Introduction

Welcome to **VideoSoft VSFlexGrid Pro 6.0®.**

VSFlexGrid Pro 6.0 includes **VSFlexGrid**, a full-featured grid control and **VSFlexString**, a powerful regular expression engine.

VSFlexGrid incorporates the latest in Microsoft data-binding technologies--ADO 2.0 and OLEDB, as well as DAO--giving you the flexibility to choose when to migrate your applications to the newest generation of data access methods as your needs dictate.

If you are upgrading from a previous version of **VSFlex**, look at the **New Features in VSFlexGrid Pro 6.0** section of the documentation.

VideoSoft custom controls are innovative, flexible, and powerful. If you like **VSFlexGridPro**, make sure you check out our other award-winning products, which are described in the **VideoSoft Products** section.

Our distribution policy is as innovative as the controls. We want every Microsoft Visual Basic® programmer to get copies of our products and try them for as long as they wish. Those who like the products and find them useful may purchase licenses at reasonable prices. The only restriction is that unlicensed copies of the software will display a VideoSoft banner whenever they are loaded, to remind developers to license the product.

We are confident that you will like **VideoSoft VSFlexGrid Pro 6.0**. If you have any suggestions or ideas for new features that you'd like to see included in a future version, or ideas for new controls, please call us or write:

VideoSoft
5900-T Hollis Street
Emeryville, CA 94608
510-595-2400 (phone)
510-595-2424 (fax)
http://www.videosoft.com

Upgrading to VSFlexGrid Pro 6.0

Projects that use **VSFlex3** may be upgraded to to **VSFlexGrid Pro 6.0** using the conversion utility provided in the distribution package.

The conversion utility is a Visual Basic program called **CONVERT**, and its source code is included should you want to see exactly what it does.

The utility reads the name of an existing Visual Basic project, parses the names of all forms, then makes all the changes needed to each file. The routine saves the original files with a "bak" extension that is appended to the original file name (e.g. Form1.frm becomes Form1.frm.bak).

The utility will convert most existing projects 100 percent automatically. There is only one exception, where additional (manual) changes may be necessary. Projects that build custom outlines using the **IsSubtotal** property to create nodes and the **RowData** property to set their level must be edited to use the new **RowOutlineLevel** property instead of **RowData**. Read the list below for additional information.

The following list explains the changes needed to convert the project and why they are necessary:

Class names and GUIDs have changed

This affects declarations made inside .frm and .vbp files. It also affects the declarations of the **OLEDragDrop** events, which include a parameter of type **vsDataObject**.

These changes were made to avoid conflicts with **VSFlex3** projects. Both versions of the control may coexist on the same computer.

Some Event names and parameter lists have changed

The **Validate** event has been renamed to **ValidateEdit** in order to avoid conflicts with Visual Basic 6's ambient **Validate** event.

The **UserResized** event has been renamed to **AfterUserResize** for consistency with the new **BeforeUserResize** event.

The **Compare** event has had its parameter list changed for consistency. The *Row1* and *Row2* parameters used to be ByRef and are now ByVal.

The OLEDragDrop events have changed slightly because of the *vsDataObject* parameter, which includes the library name.

RowData and ColData are now Variants

These properties used to be of type Long. They were changed to type Variant to increase their flexibility and usefulness. Since Variants can hold long values, this should not break any existing code.

RowData is no longer used for outlining

In previous versions of the grid control, the **RowData** property was used to determine the level of outline nodes (or subtotals). This prevented the use of that property for other purposes.

To remedy this, a new property called **RowOutlineLevel** has been added to the control, and is used to determine the level of outline nodes.

Projects that build custom outlines using the **IsSubtotal** property to create nodes and the **RowData** property to set their level must be changed manually to accommodate this change. The conversion utility does not attempt to fix this.

Installing VSFlexGrid Pro 6.0

To install **VideoSoft VSFlexGrid Pro 6.0**, use the **SETUP.EXE** utility provided on the distribution CD or diskettes. When you are prompted, enter the registration key (found on the CD case or on the diskette itself) exactly as it is printed and click REGISTER to complete the registration process. You may register any other VideoSoft products for which you have purchased a registration key at this time as well.

The following files will be installed into your WINDOWS\HELP directory:

VSFLEX6.HLP: This file contains the **VSFlexGrid Pro 6.0** online help topics.
VSFLEX6.CNT: This file contains the **VSFlexGrid Pro 6.0** online help contents.

The following OCX files will be installed into your WINDOWS\SYSTEM directory:

VSFLEX6.OCX: This file contains the **VSFlexGrid Pro 6.0** ActiveX controls with ADO data-binding.
VSFLEX6D.OCX: This file contains the **VSFlexGrid Pro 6.0** ActiveX controls with DAO data-binding.

The following folders will be created by the setup utility:

VS: Main VideoSoft directory to store VideoSoft control information.
VS\VSFLEX6: Contains sample Visual Basic projects, utilities, and the README.TXT file which discusses version specific information.

Installing a demo version

If you wish to try **VideoSoft VSFlexGrid Pro 6.0** or any of our other products, and do not have a registration key, use the **SETUP.EXE** utility provided on the distribution CD. When prompted, leave the registration key box blank, then press FINISH.

The only difference between unregistered (demonstration) and registered (purchased) versions of our products is that registered versions will stamp every application you compile so a VideoSoft banner will not appear when your users run the applications.

Uninstalling VSFlexGrid Pro 6.0

To uninstall **VideoSoft VSFlexGrid Pro 6.0**, use the **UNSETUP.EXE** utility provided on the **VSFlexGrid Pro 6.0** installation CD or diskettes. Just run the **UNSETUP.EXE** utility and it will remove all **VSFLEX** files from your \Windows\System directory.

Licensing

You may use **VSFlexGrid Pro 6.0** for development with Microsoft Visual Basic 5.0 or later, and any other programming environment. You may distribute tools created with **VideoSoft VSFlexGrid Pro 6.0** free of royalties. You may include copies of the OCX files with as many copies of your application as you ship.

End-users of your applications are not licensed to use **VSFlexGrid Pro 6.0** for any development, and may not redistribute any OCX files.

You are not allowed to distribute any **VSFlexGrid Pro 6.0** files to others for development purposes. Nor are you permitted to add or transfer the **VSFlexGrid Pro 6.0** key to the registrty of your user's computer(s).

Groups of multiple developers may be interested in acquiring VideoSoft product site licenses. Please contact VideoSoft for details.

If you haven't yet registered your copy of **VideoSoft VSFlexGrid Pro 6.0** and would like to do it now, just fill out the **Order Form** included in this documentation.

Product Support

Product support for **VideoSoft VSFlexGrid Pro 6.0** is available to licensed users through the following channels:

Internet	support.vsflex@videosoft.com
Website	http://www.videosoft.com
Mail	VideoSoft 5900-T Hollis Street Emeryville, CA 94608
Phone	510-595-2400
FAX	510-595-2424

You can always find the latest version of VideoSoft **VSFlexGrid Pro 6.0** on our Web page (http://www.videosoft.com). Before calling for technical support, please identify the version of VideoSoft **VSFlexGrid Pro 6.0** that you are using to help our technicians expedite your queries. The version number appears in the About box that pops up when you double-click the **About** property in the VSFlexGrid Pro ActiveX controls.

Also, please read the **Frequently Asked Questions** section of the help file or the VSFLEX FAQ page on our Web site (http://www.videosoft.com). These resources contain answers given by our technical support staff to our customers' most commonly asked questions, and there is a good chance you may find useful information there.

New Features in VSFlexGrid Pro 6.0

This section summarizes the new features in **VideoSoft VSFlexGrid Pro 6.0**. If you are familiar with previous versions of **VSFLEX**, this section will get you up to speed quickly. For details on each new feature, check the main body of the documentation.

Masked Editing

VSFlexGrid Pro 6.0 has much more powerful editing features than before.

The new **EditMask** and **ColEditMask** properties allow you to specify input masks for automatic input feedback and validation. The mask syntax is similar to the one used by the Microsoft MaskedEdit control and by Microsoft Access.

New Edit Properties and Events

VSFlexGrid Pro 6.0 implements several properties and events similar to those provided by the built-in **TextEdit** and **ComboBox** controls. These new properties and events give you total control over editing.

The new Edit properties are: **EditMask**, **EditSelStart**, **EditSelLength**, **EditSelText**, and **EditMaxLength**. These properties also apply to the **ComboBox** editor.

The new Combo properties are: **ComboCount**, **ComboData**, **ComboIndex**, and **ComboItem**. These properties also apply to the **ListBox** editor.

There is also a new event, **ChangeEdit**, that is fired whenever the contents of the editor changes.

Translated Combos

The new **ColComboList** property has two advantages over the traditional **ComboList** property:

1) You set **ColComboList** once for each column, and you no longer have to worry about setting it in response to the **BeforeEdit** event. This makes your code cleaner and faster.

2) The **ColComboList** may be used as a data dictionary.

For example, say you have a column that holds the employee type, which could be one of the following: Full-time, Part-time, Contractor, Intern, or Other. These will often come from a database, where they will have a unique entry ID.

You may associate the entry IDs with items on the list, and the control will store the IDs and translate them automatically before displaying them.

Multi-Column Combos

The syntax of the **ComboList** property has been extended to allow for multi-column lists to be displayed in the drop-down part of the list, and for items to have arbitrary data items attached to them.

Drop-down/Pop-up Buttons

The **VSFlexGrid** control displays drop-down buttons automatically for cells with associated combo boxes or drop-down lists. The user may edit the cells directly, by clicking the button with the mouse. (In previous versions, the user had to start editing the cell before the box would appear. You may revert to the old behavior by setting the **ShowComboButton** property to False.)

In addition, you may now display pop-up buttons in cells. Just set the **ComboList** property to an ellipsis ("...") in

response to the **BeforeEdit** event and a pop-up button will appear on the cell. If the user clicks the button, the control will fire a **CellButtonClick** event to which you may respond accordingly.

Cell Property

This new property allows reading or writing any cell properties directly to individual cells or ranges (without selecting them).

There are more than 20 cell properties that you access, including text, alignment, fonts, colors, pictures, and even a new *CellData* value that you may use to store custom information with individual calls.

For example, the code below makes an entire range of cells boldface:

```
fa.Cell(flexcpFontBold, 1, 1, 10) = True
```

RowData, ColData are now Variants

You are no longer limited to storing long values in the **RowData** and **ColData** properties. Now these properties are Variants, which means you may associate virtually any data (e.g. strings, objects) with individual rows and columns.

Note that Variants can hold Long values, so most existing VSFlex3 code you have should work with no changes, with one exception: If you used these properties in **VSFlex3** and assigned Variant values to either **RowData** or **ColData**, the values were automatically converted into Longs. If your code relied on this conversion it may break, since now the actual Variants are assigned to **RowData** and **ColData** with no conversion. This distinction is especially important when dealing with objects.

For example, consider the following code:

```
Dim rs as RecordSet
fa.RowData(r) = rs!Field(0)
```

In VSFlex3, this would assign a numeric value to **RowData**. This could be the value of the Field object's default property, or zero if the field is empty.

VSFlexGridPro 6.0 will store a reference to the object instead.

If you want to maintain the original behavior, you need to perform the conversion explicitly, as the code below shows:

```
Dim rs as RecordSet
fa.RowData(r) = CLNG(rs!Field(0))
```

FindRow

The new **FindRow** method allows you to look up specific rows based on their **RowData** values. You can also search rows based on the cell data values for a specific column.

The search is much faster and more convenient than a Visual Basic loop.

Improved Outlining

The new **OutlineCol** property allows the outline tree to be displayed in any column, including those which hold data. The control will take care of the indentation for you.

The outline/subtotal levels are no longer stored in the **RowData** property. This means you may use **RowData** freely, even when dealing with outlines. To retrieve or set the outline/subtotal levels, use the new **RowOutlineLevel** property.

Note that you may need to modify your existing code slightly when porting old projects: if you used the **RowData** property to set outline levels, you should change that to use the new **RowOutlineLevel** property instead.

Clicking on outline symbols collapses or expands individual branches, as before. Shift or control clicking on the symbols collapses the entire outline to the level of the branch clicked. This new behavior is similar to clicking the buttons on the **OutlineBar**.

The outline tree is drawn with dotted lines, similar to the standard Windows Tree control, and the drawing is smoother and faster than before.

Import/Export Comma- or Tab-Delimited Files

The **SaveGrid** and **LoadGrid** methods have been upgraded. Now they allow saving and loading of Excel-compatible comma-delimited or tab-delimited text files.

Bind VSFlexGrid to Visual Basic Arrays or to other VSFlexGrids

The new **BindToArray** method allows you to bind a **VSFlexGrid** control to a Visual Basic array of variants. Then you don't have to copy data between the array and the control: the control displays values read from the array and writes them back into it automatically.

The array must have at least two dimensions and it must be an array of variants. If the array has more than two dimensions, you may display one "page" of it at a time, and you may easily "flip pages".

The **BindToArray** method also allows you to bind a **VSFlexGrid** to another. This way, you may create different "views" of the same data without having to keep duplicate copies of the data.

ExplorerBar

The new **ExplorerBar** property allows users to use column headings to sort and pivot columns without any code.

By default, the **ExplorerBar** works like the one in Microsoft's Internet Explorer 4: One click sorts the column in ascending order, the next in descending order. Any non-fixed column may be dragged to any non-fixed position.

RowHidden, ColHidden

Now you can hide rows and columns by setting the **RowHidden** and **ColHidden** properties. This is better than setting **RowHeight** or **ColWidth** to zero, because you may hide and unhide rows and columns without having to save/restore their original dimensions.

AutoSearch

Set the **AutoSearch** property to *flexSearchFromTop* or *flexSearchFromCursor*, and the control will look for data (in the current column) as the user types what she is looking for. The search is case-insensitive, and partial matches are displayed as the user types.

Improved Formatting

The **VSFlexGrid** control allows you to format dates using Visual Basic-like formatting strings. Just set the **ColFormat** property to "Short Date", for example. Or define your own custom format for dates using the same familiar syntax used with Visual Basic's **Format** function.

The **VSFlexGrid** control supports number scaling the same way Visual Basic does. Include a percent sign in the format and the value shown will be multiplied by 100, with a trailing percentage sign added. Include a decimal right after the thousand separator (",.") and the value will be divided by 1,000.

Automatic CheckBoxes

When a column has its **ColDataType** property set to *flexDTBoolean*, the control will automatically display its values as check boxes. The control will automatically map cell contents onto boolean values and vice-versa. This feature is especially convenient in bound mode when editing recordsets that contain boolean fields.

You may also assign custom strings to represent boolean values. To do this, set the **ColFormat** property to a string containing the values you want to display, separated by a semicolon (e.g. "True;False", "Si;No", "Ja;Nein", "Oui;Non").

ScrollTips

The new **ScrollTips** property allows you to display a tooltip over the vertical scrollbar as the user moves the scroll thumb, just like Excel and Word do. This makes it easy for users to browse and find specific rows on large data sets.

Enumerate selected rows

When you set the **SelectionMode** property to *flexSelectionListBox*, the control allows you to select rows by control-clicking them.

You can now enumerate the selected rows using two new properties, **SelectedRows** and **SelectedRow**. This is much faster than scanning the entire control for selected rows.

Automatic Auditing

The **VSFlexGrid** control keeps track of the state of each row for you. The new **RowStatus** property is set automatically to reflect the status of the row (new, modified, updated). The **RowStatus** property is read/write, so you may define and assign your own constants to it.

Miscellaneous Improvements

The **VSFlexGrid** control implements a number of significant miscellaneous improvements:

Proportionally-sized scrollbars show how large the visible area of the document is compared to the entire document.

The **AutoSize** method has a new optional parameter, *ExtraSpace*, that allows you to specify extra width or height, in Twips, for the columns or rows being resized.

Merging has been improved. Painting is faster, merged cells may be highlighted when selected, and the **MergeCells** property has two new settings: *flexFixedOnly* and *flexSpill.*

There are new events for better UI control: **BeforeMouseDown** and **BeforeUserResize**.

There is a new Explorer-style setting for the **GridLines** and **GridLinesFixed** properties. It gives them a 3D look similar to the Windows common controls.

New **RowHeightMax**, **ColWidthMax**, and **ColWidthMin** properties to work with **RowHeightMin**.

Faster painting, smoother scrolling, especially when displaying merged cells.

Stable sorting algorithm: The sorting keeps the relative order of records when the sorting key is the same. For example, if you sort a list of files by name, then by extension, file names will still be sorted within each extension group.

More robust validation: When the **ValidateEdit** event fails, the user is returned to the same cell, and is back in

edit mode. Previously, the user was returned to the cell, but not in edit mode. You had to write code to enable this.

The **Subtotal** method has an additional optional parameter (*TotalOnly*) that allows you to specify whether subtotal rows should include only a title and a subtotal or whether they should also include data. (The latter is the default).

Better support for pictures in cells, custom-sized icons and palette support.

FlexString Improvements

Now you can use an optional index with the **MatchStart**, **MatchLength**, **MatchString**, **TagStart**, **TagLength**, and **TagString** properties. This makes your code more compact and more readable.

No Dependencies

VSFlexGrid Pro 6.0 does not depend on any MFC .DLLs or separate OCXs. This makes deployment much easier, since you need only include the VSFLEX6.OCX file with your application.

Overview

The **VideoSoft VSFlexGrid Pro 6.0** package consists of two ActiveX controls:

VSFlexGrid
A powerful, full-featured grid. It provides new ways to display, edit, format, organize, summarize, and print tabular data. VideoSoft VSFlexGrid Pro supports comma- and tab-delimited files from Microsoft Access and Excel, 2D and 3D arrays, automatic and multiple totaling and subtotaling, mouse-activated scroll tips, and an innovative grid-to-grid data binding feature. VSFlexGrid incorporates the latest in Microsoft data-binding technologies--ADO 2.0 and OLE DB, as well as DAO, support OLE drag-and-drop editing, and works equally well in both bound and unbound modes.

VSFlexGrid adds versatile data-presentation tools to your database applications that maximize end-user customization capabilities with features that include in-cell editing, cell merging, Outlook-style sorting by column headings, and advanced outlining capabilities. Featuring automatic auditing which tracks data changes, improved object model for easier cell formatting, translated drop-downs, combo box drop-downs and multi-column drop-downs, VSFlexGrid is easy to use, has a small footprint and is the fastest grid control on the market.

VSFlexString
A flexible regular expression engine. It features pattern matching as well as regular expression text matching. **vsFlexString's** automatic replace capabilities immediately replaces all matches with the new assigned string. And Tag matching capabilities determine which parts of the string matched what parts of the pattern.

VSFlexGrid features an optional index with the **MatchStart**, **MatchLength**, **MatchString**, **TagStart**, **TagLength**, and **TagString** properties to make your code more compact and more readable.

VSFlexGrid QuickStart

This section takes you step by step through the creation of three Visual Basic projects using the **VSFlexGrid** control:

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

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

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

OLE Drag and Drop Demo
How to implement automatic and custom OLE Drag and Drop.

Visual C++ Demo
How to handle optional parameters and picture properties in C++.

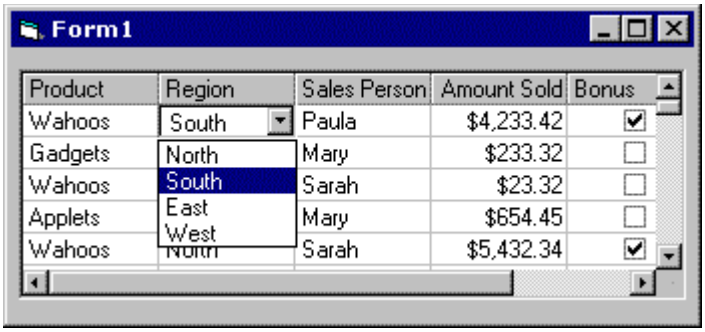
These are simple programs that focus on using the **VSFlexGrid** control. We tried to reduce the amount of coding to a minimum, just enough to show how common tasks can be easily accomplished with the **VSFlexGrid**. For more realistic and complex projects, please refer to the samples on the distribution CD or disks.

Edit Demo

This sample starts with a basic data-entry grid, then adds the following features:

- Data formatting
- Check boxes
- Drop-down lists
- Input masks
- Complex data validation
- Clipboard support

Here is what the final application will look like:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid Pro 6.0** (if you don't know how to add OCX files to a project, consult the Visual Basic documentation). The **VSFlexGrid** icon will be added to the Visual Basic toolbox.

Create a **VSFlexGrid** object on the form by clicking the **VSFlexGrid** icon on the toolbox, then clicking on the form and dragging until the object is the proper size.

Next, use the Visual Basic properties window to set the following control properties:

```
(Name) = fa
Editable = True
Cols = 5
FixedCols = 0
FormatString = "=Product|Region|Sales Person|" & ">Amount Sold|Bonus"
```

That's it. Press F5 to run the project, and you can start typing data into the control. Press F2 or the space bar to edit existing entries, or just type new entries over existing ones.

Step 2: Data Formatting

When displaying numeric or date values, you will typically want to adopt a consistent format for the values. The **VSFlexGrid** allows you to do this using the **ColFormat** property. This property allows you to assign a format to each column. The formats are similar to the ones recognized by the Visual Basic **Format** function.

The **ColFormat** property must be assigned at runtime. A good place to do it is in the Form_Load event, as show below:

```
Private Sub Form_Load()

  ' format column 3 (Amount Sold) to display currency
  fa.ColFormat(3) = "$#,###.00"

End Sub
```

This code assigns a format to column 3 (*Amount Sold*). The format specifies that values should be displayed with a currency sign, thousand separators, and two decimals.

The **ColFormat** property does not affect the cell content, only the way it is displayed. You may change the format freely without modifying the underlying data.

Step 3: Check Boxes

When displaying boolean (True/False) values, you have the option of using check boxes instead of True/False strings or 1/0 values. This has the advantage of preventing users from entering bad values.

Column 4 (*Bonus*) contains boolean values (either someone gets a bonus or not). To display the values as checkboxes, set the **ColDataType** property to *flexdtBoolean*. The control will automatically display and manage the check boxes.

The **ColDataType** property must be assigned at runtime. Change the Form_Load routine as shown below:

```
Private Sub Form_Load()

  ' format column 3 (Amount Sold) to display currency
  fa.ColFormat(3) = "$#,###.00"

  ' make column 4 (Bonus) a boolean column
  fa.ColDataType(4) = flexdtBoolean

End Sub
```

Users may toggle the check boxes by clicking them or by selecting them with the keyboard and then hitting enter or space. Press F5 to run the project again, then type a few sales amounts and give bonuses to some people.

Step 4: 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 full-time sales people (Mary, Sarah, and Paula).

Typing repetitive data would be inefficient and error-prone. A much better approach would be to use drop-down lists to let users pick the appropriate entry from lists. The **VSFlexGrid** allows you to assign a list of choices to each column using the **ColComboList** property. The list consists of a string with choices, separated by pipe characters ("|").

The **ColComboList** property must be assigned at runtime. Change the Form_Load routine as shown below:

```
Private Sub Form_Load()

  ' format column 3 (Amount Sold) to display currency
  fa.ColFormat(3) = "$#,###.00"

  ' make column 4 (Bonus) a boolean column
  fa.ColDataType(4) = flexdtBoolean

  ' assign combo lists to each column
  fa.ColComboList(0) = "Applets|Wahoos|Gadgets"
  fa.ColComboList(1) = "North|South|East|West"
  fa.ColComboList(2) = "|Mary|Paula|Sarah"

End Sub
```

Notice how the last **ColComboList** string starts with a pipe. This will allow users to type additional names that are not on the list. In other words, these values will be edited using a drop-down combo, as opposed to a drop-down list as the others. There are syntax options to create multi-column lists and translated lists as well. See the control reference for more details.

Press F5 to run the project again, then move the cursor around. When you move the cusrsor to one of the

columns that have combo lists, a drop-down button becomes visible. You may click on it to show the list, or simply type the first letter of an entry to highlight it on the list.

Step 5: Input Masks

When picking data from a list, there's usually little need for data validation. When input values are typed in, however, you will often want to make sure it is valid.

In our example, we would like to prevent users from typing text or negative values in column 3 (*Amount Sold*). You can do this using the **ColEditMask** property, which assigns an input mask to a column that governs what the user can type into that field.

