You may print this form by selecting the File|Print command).

TO: VIDEOSOFT 5900 Hollis Street, Suite T Emeryville, CA 94608	To order by phone, call 510/595-2400 510/595-2424 (fax)

Please register my copy of the following VideoSoft products. I am enclosing a check or money order for the amount of (all prices in US$):

OCX Version (includes 32-bit, 16-bit, and VBX versions)

vsOCX	99.00	x ___ copies	_____
vsView2	149.00	x ___ copies	_____
vsFlex2	149.00	x ___ copies	_____

VBX Version (16-bit only)

vsVBX	45.00	x ___ copies	_____
vsView2	99.00	x ___ copies	_____
vsFlex	99.00	x ___ copies	_____

Tax (CA residents only, please add 8.25% sales tax)	_____
Shipping	6.00
Total Order Amount	_____

Note: Call us for details on site licenses and volume discounts.

Name:

Company:

Street:

City, State, ZIP:

Country:

Phone/Fax:

Where did you hear about the VideoSoft products?