The **ColEditMask** property must be assigned at runtime. Change the Form_Load routine as shown below:

```
Private Sub Form_Load()

  ' format column 3 (Amount Sold) to display currency
  fa.ColFormat(3) = "$#,###.00"

  ' assign edit mask to column 3 (Amount Sold)
  fa.ColEditMask(3) = "######.##"

  ' make column 4 (Bonus) a boolean column
  fa.ColDataType(4) = flexdtBoolean

  ' assign combo lists to each column
  fa.ColComboList(0) = "Applets|Wahoos|Gadgets"
  fa.ColComboList(1) = "North|South|East|West"
  fa.ColComboList(2) = "|Mary|Paula|Sarah"

End Sub
```

The edit mask ensures that the user will not type anything into column 3 except numbers. The syntax for the **ColEditMask** property allows you to specify several types of input. See the control reference for details.

Step 6: Complex Data Validation

Input masks are convenient to help users input properly formatted data. They also help with simple data validation tasks. In many situations, however, you may need to perform more complex data validation. In these cases, you should use the **ValidateEdit** event.

For example, let's say some anti-trust regulations prevent us from being able to sell Applets in the North region. To prevent data-entry mistakes and costly lawsuits, we want to prevent users from entering this combination into the control. We can do it with the following routine:

```
Private Sub fa_ValidateEdit(ByVal Row As Long, _
                            ByVal Col As Long, Cancel As Boolean)
  Dim rgn As String, prd As String

  ' collect the data we need
  Select Case Col
    Case 0
      prd = fa.EditText
      rgn = fa.TextMatrix(Row, 1)
    Case 1
      prd = fa.TextMatrix(Row, 0)
      rgn = fa.EditText
  End Select

  ' we can't sell Applets in the North Region...
  If prd = "Applets" And rgn = "North" Then
    MsgBox "Regulation #12333AS/SDA-23 " & _
           "Prevents us from selling " & prd & _
           " in the " & rgn & " Region. Please verify input."
    Cancel = True
  End If
End Sub
```

The function starts by gathering the input that needs to be validated. Note that the values being checked are retrieved using the **EditText** property. This is necessary because they have not yet been applied to the control.

If the test fails, the function displays a warning and then sets the *Cancel* parameter to True, which cancels the edits and puts the cell back in edit mode so the user can try again.

Press F5 to run the project again, then try inputting some bad values. You will see that the control will reject them.

Step 7: Clipboard Support

The Windows clipboard is a very useful device for transfering information between applications. Adding clipboard support to **VSFlexGrid** projects is very easy. All it takes is the following code:

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

  ' copy: ctrl-C, ctrl-X, ctrl-ins
  If KeyCode = vb Key C And Shift = 2 Then Cpy = True
  If KeyCode = vb Key X And Shift = 2 Then Cpy = True
  If KeyCode = vb Key Insert And Shift = 2 Then Cpy = True

  ' paste: ctrl-V, shift-ins
  If KeyCode = vb Key V And Shift = 2 Then Pst = True
  If KeyCode = vb Key Insert 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.

Another great Windows feature that is closely related to clipboard operations is OLE Drag and Drop. **VSFlexGrid** has two properties, **OleDragMode** and **OLEDropMode**, that help implement this feature. Just set both properties to their automatic settings and you will be able to drag selections by their edges and drop them into other applications such as Microsoft Excel, or drag ranges from an Excel spreadsheet and drop them into the **VSFlexGrid** control.

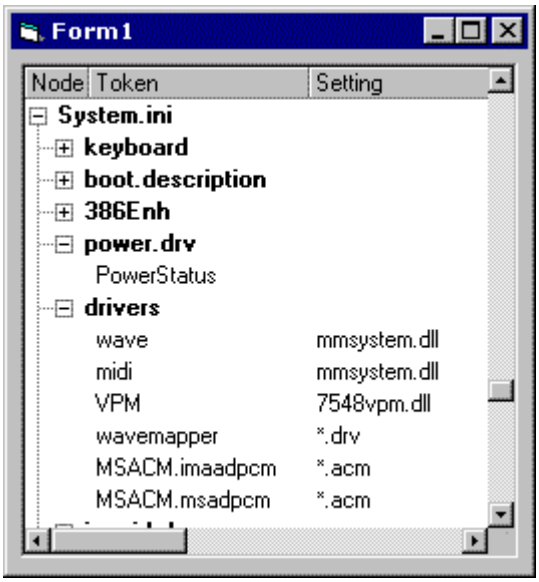
Press F5 to run the project again, then try copying and pasting some data. You will notice that it is possible to paste invalid data, because our paste code does not do any data validation. This is left as an exercise for the reader.

Outline Demo

This sample shows how you can use the **VSFlexGrid** as an outliner to display structured (or hierarchical) data.

When used as an outliner, the **VSFlexGrid** control behaves like a Tree control, displaying nodes that can be collapsed or expanded to show branches containing subordinate data.

The sample reads several .INI files and presents each one as a node. Each file node has a collection of sub-nodes that contain sections within the corresponding .INI file. Each section node contains branches that show the tokens and settings stored in the corresponding section. Here is how the final project will look:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid Pro 6.0** (if you don't know how to add OCX files to a project, consult the Visual Basic documentation). The **VSFlexGrid** icon will be added to the Visual Basic toolbox.

Create a **VSFlexGrid** object on the form by clicking the **VSFlexGrid** icon on the toolbox, then clicking on the form and dragging until the object is the proper size.

Next, use the Visual Basic properties window to set the following control properties:

```
(Name) = fa
Cols = 3
ExtendLastCol = True
FixedCols = 0
Rows = 1
FormatString = "Node|Token|Setting"
OutlineBar = flexOutlineBarComplete
GridLines = flexGridNone
MergeCells = flexMergeSpill
AllowUserResising = flexResizeColumns
```

We set the **OutlineBar** property to be able to see the outline tree. You can create outlines without trees, but the user will not be able to collapse and expand the nodes (unless you write code to do it).

We also set the **MergeCells** property to *flexMergeSpill*, so long entries may extend into adjacent empty cells. This is often a good setting to use when building outlines.

Now the control is ready. We can start adding some code to it.

Step 2: Read the Data and Build the Outline

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

```
Private Sub Form_Load()

  ' suspend repainting to increase speed
  fa.Redraw = False

  ' populate the control
  AddNode "Win.ini"
  AddNode "System.ini"
  AddNode "vb.ini"

  ' expand outline, resize to fit, collapse outline
  fa.Outline -1
  fa.AutoSize 1, 2
  fa.Outline 1

  ' repainting is back on
  fa.Redraw = True
End Sub
```

The routine starts by setting the **Redraw** property to False. This suspends repainting while we populate the control, and increases speed significantly.

Then the **AddNode** routine is called to populate the control with the contents of three .INI files which you are likely to have on your system: Win, System, and Vb. The **AddNode** routine is shown below.

Finally, the outline is totally expanded, the **AutoSize** method is called to adjust column widths to their contents, and the outline is collapsed back to level 1 so the file and section nodes will be displayed.

The **AddNode** routine does most of the work. It reads an .INI file and populates the control, creating nodes and branches according to the contents of the file. Here is the **AddNode** routine:

```
Sub AddNode(inifile As String)
  Dim ln As String, p As Integer
  With fa

    ' create file node
    .AddItem inifile
    .IsSubtotal(Rows - 1) = True
    .Cell(flexcpFontBold, Rows - 1, 0) = True

    ' read ini file
    Open "c:\windows\" & inifile For Input As #1
    While Not EOF(1)
      Line Input #1, ln

      ' if this is a section, add node
      If Left(ln, 1) = "[" Then
        .AddItem Mid(ln, 2, Len(ln) - 2)
        .IsSubtotal(Rows - 1) = True
        .RowOutlineLevel(Rows - 1) = 1
        .Cell(flexcpFontBold, Rows - 1, 0) = True

      ' if this is regular data, add branch
      ElseIf InStr(ln, "=") > 0 Then
        p = InStr(ln, "=")
        .AddItem vbTab & Left(ln, p - 1) & vbTab & Mid(ln, p + 1)
      End If
    Wend
    Close #1
  End With
End Sub
```

The **AddNode** routine is a little long, but it is fairly simple. It starts by adding a row containing the name of the .INI file being read. It marks the row as a subtotal using the **IsSubtotal** property so the control will recognize it as an outline node.

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 **RowOutlineLevel** property to 1, indicating this node is a child of the previous level-0 node (the one that contains the file name).

Finally, lines containing data are parsed into tokens and settings and then added to the control. They are not marked as subtotals.

Step 3: Use the Outline

Press F5 to run the project, and you will see the outline in action. If you click on one of the nodes, it will expand or collapse to show or hide the data under it.

You may also shift-click on a node to expand the entire outline to the node's level, or shift-ctrl-click on a node to collapse the entire outline to that level. For example, if you shift-click on a file name, you will see all file names and all sections, but no token data. If you shift-ctrl-click on a file name, you will see all file names, and nothing else.

Step 4: Custom Mouse and Keyboard Handling

The **VSFlexGrid** provides the expanding and collapsing for you, but you may extend and customize its behavior. Every time a branch is expanded or collapsed, the control 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.

For example, the following code allows users to expand and collapse outline branches by double-clicking on a row itself, rather than on the outline bar. Here's the code to do it:

```
Private Sub fa_DblClick()
  Dim r As Long
  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 collapses its entire parent node.

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 = vbKeyReturn Then fa_DblClick
End Sub
```

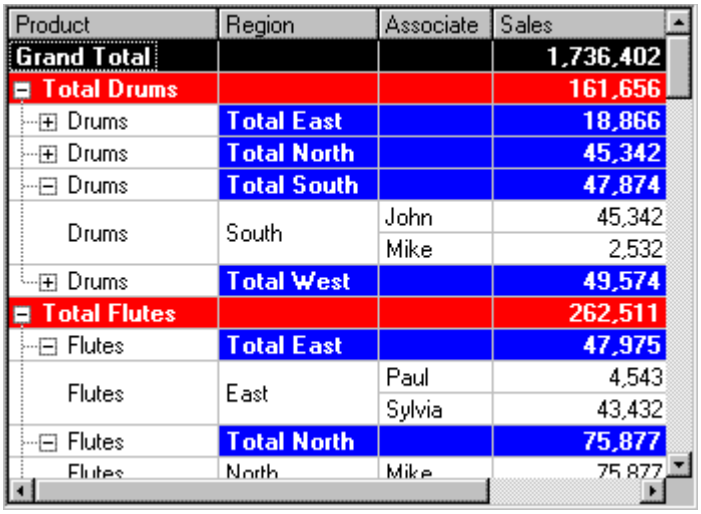
This closes the Outline demo. Press F5 to run the project one last time and test the additional mouse and keyboard handling.

Data Analysis Demo

This sample starts with a grid containing sales data for different products, regions, and salespeople, then adds the following features:

- Dynamic layout (column order)
- Automatic sorting
- Cell merging
- Automatic subtotals
- Outlining

Here is how the final application will look:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid Pro 6.0** (if you don't know how to add OCX files to a project, consult the Visual Basic documentation). The **VSFlexGrid** icon will be added to the Visual Basic toolbox.

Create a **VSFlexGrid** object on the form by clicking the **VSFlexGrid** icon on the toolbox, then clicking on the form and dragging until the object is the proper size.

Next, use the Visual Basic properties window to set the control name to **fa**.

2: Initialize and populate the grid

There are many methods available to populate a **VSFlexGrid** control. Often, you will simply connect it to a database using the **DataSource** property. Or you could load the data from a file using the **LoadGrid** method. Finally, you may use the **AddItem** method to add rows or the **Cell** property to assign data to cells.

In this demo, we will generate some random data and assign it to the control using the **Cell** property. This is done at the Form_Load event:

```
Private Sub Form_Load()

  ' initialize the control
  fa.Cols = 4
  fa.FixedCols = 0
  fa.GridLinesFixed = flexGridExplorer
  fa.AllowUserResizing = flexResizeBoth
  fa.ExplorerBar = flexExMove

  ' define some sample data
  Const slProduct = "Product|Flutes|Saxophones|Drums|" & _
```

```
                    "Guitars|Trombones|Keyboards|Microphones"
  Const slAssociate = "Associate|John|Paul|Mike|Paula|Sylvia|Donna"
  Const slRegion = "Region|North|South|East|West"
  Const slSales = "Sales|14323|2532|45342|43432|75877|4232|4543"

  ' populate the control with the data
  FillColumn fa, 0, slProduct
  FillColumn fa, 1, slAssociate
  FillColumn fa, 2, slRegion
  FillColumn fa, 3, slSales
  fa.ColFormat(3) = "#,###"

End Sub
```

This routine uses a helper function called **FillColumn** that fills an entire column with data drawn randomly from a list. This is a handy function for demos, and here is the code:

```
Sub FillColumn(fa As vsFlexGrid, ByVal c As Long, ByVal s As String)
  Dim r As Long, i As Long, cnt As Long
  ReDim lst(0) As String

  ' build list of data values
  cnt = 0
  i = InStr(s, "|")
  While i > 0
    lst(cnt) = Left(s, i - 1)
    s = Mid(s, i + 1)
    cnt = cnt + 1
    ReDim Preserve lst(cnt) As String
    i = InStr(s, "|")
  Wend
  lst(cnt) = s

  ' set values by randomly picking from the list
  fa.Cell(flexcpText, 0, c) = lst(0)
  For r = fa.FixedRows To fa.Rows - 1
    i = (Rnd() * 1000) Mod cnt + 1
    fa.Cell(flexcpText, r, c) = lst(i)
  Next

  ' do an autosize on the column we just filled
  fa.AutoSize c, , , 300
End Sub
```

This concludes the first step. Press F5 to run the project, and you will see a grid loaded with data. Because the **ExplorerBar** property is set to *flexExMove*, you may drag column headings around to reorder the columns.

The data presented is almost useless, however, because it is not presented in an organized way. We will fix that next.

Step 2: Automatic Sorting

The first step in organizing the data is sorting it. Furthermore, we would like the data to be sorted automatically whenever the user reorders the columns.

After the user reorders the columns, the **VSFlexGrid** control fires the **AfterMoveColumn** event. We will add an event handler to sort the data using the **Sort** property. (Note that if the grid were bound to a database, you would need to set the **DataMode** property to *flexDMFree* to be able to sort using the **Sort** property.)

Here is the code:

```
Private Sub fa_AfterMoveColumn(ByVal Col As Long, Position As Long)

  ' sort the data from first to last column
  fa.Select 1, 0, 1, fa.Cols - 1
  fa.Sort = flexSortGenericAscending
  fa.Select 1, 0
```

```
End Sub
```

The **AfterMoveColumn** 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.

To start with a sorted grid, we will also add a call to the **AfterMoveColumn** routine to the end of the Form_Load handler.

```
Private Sub Form_Load()

  ' initialize the control
  ' …
  ' define some sample data
  ' …
  ' populate the control with the data
  ' …

  ' organize the data
  fa_AfterMoveColumn 0, 0

End Sub
```

Press F5 to run the project again, and try reordering the columns by dragging their headings around. Whenever you move a column, the data is automatically sorted, which makes it much easier to interpret. But we're just getting started.

Step 3: Cell Merging

The ability to dynamically merge cells is one of the features that sets the **VSFlexGrid** apart from other grid controls. Merging cells groups them visually, making the data easier to interpret.

To implement cell merging, we need only add two lines of code to the Form_Load event handler:

```
Private Sub Form_Load()

  ' initialize the control
  ' …
  ' define some sample data
  ' …
  ' populate the control with the data
  ' …

  ' set up cell merging (all columns)
  fa.MergeCells = flexMergeRestrictAll
  fa.MergeCol(-1) = True

  ' organize the data
  ' …

End Sub
```

The new code sets the **MergeCells** property, which works over the entire control, then sets the **MergeCol** property to True for all columns (the -1 index may be used as a wildcard for all properties that apply to rows and columns).

Press F5 again to run the project. This time it looks very different from a typical grid. The cell merging makes groups of data stand out visually and help interpret the information.

Step 4: Automatic Subtotals

Now that the data is sorted and grouped, we will add code to calculate subtotals. With the subtotals, the user will be able to see what products are selling more, in what regions, and which salespeople are doing a good job.

Adding subtotals to a **VSFlexGrid** control is easy. The **Subtotal** method handles most of the details.

The subtotals need to be recalculated after each sort, so we will add the necessary code to the

AfterMoveColumn event. Here is the revised code:

```
Private Sub fa_AfterMoveColumn(ByVal Col As Long, Position As Long)

  ' suspend repainting to get more speed
  fa.Redraw = False

  ' sort the data from first to last column
  fa.Select 1, 0, 1, fa.Cols - 1
  fa.Sort = flexSortGenericAscending
  fa.Select 1, 0

  ' calculate subtotals
  fa.Subtotal flexSTClear
  fa.Subtotal flexSTSum, -1, 3, , 1, vbWhite, True
  fa.Subtotal flexSTSum, 0, 3, , vbRed, vbWhite, True
  fa.Subtotal flexSTSum, 1, 3, , vbBlue, vbWhite, True

  ' autosize
  fa.AutoSize 0, fa.Cols - 1, , 300

  ' turn repainting back on
  fa.Redraw = True

End Sub
```

This code starts by setting the **Repaint** property to False. This suspends all repainting while we work on the grid, which avoids flicker and increases speed.

Then the subtotals are calculated using the **Subtotal** method. The first call removes any existing subtotal rows, cleaning up the grid. The next three calls add subtotal rows. We start by adding a grand total, then subtotals on sales grouped by columns 0 and 1. (For now, we are assuming that sales figures will be on column 3.)

After adding the subtotals, we use the **AutoSize** method to make sure all columns are wide enough to display the new data.

Finally, the **Redraw** property is set back to True, at which point the changes become visible.

If you run the project now, you will see that it almost works. The problem is that we are assuming that sales figures will be on column 3, and if the user moves the figures to the left, the subtotals will just add up to zero.

To prevent this from happening, we can trap the **BeforeMoveColumn** event and prevent the user from moving the sales figures column.

Here is the code:

```
Private Sub fa_BeforeMoveColumn(ByVal Col As Long, Position As Long)

  ' don't move sales figures
  If Col = fa.Cols - 1 Then Position = -1

End Sub
```

We should also prevent the sales column from having merged cells. Merging these values could be confusing because identical amounts would be merged and appear to be a single entry. To do this, we need to go back to the Form_Load event handler and add one line of code:

```
Private Sub Form_Load()

  ' initialize the control
  ' …
  ' define some sample data
  ' …
  ' populate the control with the data
  ' …

  ' set up cell merging (all columns)
  fa.MergeCells = flexMergeRestrictAll
```

```
    fa.MergeCol(-1) = True
    fa.MergeCol(fa.Cols - 1) = False

    ' organize the data
    ' …

  End Sub
```

We are done with the subtotals. If you run the project now, you will see how easy it is to understand the picture behind the sales figures. You can organize the data by product, by region, or by salesperson and quickly see who is selling what and where.

We are now almost done with this demo. The last step is to add outlining to the control, so users can hide or show details and get an even clearer picture.

Step 5: Outlining

The outlining capabilities of the **VSFlexGrid** control rely on subtotals. When outlining, each subtotal row is treated as a node that can be collapsed or expanded. Nested subtotals are treated as nested nodes. Any rows that are not subtotal rows are treated as branches, which contain detail data.

Because we have already implemented subtotals, adding the outline capabilities is just a matter of adding one more line of code to the Form_Load event handler. The new code sets the **OutlineBar** property, which displays a tree structure with buttons that the user may click to collapse or expand the outline. Here is what the Form_Load routine should look like by now:

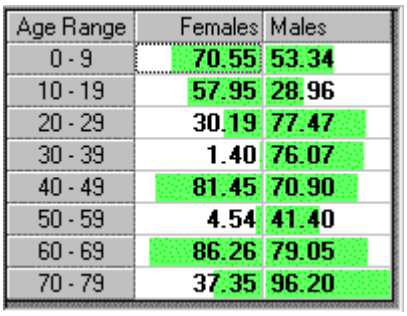
```
  Private Sub Form_Load()

    ' initialize the control
    ' …
    ' define some sample data
    ' …
    ' populate the control with the data
    ' …
    ' set up cell merging (all columns)
    ' …

    ' set up outlining
    fa.OutlineBar = flexOutlineBarComplete

    ' organize the data
    ' …

  End Sub
```

That concludes this demo. Run the project one last time and try clicking on the outline buttons. Clicking will toggle the state of the node between collapsed and expanded. Shift-clicking or ctrl-shift-clicking will set the outline level for the entire control.

Cell Flooding Demo

This example demonstrates how to use the **Cell** property to format individual cells. The demo uses flooding to create a display combining numbers and bars.

Here is how the final application will look:



Age Range	Females	Males
0 - 9	70.55	53.34
10 - 19	57.95	28.96
20 - 29	30.19	77.47
30 - 39	1.40	76.07
40 - 49	81.45	70.90
50 - 59	4.54	41.40
60 - 69	86.26	79.05
70 - 79	37.35	96.20

This project is very simple. It consists of a single routine, the Form_Load event handler. Here is the code, followed by some comments:

```
Private Sub Form_Load()
  Dim i As Long
  Dim max As Double

  ' initialize array with random data
  Dim count(1, 7) As Single
  For i = 0 To 7
      count(0, i) = Rnd * 100
      count(1, i) = Rnd * 100
  Next

  ' initialize control
  fa.Cols = 3
  fa.Rows = 9
  fa.FloodColor = RGB(100, 255, 100)
  fa.ColAlignment(0) = flexAlignCenterCenter
  fa.ColAlignment(1) = flexAlignRightCenter
  fa.ColAlignment(2) = flexAlignLeftCenter
  fa.Cell(flexcpText, 0, 0) = "Age Range"
  fa.Cell(flexcpText, 0, 1) = "Females"
  fa.Cell(flexcpText, 0, 2) = "Males"
  fa.ColFormat(-1) = "#.##"

  ' make data bold
  fa.Cell(flexcpFontBold, 1, 1, _
         fa.Rows - 1, fa.Cols - 1) = True

  ' place text in cells, keep track of maximum
  For i = 0 To 7
    fa.Cell(flexcpText, i + 1, 0) = _
                    10 * i & " - " & (10 * (i + 1) - 1)
    fa.Cell(flexcpText, i + 1, 1) = count(0, i)
    fa.Cell(flexcpText, i + 1, 2) = 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 each cell's flood percentage,
  ' 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.Cell(flexcpFloodPercent, i + 1, 1) = _
                     -100 * count(0, i) / max
    fa.Cell(flexcpFloodPercent, i + 1, 2) = _
                      100 * count(1, i) / max
```

```
    Next

End Sub
```

The code starts by declaring and populating an array with random data. The data will be used later to populate the control.

Then the control is initialized. The code sets the number of rows and columns, column alignments, column titles, and the format that is to be used when displaying data. Note that when setting the **ColFormat** property, the -1 index is used as a wildcard so the setting is applied to all columns.

The **Cell** property is then used to set the font of the scrollable area to bold. It takes only a single statement, because the **Cell** property accepts a whole range as a parameter.

Next, the array containing the data is copied to the control (again using the **Cell** property). The code keeps track of the maximum value assigned to any cell in order to scale the flood percentages later.

Finally, the **Cell** property is used one last time to set the flood percentages. The percentages on the first column are set to negative values, which causes the bars to be drawn from right to left. The percentages on the second column are set to positive values, which causes the bars to be drawn from left to right.

Cell ToolTip Demo

The example below shows how you can use the **MouseRow** and **MouseCol** properties to implement tooltips with text that changes as the mouse moves over the control.

```
Sub fa_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
  Static r As Long, c As Long
  Dim nr As Long, nc As Long

  ' get coordinates
  nr = fa.MouseRow
  nc = fa.MouseCol

  ' update tooltip text
  If c <> nc Or r <> nr Then
    r = nr
    c = nc
    fa.ToolTipText = "Row:" & r & " Col:" & c
  End If

  ' other processing...
  ' ...
End Sub
```

The code keeps track of the last cell for which tooltips were displayed, and refreshes the **ToolTipText** only when needed. This is done to avoid flicker.

RenderControl Demo

The example below shows how you can print a report based on a **VSFLEX Grid Pro 6.0** control using VideoSoft's **VSPrinter** control.

The **VSPrinter** control is part of **VSVIEW 3.0**, a separate VideoSoft product.

The example assumes you have a **VSFlexGrid** control named **fa** and a **VSPrinter** control named **vp** on your form.

```
Sub PrintFlexGrid()
  vp.StartDoc
  vp.RenderControl = fa.hWnd
  vp.EndDoc
End Sub
```

The routine above is really all you need in order to print simple reports. By setting some properties on the **VSPrinter** control, the report may be shown on a print preview window, rendered on the printer, or saved to a file.

For printing complex reports, the **VSFlexGrid** control exposes events that allow you to control page breaks and to supply header rows which get printed at the top of each new page. The code below illustrates the use of these events:

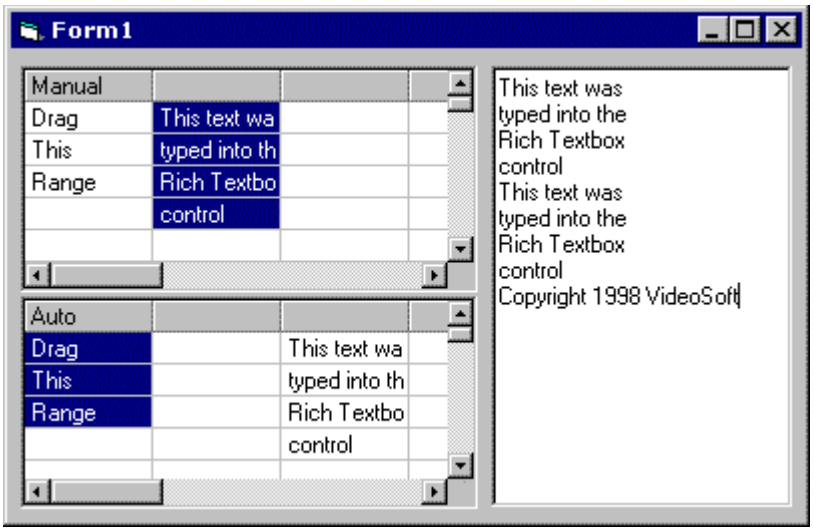
```
' BeforePageBreak: controls page breaks
' we assume we have subtotals above details, and prevent subtotal rows from
' being the last on a page
Private Sub fa_BeforePageBreak(ByVal Row As Long, BreakOK As Boolean)
  With fa

    ' if this row is a subtotal heading, we can't break here
    If .IsSubtotal(Row) Then
      BreakOK = False
    End If
  End With
End Sub


' GetHeaderRow: supplies header rows for new pages
' we assume we have title rows with RowData set to -1 that we want to show
' above the data
Private Sub fa_GetHeaderRow(ByVal Row As Long, HeaderRow As Long)
  Dim r As Long

  ' ignore if the top row is a header already
  If fa.RowData(Row) = -1 Then Exit Sub

  ' we need a header, so find one
  For r = fa.FixedRows To fa.Rows - 1
    If fa.RowData(r) = -1 Then
      HeaderRow = r
      Exit Sub
    End If
  Next
End Sub
```

OLE Drag and Drop Demo

This sample shows how to implement automatic and manual OLE drag and drop using **VSFlexGrid Pro 6.0**.

OLE drag and drop can be a little confusing at first, because of all the properties, methods, objects and events that may be involved in the process. However, you only need to handle a few of these events in order to make OLE drag and drop work for you. This demo illustrates the basic concepts and procedures you will need.

Here is how the final application will look:



The three controls shown are OLE drag drop sources and targets. This means you can drag data from one control to the others, or between any of the controls and external applications.

Step 1: Create the Controls

Start a new Visual Basic project including **VSFlexGrid Pro 6.0** (if you don't know how to add OCX files to a project, consult the Visual Basic manual). The **VSFlexGrid** icon will be added to the Visual Basic toolbox.

Create two **VSFlexGrid** objects on the form by clicking the **VSFlexGrid** icon on the toolbox, then clicking on the form and dragging until the objects are the proper size.

Set the name of the **VSFlexGrid** controls to **faDDManual** and **faDDAuto**.

Now add a Microsoft Rich Textbox control to the form (register the **Richtx32.ocx** file if this control is not on your custom control list).

Step 2: Initialize the Controls

We could have set the initial properties of the **faDDManual** and **faDDAuto** controls using the Visual Basic properties window, but chose to do it using the Form_Load event instead. Here is the routine that initializes the controls:

```
Private Sub Form_Load()

  ' initialize manual control
  With faDDManual
    .Cell(flexcpText, 0, 0) = "Manual"
    .FixedCols = 0
    .Editable = True
    .OLEDragMode = flexOLEDragManual
```

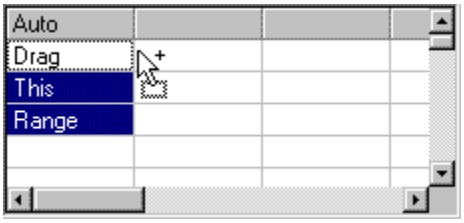
```
    .OLEDropMode = flexOLEDropManual
  End With

  ' initialize auto control
  With faDDAuto
    .Cell(flexcpText, 0, 0) = "Auto"
    .FixedCols = 0
    .Editable = True
    .OLEDragMode = flexOLEDragAutomatic
    .OLEDropMode = flexOLEDropAutomatic
  End With

End Sub
```

The code makes both grids editable, so you can type into them, and sets the **OLEDragMode** and **OLEDropMode** properties to make each control an OLE drag-and-drop source and a target.

There is no need to initialize the Rich Editbox, since its **OLEDragMode** and **OLEDropMode** properties are set to automatic by default.

That's all it takes to implement automatic OLE drag and drop. If you run the project now, you will be able to drag text from the Rich Editbox into the **faDDAuto** grid. You may also drag files from the Window Explorer, ranges from Microsoft Excel, or text from Microsoft Word.

You can also drag selections from the **faDDAuto** grid into any OLE drop target (including other areas of the same control). To do this, select a range and move the mouse cursor to an edge around the selection. The cursor will turn into a default OLE drag cursor, as the picture below shows. Click the left mouse button and start dragging. The cursor will give you visual feedback whenever you move it over an OLE drop target.



As you can see, implementing automatic OLE drag and drop is easy. Just set the **OLEDragMode** and **OLEDropMode** properties to automatic and you are done.

Sometimes you may want to customize the way in which OLE drag and drop works. This sample shows how you can do that by customizing both the drag (OLE source) behavior and the drop (OLE target) behavior of the **faDDManual** control.

Step 3: Manual OLE Drag

We will customize the behavior of the **faDDManual** control as an OLE drag source in two ways:
1) We will initiate dragging whenever the user clicks on the current cell, and
2) We will add a copyright notice to the contents being dragged from the control.

Because the **OLEDragMode** property of the **faDDManual** control is set to *flexOLEDragManual*, you need to initiate the OLE dragging operation with code, using the **OLEDrag** method. To do this we will add code to handle the **BeforeMouseDown** event. When the user clicks on the active cell, we call the **OLEDrag** method. Here is the code:

```
Private Sub faDDManual_BeforeMouseDown(ByVal Button As Integer, _
                        ByVal Shift As Integer, _
                        ByVal X As Single, ByVal Y As Single, _
                        Cancel As Boolean)
  With faDDManual

    ' if the click was on the active cell, start dragging
```

```
    If .MouseRow = .Row And .MouseCol = .Col Then

      ' use OLEDrag method to start manual OLE drag operation
      ' this will fire the OLEStartDrag event, which we will use
      ' to fill the DataObject with the data we want to drag.
      faDDManual.OLEDrag

      ' tell grid control to ignore mouse movements until the
      ' mouse button goes up again
      Cancel = True
    End If

  End With
End Sub
```

The code above checks whether the user clicked on the active cell. If so, it calls the **OLEDrag** method and sets the **Cancel** parameter to True.

Note that we have not specified what the data is. In automatic mode, the control assumed that you wanted to drag the current selection. In manual mode, you are responsible for providing the data.

When the **OLEDrag** method is called, the control fires the **OLEStartDrag** event, which gives you access to a **DataObject** object. You must store the data that will be dragged into the **DataObject** so that the target object can get to it. Here is the code:

```
Private Sub faDDManual_OLEStartDrag(Data As VSFlex6Ctl.vsDataObject,
            AllowedEffects As Long)

  ' set contents of data object for manual drag
  Dim s$
  s = faDDManual.Clip & vbCrLf & "Copyright 1998 VideoSoft"
  Data.SetData s, vbCFText

End Sub
```

The code takes the current selection (contained in the **Clip** property), appends a copyright notice to it, and then assigns it to the *Data* parameter. This is the data that will be exposed to the OLE drop targets.

If you run the project now, and type some data into the **faDDManual** control, you will be able to drag it to one of the other controls on the form. Notice how the copyright notice gets appended to the selection when you make the drop.

Step 3: Manual OLE Drop

We will customize the behavior of the **faDDManual** control as an OLE drop target so that when a list of files is dropped, it opens the first file on the list and displays the contents of the first 10 lines in that file. (The default behavior is to treat lists of files as text, and paste the file names.)

When the user drops an OLE data object on a **VSFlexGrid** control with the **OLEDropMode** property set to *flexOLEDropManual*, the control fires the **OLEDragDrop** event. The data object being dropped is passed as a parameter (*Data*) that you may query for the type of data you want.

The routine below checks to see if *Data* contains a list of files. If so, it opens the first file on the list and reads the contents of its first 10 lines. If the *Data* parameter does not contain any files, then the routine tries to get its text contents. Either way, the routine transfers the data to the grid using the Clip property.

Here is the routine:

```
Private Sub faDDManual_OLEDragDrop(Data As VSFlex6Ctl.vsDataObject, _
            Effect As Long, _
            ByVal Button As Integer, _
            ByVal Shift As Integer, _
            ByVal X As Single, ByVal Y As Single)
  Dim r As Long, c As Long, i As Integer, s As String

  With faDDManual
```

```
    ' get drop location
    r = .MouseRow
    c = .MouseCol

    ' if we're dropping files, open the file and paste contents
    If Data.FileCount > 0 Then
      On Error Resume Next
      Open Data.Files(0) For Input As #1
      For i = 0 To 10
        Line Input #1, s
        .Cell(flexcpText, r + i, c) = s
      Next
      Close #1

    ' drop text using the Clip property
    ElseIf Data.GetFormat(vbCFText) Then
      s = Data.GetData(vbCFText)
      .Select r, c, .Rows - 1, .Cols - 1
      .Clip = s
      .Select r, c

    ' we don't accept anything else
    Else
      MsgBox "Sorry, we only accept text and files..."
    End If
  End With

End Sub
```

That concludes this demo. Run the project again and try dragging and dropping between the controls and other applications.

Visual C++ Demo

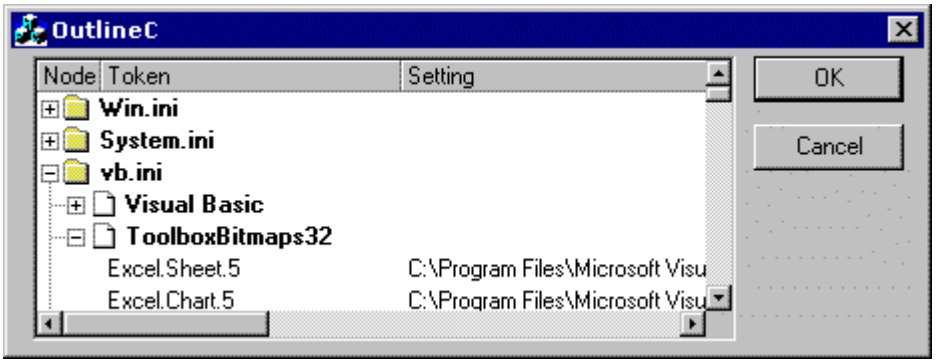
The **VSFlexGrid Pro 6.0** documentation is geared toward Visual Basic users. However, many other development environments are capable of hosting ActiveX controls, including Microsoft Visual C++, Internet Explorer, Microsoft Access, and others.

This demo shows the basic techniques you will need to use **VSFlexGrid Pro 6.0** with Visual C++. (The other environments mentioned are very similar to Visual Basic.)
To follow this demo, you must know how to use the Visual C++ development environment and you must also know C++ (or at least C).

The Visual C++ sample project is similar to the Visual Basic **Outline Demo** that is also a part of this documentation, but it adds a few extra bells and whistles (such as cell pictures), just to show how this is done in C++.

The sample reads several INI files and presents each one as a node. Each file node has a collection of sub-nodes that contain sections within the corresponding INI file. Each section node contains branches that show the tokens and settings stored in the corresponding section.

Here is what the final application will look like:



Step 1: Create the project

Start Microsoft Visual C++ and select **File|New**. You will see a tabbed dialog that lists the types of files you may create.

Select the **Projects** tab, then click the **MFC AppWizard (EXE)** option and type the path where you want to place the new project. Also type in the project name, **OutlineC**.

Click **OK** and the MFC App Wizard will appear. On the first page, click the **Dialog Based** option button. Click **Finish** to accept all other defaults and create the project. You will see a dialog with some information about the new project. Click **OK**.

Step 2: Add the VSFlexGrid Control to the Project

Now that the project has been created, we need to add the **VSFlexGrid** control to the project. (This is equivalent to adding the control to the Visual Basic toolbox.) The exact steps may vary a little depending on the version of Visual C++ you are using.

In VC++ 5, select the **Project|Add and Controls…** menu. You will see a list of elements that you can add to the project. Select **Registered ActiveX Controls** by double clicking it. A list of registered ActiveX controls will appear. If the **VSFlexGrid** control does not appear on the list, you need to register it.

Select the VideoSoft **VSFlexGrid** control, then click **Insert**.

You will see a dialog informing you of the classes that will be generated by the Wizard: **CvsFlexGrid**, **COleFont**, and **COlePicture**. These classes are wrappers that the Wizard creates for you based on information it retrieves from the control's type library. Click **OK** to proceed, then **Close** to dismiss the components dialog.

Go to the VC++ workspace window and select the **Files** pane. You will see that VC++ added a few files to the project, including *vsflexgrid.h* and *vsflexgrid.cpp*. If you open these files, you will see that they define members to access every property and method of the underlying object. For example, the **Row** property of the **VSFlexGrid** ActiveX control is read or set using the **GetRow** and **SetRow** methods of the **CvsFlexGrid** C++ class.

Step 3: Create the VSFlexGrid Control

Go to the VC++ workspace window and select the **Resources** pane. Because this is a dialog-based MFC application, you can design the application by dragging and dropping controls on the main dialog (or form). It's almost like designing a form in Visual Basic.

Open the main dialog (called IDD_OUTLINEC_DIALOG) by double-clicking on it. Then delete the TODO label, pick a **VSFlexGrid** control from the toolbox and drop it on the form. Adjust the size of the dialog and the control until you are happy with the layout.

Now right-click on the control and select **Properties…** from the popup menu. Select the **All** pane and click the pushpin to keep the window on top of the others while you initialize the control's properties. Use the properties window to apply these settings (the same we used in the **Outline** demo):

```
Cols = 3
ExtendLastCol = True
FixedCols = 0
Rows = 1
FormatString = "Node|Token|Setting"
OutlineBar = flexOutlineBarComplete
GridLines = flexGridNone
MergeCells = flexMergeSpill
AllowUserResising = flexResizeColumns
```

Save your project and press F5 to run it. Visual C++ will build the project and you will see that the control is created and initialized properly. Click **OK** or **Cancel** when you are done.

Step 3: Create a Member Variable to Access the Control

Remember how the wizard created wrapper classes to enable easy access to the control? Now we will create a member variable **m_fa** of type **CvsFlexGrid**. This variable will be attached to the control on the form, and it will allow us to read and set the object's properties, trap events and so on.

Return to the VC++ workspace window and select the **Resources** pane. Open the main dialog (called IDD_OUTLINEC_DIALOG) by double-clicking on it.

Now hold down the CONTROL key and double-click on the **VSFlexGrid** control. You will see a dialog prompting you to enter a variable name. Type **m_fa** and click **OK**. The Wizard will create the variable and initialize it for you.

Step 4: Read the Data and Build the Outline

In the **Outline** sample, we placed the code to read the data in the Form_Load event. In this sample we will use the **OK** button instead.

Open the dialog in the Visual C++ resource editor, then type CTRL-W (for Wizard). You will see a dialog that lets you add event handlers to each element on the form.
On the **Object Ids** list, select **IDOK**. On the **Messages** list, select **BN_CLICKED**. Now click the **Add Function** button, and then the **Edit Code** button.

This will open the code editor. You will see that the Wizard already added the function declaration for you. Now type the following code:

```
void COutlineCDlg::OnOK()
{
  // TODO: Add extra validation here
  // comment the following line to avoid closing the
  // dialog when the user clicks OK:
  //CDialog::OnOK();

  // initialize variant to use as optional parameter
  COleVariant varDefault;
  V_VT(&varDefault) = VT_ERROR;

  // suspend repainting to increase speed
  m_fa.SetRedraw(FALSE);

  // populate the control
  AddNode("Win.ini");
  AddNode("System.ini");
  AddNode("vb.ini");

  // expand outline, resize to fit, collapse outline
  m_fa.Outline(-1);
  COleVariant vCol((short)2, VT_I2);
  m_fa.AutoSize(1, vCol, varDefault, varDefault);
  m_fa.Outline(1);

  // repainting is back on
  m_fa.SetRedraw(TRUE);
}
```

The first thing to notice is that you should comment out the line that calls the default handler for this event (**CDialog::OnOK**()). The default handler closes the dialog when the user clicks **OK**, which is not what we want here.

Next, we declare a **varDefault** variable of type variant and initialize it with type VT_ERROR.

This is necessary because many of the methods in the **VSFlexGrid** control take optional parameters. In Visual Basic, optional means you don't have to supply them at all. In Visual C++, optional means you don't have to supply the *value*, but the parameter must still appear in the function calls. This is what the **varDefault** variable does: it is a parameter without a value. (You may prefer to modify the **CvsFlexGrid** wrapper classes and overload the methods to user friendlier parameter lists. We chose not to do it here to keep the example simple.)

The code then calls the **AddNode** function to populate the grid, just like the Visual Basic version of the program did. The **AddNode** function will be discussed later.
Finally, the code calls the **AutoSize** method, which takes three variant parameters. One of them holds the value 2 (the last column to be autosized) and the others use **varDefault**, which means the control will use default values.

Like before, the **AddNode** routine does most of the work. It reads an INI file and populates the control, creating nodes and branches according to the contents of the file. Here is the C++ version of the **AddNode** routine (remember to add its declaration to the **OutlineCDlg.h** file):

```
void COutlineCDlg::AddNode(LPSTR inifile)
{
  long row;

  // initialize variant to use as optional parameter
  COleVariant varDefault;
  V_VT(&varDefault) = VT_ERROR;

  // create file node
  m_fa.AddItem(inifile, varDefault);
  row = m_fa.GetRows() - 1;
  m_fa.SetIsSubtotal(row, TRUE);
  m_fa.Select(row, 0, varDefault, varDefault);
  m_fa.SetCellFontBold(TRUE);

  // read ini file
  CString fn = (CString)"c:\\windows\\" + (CString)inifile;
```

```
    FILE* f = fopen(fn, "rt");
    while (f && !feof(f)) {
        char ln[201];
        fgets(ln, 200, f);

        // if this is a section, add section node
        if (*ln == '[') {
            char* p = strchr(ln, ']');
            if (p) *p = 0;
            m_fa.AddItem(ln + 1, varDefault);
            row = m_fa.GetRows() - 1;
            m_fa.SetIsSubtotal(row, TRUE);
            m_fa.SetRowOutlineLevel(row, 1);
            m_fa.Select(row, 0, varDefault, varDefault);
            m_fa.SetCellFontBold(TRUE);

        // if this is regular data, add branch
        } else if (strchr(ln, '=')) {
            char* p = strchr(ln, '=');
            *p = 0;
            CString str = (CString)"\t" + (CString)ln +
                          (CString)"\t" + (CString)(p + 1);
            m_fa.AddItem(str.GetBuffer(0), varDefault);
        }
    }
    if (f) fclose(f);
}
```

This routine is a line-by-line translation of the Visual Basic **AddNode** routine presented in the **Outline** demo. It uses the MFC **CString** class to create some of the strings, and a few additional variants for parameters.

The routine starts by adding a row containing the name of the INI file being read. It marks the row as a subtotal using the **SetIsSubtotal** method so the control will recognize it as an outline node.

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 marks them as subtotals the same way it marked the file name. The difference is that here the **SetRowOutlineLevel** method is used to indicate that this node is a child of the previous level-0 node (the one that contains the file name).

Finally, lines containing data are parsed into token and setting and then added to the control. They are not marked as subtotals.

Step 5: Use the Outline

Press F5 to run the project, click the **OK** button, and you will see the outline in action. If you click on one of the nodes, it will expand or collapse to show or hide the data under it.
You may also SHIFT-click on a node to expand the entire outline to the node's level, or SHIFT-CTRL-click on a node to collapse the entire outline to that level. For example, if you SHIFT-click on a file name, you will see all file names and all sections, but no token data. If you SHIFT-CTRL-click on a file name, you will see all file names, and nothing else.

Step 6: Custom Mouse and Keyboard Handling

To add custom mouse and keyboard handling similar to those implemented in the Visual Basic version of the **Outline** demo, we need to handle the **DblClick** and **KeyPress** events.

Adding the event handlers is easy: click CTRL-W to invoke the Wizard, select the **VSFLEXGRID1** object on the **Object Ids** list, then select each event and click the **Add Function** button. When you are done, click the **Edit Code** button and type the following code:

```
  #define flexOutlineExpanded  0
  #define flexOutlineSubtotals 1
  #define flexOutlineCollapsed 2
```

```
void COutlineCDlg::OnDblClickVsflexgrid1()
{
  // double clicking on a row expands or collapses it
  long r = m_fa.GetRow();
  if (m_fa.GetIsCollapsed(r) == flexOutlineCollapsed)
      m_fa.SetIsCollapsed(r, flexOutlineExpanded);
  else
      m_fa.SetIsCollapsed(r, flexOutlineCollapsed);
}

void COutlineCDlg::OnKeyPressVsflexgrid1(short FAR* KeyAscii)
{
  if (*KeyAscii == VK_RETURN) {
    OnDblClickVsflexgrid1();
    *KeyAscii = 0;
  }
}
```

Again, the code is a line-by-line translation of the Visual Basic **Outline** example.

Step 7: Cell Pictures

The final step shows how you can add cell pictures using C++.

First of all, you need to use the VC++ resource editor and add two bitmap resources to the project. Make the bitmaps approximately 15 by 15 pixels in size and name them **IDB_FILE** and **IDB_SECTION**.

Then, make the following changes to the **AddNode** routine (the changes are marked in boldface):

```
#include <afxctl.h>
void COutlineCDlg::AddNode(LPSTR inifile)
{
  long row;

  // initialize pictures
  CPictureHolder picFile, picSection;
  picFile.CreateFromBitmap(IDB_FILE);
  picSection.CreateFromBitmap(IDB_SECTION);

  // initialize variant to use as optional parameter
  …

  // create file node
  …
  m_fa.SetCellFontBold(TRUE);
  m_fa.SetCellPicture(picFile.GetPictureDispatch());

  // read ini file
  …
      // if this is a section, add section node
      …
          m_fa.SetCellFontBold(TRUE);
          m_fa.SetCellPicture(picSection.GetPictureDispatch());

      // if this is regular data, add branch
      …
  }
  if (f) fclose(f);
}
```

The first line added includes the MFC header file **afxctl.h**. This file defines **CPictureHolder**, a handy class for manipulating OLE pictures.

The next three lines added declare a **CPictureHolder** variable for each bitmap, and load the bitmaps using the **CreateFromBitmap** method.

Finally, the **SetCellPicture** method is used to assign the pictures to cells that are file and section nodes.

This concludes this demo. Run the project once again to see the final result.

VSFlexString QuickStart

This section takes you through four examples and the step-by-step creation of a Visual Basic project using the **vsFlexArray** control:

Regular Expressions
Illustrates the notation used in regular expression text matching.

Matching Demo
An example of **vsFlexString's** text matching capabilites.

Replacing Demo
An example of **vsFlexString's** automatic replace capabilites.

Tag Match Demo
An example using **vsFlexString's** tag matching capabilities.

Expression Evaluator Demo
This sample takes you step by step through the creation of a Visual Basic project using the **vsFlexString** control. It features pattern matching and shows how **vsFlexString** can be used to implement a mathematical expression evaluator.

These are simple programs that focus on using the **vsFlexString** 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 **vsFlexString**. For more realistic (and ambitious) projects, please check out the samples on the distribution CD or disks.

Regular Expressions

The **VSFlexString** control finds patterns in strings. The pattern being searched (stored in the **Pattern** property) is 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. The **VSFlexString** control recognizes the following operators (special characters):

Char	Description
^	Matches the beginning of a string.
$	Matches the end of a string.
.	Matches any character.
[]	Character class (any of).
[^]	Complemented character class (any but).
*	Repeat previous zero or more times.
+	Repeat previous one or more times.
?	Repeat previous zero or one time.
\	Treat next character as a literal (e.g. \. means period).
{}	Tagged match.

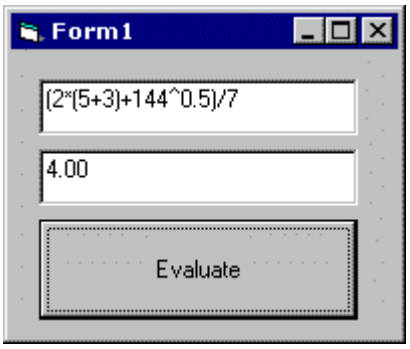
The following examples illustrate how these characters are used:

Pattern	Description
^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 followed zero or more of digits.
[A-Z][0-9]+	Any upper-case letter followed one or more of digits.
[A-Z][0-9]?	Any upper-case letter followed by one optional digit.
[+-]?[0-9]+	Any integer preceded by an optional sign.

Expression Evaluator Demo

This sample project illustrates some of **VSFlexString's** pattern matching capabilities. It shows how **VSFlexString** 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.

Here is how the final application will look:



Step 1: Create the Controls

Start a new Visual Basic project including **VideoSoft VSFLEX Grid Pro 6.0** (if you don't know how to add OCXs to a project, consult the Visual Basic documentation). The **VSFlexString** icon will be added to the Visual Basic toolbox.

Create a **VSFlexString** object on the form by clicking the **VSFlexString** icon on the toolbox, then clicking on the form and dropping it on the form. Also create two text boxes and a command button. Arrange the controls and resize the form so it looks like the picture above.

Click on the **VSFlexString** control and use the Visual Basic properties window to change its name to **fs**.

Step 2: Evaluating Expressions

This project consists basically of a single recursive function that uses the **VSFlexString** control to evaluate the expressions typed in the text box.

This function, which we will write later, needs to be connected to the command button through the Command1_Click event handler. All we need is a single line of code:

```
Sub Command1_Click ()

  ' evaluate the expression in Text1 and
  ' show the result in Text2
  Text2 = Format(Eval(Text1), "0.00")

End Sub
```

That leaves only the **Eval** function, which takes a string containing a mathematical expression as a parameter and returns a value. Here is the code that implements the **Eval** function:

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

  ' get ready to parse
  fs = Trim(s)    ' set breakpoint on this line

  ' interpret sub-expressions enclosed in parentheses
  fs.Pattern = "{.*}({[^()]*}){.*}"
  If fs.MatchCount > 0 Then
```

```
    s1 = fs.TagString(0)    ' stuff to the left
    s2 = fs.TagString(1)    ' sub-expression
    s3 = fs.TagString(2)    ' stuff to the right
    Debug.Print "match: "; s1; " #<(># "; s2; " #<)># "; s3
    v = Eval(s2)            ' evaluate sub-expression
    Eval = Eval(s1 + Format(v) + s3)
    Exit Function
  End If

  ' add and subtract (high-priority operators)
  fs.Pattern = "{.+}{[+-]}{.+}"
  If fs.MatchCount > 0 Then
    s1 = fs.TagString(0)       ' operand 1
    s2 = fs.TagString(2)       ' operand 2
    Debug.Print "match: "; s1; " #<+->#  "; s2
    Select Case fs.TagString(1)
      Case "+": Eval = Eval(s1) + Eval(s2)
      Case "-": Eval = Eval(s1) - Eval(s2)
    End Select
    Exit Function
  End If

  ' multiply and divide (lower-priority operators)
  fs.Pattern = "{.+}{[*/]}{.+}"
  If fs.MatchCount > 0 Then
    s1 = fs.TagString(0)       ' operand 1
    s2 = fs.TagString(2)       ' operand 2
    Debug.Print "match: "; s1; " #<*/>#  "; s2
    Select Case fs.TagString(1)
      Case "*": Eval = Eval(s1) * Eval(s2)
      Case "/": Eval = Eval(s1) / Eval(s2)
    End Select
    Exit Function
  End If

  ' power (lowest-priority operator)
  fs.Pattern = "{.+}^{.+}"
  If fs.MatchCount > 0 Then
    s1 = fs.TagString(0)       ' operand 1
    s2 = fs.TagString(1)       ' operand 2
    Debug.Print "match: "; s1; "#<^>#"; s2
    Eval = Eval(s1) ^ Eval(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
    Eval = Val(s)
  Else
    Debug.Print "Eval Error: "; fs: Beep
  End If
End Function
```

This routine handles all basic operators taking into account their precedence (i.e., power before division before sum). It also handles sub-expressions contained in parentheses.

If you understand how **VSFlexString** works, the **Eval** function is pretty simple. It consists of a pattern that repeats itself. The **VSFlexString** is used to parse each expression into its components, according to operator priority rules, then **Eval** is called recursively to evaluate each component.

The typical pattern has this format: "{.+}{[*/]}{.+}". The "{+.}" matches runs of one or more characters. The "{[*/]}" matches a single asterisk or a slash. The other patterns have similar interpretations.

Press F5 to run the project and type an expression such as "(2*(5+3)+144^0.5)/7". Then click the command button and the result (4) will appear on the second text box. The debug window will show a trace of the **Eval** function. Here's a commented version of the output:

```
match: (2* #<(># 5+3 #<)># +144^0.5)/7 found sub-expression
```

```
match: 5 #<+-># 3                       found +
match:  #<(># 2*8+144^0.5 #<)># /7      found sub-expression
match: 2*8 #<+-># 144^0.5               found +
match: 144 #<^># 0.5                    found ^
match: 2 #<*/># 8                       found *
match: 28 #<*/># 7                      found /
```

The trace shows the order in which matches were found and operations executed. You may want to place a breakpoint at the top of the **Eval** routine and see what happens after each match.

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.

Matching Demo

Whenever a string is assigned to either the **Text** or **Pattern** properties, the **VSFlexString** control scans the text to find as many matches as it can. The number of matches found is stored in the **MatchCount** property. Information about individual matches can be retrieved using the **MatchString**, **MatchStart**, and **MatchLength** properties.

For example, the following code scans a string that consists of clients names and phone numbers, separated by commas. It then prints a list of the clients in the San Francisco area (area code 415).

The pattern used assumes that all area codes are three digit numbers enclosed in parentheses. The entries we are interested in are strings that do not contain commas and that do contain the string "(415)".

```
ClientList = "John Doe: (415) 555-1212," & _
             "Mary Smith: (212) 555-1212," & _
             "Dick Tracy: (412) 555-1212," & _
             "Martin Long: (415) 555-1212," & _
             "Leo Getz: (510) 555-1212," & _
             "Homer Simpson: (415) 555-1212"
fs.Text = ClientList
fs.Pattern = "[^,]*(415)[^,]*"
Debug.Print fs.MatchCount " match(es) found."
For i = 0 to fs.MatchCount - 1
  Debug.Print "found: "; fs.MatchString(i)
Next

  found: John Doe: (415) 555-1212
  found: Martin Long: (415) 555-1212
  found: Homer Simpson: (415) 555-1212
```

Replacing Demo

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

```
ClientList = "John Doe: (415) 555-1212," & _
             "Mary Smith: (212) 555-1212," & _
             "Dick Tracy: (412) 555-1212," & _
             "Martin Long: (415) 555-1212," & _
             "Leo Getz: (510) 555-1212," & _
             "Homer Simpson: (415) 555-1212"
fs.Text = ClientList
fs.Pattern = "(415)"
fs.Replace = "(510)"
```

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

This is convenient, but **VSFlexString** goes way beyond simple search and replace. You can use **tags** to control each part of each match. For an example, see the **Tag Matches Demo**.

Tag Matches Demo

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

For example, say you have a database that contains people's names. But the same name may be stored as "John Doe", "John Francis Doe", "John F. Doe", or "Doe, John". You could use the following code to clean the data, converting all entries to the latter type:

```
Private Function CleanName(n$) As String

  ' assign pattern
  '              |tag(0)---|      |tag(1)---|
  fs.Pattern = "^{[A-Za-z]+}[^,]* {[A-Za-z]+}$"

  ' try match
  fs.Text = n

  ' if a match was found, replace name with
  ' tag1 (last name), comma, tag0 (first name)
  If fs.MatchCount > 0 Then
    CleanName = fs.TagString(1) & ", " & fs.TagString(0)

  ' otherwise, return original name
  Else
    CleanName = n
  End If

End Function
```

You may test the function using the Visual Basic debug (immediate mode) window:

```
? CleanName("John Doe")
  Doe, John
? CleanName("John    Doe")
  Doe, John
? CleanName("Doe, John")
  Doe, John
? CleanName("John F. Doe")
  Doe, John
? CleanName("John Francis Doe")
  Doe, John
? CleanName("John Francis Jr.")
  John Francis Jr.
```

Pretty neat, huh? Note that the last try fails, because the last name is not supposed to contain periods. The function just returns the original string.

To understand how this works, you need to understand the pattern. (This one is not trivial.)

```
  '              |tag(0)---|      |tag(1)---|
  fs.Pattern = "^{[A-Za-z]+}[^,]* {[A-Za-z]+}$"
```

The initial character (`"^"`) matches the beginning of a string.

The next part, (`"[A-Za-z]+"`) means *a sequence of one or more letters*. This will match the first name. By enclosing this expression in curly brackets, we are telling the **VSFlexString** control to tag it, so we can refer to it later.

The next part, (`"[^,]* "`), means *a sequence of zero or more non-comma characters followed by a space*. This will match optional middle names and initials. It will also prevent matches when the input contains commas (we assume it is already properly formatted in this case).

The next part is similar to the one used to match the first name. This one will match the last name.

Finally, the trailing character (`"$"`) matches the end of a string.

Whenever a match occurs with this pattern, the tagged parts of the match can be retrieved using the **TagString** property. In this case, we have two tags: **TagString**(0) matches the first name, and **TagString**(1) matches the last name. With these, it is easy to rewrite the name in the format we want.

Writing the patterns is not difficult, but it does require some practice. This sample is a good starting point.

vsFlexGrid Object

Properties Methods Events

Object Name: VSFlexGrid
Description: :-) VideoSoft VSFlexGrid 6.0
Properties: 145
Events: 35
Methods: 15

Before you can use a **VSFlexGrid** object in your application, you must add the **VSFLEX6.OCX** file to your project.

To distribute applications you create with the **VSFlexGrid** object, you must install and register it on the user's computer. The Setup Wizard provided with Visual Basic provides tools to help you do that. Please refer to the Visual Basic manual for details.

AddItem Method

See Also Examples Applies to

Adds a row to the control.

Syntax

[*form*!]*vsFlexGrid*.**AddItem** *Item* As String, [*Row* As Variant]

Remarks

The parameters for the **AddItem** method are described below:

Item As String
String expression to add to the control. Use the tab character (vbTab or Chr$(9)) to separate multiple strings you want inserted into each column of a newly added row.

Row As Long (optional)
Zero-based index representing the position within the control where the new row is placed. If Row is omitted, the new row is appended as the last one.

AfterDataRefresh Event

See Also Examples Applies to

Fired after reading data from the record source.

Syntax

Private Sub *vsFlexGrid*_**AfterDataRefresh**()

Remarks

This event is useful when the control is bound and you want to work on data that comes from a database. For example, you may want to calculate subtotals or format individual cells.

You cannot do this in the Form_Load event because the data has not been read at that point of execution.

AfterEdit Event

See Also Examples Applies to

Fired after the control exits cell edit mode.

Syntax

Private Sub *vsFlexGrid*_**AfterEdit**(ByVal *Row* As Long, ByVal *Col* As Long)

Remarks

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

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

The **AfterEdit** event is not useful for validation, because it is fired after the changes have been applied to the control.

To validate user-entered data, use the **ValidateEdit** event instead.

AfterMoveColumn Event

See Also Examples Applies to

Fired after a column is moved by dragging on the ExplorerBar.

Syntax

Private Sub *vsFlexGrid*_**AfterMoveColumn**(ByVal *Col* As Long, *Position* As Long)

Remarks

This event is only fired if the column was moved by dragging it in the **ExplorerBar**. It is not fired after moving columns with the **ColPosition** property.

This event is useful if you want to sort or recalculate subtotals on the grid after moving its columns.

AfterSort Event

See Also Examples Applies to

Fired after a column is sorted by a click on the ExplorerBar.

Syntax

Private Sub *vsFlexGrid*_**AfterSort**(ByVal *Col* As Long, *Order* As Integer)

Remarks

This event is only fired if the sorting was caused by a click on the **ExplorerBar**. It is not fired after sorting with the **Sort** property.

This event is useful if you want to recalculate subtotals on the grid after sorting a column.

AfterUserResize Event

See Also Examples Applies to

Fired after the user resizes a row or a column.

Syntax

Private Sub *vsFlexGrid*_**AfterUserResize**(ByVal *Row* As Long, ByVal *Col* As Long)

Remarks

The user may resize rows and columns depending on the setting of the **AllowUserResizing** property.

If the user resized a row, the Row parameter contains the index of the row that was resized and the Col parameter contains -1. If the user resized a column, the Col parameter contains the index of the column that was resized and the Row parameter contains -1.

AllowBigSelection Property

See Also Examples Applies to

Returns or sets whether clicking on the fixed area will select entire columns and rows.

Syntax

[*form*!]*vsFlexGrid*.**AllowBigSelection**[= {**True | False**}]

Remarks

If the **AllowBigSelection** property is set to True, clicking on the top left fixed cell selects all cells in the sheet.

Data Type

Boolean

Default Value

True

AllowSelection Property

See Also Examples Applies to

Returns or sets whether the user can select ranges of cells with the mouse and keyboard.

Syntax

[*form*!]*vsFlexGrid*.**AllowSelection**[= {**True | False**}]

Remarks

Set this property to False to prevent users from extending the selection by clicking and dragging or using the cursor keys.

This is useful if you are using **VSFlexGrid** to implement some custom user interface elements such as menus and property sheets.

Data Type

Boolean

Default Value

True

AllowUserResizing Property

See Also Examples Applies to

Returns or sets whether the user is allowed to resize rows and columns with the mouse.

Syntax

[*form*!]*vsFlexGrid*.**AllowUserResizing**[= *AllowUserResizeSettings*]

Remarks

Valid settings for the **AllowUserResizing** property are:

Value	Constant
0	*flexResizeNone*
1	*flexResizeColumns*
2	*flexResizeRows*
3	*flexResizeBoth*

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

If you intend to use this property, you should not set the Gridlines property to FlexGrid none.

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 a 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 pixel, not to zero. For example:

```
fa.ColWidth(5) = Screen.TwipsPerPixelX
```

The **BeforeUserResize** event is fired before resizing starts, and may be used to prevent resizing of specific rows and columns. The **AfterUserResize** event is fired after resizing, and may be used to validate the user's action.

Data Type

AllowUserResizeSettings (Enumeration)

Default Value

flexResizeNone (0)

Archive Method

See Also Examples Applies to

Adds, extracts, or deletes files from a vsFlex archive file.

Syntax

[*form*!]*vsFlexGrid*.**Archive** *ArcFileName* As String, *FileName* As String, *Action* As ArchiveSettings

Remarks

This method allows you to combine several files into one, optionally compressing the data. This is especially useful for applications that store data in several grids.

To save the grid to a file, use the **SaveGrid** method. To load the data back from the file, use the **LoadGrid** method. To obtain information from an archive file, use the **ArchiveInfo** property.

The parameters for the **Archive** method are described below:

ArcFileName As String
This parameter contains the name of the archive file, including its path.

FileName As String
This parameter contains the name of the file to be added, deleted, or extracted from the archive.

Action As ArchiveSettings
This parameter can be one of the following:

Constant	Description
arcAdd	Adds the file *FileName* to the archive *ArcFileName*, compressing it. If the archive file does not exist, it is created. If the file is already present in the archive, it is overwritten with the new contents.
arcStore	Adds the file *FileName* to the archive *ArcFileName*, without compressing it. If the archive file does not exist, it is created. If the file is already present in the archive, it is overwritten with the new contents.
arcDelete	Removes the file *FileName* from the archive *ArcFileName*.
arcExtract	Creates a copy of the file *FileName* on the disk. The file is created on the directory specified in the *FileName* parameter, or in the archive directory if no path is specfied.

ArchiveInfo Property

See Also Examples Applies to

Gets information from a vsFlex archive file.

Syntax

[*form*!]*vsFlexGrid*.**ArchiveInfo**(*ArcFileName* As String, *InfoType* As ArchiveInfoSettings, [*Index* As Variant])

Remarks

This property returns information from an archive file created with the **Archive** method.

The parameters for the **ArchiveInfo** property are described below:

ArcFileName as String
This parameter contains the name of the archive file, including its path.

InfoType As ArchiveInfoSettings
This parameter can be one of the following:

Constant	Description
arcFileCount	Returns the number of files in the archive.
arcFileName	Returns the name of a file in the archive.
arcFileSize	Returns the original size of a file in the archive.
arcFileCompressedSize	Returns the compressed size of a file in the archive.

Index As Integer (optional)
This parameter specifies which file in the archive should be processed. It ranges from zero to the number of files in the archive minus 1.

For example, the code below lists the contents of an archive file.

```
Sub ArcList(fn$)
  Dim i As Long, cut As Long
  With fa
    On Error Resume Next
    cnt = .ArchiveInfo(fn, arcFileCount)
    Debug.Print "Archive "; fn; " ("; cnt; " files)"
    Debug.Print "Name", "Size", "Compressed"
    For i = 0 To cnt - 1
      Debug.Print .ArchiveInfo(fn, arcFileName, i),
      Debug.Print .ArchiveInfo(fn, arcFileSize, i),
      Debug.Print .ArchiveInfo(fn, arcFileCompressedSize, i)
    Next
    If Err > 0 Then MsgBox "An error occurred while processing " & fn
  End With
End Sub
```

Data Type

Variant

AutoResize Property

See Also Examples Applies to

Returns or sets whether column widths will be automatically adjusted when data is loaded.

Syntax

[*form*!]*vsFlexGrid*.**AutoResize**[= {**True | False**}]

Remarks

If the **AutoResize** property is set to True, the control automatically resizes its columns to fit the widest entry every time new data is read from the database. This occurs by default when the control is loaded and every time the data source is refreshed.

This property only works when the control is bound to a database. If the control is not bound to a database, you may use the **AutoSize** method to adjust column widths after changes are made to the grid contents.

Data Type

Boolean

Default Value

True

AutoSearch Property

See Also Examples Applies to

Returns or sets whether the control will search for entries as they are typed.

Syntax

[*form*!]*vsFlexGrid*.**AutoSearch**[= *AutoSearchSettings*]

Remarks

The **AutoSearch** property allows the user to look for data by typing the string for which they are looking.

If **AutoSearch** is on, the control will search the current column as the user types. The control will move the selection and highlight partial matches. The search is case-insensitive.

To cancel a search, the user may press Escape or simply move the selection with the mouse or cursor keys.

If **AutoSearch** is on and the **Editable** property is set to True, the user will need to hit Enter, Space, or F2 to start editing cells. Other keys are used for searching.

The effects of the settings for the **AutoSearch** property are described below:

Constant	Description
flexSearchNone	Turns **AutoSearch** off.
flexSearchFromTop	**AutoSearch** from the first row.
flexSearchFromCursor	**AutoSearch** from the current row.

Data Type

AutoSearchSettings (Enumeration)

Default Value

flexSearchNone (0)

AutoSize Method

See Also Examples Applies to

Resizes column widths or row heights to fit cell contents.

Syntax

[*form*!]*vsFlexGrid*.**AutoSize** *Col1* As Long, [*Col2* As Variant], [*Equal* As Variant], [*ExtraSpace* As Variant]

Remarks

The parameters for the **AutoSize** method are described below:

Col1 As Long, *Col2* As Long
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. *Col2* is optional. If it is omitted, only *Col1* is resized.

Equal As Variant (optional)
If True, all columns between *Col1* and *Col2* are set to the same width. If False, then each column is resized independently. This parameter is optional and defaults to False.

ExtraSpace As Variant (optional)
Allows you to specify extra spacing, in twips, to be added in addition to the minimum required to fit the widest entry. This is often useful if you wish to leave extra room for pictures or margins within cells.

The **AutoSize** method may also be used to resize row heights. This is useful when text is allowed to wrap within cells (see the **WordWrap** property) or when cells have fonts of different sizes (see the **Cell** property).

The **AutoSizeMode** property determines whether AutoSize will adjust column widths or row heights.

AutoSizeMode Property

See Also Examples Applies to

Returns or sets whether AutoSize will adjust column widths or row heights to fit cell contents.

Syntax

[*form*!]*vsFlexGrid*.**AutoSizeMode**[= *AutoSizeSettings*]

Remarks

Valid settings for the **AutoSizeMode** property are:

Value	Constant
0	*flexAutoSizeColWidth*
1	*flexAutoSizeRowHeight*

The effect of the settings for the **AutoSizeMode** property are described below:

flexAutoSizeColWidth
This setting causes the **AutoSize** method to adjust the widths of the specified columns to accommodate the longest entry in each column.

flexAutoSizeRowHeight
This setting causes the **AutoSize** method to adjust the height of each row in the specified columns to accommodate the longest entry in each row. This is useful when text is allowed to wrap within cells (see the **WordWrap** property) or when cells have fonts of different sizes (see the **Cell** property).

Data Type

AutoSizeSettings (Enumeration)

Default Value

flexAutoSizeColWidth (0)

BackColor* Property

See Also Examples Applies to

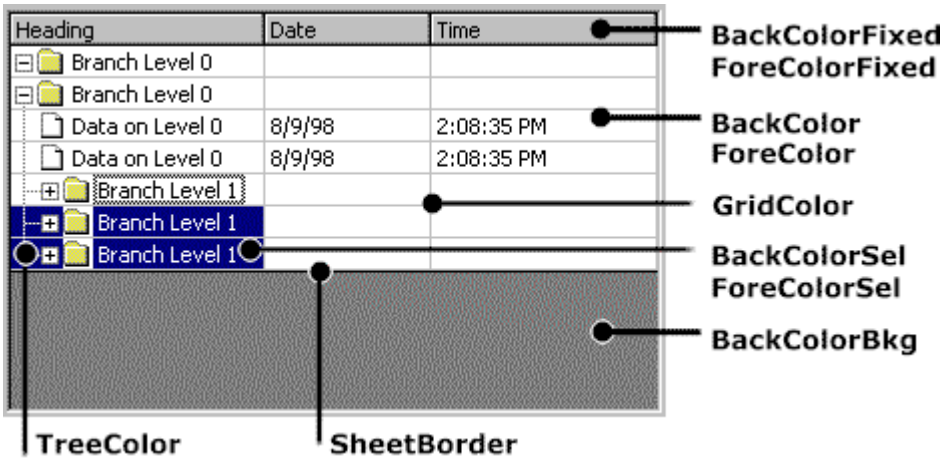
Returns or sets the background color of the non-fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**BackColor**[= *colorref&*]

Remarks

The **VSFlexGrid** control has several properties that allow you to customize its colors. The picture below shows these properties and to which part of the control each one refers:



To set the background color of individual cells or ranges, use the **Cell**(*flexcpBackColor*) property. To set the background color of the current selection, use the **CellBackColor** property.

Data Type

Color

BackColorAlternate Property

See Also Examples Applies to

Returns or sets the background color for alternate rows (set to 0 to disable).

Syntax

[*form*!]*vsFlexGrid*.**BackColorAlternate**[= *colorref&*]

Remarks

If you set the **BackColorAlternate** property to a value other than False (zero), the color specified is used to paint every other row in the control, creating a checkbook look.

Using this property is faster and more efficient than using the **CellBackColor** property to paint every other row. Besides, the alternating colors are preserved even if you sort the grid or add and remove rows.

Data Type

Color

Default Value

Windows (System Color)

BackColorBkg Property

See Also Examples Applies to

Returns or sets the background color of the area not covered by any cells.

Syntax

[*form*!]*vsFlexGrid*.**BackColorBkg**[= *colorref&*]

Remarks

See the **BackColor** property for a diagram.

Data Type

Color

Default Value

Button Shadow

BackColorFixed Property

See Also Examples Applies to

Returns or sets the background color of the fixed rows and columns.

Syntax

[*form*!]*vsFlexGrid*.**BackColorFixed**[= *colorref&*]

Remarks

See the **BackColor** property for a diagram.

Data Type

Color

Default Value

Button Face (System Color)

BackColorSel Property

See Also Examples Applies to

Returns or sets the background color of the selected cells.

Syntax

[*form*!]*vsFlexGrid*.**BackColorSel**[= *colorref*&]

Remarks

See the **BackColor** property for a diagram.

Data Type

Color

BeforeDataRefresh Event

See Also Examples Applies to

Fired before reading data from the record source.

Syntax

Private Sub *vsFlexGrid*_**BeforeDataRefresh**(*Cancel* As Boolean)

Remarks

This event is fired when the control is bound, right before a batch of data is loaded from the database.

You may trap this event and prevent the data from being loaded if you wish. You may later force the data to be loaded by using the **DataRefresh** method.

BeforeEdit Event

See Also Examples Applies to

Fired before the control enters cell edit mode.

Syntax

Private Sub *vsFlexGrid*_**BeforeEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *Cancel* As Boolean)

Remarks

This event is fired immediately before the control enters cell-editing mode. It allows you to prevent editing, to supply a list of choices for a combo list with the **ComboList** property, or to specify an edit mask with the **EditMask** property.

The **BeforeEdit** event occurs after the **KeyDown** event. Unless the edit mode is cancelled by setting the *Cancel* parameter to True in the **BeforeEdit** event, **KeyDown** is followed by **KeyPressEdit** event, not by **KeyPress**.

The parameters for the **BeforeEdit** event are described below:

Row As Long, *Col* As Long
These parameters specify which cell is about to be edited.

Cancel As Boolean
This 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.

BeforeMouseDown Event

See Also Examples Applies to

Fired before the control processes the MouseDown event.

Syntax

Private Sub *vsFlexGrid*_**BeforeMouseDown**(ByVal *Button* As Integer, ByVal *Shift* As Integer, ByVal *X* As Single, ByVal *Y* As Single, *Cancel* As Boolean)

Remarks

The parameters for this event are identical to the ones in the **MouseDown** event, plus an additional *Cancel* parameter that allows you to prevent the default processing.

This event is useful if you want to process some mouse actions yourself, instead of relying on the control's default processing.

For example, the following routine detects shift-clicks and uses them to build and save a list of selected rows. Then it initiates a drag operation using Visual Basic's **Drag** method, and cancels the default processing so the control does not modify the selection.

```
Private Sub fa_BeforeMouseDown(ByVal Button As Integer, _
                    ByVal Shift As Integer, _
                    ByVal X As Single, ByVal Y As Single, Cancel As Boolean)

  ' use shift to drag (ctrl selects)
  If Shift <> 1 Then Exit Sub

  ' build a list of what we'll be dragging
  Dim i As Long
  fa.Tag = ""
  For i = 0 To fa.SelectedRows - 1
    fa.Tag = fa.Tag & vbCrLf & vbTab & fa.Cell(flexcpText, fa.SelectedRow(i), 0)
  Next

  ' start dragging
  fa.Drag

  ' cancel remaining mouse events
  Cancel = True
End Sub
```

BeforeMoveColumn Event

See Also Examples Applies to

Fired before a column is moved by dragging on the ExplorerBar.

Syntax

Private Sub *vsFlexGrid*_**BeforeMoveColumn**(ByVal *Col* As Long, *Position* As Long)

Remarks

This event is only fired if the column was moved by dragging it into the **ExplorerBar**. It is not fired after before moving with the **ColPosition** property.

This event is useful when you want to prevent the user from moving certain columns to invalid positions. You may do so by modifying the value of the *Position* parameter. For example, if you set *Position* = *Col*, the column will not be moved.

BeforePageBreak Event

See Also Examples Applies to

Fired while printing the control to control page breaks.

Syntax

Private Sub *vsFlexGrid*_**BeforePageBreak**(ByVal *Row* As Long, *BreakOK* As Boolean)

Remarks

This event is fired while the control is being rendered on a page or print preview window using VideoSoft's **VSPrinter** control. If you are not using **VSPrinter** to render the control, you do not need to handle this event at all.

Set the *BreakOK* parameter to True to indicate that row number *Row* is an acceptable place to insert a page break, or set it to False to indicate otherwise.

See also the **GetHeaderRow** event and the **RenderControl Demo**.

For more information on using **VSPrinter** to render other controls, refer to the **VSPrinter** documentation.

BeforeScrollTip Event

See Also Examples Applies to

Fired before a scroll tip is shown so you can set the ScrollTipText property.

Syntax

Private Sub *vsFlexGrid*_**BeforeScrollTip**(ByVal *Row* As Long)

Remarks

This event is fired only if the **ScrollTips** property is set to True. It allows you to set the **ScrollTipText** property to a descriptive string for the given row.

For more details, see the **ScrollTips** property.

BeforeSort Event

See Also Examples Applies to

Fired before a column is sorted by a click on the ExplorerBar.

Syntax

Private Sub *vsFlexGrid*_**BeforeSort**(ByVal *Col* As Long, *Order* As Integer)

Remarks

This event is only fired if the sorting was caused by a click on the **ExplorerBar**. It is not fired before sorting with the **Sort** property.

This event is useful when you want to prevent the user from sorting certain columns or to specify custom sorting orders for specific columns. You may do so by modifying the value of the *Order* parameter.

BeforeUserResize Event

See Also Examples Applies to

Fired before the user starts resizing a row or column, allows cancel.

Syntax

Private Sub *vsFlexGrid*_**BeforeUserResize**(ByVal *Row* As Long, ByVal *Col* As Long, *Cancel* As Boolean)

Remarks

The user may resize rows and columns depending on the setting of the **AllowUserResizing** property.

If the user is about to start resizing a row, the *Row* parameter contains the index of the row to be resized and the *Col* parameter contains -1. If the user is about to start resizing a column, the *Col* parameter contains the index of the column to be resized and the *Row* parameter contains -1.

You may prevent the user from resizing specific rows and columns by setting the *Cancel* parameter to True.

BindToArray Method

See Also Examples Applies to

Binds the grid to an array of variants to be used as storage.

Syntax

[*form*!]*vsFlexGrid*.**BindToArray** *VariantArray* As Variant, [*RowDim* As Variant], [*ColDim* As Variant], [*PageDim* As Variant], [*CurrentPage* As Variant]

Remarks

This method allows you to bind the VSFlexGrid control to a Visual Basic array of Variants. Then you don't have to copy data between the array and the control: the control displays values read from the array and writes them back into it automatically.

The array must have at least two dimensions and it must be an array of Variants. If the array has more than two dimensions, you may use the control to display one "page" of it at a time, and you may easily "flip pages".

The parameters on this method allow you to control how the rows and columns map onto the array's dimensions. By default, columns bind to the first array dimension (0) and rows bind to the second array dimension (1). This is the order used by ADO when returning recordsets.

The advantage of this default setting is that you may add or remove rows while preserving existing data using Visual Basic's **Redim Preserve** statement, which only allows the last dimension to be modified. If you don't like the default setting, you may define things differently.

The mapping is always from LBound to UBound on all dimensions. If you want to hide some rows or columns, set their height or width to zero. The binding does not apply to fixed rows or columns. It works only for the scrollable (data) part of the control.

After you change an array that is bound to a flex (any values in it or its dimensions), you should tell the control to repaint itself so the changes become visible to the user. You may do this with the **Refresh** method or by using the **BindToArray** method again.

To unbind the control, call the **BindToArray** method with a Null parameter:

fa.`BindToArray Null`).

The example below illustrates several variations on this theme. The demo project included in the distribution package has more examples.

```
' ** Two-dimensional binding:
Dim arr(4, 8)

' Default binding:
fa.BindToArray arr
' fa now has 5 non-fixed columns (0-4) and 9 non-fixed rows (0-8).
' fa and arr are mapped like this:
'   arr(i, j) = fa.TextArray(j - fa.FixedRows, i - fa.FixedCols)

' Transposed binding:
fa.BindToArray arr, 0, 1
' fa now has 9 non-fixed columns (0-8) and 5 non-fixed rows (0-4).
' fa and arr are mapped like this:
'   arr(i, j) = fa.TextArray(i - fa.FixedRows, j - fa.FixedCols)

' ** Three-dimensional binding (aka cube, notebook):
ReDim arr(4, 8, 12)

' Default binding:
fa.BindToArray a
' by default, the last dimension becomes the "pages", and the
' current page is the first (0), so
' fa now has 5 non-fixed columns (0-4) and 9 non-fixed rows (0-8).
' fa and arr are mapped like this:
```

```
'   arr(i, j, 0) = fa.TextArray(j - fa.FixedRows, i - fa.FixedCols)

' Page Flipping:
fa.BindToArray a, , , , 2
' the row, col, and page settings are the default, and the current
' page is 2 (instead of the default 0), so
' fa now has 5 non-fixed columns (0-4) and 9 non-fixed rows (0-8).
' fa and arr are mapped like this:
'   arr(i, j, 2) = fa.TextArray(j - fa.FixedRows, i - fa.FixedCols)
```

The BindToArray method also allows you to bind the control to another **VSFlexGrid** control. This way, you may create different "views" of the same data without having to keep duplicate copies of the data. The syntax is the same:

```
fa.BindToArray faSource
```

In this case, the *fa* control will display the data stored in the *faSource* control. Changes to cells in either control will reflect on the other.

When binding to another **VSFlexGrid** control, the fixed cells are bound as well as the scrollable ones. The binding only applies to the data, not to the cell formats.

BottomRow Property

See Also Examples Applies to

Returns the zero-based index of the last row displayed in the control.

Syntax

val& = [*form*!]*vsFlexGrid*.**BottomRow**

Remarks

The bottom row returned may be only partially visible.

You cannot set this property. To scroll the contents of the control through code, set the **TopRow** and **LeftCol** properties instead. Or you may bring a cell into view by reading the **CellTop** property.

Data Type

Long

Cell Property

See Also Examples Applies to

Sets or returns cell properties for an arbitrary range.

Syntax

[*form*!]*vsFlexGrid*.**Cell**(*Setting* As CellPropertySettings, [*R1* As Long], [*C1* As Long], [*R2* As Long], [*C2* As Long]) [= *Value*]

Remarks

The **Cell** property allows allows you to read or set cell properties directly to individual cells or ranges (without selecting them).

The parameters for the **Cell** property are described below:

Setting As CellPropertySettings
This parameter determines which cell property will be read or set. The settings available are:

Constant	**Gets or Sets which cell property**
flexcpText	Text (or clip string for selections).
flexcpValue	Numerical value of the cell's text (read-only)
flexcpTextDisplay	Formatted text (read only)
flexcpData	User-defined Variant attached to cell
flexcpFont	Entire font
*flexcpFont**	Font properties (see **CellFontName** etc)
flexcpTextStyle	Text style (see **CellTextStyle**)
flexcpAlignment	Text alignment (see **CellAlignment**)
flexcpPicture	Cell Picture (see **CellPicture**)
flexcpPictureAlignment	Picture alignment (see **CellPictureAlignment**)
flexcpChecked	Check box (see **CellChecked**)
flexcpBackColor	Back color (see **CellBackColor**)
flexcpForeColor	Fore color (see **CellForeColor**)
flexcpFloodPercent	Flood percent (see **CellFloodPercent**)
flexcpFloodColor	Flood color (see **CellFloodColor**)
flexcpCustomFormat	Whether a cell has custom formatting

Row1 As Long (optional)
The *Row1*, *Col1*, *Row2*, and *Col2* parameters are optional. When reading cell properties, only cell (*Row1, Col1*) is used. When setting, the whole range is affected. The only exception is when you read the *flexcpText* property of a range. In this case, a clip string is returned containing the text in the whole selection.

The default value for *Row1* and *Col1* is the current row and the current column (**Row** and **Col** properties). Thus, if they are not supplied, the current cell is used.

The default value for *Row2* and *Col2* is *Row1* and *Col1*. Thus, if they are not supplied, a single cell is used.

For example:

```
' set the font to bold on cell (1,1)
fa.Cell(flexcpFontBold, 1, 1) = True

' set the font to bold on cells (1,1)-(10,1)
fa.Cell(flexcpFontBold, 1, 1, 10) = True
```

Most of the settings listed above can also be read or set through other properties (e.g. **Text**, **TextArray**, etc). Using the **Cell** property is often more convenient, however, because you it lets you specify the cell range.

A couple of settings are not accessible through other properties and deserve additional comments:

flexcpTextDisplay
This setting allows you to get the formatted contents of the cell, as it is displayed to the user. For example, if a cell contains the string "1234" and the **ColFormat** property is set to "#,###.00", this setting will return "1,234.00".

flexcpData
This settings allows you to attach custom information to individual cells, the same way the **RowData** and **ColData** properties allow you to attach custom information to rows and columns. Note that in **VSFlexGrid Pro 6.0**, these values are Variants, which means you may associate virtually any type of data to a cell, including strings, longs, objects, arrays, etc.

flexcpFont
This setting allows you to assign fonts to cells in one step. This is much more efficient than setting each font property individually. For example, instead of writing:

```
fa.CellFontName = "Arial"
fa.CellFontSize = 8
fa.CellFontBold = True
```

you may write

```
fa.Cell(flexcpFont) = Text1.Font
```

flexcpCustomFormat
This setting returns a Boolean value that indicates whether a cell has any custom formatting associated with it (e.g. back color, font, data, etc). You may also set this to False to clear any custom formatting a cell may have.

Data Type
Variant

CellAlignment Property

See Also Examples Applies to

Returns or sets the alignment of text in the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellAlignment**[= *AlignmentSettings*]

Remarks

Valid settings for the **CellAlignment** property are:

Value	Constant
0	*flexAlignLeftTop*
1	*flexAlignLeftCenter*
2	*flexAlignLeftBottom*
3	*flexAlignCenterTop*
4	*flexAlignCenterCenter*
5	*flexAlignCenterBottom*
6	*flexAlignRightTop*
7	*flexAlignRightCenter*
8	*flexAlignRightBottom*
9	*flexAlignGeneral*

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property. To set the alignment of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

AlignmentSettings (Enumeration)

CellBackColor Property

See Also Examples Applies to

Returns or sets the background color of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellBackColor**[= *colorref&*]

Remarks

Setting this property to zero (black) causes the control to paint the cell using the standard colors (set by the BackColor and BackColorAlternate properties). Therefore, to set this property to black, use RGB(1,1,1) instead of RGB(0,0,0).

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property. To set the back color of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

Color

CellBorder Method

See Also Examples Applies to

Draws a border around and within the selected cells.

Syntax

[*form*!]*vsFlexGrid*.**CellBorder** *Color* As Color, *Left* As Integer, *Top* As Integer, *Right* As Integer, *Bottom* As Integer, *Vertical* As Integer, *Horizontal* As Integer

Remarks

The **CellBorder** method allows you to draw borders around groups of cells. It works on the current selection, so in order to use it, you must start by selecting the group of cells where the border is to be drawn. Then call the **CellBorder** method using the following parameters:

Color As Color
This parameter determines the color of the border.

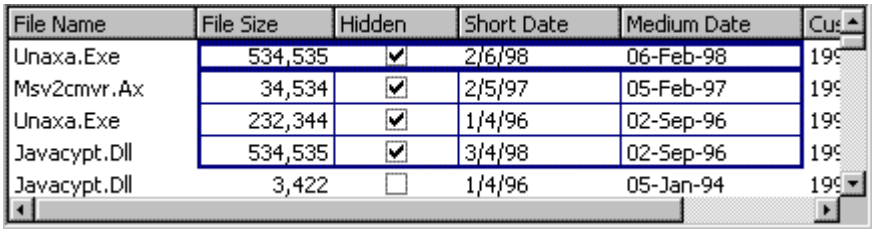
Left, *Top*, *Right*, *Bottom* As Integer
These parameters specify the width, in pixels, of the border to be drawn around the selection. Specify zero to remove the border, or any negative number to preserve the existing border.

Vertical, *Horizontal* As Integer
These parameters specify the width, in pixels, of the borders to be drawn inside the selection in the vertical and horizontal directions. Specify zero to remove the border, or any negative number to preserve the existing border.

For example, the code below draws blue borders around a selected range:

```
Private Sub Form_Load()
  With fa
    ' draw borders around a table
    .Select 1, 1, 4, 4
    .CellBorder RGB(0, 0, 125), 2, 3, 2, 2, 1, 1

    ' apply special formatting to first line of table
    .Select 1, 1, 1, 4
    .CellBorder RGB(0, 0, 125), -1, -1, -1, 3, 0, 0

  End With
End Sub
```

The result looks like this:



File Name	File Size	Hidden	Short Date	Medium Date	Cus
Unaxa.Exe	534,535	☑	2/6/98	06-Feb-98	199
Msv2cmvr.Ax	34,534	☑	2/5/97	05-Feb-97	199
Unaxa.Exe	232,344	☑	1/4/96	02-Sep-96	199
Javacypt.Dll	534,535	☑	3/4/98	02-Sep-96	199
Javacypt.Dll	3,422	☐	1/4/96	05-Jan-94	199

CellButtonClick Event

See Also Examples Applies to

Fired after the user clicks a cell button.

Syntax

Private Sub *vsFlexGrid*_**CellButtonClick**(ByVal *Row* As Long, ByVal *Col* As Long)

Remarks

This event is fired when the user clicks an edit button on a cell. Typically, this event is used to pop up a custom editor for the cell (e.g. dialogs for selecting colors, dates, files, pictures, and so on.).

Edit buttons are displayed on the right side of a cell, with an ellipsis caption ("..."). (They are similar to the buttons displayed in the Visual Basic property window next to picture properties.)

To create an edit button on a cell, you must set the **Editable** property to True and set the **ComboList** (or **ColComboList**) property to an ellipsis.

For example, the following code assigns edit buttons to the first column of a grid, then traps the **CellButtonClick** event to show a color-pick dialog and to assign the selected color to the cell background:

```
Private Sub Form_Load()
  With fa
Private Sub fa_CellButtonClick(ByVal Row As Long, ByVal Col As Long)
  With CommonDialog1
    .ShowColor
    fa.Cell(flexcpBackColor, Row, Col) = .Color
  End With
End Sub
```

CellChecked Property

See Also Examples Applies to

Returns or sets whether a grid cell has a check mark in it.

Syntax

[*form*!]*vsFlexGrid*.**CellChecked**[= *CellCheckedSettings*]

Remarks

Valid settings for the **CellChecked** property are:

Constant	Description
flexNoCheckbox	The cell has no check box. This is the default setting.
flexChecked	The cell has a check box that is checked.
flexUnchecked	The cell has a check box that is not checked.

If the cell has a check box and the **Editable** property is set to True, the user can toggle the check boxes by clicking them with the mouse or by hitting the space or return keys on the keyboard. Either way, the **AfterEdit** event is fired after the toggle so you can take appropriate action.

The check box may appear on the left, right, or center of the cell, depending on the setting of the **CellPictureAlignment** property.

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property. To set check box values of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

CellCheckedSettings (Enumeration)

Default Value

flexNoCheckbox

CellFloodColor Property

See Also Examples Applies to

Returns or sets the color to be used for flooding a cell.

Syntax

[*form*!]*vsFlexGrid*.**CellFloodColor**[= *colorref&*]

Remarks

This property overrides the **FloodColor** property to determine the color to be used for flooding individual cells. For performance reasons, these colors are always mapped to the nearest solid color.

Setting this property to zero (black) causes the control to paint the cell using the standard colors (set by the **FloodColor** property). Thus, to set this property to black, use RGB(1,1,1) instead of RGB(0,0,0).

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the flood color of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

The **Cell Flooding Demo** shows how this property is used.

Data Type

Color

CellFloodPercent Property

See Also Examples Applies to

Returns or sets the percentage of flooding for a cell.

Syntax

[*form*!]*vsFlexGrid*.**CellFloodPercent**[= *value As Integer*]

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** property or **CellFloodColor** property.

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. To set the flood color of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

The **Cell Flooding Demo** shows how this property is used.

Data Type

Integer

CellFontBold Property

See Also Examples Applies to

Returns or sets the Bold attribute of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontBold**[= {**True | False**}]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

Boolean

CellFontItalic Property

See Also Examples Applies to

Returns or sets the Italic attribute of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontItalic**[= {**True | False**}]

Remarks

Changing this property affects the current cell or the current selection, depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

Boolean

CellFontName Property

See Also Examples Applies to

Returns or sets the name of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontName**[= *value As String*]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Setting this property to an empty string resets the cell formatting and causes the default font to be used.

Data Type

String

CellFontSize Property

See Also Examples Applies to

Returns or sets the size of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontSize**[= *value As Single*]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Setting this property to zero resets the cell formatting and causes the default font to be used.

Data Type

Single

CellFontStrikethru Property

See Also Examples Applies to

Returns or sets the Strikethru attribute of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontStrikethru**[= {**True | False**}]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

Boolean

CellFontUnderline Property

See Also Examples Applies to

Returns or sets the Underline attribute of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontUnderline**[= {**True | False**}]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

Boolean

CellFontWidth Property

See Also Examples Applies to

Returns or sets the width of the font of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellFontWidth**[= *value As Single*]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Setting this property to zero causes the default font width to be used.

Data Type

Single

CellForeColor Property

See Also Examples Applies to

Returns or sets the foreground color of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellForeColor**[= *colorref&*]

Remarks

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the font of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Setting this property to zero (black) causes the control to paint the cell using the standard color (set by the **ForeColor** property). Thus, to set this property to black, use RGB(1,1,1) instead of RGB(0,0,0).

Data Type

Color

CellHeight Property

See Also Examples Applies to

Returns the height of the selected cell, in twips. Also brings the cell into view, scrolling if necessary.

Syntax

val& = [*form*!]*vsFlexGrid*.**CellHeight**

Remarks

The **CellHeight** property, **CellWidth** property, **CellTop** property, and **CellLeft** property 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.

Data Type

Long

CellLeft Property

See Also Examples Applies to

Returns the left (x) coordinate of the selected cell relative to the control, in twips. Also brings the cell into view, scrolling if necessary.

Syntax

val& = [*form*!]*vsFlexGrid*.**CellLeft**

Remarks

The **CellHeight** property, **CellWidth** property, **CellTop** property, and **CellLeft** property 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.

Data Type

Long

CellPicture Property

See Also Examples Applies to

Returns or sets the picture displayed in a selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellPicture**[= *Picture*]

Remarks

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

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To assign pictures to an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

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

Data Type

Picture

CellPictureAlignment Property

See Also Examples Applies to

Returns or sets the alignment of the pictures in the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellPictureAlignment**[= *PictureAlignmentSettings*]

Remarks

Valid settings for the **CellPictureAlignment** property are:

Value	Constant
0	*flexPicAlignLeftTop*
1	*flexPicAlignLeftCenter*
2	*flexPicAlignLeftBottom*
3	*flexPicAlignCenterTop*
4	*flexPicAlignCenterCenter*
5	*flexPicAlignCenterBottom*
6	*flexPicAlignRightTop*
7	*flexPicAlignRightCenter*
8	*flexPicAlignRightBottom*
9	*flexPicAlignStretch*
10	*flexPicAlignTile*

This property also governs the alignment of check boxes in the cells (see the **CellChecked** property).

Changing this property affects the current cell or the current selection depending on the setting of the **FillStyle** property. To set the picture alignment of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

PictureAlignmentSettings (Enumeration)

CellTextStyle Property

See Also Examples Applies to

Returns or sets 3D effects for text in a selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**CellTextStyle**[= *TextStyleSettings*]

Remarks

The effect of the settings for the **CellTextStyle** property are described below:

Value	Constant
0	*flexTextFlat*
1	*flexTextRaised*
2	*flexTextInset*
3	*flexTextRaisedLight*
4	*flexTextInsetLight*

Constants *flexTextRaised* and *flexTextInset* work best for large and bold fonts. Constants *flexTextRaisedLight* 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. To set the picture alignment of an arbitrary range of cells (not necessarily the current selection), use the **Cell** property instead.

Data Type

TextStyleSettings (Enumeration)

CellTop Property

See Also Examples Applies to

Returns the top (y) coordinate of the selected cell relative to the control, in twips. Also brings the cell into view, scrolling if necessary.

Syntax

val& = [*form*!]*vsFlexGrid*.**CellTop**

Remarks

The **CellHeight** property, **CellWidth** property, **CellTop** property, and **CellLeft** property 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.

Data Type

Long

CellWidth Property

See Also Examples Applies to

Returns the width of the selected cell, in twips. Also brings the cell into view, scrolling if necessary.

Syntax

val& = [*form*!]*vsFlexGrid*.**CellWidth**

Remarks

The **CellHeight** property, **CellWidth** property, **CellTop** property, and **CellLeft** property 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.

Data Type

Long

ChangeEdit Event

See Also Examples Applies to

Fired after the text in the editor has changed.

Syntax

Private Sub *vsFlexGrid*_**ChangeEdit**()

Remarks

This event is fired while in edit mode, whenever the contents of the editor is modified or a new selection is made from a drop-down list.

Clear Method (vsFlexGrid Object)

See Also Examples Applies to

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

Syntax

[*form*!]*vsFlexGrid*.**Clear** [*Where* As Variant], [*What* As Variant]

Remarks

The parameters for the **Clear** method are described below:

Where (optional)
This parameter specifies what part of the control should be cleared. Valid settings are:

Constant	Description
flexClearEverywhere	Clear everywhere (default)
flexClearScrollable	Clear scrollable region
flexClearSelection	Clear selection

What (optional)
This parameter specifies what should be cleared. Valid settings are:

Constant	Description
flexClearEverything	Clear everything (default)
flexClearText	Clear text only
flexClearFormatting	Clear formatting (including pictures)
flexClearData	Clears all data (RowData, ColData, CellData)

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

ClientHeight Property

See Also Examples Applies to

Returns the height of the control's client area, in twips.

Syntax

val& = [*form*!]*vsFlexGrid*.**ClientHeight**

Remarks

The **ClientHeight** and **ClientWidth** property are useful for setting column widths and row heights proportionally to the size of the control.

Data Type

Long

ClientWidth Property

See Also Examples Applies to

Returns the width of the control's client area, in twips.

Syntax

val& = [*form*!]*vsFlexGrid*.**ClientWidth**

Remarks

The **ClientHeight** and **ClientWidth** properties are useful for setting column widths and row heights proportionally to the size of the control.

The example below 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.

```
' 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

See Also Examples Applies to

Returns or sets the contents of a range.

Syntax

[*form*!]*vsFlexGrid*.**Clip**[= *value As String*]

Remarks

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

When a string is assigned to the **Clip** property, 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 extraneous portion of the clip string is ignored.

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

```
' build clip string
Dim s$
s = "1st" & vbTab & "a" & vbCr
s = s & "2nd" & vbTab & "b"

' paste it over current selection
fa.Clip = s
```

Data Type

String

Col Property

See Also Examples Applies to

Returns or sets the zero-based index of the current column.

Syntax

[*form*!]*vsFlexGrid*.**Col**[= *value As Long*]

Remarks

Use the **Row** and **Col** 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.

Setting the **Row** and **Col** properties automatically resets **RowSel** property and **ColSel** property, 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**. Alternatively, you may use the **Select** method to do it all with a single statement.

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

Data Type

Long

ColAlignment Property

See Also Examples Applies to

Returns or sets the alignment of the given column.

Syntax

[*form*!]*vsFlexGrid*.**ColAlignment**(*Col* As Long)[= *AlignmentSettings*]

Remarks

Valid settings for the **ColAlignment** property are:

Value	Constant
0	*flexAlignLeftTop*
1	*flexAlignLeftCenter*
2	*flexAlignLeftBottom*
3	*flexAlignCenterTop*
4	*flexAlignCenterCenter*
5	*flexAlignCenterBottom*
6	*flexAlignRightTop*
7	*flexAlignRightCenter*
8	*flexAlignRightBottom*
9	*flexAlignGeneral*

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 *Col* is -1 the setting is applied to all columns.

To set the alignment of the fixed parts of a column, use the **FixedAlignment** property. To set individual cell alignments, use the **CellAlignment** or the **Cell** properties. To set column alignments at design time, use the **FormatString** property.

Data Type

AlignmentSettings (Enumeration)

Default Value

flexAlignGeneral (9)

ColComboList Property

See Also Examples Applies to

Returns or sets the list to be used as a drop-down on the specified column.

Syntax

[*form*!]*vsFlexGrid*.**ColComboList**(*Col* As Long)[= *value As String*]

Remarks

This property is similar to the **ComboList** property, except it applies to entire columns. This is often more convenient that using the **ComboList** property because you may set the **ColComboList** property once for each column, whereas the **ComboList** property normally needs to be set in the **BeforeEdit** event.

Another difference is that the **ColComboList** property acts as a data dictionary, allowing you to map numeric values to string entries. The control will hold the numeric values, but will display the associated strings. This mapping is useful for displaying numeric fields that correspond to entries on a list or on a database table.

For example, you may have a column that holds the employee type, which could be one of the following: Full-time, Part-time, Contractor, Intern, or Other.

These values will often come from a database, where they will have a unique entry ID. These should be included in the **ColComboList** string using the following syntax:

```
ColComboList(colEmployeeType) = & _
  "#1;Full time|#23;Part time|#65;Contractor|#78;Intern|#0;Other"
```

After editing, the column will contain the numbers for each entry (i.e. 1 for full-time, 23 for part-time, 65 for contractor etc.). The control will display the full text, however.

This translation is optional. If you omit the entry ID, the control will store the full text.

You may retrieve the number using the **Cell**(*flexcpText*), **Text**, or **TextMatrix** properties. You may retrieve the associated text using the **Cell**(*flexcpTextDisplay*) property. For example:

```
Debug.Print fa.Cell(flexcpText), fa.Cell(flexcpTextDisplay)
23          Part time
```

For more details on list syntax, including multi-column lists, see the **ComboList** property.

Data Type

String

ColData Property

See Also Examples Applies to

Returns or sets a user-defined variant associated with the given column.

Syntax

[*form*!]*vsFlexGrid*.**ColData**(*Col* As Long)[= *value As Variant*]

Remarks

The **RowData** and **ColData** properties allow you to associate values with each row or column on the control.

A typical use for these properties is to keep indices into an array of data structures associated with each row, or pointers to objects represented by the data in the row or column. The values assigned will remain current even if you sort the control or move its columns.

You may also associate values to individual cells using the **Cell** property.

Because these properties hold Variants, you have extreme flexibility in the types of information you may associate with each row, column or cell. The examples below shows some valid uses for these properties.

Store a long that represents a unique ID:

```
fa.RowData(i) = 212
```

Store a string that holds non-numeric information:

```
fa.RowData(i) = "Hello"
```

Store a pointer to another control:

```
fa.RowData(i) = ListBox1
Debug.Print fa.RowData(i).List(0), fa.RowData(i).List(1)
First Item        Second Item
```

Store a pointer to an object:

```
Dim x As Collection
Set x = New Collection
x.Add "Arnold"
x.Add "Billy"
x.Add "Cedric"
fa.RowData(i) = x
Debug.Print fa.RowData(i).Item(2)
  Billy
```

Upgrade Note:

In previous versions, these properties were of type Long. Using Variants means you have more flexibility. Also, previous versions used the **RowData** property in subtotaling and outlining, to store the outline level. This is no longer the case. The outline level is now set with the **RowOutlineLevel** property.

Data Type

Variant

ColDataType Property

See Also Examples Applies to

Returns or sets the data type for the column.

Syntax

[*form*!]*vsFlexGrid*.**ColDataType**(*Col* As Long)[= *DataTypeSettings*]

Remarks

Valid settings for the **ColDataType** property are:

Value	Constant
0	*flexDTEmpty*
2	*flexDTShort*
3	*flexDTLong*
4	*flexDTSingle*
5	*flexDTDouble*
6	*flexDTCurrency*
7	*flexDTDate*
8	*flexDTString*
11	*flexDTBoolean*
20	*flexDTLong8*
30	*flexDTStringC*
31	*flexDTStringW*

This property is automatically set for each column when the control is data bound, so you can determine the data type of each field. When not in bound mode, you may set this property using code.

There are two column types that receive special treatment from the control:

If a column is of type *flexDTDate*, the control takes that into account when sorting the column.

If a column is of type *flexDTBoolean*, the control will display check boxes instead of strings. The mapping between strings and check boxes follows the rules for Variant conversion: any non-zero value and the "True" string are displayed as checked boxes; zero values are displayed as unchecked boxes.

For example:

```
fa.ColDataType(1) = flexDTBoolean
fa.TextMatrix(1, 1) = 1         ' checked
fa.TextMatrix(2, 1) = True      ' checked
fa.TextMatrix(3, 1) = "True"    ' checked
fa.TextMatrix(4, 1) = 0         ' not checked
fa.TextMatrix(5, 1) = "False"   ' not checked
fa.TextMatrix(6, 1) = "foobar"  ' not checked
```

If you want to display custom strings for boolean values instead of check boxes, set the **ColFormat** property to a string containing the values you want to display for True and False values, separated by a semicolon. For example:

```
fa.ColDataType(2) = flexDTBoolean
fa.ColFormat(2) = "Yes;Not Available" ' or "True;False", "On;Off", "Yes;No", etc.
```

Data Type

DataTypeSettings (Enumeration)

Default Value

flexDTEmpty (0)

ColEditMask Property

See Also Examples Applies to

Returns or sets the input mask used to edit cells on the specified column.

Syntax

[*form*!]*vsFlexGrid*.**ColEditMask**(*Col* As Long)[= *value As String*]

Remarks

This property is similar to the **EditMask** property, except it applies to entire columns. This is often more convenient than using the **EditMask** property because you may set the **ColEditMask** property once for each column, whereas the **EditMask** property normally needs to be set in the **BeforeEdit** event.

For more details and syntax documentation, see the **EditMask** property.

Data Type

String

ColFormat Property

See Also Examples Applies to

Returns or sets the format used to display numeric values.

Syntax

[*form*!]*vsFlexGrid*.**ColFormat**(*Col* As Long)[= *value As String*]

Remarks

This property allows you to define a format to be used for displaying numerical, boolean, or date/time values. The syntax for the format string is similar but not identical to the syntax used with Visual Basic's **Format** command.

Formatting Numbers:

The characters used to format numerical values are as follows:

Char	Description
$	A locale-dependent currency sign is prepended to the output.
,	Locale-dependent thousand separators are added to the output.
(	Negative values are displayed enclosed in parentheses.
.	The number of decimals is determined by the number of "0" or "#" characters after the decimal point.
%	The value is multiplied by 100 and followed by a percent sign.
,.	The value is divided by 1000 and displayed with thousand separators.

Formatting Boolean Values:

If a column's **ColDataType** property is set to *flexDTBoolean*, the control will display checkboxes by default. If you want to represent the boolean values in other ways (e.g. True/False, On/Off, Yes/No), then set the **ColFormat** property to a string containing the values you want to display for True and False values, separated by a semicolon. For example:

```
fa.ColDataType(2) = flexDTBoolean
fa.ColFormat(2) = "Yes;Not Available" ' or "True;False", "On;Off", "Yes;No", etc.
```

Formatting Dates and Times:

The characters used to format date/time values is the same as the one used with Visual Basic's **Format** command (including predefined strings such as "Short Date").

The **ColFormat** property does not modify the underlying data, only the way it is displayed. You may retrieve the data using the **Cell**(*flexcpText*), **Text**, or **TextMatrix** properties. You may retrieve the display text using the **Cell**(*flexcpTextDisplay*) property. For example:

```
fa.Cell(flexcpText, 1, 1) = "-12345"

fa.ColFormat(1) = "#,###.00"
Debug.Print fa.Cell(flexcpTextDisplay, 1, 1)
  -12,345.00
fa.ColFormat(1) = "($#,###.00)"
Debug.Print fa.Cell(flexcpTextDisplay, 1, 1)
  ($12,345.00)
fa.ColFormat(1) = "($#,.00)"
Debug.Print  fa.Cell(flexcpTextDisplay, 1, 1)
  ($12.35)

fa.Cell(flexcpText, 1, 1) = "6 Aug 98"
```

```
fa.ColFormat(1) = "Short Date"
Debug.Print  fa.Cell(flexcpTextDisplay, 1, 1)
  8/6/98
fa.ColFormat(1) = "Long Date"
Debug.Print  fa.Cell(flexcpTextDisplay, 1, 1)
  Thursday, August 06, 1998
```

Data Type

String

ColHidden Property

See Also Examples Applies to

Returns or sets whether a column is hidden.

Syntax

[*form*!]*vsFlexGrid*.**ColHidden**(*Col* As Long)[= {**True | False**}]

Remarks

Use the **ColHidden** property to hide and display columns. This is a better approach than setting the column's **ColWidth** property to zero, because it allows you to display the column later with its original width.

Hidden columns are ignored by the **AutoSize** method.

Data Type

Boolean

ColIsVisible Property

See Also Examples Applies to

Returns whether a given column is currently within view.

Syntax

val% = [*form*!]*vsFlexGrid*.**ColIsVisible**(*Col* As Long)

Remarks

The **ColIsVisible** and **RowIsVisible** properties are used to determine whether the specified column or row is within the visible area of the control or whether it has been scrolled off the visible part of the control.

If a column has zero width or is hidden but is within the scrollable area, **ColIsVisible** will return True.

Data Type

Boolean

Collapsed Event

Fired after the user expands or collapses a row group in an outline.

Syntax

Private Sub *vsFlexGrid*_**Collapsed**()

Remarks

This event is fired after the collapsed state of a row or group of rows changes.

This event may be caused by a call to the **Outline** method, by setting the **IsCollapsed** property, or by user interaction with the **OutlineBar**.

See the **Outline Demo** for an example.

ColPos Property

See Also Examples Applies to

Returns the left (x) coordinate of a column relative to the edge of the control, in twips.

Syntax

val& = [*form*!]*vsFlexGrid*.**ColPos**(*Col* As Long)

Remarks

This property is similar to the **CellLeft** property, except **ColPos** applies to an arbitrary column and will not cause the control to scroll. The **CellLeft** property applies to the current selection and reading it will make the current cell visible, scrolling the contents of the control if necessary.

Data Type

Long

ColPosition Property

See Also Examples Applies to

Moves a given column into a new position.

Syntax

[*form*!]*vsFlexGrid*.**ColPosition**(*Col* As Long)[= *NewPosition As Long*]

Remarks

The *Col* and *NewPosition* must be valid column numbers (in the range 0 to **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.

```
Sub fa_Click ()
  Dim col As Long

  ' find out which column was clicked
  col = fa.MouseCol

  ' move it all the way to the left
  fa.ColPosition(col) = fa.FixedCols
End Sub
```

The **ColPosition** property gives you programmatic control over the column order. You may also use the **ExplorerBar** property to allow users to move columns with the mouse.

Data Type

Long

Cols Property

See Also Examples Applies to

Returns or sets the total number of columns in the control.

Syntax

[*form*!]*vsFlexGrid*.**Cols**[= *value As Long*]

Remarks

Use the **Rows** and **Cols** properties to get the dimensions of the control or to resize the control dynamically at runtime.

The minimum number of rows and columns is 0. The maximum number is limited by the memory available on your computer.

If the control runs out of memory while trying to add rows, columns, or cell contents, it will trigger a Visual Basic error. To make sure your code works properly when dealing with large controls, you should add error-handling code to your programs.

Data Type

Long

ColSel Property

See Also Examples Applies to

Returns or sets the extent of a range of columns.

Syntax

[*form*!]*vsFlexGrid*.**ColSel**[= *value As Long*]

Remarks

Use the **RowSel** and **ColSel** 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 is 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**.

Note:

Whenever you set the **Row** and **Col** properties, **RowSel** and **ColSel** are automatically reset so the cell with coordinates (Row, Col) becomes the current selection. Therefore, 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 to do it all with a single statement).

Data Type

Long

ColSort Property

See Also Examples Applies to

Returns or sets the sorting order for each column (for use with the Sort property).

Syntax

[*form*!]*vsFlexGrid*.**ColSort**(*Col* As Long)[= *SortSettings*]

Remarks

This property allows you to specify different sorting orders for each column on the grid. The most common settings for this property are *flexSortGenericAscending* and *flexSortGenericDescending*. For a complete list of possible settings, see the **Sort** property.

To perform the sort using the settings assigned to each column, set the **Sort** property to *flexSortUseColSort*.

To sort dates, set the column's **ColDataType** property to *flexDTDate*.

Data Type

SortSettings (Enumeration)

Default Value

flexSortNone (0)

ColWidth Property

See Also Examples Applies to

Returns or sets the width of the specified column in twips.

Syntax

[*form*!]*vsFlexGrid*.**ColWidth**(*Col* As Long)[= *value As Long*]

Remarks

Use this property to set the width of a column at runtime. To set column widths at design time, use the **FormatString** property. To set width limits for all columns, use the **ColWidthMin** and **ColWidthMax** properties.

If *Col* is -1, then the specified width is applied to all columns.

If you specify a width of -1, the column width is reset to its default value, which depends on the control's current font.

To set column widths automatically, based on the contents of the control, use the **AutoSize** method.

If you specify a width of 0, the column becomes invisible. If you want to hide a column, however, consider using the **ColHidden** property instead. This allows you to make the column visible again with the same width it had before it was hidden. Also, hidden columns are ignored by the **AutoSize** method.

Data Type

Long

ColWidthMax Property

See Also Examples Applies to

Returns or sets the maximum column width, in twips.

Syntax

[*form*!]*vsFlexGrid*.**ColWidthMax**[= *value As Long*]

Remarks

Set this property to a non-zero value to set a maximum limit to column widths. This is often useful when you use the **AutoSize** method to automatically set column width to prevent extremely long entries from making columns too wide.

See also the **ColWidthMin**, **RowHeightMax**, and **RowHeightMin** properties.

Data Type

Long

Default Value

0

ColWidthMin Property

See Also Examples Applies to

Returns or sets the minimum column width, in twips.

Syntax

[*form*!]*vsFlexGrid*.**ColWidthMin**[= *value As Long*]

Remarks

Set this property to a non-zero value to set a minimum limit to column widths. This is often useful when you use the **AutoSize** method to automatically set column widths to prevent empty columns from becoming too narrow.

See also the **ColWidthMax**, **RowHeightMax**, and **RowHeightMin** properties.

Data Type

Long

Default Value

0

ComboCount Property

See Also Examples Applies to

Returns the number of items in the editor's combo list.

Syntax

val& = [*form*!]*vsFlexGrid*.**ComboCount**

Remarks

The **ComboCount** property allows you to customize editing when using drop-down or combo lists. It is valid only while the user is editing a value using a list.

For example, the code below traps the Home key and selects a specific name instead of moving the cursor to the first item on the list. The example also illustrates the use of other related properties, **ComboItem** and **ComboIndex**.

```
Private Sub fa_KeyDownEdit(ByVal Row As Long, _
                           ByVal Col As Long, _
                           KeyCode As Integer, _
                           ByVal Shift As Integer)
  Dim i As Long

  ' make sure we're editing with a list and the home key was pressed
  If Col = 2 And KeyCode = vb Key Home Then

    ' eat the key
    KeyCode = 0

    ' select "Cedric"
    For i = 0 To vsFlexGrid1.ComboCount - 1
      If vsFlexGrid1.ComboItem(i) = "Cedric" Then
        vsFlexGrid1.ComboIndex = i
      End If
    Next
  End If
End Sub
```

Data Type

Long

ComboData Property

See Also Examples Applies to

Returns the long value associated with an item in the editor's combo list.

Syntax

val& = [*form*!]*vsFlexGrid*.**ComboData**([*Index* As Variant])

Remarks

You may assign data values to list items when you define the list, using the **ComboList** or **ColComboList** properties.

Assigning data values to list items serves two purposes:

1) If you do it using the **ColComboList** property, the control stores the data value instead of the string. See the **ColComboList** property for details.

2) If you do it using the **ComboList** property, the control does not perform any mapping. In this case, the value is available for use by the programmer, for example to store an index into an array or a database record ID.

Data Type

Long

Default Value

-1

ComboIndex Property

See Also Examples Applies to

Returns or sets the zero-based index of the current selection in the editor's combo list.

Syntax

[*form*!]*vsFlexGrid*.**ComboIndex**[= *value As Long*]

Remarks

The **ComboIndex** property allows you to customize editing when using drop-down or combo lists. It is valid only while the user is editing a value using a list.

See the **ComboCount** property for an example.

Data Type

Long

ComboItem Property

See Also Examples Applies to

Returns the string associated with an item in the editor's combo list.

Syntax

val$ = [*form*!]*vsFlexGrid*.**ComboItem**([*Index* As Variant])

Remarks

The **ComboItem** property allows you to customize editing when using drop-down or combo lists. It is valid only while the user is editing a value using a list.

See the **ComboCount** property for an example.

Data Type

String

ComboList Property

See Also Examples Applies to

Returns or sets the list to be used as a drop-down when editing a cell.

Syntax

[*form*!]*vsFlexGrid*.**ComboList**[= *value As String*]

Remarks

The **ComboList** property controls the type of editor to be used when editing a cell. You may use a text box, drop-down list, drop-down combo, or an edit button to pop up custom editor forms.

To use the **ComboList** property, set the **Editable** property to True, and respond to the **BeforeEdit** event by setting the **ComboList** property to a string containing the proper options, described below.

Editing Options

To edit the cell using a regular text box, set the **ComboList** property to an empty string (""). You may also define an edit mask using the **EditMask** property.

To edit the cell using a drop-down list, set the **ComboList** property to a string containing the available options, separated by pipe characters ("|"). For example:

```
ComboList = "ListItem 1|ListItem 2".
```

To edit the cell using a drop-down combo, set the **ComboList** property to a string containing the available options, separated by pipe characters ("|") and starting with a pipe character. For example:

```
ComboList = "|ComboItem 1|ComboItem 2".
```

You can also use edit masks with drop-down combos using the **EditMask** property.

To display an edit button, set the **ComboList** property to a string containing an ellipsis (...). Edit buttons look like regular push buttons, aligned to the right of the cell, with an ellipsis as a caption. When the user clicks on the edit button, the control fires the **CellButtonClick** event. For example:

```
ComboList = "..."
```

List Syntax

In addition to the basic list syntax described above, you may create lists that define multi-column drop-downs and translated lists (lists where each item has an associated numerical value).

To define multi-column lists, separate columns with tab characters (Chr(9), or vbTab). When you define a multi-column combo, only one column is displayed in the cell (the others are visible only on the drop-down list). By default, the first column is the one that is displayed in the cell. To display a different column instead, add a string with the format `"*nnn;"` to the first item, where `nnn` is the zero-based index of the column to be displayed.

To create a translated list, attach a numerical value to each list item by adding a string with format `"#xxx;"` to the beginning of the row, where `xxx` is the numerical value. This value may be read while editing the cell using the **ComboData** property.

For example:

```
s = "|#10*1;Getz"   & vbTab & "Stan"  & vbTab & "1 Sansome" & vbTab & "972-4323" & _
    "|#20;Mindelis" & vbTab & "Nuno"  & vbTab & "2 5th"     & vbTab & "972-2321" & _
    "|#30;Davis"    & vbTab & "Miles" & vbTab & "1 High"    & vbTab & "345-2342" & _
    "|#40;Johnson"  & vbTab & "Bob"   & vbTab & "5 Hemlock" & vbTab & "342-2321"
fa.ComboList = s
```

The code above will display a drop-down combo with four columns. The items will have associated data values 10, 20, 30, and 40. The value displayed in the cells will be the one in column 1 (first name). Because the first character is a pipe, the box will be a drop-down combo, as opposed to a drop-down list box.

What is the difference between ComboList and ColComboList?

The **ComboList** and **ColComboList** properties are closely related. They have the same function, and the syntax used to define the lists is exactly the same. There are two differences:

The **ColComboList** property applies to an entire column. It may be set once, when the control is loaded, and then you can forget about it. The **ComboList** property applies to the current cell only. To use it, you need to trap the **BeforeEdit** event and set **ComboList** to the list that is applicable to the call about to be edited.

The **ColComboList** property performs data translation. If data values are supplied, they are stored on the grid, not the actual string. The **ComboList** property does not perform this translation.

If all cells in a column are items picked from the same list, as is the case in most database applications, use the **ColComboList** property. You will not need to handle the **BeforeEdit** event and your code will be cleaner and more efficient. Also, you have the option of using data translation, which simplifies the code and increases data integrity.

If different cells in the same column have different lists, as for example in a property window, then you should use the **ColComboList** property instead. You will need to trap the **BeforeEdit** event and you will have the automatic value translation.

Data Type

String

Compare Event

See Also Examples Applies to

Fired when the Sort property is set to flexSortCustom, to allow custom comparison of rows.

Syntax

Private Sub *vsFlexGrid*_**Compare**(ByVal *Row1* As Long, ByVal *Row2* As Long, *Cmp* As Integer)

Remarks

When the **Sort** property is set to *flexSortCustom*, this event is fired several times, to compare pairs of rows. The event handler should compare rows *Row1* and *Row2* and return the result in the *Cmp* parameter:

Value	Description
-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 your data sets are small.

Usually, there are good alternatives to a custom sort:

If you are sorting dates, set the **ColDataType** property to *flexDTDate* and the generic sorting settings will sort the dates correctly.

If you are sorting international strings, the generic and string settings will sort the value correctly.

If you want to sort based on arbitrary criteria (e.g. "Urgent", "High", "Medium", "Low"), use a hidden column with numerical values that correspond to the criteria you are using.

DataMember Property

See Also Examples Applies to

Returns or sets the data member.

Syntax

[*form*!]*vsFlexGrid*.**DataMember**[= *value As String*]

Remarks

This property is available only in the ADO version of the **VSFlexGrid** control. The DAO version is provided for compatibility with older version of the control. The ADO version supports OLEDB, the new Microsoft standard for database connectivity.

The **DataMember** property is used when the **DataSource** property is set to a source defined with the Visual Basic Data Environment. It contains the name of the data member to retrieve from the object referenced by the **DataSource** property.

The Data Environment maintains collections of data (data sources) containing named objects (data members) that will be represented as Recordset objects. The **DataMember** property determines which object specified by the **DataSource** property will be bound to the control.

Note that if you are binding the control to a data control, you don't need to set this property. Data controls contain only one data member which is used by default.

See also the **DataSource** and **DataMode** properties.

Data Type

String

DataMode Property

See Also Examples Applies to

Returns or sets the type of data binding used by the control when it is connected to a data source (read-only or read/write).

Syntax

[*form*!]*vsFlexGrid*.**DataMode**[= *DataModeSettings*]

Remarks

Valid settings for the **DataMode** property are:

Value	Constant
0	*flexDMFree*
1	*flexDMBound*

The effect of the settings for the DataMode property are described below:

flexDMFree
This setting causes the data to be read from the database when the program starts, when the data source is refreshed, and when the user calls the **DataRefresh** method. Any direct changes to the database (edits and cursor movements) are ignored by the control. The *flexDMFree* setting is equivalent to the data binding implemented in the MSFlexGrid control.

flexDMBound
This setting causes the data in the database to be permanently synchronized with the control. The current row is linked to the database cursor, so when the **Row** property changes, the database cursor moves and vice-versa. All edits to the control contents are updated in the database and vice-versa. The *flexDMBound* setting is similar to the data binding implemented in the Microsoft DBGrid control.

When the **DataMode** property is set to *flexDMBound*, some properties and methods are disabled or their behavior is restricted:

AddItem
The second parameter of the **AddItem** method, the position where the new row should be inserted, is ignored. New rows are always appeded at the bottom of the database.

Rows, **Cols**
These properties become read-only. You may add or remove records from the database one at a time using the **AddItem** and **RemoveItem** methods.

FixedRows, **FixedCols**
These properties become read-only at runtime. You need to decide how many fixed rows and columns you want at design time.

Sort, **RowPosition**
These properties are disabled. You may sort the database records by modifying the SQL statement in the data source.

IsSubtotal
This property becomes read-only. You may add or clear subtotals using the **Subtotal** method.

Data Type

DataModeSettings (Enumeration)

Default Value

flexDMFree (0)

DataRefresh Method

See Also Examples Applies to

Forces the control to re-fetch all data from its data source.

Syntax

[*form*!]*vsFlexGrid*.**DataRefresh**

Remarks

If you trap the **BeforeDataRefresh** event and refuse to load new data from the database, you may later want to force a data refresh by using this method.

DataSource Property

See Also Examples Applies to

Returns or sets the data source.

Syntax

[*form*!]*vsFlexGrid*.**DataSource**[= *DataSource*]

Remarks

This property behaves differently in the DAO and ADO versions of the **VSFlexGrid** control. The DAO version is provided for compatibility with older version of the control. The ADO version supports OLEDB, the new Microsoft standard for database connectivity.

DAO version (VSFlex6d.ocx)

This property can only be set at design time. Use Visual Basic's properties window to set the **DataSource** property to a **Data** control already on the form. Once this property is set, the contents of the grid will be updated whenever the associated **Data** control is refreshed or when the **DataRefresh** method is called.

OLEDB/ADO version (VSFlex6.ocx)

The *DataSource* parameter is a reference to an object that qualifies as a data source, including ADO Recordset objects and classes or user controls defined as data sources.

You may set the **DataSource** property at design time using the properties window. When you select the **DataSource** property, you will get a drop-down list enumerating the sources available. These include sources defined with Visual Basic's Data Environment as well as any controls defined as data sources, such as the Microsoft ADO data control.

You may also set the **DataSource** property at runtime using the Visual Basic **Set** statement, as shown below:

```
' ADODC1 is a Microsoft ADO Data control
Set fa.DataSource = ADODC1
```

See also the **DataMember** and **DataMode** properties.

Data Type

DataSource

DrawCell Event

See Also Examples Applies to

Fired when the OwnerDraw property is set to allow custom cell drawing.

Syntax

Private Sub *vsFlexGrid*_**DrawCell**(ByVal *hDC* As Long, ByVal *Row* As Long, ByVal *Col* As Long, ByVal *Left* As Long, ByVal *Top* As Long, ByVal *Right* As Long, ByVal *Bottom* As Long, *Done* As Boolean)

Remarks

The **DrawCell** event is fired before the contents of a cell are painted.

The **DrawCell** event is fired only if the **OwnerDraw** property is set to a non-zero value, to allow for custom drawing on selected cells.

The parameters for the **DrawCell** event are described below:

hDC As Long
This parameter contains a handle to the control's device context. The *hDC* parameter is required by all Windows GDI calls.

Row, *Col* As Long
These parameters define the cell that is about to be drawn.

Left, *Top*, *Right*, *Bottom* As Long
These parameters define the rectangle that contains the cell. The coordinates are given in pixels, so they can be used directly in the GDI calls.

Done As Boolean
This parameter should be set to True to indicate that the event did, in fact, handle the drawing. Set it to False to indicate that you don't want to paint this particular cell and the control should handle it instead.

Note:
Owner-drawn cells are a fairly advanced feature that requires knowledge of the Windows GDI calls. If you decide to use this feature, our technical support technicians will probably not be able to help you with problems you may encounter. Efficient painting is also fundamental to the perceived speed of your application, so use this feature only if you really need it, and make sure your own painting code is as fast as possible.

For an example, refer to the **OwnerDraw** demo included in the distribution package.

Editable Property

See Also Examples Applies to

Returns or sets whether the control allows in-cell editing.

Syntax

[*form*!]*vsFlexGrid*.**Editable**[= {**True | False**}]

Remarks

If the **Editable** property is set to True, the control provides in-cell editing.

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.

You may choose to use a regular edit box, drop-down list or drop-down combo, depending on the setting of the **ComboList** and **ColComboList** properties. You may also specify an editing mask using the **EditMask** and **ColEditMask** properties. Set these properties in response to the **BeforeEdit** event.

You may perform data validation in response to the **ValidateEdit** event, and perform post-editing work such as resorting the control in response to the **AfterEdit** event.

Data Type

Boolean

Default Value

False

EditCell Method

See Also Examples Applies to

Activates edit mode.

Syntax

[*form*!]*vsFlexGrid*.**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. You may even use it to allow editing of fixed cells.

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

```
Sub fa_MouseDown(Button As Integer, Shift As Integer, X!, Y!)
  If Button = vb Right Button Then
    fa.Select fa.MouseRow, fa.MouseCol
    fa.EditCell
  End If
End Sub
```

EditMask Property

See Also Examples Applies to

Returns or sets the input mask used to edit cells.

Syntax

[*form*!]*vsFlexGrid*.**EditMask**[= *value As String*]

Remarks

The **EditMask** property allows you to specify an input mask for automatic input formatting and validation. The mask syntax is similar to the one used by the Microsoft MaskedEdit control and by Microsoft Access.

Set the **EditMask** property in response to the **BeforeEdit** event, in the same way you would set the **ComboList** property.

If the same mask is used to edit all values in a column, use the **ColEditMask** property instead. This tends to simplify the code because you don't need to trap the **BeforeEdit** event.

When the user is done editing, the **ValidateEdit** event will be fired as usual. The *Cancel* parameter will be set to True if the mask was not filled out properly, so in most cases you don't event need to implement the handler. The default behavior ensures that only valid data will be entered.

The **EditMask** must be a string composed of the following symbols:

1) Wildcards

0	digit
9	digit or space
#	digit or sign
L	letter
?	letter or space
A	letter or digit
a	letter, digit, or space
&	any character

2) Localized characters

.	localized decimal separator
,	localized thousand separator
:	localized time separator
/	localized date separator

3) Command characters

\	next character is taken as a literal (not a special character)
>	translate letters to uppercase
<	translate letters to lowercase
;	group delimiter (see below)

The group delimiter character is used to control additional options. If present in the mask string, then the part of the mask to the left of the first delimiter is used as the actual mask. The part to the right is interpreted in this way:

1 - if a lowercase 'q' is present, the control edits in 'quiet' mode (no beeps on invalid characters),
2 - the last character is used as a placeholder (instead of the default underscore).

For example:

```
' set the mask so the user can enter a phone number,
' with optional area code, and a state in capitals.
' this will beep on invalid keys.
fa.EditMask = "(###) 000-0000 St\ate\: >LL"

' similar mask, but in quiet mode (no beep for wrong keys)
```

```
' and with an asterisk instead of underscore for a placeholder:
fa.EditMask = "(###) 000-0000 St\ate\: >LL;q;*"
```

Here are some commented examples:

`"St\ate\; >LL"`
Is a valid format. The 'a' and ';' characters are escaped and thus taken as literals. The '>' is used to ensure that the next two characters will be represented in uppercase.

`"St\ate\; >LL;q;*"`
Is a valid format. It is similar to the previous example, but the 'q' after the delimiter puts the control in quiet mode. An asterisk '*' is used as placeholder instead of the underscore, because that is the last character after the delimiter.

`"St; >LL"`
This is an invalid format. The mask itself is just "St" (the part to the left of the ';' delimiter. There are no wildcards, so the user can't type anything. If he could, the placeholder character would be "L" (last character after the ';' delimiter).

`"; >LL"`
This is an invalid format. The first character is a delimiter, so there is no real mask at all.

Data Type
String

EditMaxLength Property

See Also Examples Applies to

Returns or sets the maximum number of characters that can be entered in the editor.

Syntax

[*form*!]*vsFlexGrid*.**EditMaxLength**[= *value As Long*]

Remarks

Set this property in the **BeforeEdit** event to limit the length of the text that may be entred while editing a cell.

Setting **EditMaxLength** to 0 allows editing of strings up to about 32k characters.

Changing this property while editing a cell does not affect the contents of the editor but will affect subsequent editing.

Data Type

Long

Default Value

0

EditSelLength Property

See Also Examples Applies to

Returns or the number of characters selected in the editor.

Syntax

[*form*!]*vsFlexGrid*.**EditSelLength**[= *value As Long*]

Remarks

This property works in conjunction with the **EditSelStart** and **EditSelText** properties, while the control is in cell-editing mode.

Use these properties for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in the editor, or clearing text. Used in conjunction with the Visual Basic **Clipboard** object, these properties are useful for copy, cut, and paste operations.

When working with these properties, note that:

1) Setting **SelLength** less than 0 causes a runtime error.
2) Setting **SelStart** greater than the text length sets the property to the existing text length; changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
3) Setting **SelText** to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Data Type

Long

EditSelStart Property

See Also Examples Applies to

Returns or sets the starting point of text selected in the editor.

Syntax

[*form*!]*vsFlexGrid*.**EditSelStart**[= *value As Long*]

Remarks

This property works in conjunction with the **EditSelLength** and **EditSelText** properties, while the control is in cell-editing mode.

Use these properties for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in the editor, or clearing text. Used in conjunction with the Visual Basic **Clipboard** object, these properties are useful for copy, cut, and paste operations.

When working with these properties, note that:

1) Setting **SelLength** less than 0 causes a runtime error.
2) Setting **SelStart** greater than the text length sets the property to the existing text length; changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
3) Setting **SelText** to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Data Type

Long

EditSelText Property

See Also Examples Applies to

Returns or sets the string containing the current selection in the editor.

Syntax

[*form*!]*vsFlexGrid*.**EditSelText**[= *value As String*]

Remarks

This property works in conjunction with the **EditSelStart** and **EditSelLength** properties, while the control is in cell-editing mode.

Use these properties for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in the editor, or clearing text. Used in conjunction with the Visual Basic **Clipboard** object, these properties are useful for copy, cut, and paste operations.

When working with these properties, note that:

1) Setting **SelLength** less than 0 causes a runtime error.
2) Setting **SelStart** greater than the text length sets the property to the existing text length; changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
3) Setting **SelText** to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Data Type

String

EditText Property

See Also Examples Applies to

Returns or sets the text in the cell editor.

Syntax

[*form*!]*vsFlexGrid*.**EditText**[= *value As String*]

Remarks

The **EditText** property allows you to read and modify the contents of the cell editor while it is active.

This property is useful mainly for handling the **ValidateEdit** event.

Data Type

String

Ellipsis Property

See Also Examples Applies to

Returns or sets whether the control will display ellipsis (...) after long strings.

Syntax

[*form*!]*vsFlexGrid*.**Ellipsis**[= *EllipsisSettings*]

Remarks

The **Ellipsis** property determines how the control displays strings that are too long to fit the available space in a cell. By setting this property to a non-zero value, you can force the display of an ellipsis symbol ("...") to indicate that part of the string has been truncated.

The effect of the settings for the **Ellipsis** property are described below:

Constant	Description
flexNoEllipsis	Text is truncated, no ellipsis characters are displayed.
flexEllipsisEnd	Ellipsis characters displayed at the end of the string.
flexEllipsisPath	Disk path-style ellipsis, appears in the middle of the string.

Data Type

EllipsisSettings (Enumeration)

Default Value

flexNoEllipsis (0)

EnterCell Event

See Also Examples Applies to

Fired when a cell becomes active.

Syntax

Private Sub *vsFlexGrid*_**EnterCell**()

Remarks

This event is fired after a cell becomes current, either as a result of mouse/keyboard action, or when the current selection is modified programatically.

Error Event (vsFlexGrid Object)

See Also Examples Applies to

Fired after a data-access error.

Syntax

Private Sub *vsFlexGrid*_**Error**(ByVal *ErrorCode* As Long, *ShowMsgBox* As Boolean)

Remarks

This event is fired after a non-fatal data-access error. Normally, this error indicates that an update to the database failed because of the data was of the wrong type or because the value entered would violate database integrity rules.

If you do not handle this event, the control will display a message box informing the user that an error occurred. Execution will continue normally and the control will display the value as retrieved from the database.

You may trap this event to suppress the dialog box, perhaps replacing it with a custom one.

ExplorerBar Property

See Also Examples Applies to

Returns or sets whether column headers are used to sort and/or move columns.

Syntax

[*form*!]*vsFlexGrid*.**ExplorerBar**[= *ExplorerBarSettings*]

Remarks

The **ExplorerBar** property allows users to use column headings to sort and move columns without any code.

The effect of the settings for the **ExplorerBar** property are described below:

Constant	Description
flexExNone	No **ExplorerBar**. Fixed rows behave as usual.
flexExSort	Users may sort columns by clicking on their headings.
flexExMove	Users may move columns by dragging their headings.
flexExSortAndMove	Users may sort and move columns.

By default, the **ExplorerBar** works like the one in Microsoft's Internet Explorer 4: One click sorts the column in ascending order, the next in descending order. Any non-fixed column may be dragged to any non-fixed position.

The control fires events that allow you to customize this behavior. The events are **BeforeSort**, **AfterSort**, **BeforeMoveColumn**, and **AfterMoveColumn**.

You must have at least one fixed row to be able to use the **ExporerBar**.

Data Type

ExplorerBarSettings (Enumeration)

Default Value

flexExNone (0)

ExtendLastCol Property

See Also Examples Applies to

Returns or sets whether the last column should be adjusted to fit the control's width.

Syntax

[*form*!]*vsFlexGrid*.**ExtendLastCol**[= {**True | False**}]

Remarks

This property only affects painting. It does not modify the **ColWidth** property of the last column.

Data Type

Boolean

Default Value

False

FillStyle Property

See Also Examples Applies to

Returns or sets whether changes to the Text or format properties apply to the current cell or to the entire selection.

Syntax

[*form*!]*vsFlexGrid*.**FillStyle**[= *FillStyleSettings*]

Remarks

The settings for the **FillStyle** property are described below:

flexFillSingle
Setting the **Text** property or any of the cell formatting properties affects the current cell only.

flexFillRepeat
Setting the **Text** property or any of the cell formatting properties affects the entire 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.

FillStyle is ignored if SelectionMode is *flexSelectionListBox*.

Data Type

FillStyleSettings (Enumeration)

Default Value

flexFillSingle (0)

FindRow Property

See Also Examples Applies to

Returns the index of a row that contains a specified string or RowData value.

Syntax

val& = [*form*!]*vsFlexGrid*.**FindRow**(*Item* As Variant, [*Row* As Variant], [*Col* As Variant])

Remarks

The **FindRow** method allows you to look up specific rows based on their **RowData** values. You can also search rows based on the cell data values for a specific column. The search is much faster and more convenient than a Visual Basic loop.

The parameters for the **FindRow** property are described below:

Item As Variant
This parameter contains the data for which you are looking.

Row As Variant (optional)
This parameter contains the rows where the search should start. The default value is **FixedRows**.

Col As Variant (optional)
This parameter tells the control which column should be searched. By default, this value is set to -1, which means the control will look for matches against **RowData**. If *Col* is set to a value greater than -1, then the control will look for matches against the cell's data values for the given column.

If you assign a unique value to a row's **RowData** property, you can later find it quickly and easily using the **FindRow** method.

The example below shows how this method is used:

```
' assign some data to row 30 and cell (30, 5)
fa.RowData(30)="MyRow"
fa.Cell(flexcpData, 40, 5)="MyCell"

' locate a row based on its RowData value
Debug.Print fa.FindRow("MyRow")
  30

' this fails because no rows have RowData = "MyCell"
Debug.Print fa.FindRow("MyCell")
  -1

' locate a row based on cell data for column 5
Debug.Print fa.FindRow("MyCell", , 5)
  40
```

Data Type

Long

FixedAlignment Property

See Also Examples Applies to

Returns or sets the alignment for the fixed rows in a column.

Syntax

[*form*!]*vsFlexGrid*.**FixedAlignment**(*Col* As Long)[= *AlignmentSettings*]

Remarks

The **FixedAlignment** property behaves like the **ColAlignment** property except that it only affects the alignment of fixed cells. You can use this property to align headings differently than the rest of the cells.

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

For a list of valid settings, see the **ColAlignment** property.

Data Type

AlignmentSettings (Enumeration)

Default Value

flexAlignLeftTop (0)

FixedCols Property

See Also Examples Applies to

Returns or sets the total number of fixed (non-scrollable) columns.

Syntax

[*form*!]*vsFlexGrid*.**FixedCols**[= *value As Long*]

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 through. 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 headers or in database applications to show field names.

Setting **FixedCols** to a value exceeding the number of columns will result in a runtime error.

Data Type

Long

Default Value

1

FixedRows Property

See Also Examples Applies to

Returns or sets the total number of fixed (non-scrollable) rows.

Syntax

[*form*!]*vsFlexGrid*.**FixedRows**[= *value As Long*]

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.

Setting **FixedRows** to a value exceeding the number of rows will result in a runtime error.

Data Type

Long

Default Value

1

FloodColor Property

See Also Examples Applies to

Returns or sets the color used to flood cells.

Syntax

[*form*!]*vsFlexGrid*.**FloodColor**[= *colorref&*]

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 maximize performance, this color is always mapped to the nearest solid color.

To control the flooding color of individual cells, set the **Cell**(*flexcpFloodColor*) property.

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

Data Type

Color

FocusRect Property

See Also Examples Applies to

Returns or sets the type of focus rectangle to be displayed around the current cell.

Syntax

[*form*!]*vsFlexGrid*.**FocusRect**[= *FocusRectSettings*]

Remarks

The effect of the settings for the **FocusRect** property are described below:

Constant	Description
flexFocusNone	No focus rectangle is shown.
flexFocusLight	Show one pixel wide focus rectangle.
flexFocusHeavy	Show two pixels wide focus rectangle.
flexFocusSolid	Show solid rectangle (the color is determined by the **BackColorSel** property).
flexFocusRaised	Show raised frame.
flexFocusInset	Show inset frame.

If a focus rectangle is drawn, then the current cell is painted using the regular background color, as in most spreadsheets and grids. Otherwise, the current cell is painted using the selection color (**BackColorSel**).

Data Type

FocusRectSettings (Enumeration)

Default Value

flexFocusLight (1)

ForeColor* Property

See Also Examples Applies to

Returns or sets the foreground color of the non-fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**ForeColor**[= *colorref&*]

Remarks

This property works in conjunction with the **ForeColorFixed**, and **ForeColorSel** properties to specify the color used to draw text.

ForeColor determines the color used to draw text in the scrollable area of the control.

ForeColorFixed determines the color used to draw text in the fixed rows and columns.

ForeColorSel determines the color used to draw text in selected cells.

You may set the text color of individual cells using the **Cell**(*flexCPForeColor*) property.

Data Type

Color

ForeColorFixed Property

See Also Examples Applies to

Returns or sets the foreground color of the fixed rows and columns.

Syntax

[*form*!]*vsFlexGrid*.**ForeColorFixed**[= *colorref&*]

Remarks

This property works in conjunction with the **ForeColor** and **ForeColorSel** properties to specify the color used to draw text.

ForeColor determines the color used to draw text in the scrollable area of the control.

ForeColorFixed determines the color used to draw text in the fixed rows and columns.

ForeColorSel determines the color used to draw text in selected cells.

You may set the text color of individual cells using the **Cell**(*flexCPForeColor*) property.

Data Type

Color

ForeColorSel Property

See Also Examples Applies to

Returns or sets the foreground color of the selected cells.

Syntax

[*form*!]*vsFlexGrid*.**ForeColorSel**[= *colorref&*]

Remarks

This property works in conjunction with the **ForeColor** property and **ForeColorFixed** property to specify the color used to draw text.

ForeColor determines the color used to draw text in the scrollable area of the control.

ForeColorFixed determines the color used to draw text in the fixed rows and columns.

ForeColorSel determines the color used to draw text in selected cells.

You may set the text color of individual cells using the **Cell**(*flexCPForeColor*) property.

Data Type

Color

FormatString Property

See Also Examples Applies to

Assigns column widths, alignments, and fixed row and column text.

Syntax

[*form*!]*vsFlexGrid*.**FormatString**[= *value As String*]

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.

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

See the **FormatString Demo** for some examples.

Data Type

String

GetHeaderRow Event

See Also Examples Applies to

Fired while printing the control to set repeating header rows.

Syntax

Private Sub *vsFlexGrid*_**GetHeaderRow**(ByVal *Row* As Long, *HeaderRow* As Long)

Remarks

This event is fired while the control is being rendered on a page or print preview window using VideoSoft's **VSPrinter** control. If you are not using **VSPrinter** to render the control, you do not need to handle this event at all.

The **GetHeaderRow** event allows you to set a repeating header at the top of each page. While printing, the **GetHeaderRow** event is fired at the beginning of each page (except the first) and you can return the number of a row that should be used as a header on each page. This is especially useful for printing complex reports that require control over page breaks.

The parameters for the **GetHeaderRow** event are described below:

Row As Long
This parameter contains the number of the row that will be the first on a page.

HeaderRow As Long
This parameter is initially set to -1, meaning no heading row is needed. If you want a header row on the page, set *HeaderRow* to the number of a row to be used as the header.

For more details, see the **BeforePageBreak** event and the **RenderControl Demo**.

GetMergedRange Method

See Also Examples Applies to

Returns the range of merged cells that includes a given cell.

Syntax

[*form*!]*vsFlexGrid*.**GetMergedRange** *Row* As Long, *Col* As Long, *R1* As Long, *C1* As Long, *R2* As Long, *C2* As Long

GridColor Property

See Also Examples Applies to

Returns or sets the color used to draw the grid lines between the non-fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**GridColor**[= *colorref&*]

Remarks

The **GridColor** property is ignored when **GridLines** property is set to one of the 3D styles. Raised and inset grid lines are always drawn using the system-defined colors for shades and highlights.

Data Type

Color

GridColorFixed Property

See Also Examples Applies to

Returns or sets the color used to draw the grid lines between the fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**GridColorFixed**[= *colorref&*]

Remarks

The **GridColorFixed** property is ignored when **GridLines** property is set to one of the 3D styles. Raised and inset grid lines are always drawn using the system-defined colors for shades and highlights.

Data Type

Color

GridLines Property

See Also Examples Applies to

Returns or sets the type of lines to be drawn between non-fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**GridLines**[= *GridStyleSettings*]

Remarks

Valid settings for the **GridLines** property are:

Value	Constant	Description
0	*flexGridNone*	No grid lines
1	*flexGridFlat*	Regular grid lines
2	*flexGridInset*	Inset grid lines
3	*flexGridRaised*	Raised grid lines
4	*flexGridFlatHorz*	Regular horizontal grid lines
5	*flexGridInsetHorz*	Inset horizontal grid lines
6	*flexGridRaisedHorz*	Raised horizontal grid lines
7	*flexGridSkipHorz*	Alternating horizontal grid lines
8	*flexGridFlatVert*	Regular vertical grid lines
9	*flexGridInsetVert*	Inset vertical grid lines
10	*flexGridRaisedVert*	Raised vertical grid lines
11	*flexGridSkipVert*	Alternating vertical grid lines
12	*flexGridExplorer*	Button-like 3D edges

The **GridColor** property 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 using the system-defined colors for shades and highlights.

Data Type

GridStyleSettings (Enumeration)

Default Value

flexGridFlat (1)

GridLinesFixed Property

See Also Examples Applies to

Returns or sets the type of lines to be drawn between fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**GridLinesFixed**[= *GridStyleSettings*]

Remarks

Valid settings for the **GridLinesFixed** property are thes same as for the **GridLines** property.

The **GridColorFixed** property determines the color of the grid lines when the **GridLineFixed** property is set to one of the flat styles (*flexGridFlat*, *flexGridFlatHorz*, *flexGridVert*). Raised and inset grid lines are always drawn using the system-defined colors for shades and highlights.

Data Type

GridStyleSettings (Enumeration)

Default Value

flexGridInset (2)

GridLineWidth Property

See Also Examples Applies to

Returns or sets the width of the grid lines, in pixels.

Syntax

[*form*!]*vsFlexGrid*.**GridLineWidth**[= *value As Integer*]

Remarks

The **GridLineWidth** property determines the thickness, in pixels, of the grid lines when the **GridLineWidth** property or **GridLinesFixed** property is set to one of the flat styles (*flexGridFlat*, *flexGridFlatHorz*, *flexGridFlatVert*). Raised and inset grid lines are always one pixel wide.

Data Type

Integer

Default Value

1

HighLight Property

See Also Examples Applies to

Returns or sets whether selected cells will be highlighted.

Syntax

[*form*!]*vsFlexGrid*.**HighLight**[= *ShowSelSettings*]

Remarks

Valid settings for the **HighLight** property are:

Value	Constant
0	*flexHighlightNever*
1	*flexHighlightAlways*
2	*flexHighlightWithFocus*

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.

Highlighting ranges that contain merged cells may lead to non-rectangular shapes being highlighted. If this is undesirable, you may disable it by setting the **HighLight** property to *flexHighlightNever* or by setting the **AllowSelection** property to False.

Data Type

ShowSelSettings (Enumeration)

Default Value

flexHighlightAlways (1)

IsCollapsed Property

See Also Examples Applies to

Returns or sets whether an outline row is collapsed or expanded.

Syntax

[*form*!]*vsFlexGrid*.**IsCollapsed**(*Row* As Long)[= *CollapsedSettings*]

Remarks

The effect of the settings for the **IsCollapsed** property are described below:

Constant	Description
flexOutlineExpanded	Show all subordinate rows
flexOutlineSubtotals	Show only subordinate nodes
flexOutlineCollapsed	Hide all subordinate rows

You may read this property to determine whether a row is visible or has been collapsed and is therefore hidden from view. You may set it to expand or collapse an outline branch programmatically.

When an outline branch is collapsed or expanded, either through code or as a result of a mouse action, the control fires the **Collapsed** event.

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

For more details on creating and using outlines, see the **Outline Demo**.

Data Type

CollapsedSettings (Enumeration)

IsSelected Property

See Also Examples Applies to

Returns or sets whether a row is selected (for listbox-type selections).

Syntax

[*form*!]*vsFlexGrid*.**IsSelected**(*Row* As Long)[= {**True | False**}]

Remarks

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

To implement this type of row selection, you will typically 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** property 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 you write the code to do it).

Data Type

Boolean

IsSubtotal Property

See Also Examples Applies to

Returns or sets whether a row contains subtotals (as opposed to data).

Syntax

[*form*!]*vsFlexGrid*.**IsSubtotal**(*Row* As Long)[= {**True | False**}]

Remarks

This property allows you to determine whether a given row is a regular row or a subtotal row, or to create subtotal rows manually (as opposed to using the **Subtotal** method).

There are two differences between subtotal rows and regular rows:

1) Subtotal rows may be added and removed automatically with the **Subtotal** method.
2) When using the control as an outliner, subtotal rows behave as outline nodes, while regular rows behave as branches.

You may use this property to build custom outlines. This requires three steps:

1) Set the **IsSubtotal** property to True for all outline nodes.
2) Set the **RowOutlineLevel** property for each outline node.
3) Set the **OutlineBar** and **OutlineCol** properties if you want to display an outline tree which the user can use to collapse and expand the outline.

For more details, see the **Outline Demo**.

Data Type

Boolean

KeyDownEdit Event

See Also Examples Applies to

Fired when the user presses a key in cell-editing mode.

Syntax

Private Sub *vsFlexGrid*_**KeyDownEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *KeyCode* As Integer, ByVal *Shift* As Integer)

Remarks

This event is similar to the standard **KeyDown** event, except it is fired while the grid is in edit mode.

The editor has three modes: text, drop-down combo, or drop-down list. The mode used is determined by the **ComboList** and **ColComboList** properties.

While editing with the text editor or with a drop-down combo, you may set or retrieve the contents of the editor using the **EditText** property. You may manipulate the contents of the editor using the **EditSelStart**, **EditSelLength**, and **EditSelText** properties.

While editing with drop-down lists or drop-down combos, you may set or retrieve the contents of the editor using the **ComboItem**, **ComboIndex**, **ComboCount**, and **ComboData** properties.

KeyPressEdit Event

Fired when the user presses a key in cell-editing mode.

Syntax

Private Sub *vsFlexGrid*_**KeyPressEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *KeyAscii* As Integer)

Remarks

This event is similar to the standard **KeyPress** event, except it is fired while the grid is in edit mode.

The editor has three modes: text, drop-down combo, or drop-down list. The mode used is determined by the **ComboList** and **ColComboList** properties.

While editing with the text editor or with a drop-down combo, you may set or retrieve the contents of the editor using the **EditText** property. You may manipulate the contents of the editor using the **EditSelStart**, **EditSelLength**, and **EditSelText** properties.

While editing with drop-down lists or drop-down combos, you may set or retrieve the contents of the editor using the **ComboItem**, **ComboIndex**, **ComboCount**, and **ComboData** properties.

The main use for this event is to filter keys as they are typed while the control is in cell-editing mode. For example, the code below shows how you can convert input to upper-case or restrict data entry to numeric values only.

```
Sub VSFlexGrid_KeyPressEdit(Row As Long, Col As Long, KeyAscii As Integer)
  Select Case Col

    ' column 1 entries are upper case
    ' so use VB's UCase function to convert the character
    Case 1
      KeyAscii = Asc(UCase$(Chr$(KeyAscii)))

    ' column 2 entries are numeric
    ' so set KeyAscii to 0 if it is not a digit
    Case 2
      If KeyAscii < vb Key 0 Or KeyAscii > vb Key 9 Then KeyAscii = 0
  End Select
End Sub
```

Note that you could also restrict the input of non-digits using the **EditMask** or **ColEditMask** properties.

KeyUpEdit Event

See Also Examples Applies to

Fired when the user presses a key in cell-editing mode.

Syntax

Private Sub *vsFlexGrid*_**KeyUpEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *KeyCode* As Integer, ByVal *Shift* As Integer)

Remarks

This event is similar to the standard **KeyUp** event, except it is fired while the grid is in edit mode.

The editor has three modes: text, drop-down combo, or drop-down list. The mode used is determined by the **ComboList** and **ColComboList** properties.

While editing with the text editor or with a drop-down combo, you may set or retrieve the contents of the editor using the **EditText** property. You may manipulate the contents of the editor using the **EditSelStart**, **EditSelLength**, and **EditSelText** properties.

While editing with drop-down lists or drop-down combos, you may set or retrieve the contents of the editor using the **ComboItem**, **ComboIndex**, **ComboCount**, and **ComboData** properties.

LeaveCell Event

See Also Examples Applies to

Fired before the current cell changes to a different cell.

Syntax

Private Sub *vsFlexGrid*_**LeaveCell**()

Remarks

This event is fired before the cursor leaves the current cell, either as a result of mouse/keyboard action, or when the current selection is modified programatically.

LeftCol Property

See Also Examples Applies to

Returns or sets the zero-based index of the leftmost non-fixed column displayed in the control.

Syntax

[*form*!]*vsFlexGrid*.**LeftCol**[= *value As Long*]

Remarks

Setting the **LeftCol** property causes the control to scroll through its contents horizontally so that the given column becomes the leftmost visible column. This is often useful when you want to synchronize two or more controls so that when one of them scrolls, the other scrolls as well.

To scroll vertically, use the **TopRow** property.

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 (no error will occur).

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** property and **Col** property, then bring it into view by reading the **CellTop** property. (You may restore the original selection later, if you wish.)

Data Type

Long

LoadGrid Method

See Also Examples Applies to

Loads grid contents and format from a file.

Syntax

[*form*!]*vsFlexGrid*.**LoadGrid** *FileName* As String, *LoadWhat* As SaveLoadSettings, [*FixedCells* As Variant]

Remarks

This method loads grid from a file previously saved with the **SaveGrid** method, comma-delimited text file (CSV format) such as an Excel text file, or a tab-delimited text file.

The parameters for the **LoadGrid** method are described below:

FileName As String
This parameter contains the name of the file, including the path.

LoadWhat As SaveLoadSettings
This parameter specifies what should be loaded. Valid options are:

Value	Description
flexFileAll	Load all formatting and data available in the file
flexFileData	Load only the data
flexFileFormat	Load only the formatting
flexFileCommaText	Load from comma-delimited text file
flexFileTabText	Load from tab-delimited text file

FixedCells As Variant (optional)
If this parameter is set to False (the default), then values read from a text file are not stored in the fixed cells -- only the scrollable part of the grid is used. If this parameter is set to True, then fixed rows and columns are also used to hold the data.

When loading text files, rows and columns are added to the grid if needed to accommodate the file contents.

MergeCells Property

See Also Examples Applies to

Returns or sets whether cells with the same contents will be merged into a single cell.

Syntax

[*form*!]*vsFlexGrid*.**MergeCells**[= *MergeSettings*]

Remarks

The **MergeCells** property is used in conjunction with the **MergeRow**, **MergeCol**, and **MergeCompare** properties to control whether and how cells are merged for display.

Merging cells allows you to display data in a clear, appealing way because it highlights groups of identical information. It also gives you flexibility to build tables similar to the ones you can create in HTML or using Microsoft Word, both of which support merged cells.

To create tables with merged cells, you must set the **MergeCells** property to a value other than *flexMergeNever*, and then set the **MergeRow** and **MergeCol** properties to True for the rows and columns you wish to merge (this last step is not necessary when using the *flexMergeSpill* mode).

After these properties are set, the control will automatically merge neighboring cells that have the same contents. Whenever the cell contents change, the control updates the merging state.

Valid settings for the **MergeCells** property are:

Value	Constant
0	*flexMergeNever*
1	*flexMergeFree*
2	*flexMergeRestrictRows*
3	*flexMergeRestrictColumns*
4	*flexMergeRestrictAll*
5	*flexMergeFixedOnly*
6	*flexMergeSpill*

The *flexMergeSpill* setting is a little different from the others. It is the only setting that does not require you to set the **MergeCol** and **MergeRow** properties, and that does not merge cells with identical settings. Instead, it allows cells with long entries to spill onto adjacent cells as long as they are empty. This is often useful when creating outlines. You may use a narrow column to hold group titles, which can then spill onto the cells to the right.

The picture below shows an example using the *flexMergeSpill* setting. Notice how some cells with long entries spill onto adjacent empty cells or get truncated if the adjacent cell is not empty:

Product	Associate	Region	Sales
Motor Oil SAE37-666		Lower East Side	
Motor Oil SAE37-666		Upper West Side	
Motor Oil SAE37-666		Lower East Side	
Motor Oil SAE37-666		Upper West Side	
Motor Oil SAI	Paul	Upper West	4,232
Motor Oil SAI	Paula	Upper West	45,342
Drums	Sylvia	Lower East S	45,342
Flutes	Donna	South	45,342
Flutes	John	East	43,432
Flutes	Mike	West	4,543
Flutes	Paul	North	4,543

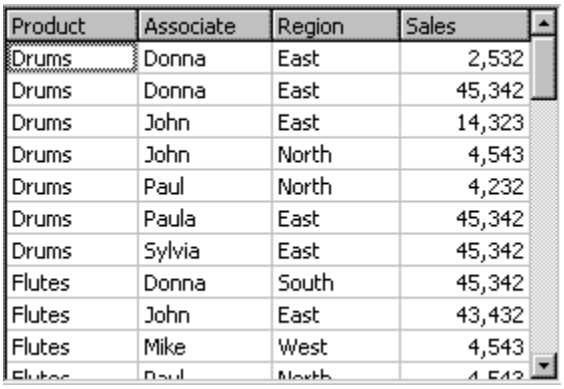
The difference between the *Free* and *Restricted* settings is whether cells with the same contents should always

be merged (*Free* settings) or only when adjacent cells to the left or to the top are also merged.

The *flexMergeFixedOnly* setting is useful if you want to create tables with merged headings, but you don't want the data to be merged.

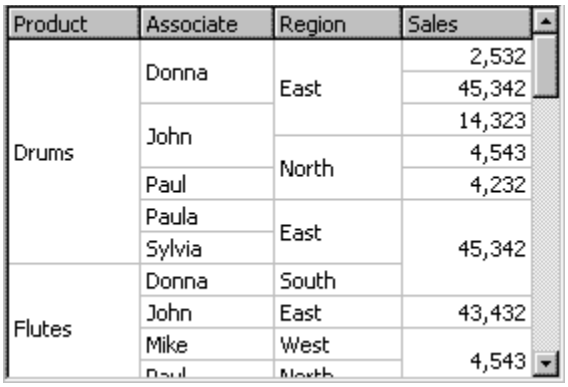
The examples below illustrate the difference.

```
' regular spreadsheet view
With fa
  .MergeCells  = flexMergeNever
  .MergeCol(0) = True: .MergeCol(1) = True: .MergeCol(2) = True
  .MergeCol(3) = False
End With
```



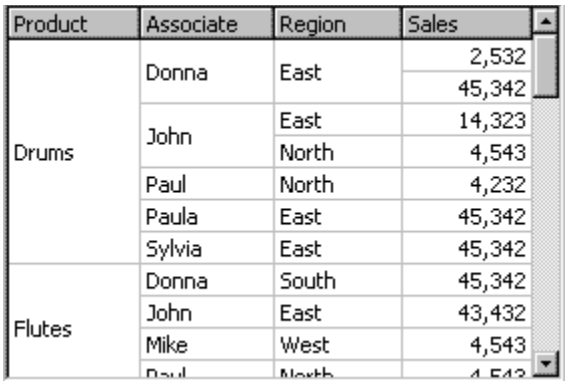
Product	Associate	Region	Sales
Drums	Donna	East	2,532
Drums	Donna	East	45,342
Drums	John	East	14,323
Drums	John	North	4,543
Drums	Paul	North	4,232
Drums	Paula	East	45,342
Drums	Sylvia	East	45,342
Flutes	Donna	South	45,342
Flutes	John	East	43,432
Flutes	Mike	West	4,543

```
' free merging: notice how the first region cell (East) merges
' across employees (Donna and John) to its left.
With fa
  .MergeCells  = flexMergeFree
  .MergeCol(0) = True: .MergeCol(1) = True: .MergeCol(2) = True
  .MergeCol(3) = False
End With
```



Product	Associate	Region	Sales
Drums	Donna	East	2,532
			45,342
	John		14,323
		North	4,543
	Paul		4,232
	Paula	East	45,342
	Sylvia		
Flutes	Donna	South	
	John	East	43,432
	Mike	West	4,543

```
' restricted merging: notice how the first region cell (East)
' no longer merges across employees to its left.
With fa
  .MergeCells  = flexMergeRestrictAll
  .MergeCol(0) = True: .MergeCol(1) = True: .MergeCol(2) = True
  .MergeCol(3) = False
End With
```



Product	Associate	Region	Sales
Drums	Donna	East	2,532
			45,342
	John	East	14,323
		North	4,543
	Paul	North	4,232
	Paula	East	45,342
	Sylvia	East	45,342
Flutes	Donna	South	45,342
	John	East	43,432
	Mike	West	4,543
	Paul	North	4,543

Data Type

MergeSettings (Enumeration)

Default Value

flexMergeNever (0)

MergeCol Property

See Also Examples Applies to

Returns or sets whether a column will have its cells merged (see also the MergeCells property).

Syntax

[*form*!]*vsFlexGrid*.**MergeCol**(*Col* As Long)[= {**True | False**}]

Remarks

The **MergeCol** property is used in conjunction with the **MergeCells**, **MergeRow**, and **MergeCompare** properties to control whether and how cells are merged for display.

The **MergeCells** property is used to enable cell merging for the entire control. After setting it to an appropriate value, the **MergeRow** and **MergeCol** properties are used to determine which rows and columns should have their cells merged.

By default, **MergeRow** and **MergeCol** are set to False, so no merging takes place. If you set them to True for a specific row or column, then adjacent cells in that row or column will be merged if their contents are equal. The rule used to compare cell contents is controlled by the **MergeCompare** property.

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

Data Type

Boolean

MergeCompare Property

See Also Examples Applies to

Returns or sets the type of comparison used when merging cells.

Syntax

[*form*!]*vsFlexGrid*.**MergeCompare**[= *MergeCompareSettings*]

Remarks

The **MergeCompare** property is used in conjunction with the **MergeCells**, **MergeRow**, and **MergeCol** properties to control whether and how cells are merged for display.

Valid setting for the **MergeCompare** property are:

Constant	Description
flexMCExact	Exact match required to merge cells
flexMCNoCase	Case-insensitive matching
flexMCTrimNoCase	Case-insensitive matching, leading and trailing blanks ignored

For more details, see the **MergeCells** property.

Data Type

MergeCompareSettings (Enumeration)

Default Value

flexMCExact

MergeRow Property

See Also Examples Applies to

Returns or sets whether a row will have its cells merged (see also the MergeCells property).

Syntax

[*form*!]*vsFlexGrid*.**MergeRow**(*Row* As Long)[= {**True | False**}]

Remarks

The **MergeRow** property is used in conjunction with the **MergeCells**, **MergeCol**, and **MergeCompare** properties to control whether and how cells are merged for display.

The **MergeCells** property is used to enable cell merging for the entire control. After setting it to an appropriate value, the **MergeRow** and **MergeCol** properties are used to determine which rows and columns should have their cells merged.

By default, **MergeRow** and **MergeCol** are set to False, so no merging takes place. If you set them to True for a specific row or column, then adjacent cells in that row or column will be merged if their contents are equal. The rule used to compare cell contents is controlled by the **MergeCompare** property.

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

Data Type

Boolean

MouseCol Property

See Also Examples Applies to

Returns the zero-based index of the column under the mouse pointer.

Syntax

val& = [*form*!]*vsFlexGrid*.**MouseCol**

Remarks

The **MouseRow** and **MouseCol** properties return the mouse pointer coordinates referenced by rows and columns.

These properties are often useful when handling the **BeforeMouseDown** event, because it is fired before the selection is updated. They are also useful when handling other mouse events that do not change the selection, such as mouse moves or right-button clicks. Finally, they are also good for detecting clicks on the fixed areas of the grid.

Typical uses for these properties include displaying help information or tooltips when the user moves the mouse over a selection, and the implementation of manual drag-and-drop manipulation of OLE objects.

Data Type

Long

MouseRow Property

See Also Examples Applies to

Returns the zero-based index of the row under the mouse pointer.

Syntax

val& = [*form*!]*vsFlexGrid*.**MouseRow**

Remarks

The **MouseRow** and **MouseCol** properties return the mouse pointer coordinates in terms of rows and columns.

These properties are often useful when handling the **BeforeMouseDown** event, because it is fired before the selection is updated. They are also useful when handling other mouse events that do not change the selection, such as mouse moves or right-button clicks. Finally, they are also good for detecting clicks on the fixed areas of the grid.

Typical uses for these properties include displaying help information or tooltips when the user moves the mouse over a selection, and the implementation of manual drag-and-drop manipulation of OLE objects.

Data Type

Long

MultiTotals Property

See Also Examples Applies to

Returns or sets whether subtotals will be displayed in a single row when possible.

Syntax

[*form*!]*vsFlexGrid*.**MultiTotals**[= {**True | False**}]

Remarks

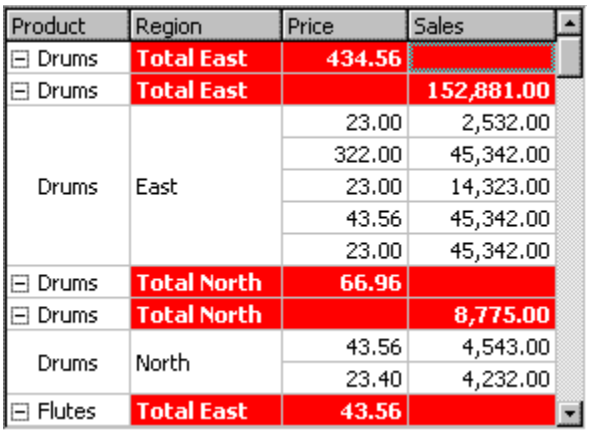
If you set the **MultiTotals** property to True, then subtotal rows created by the **Subtotal** method may contain aggregate values for multiple columns. Otherwise, new subtotal rows are created for each aggreagate value.

The examples below show the difference:

```
With fa
  .MultiTotals = True
  .Subtotal flexSTClear
  .Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
  .Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True
End With
```

Product	Region	Price	Sales
⊟ Drums	**Total East**	**434.56**	**152,881.00**
Drums	East	23.00	2,532.00
		322.00	45,342.00
		23.00	14,323.00
		43.56	45,342.00
		23.00	45,342.00
⊟ Drums	**Total North**	**66.96**	**8,775.00**
Drums	North	43.56	4,543.00
		23.40	4,232.00
⊟ Flutes	**Total East**	**43.56**	**43,432.00**
Flutes	East	43.56	43,432.00
⊟ Flutes	**Total North**	**667.00**	**125,762.00**

```
With fa
  .MultiTotals = False
  .Subtotal flexSTClear
  .Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
  .Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True
End With
```



Product	Region	Price	Sales
Drums	Total East	434.56	
Drums	Total East		152,881.00
Drums	East	23.00	2,532.00
		322.00	45,342.00
		23.00	14,323.00
		43.56	45,342.00
		23.00	45,342.00
Drums	Total North	66.96	
Drums	Total North		8,775.00
Drums	North	43.56	4,543.00
		23.40	4,232.00
Flutes	Total East	43.56	

Data Type

Boolean

Default Value

True

OLECompleteDrag Event

See Also Examples Applies to

Fired after a drop to inform the source component that a drag action was either performed or canceled.

Syntax

Private Sub *vsFlexGrid*_**OLECompleteDrag**(*Effect* As Long)

Remarks

The **OLECompleteDrag** event is the final event to be called in an OLE drag/drop operation. This event informs the source component of the action that was performed when the object was dropped onto the target component. The target sets this value through the effect parameter of the **OLEDragDrop** event. Based on this information, the source can then determine the appropriate action it needs to take. For example, if the object was moved into the target (*vbDropEffectMove*), the source should delete the object from itself after the move.

The parameter for the **OLECompleteDrag** is a long integer set by the source object identifying the action that has been performed, thus allowing the source to take appropriate action if the component was moved (such as the source deleting data if it is moved from one component to another). The possible values are the following:

Constant	Description
vbDropEffectNone	Drop operation was cancelled.
vbDropEffectCopy	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	Drop moves the data from the source to the target. The original data should be deleted.

OLEDrag Method

See Also Examples Applies to

Initiates an OLE drag operation.

Syntax

[*form*!]*vsFlexGrid*.**OLEDrag**

Remarks

When the **OLEDrag** method is called, the control's **OLEStartDrag** event event occurs, allowing it to supply data to a target component.

OLEDragDrop Event

See Also Examples Applies to

Fired when a source component is dropped onto a target component.

Syntax

Private Sub *vsFlexGrid*_**OLEDragDrop**(*Data* As vsDataObject, *Effect* As Long, ByVal *Button* As Integer, ByVal *Shift* As Integer, ByVal *X* As Single, ByVal *Y* As Single)

Remarks

The parameters for the **OLEDragDrop** event are described below:

Data As vsDataObject
An object containing formats that the source will provide and (possibly) the data for those formats. If no data is contained in the object, it is provided when the control calls the **GetData** method. The **SetData** and **Clear** methods cannot be used here.

Effect As Long
A long integer set by the target component identifying the action that has been performed (if any), thus allowing the source to take appropriate action if the component was moved (such as the source deleting the data). The possible values are:

Constant	Description
vbDropEffectNone	Drop operation was cancelled.
vbDropEffectCopy	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	Drop moves the data from the source to the target. The original data should be deleted.

Button As Integer
An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0 (**vbLeftButton**), the right button is bit 1 (**vbRightButton**), and the middle button is bit 2 (**vbMiddleButton**). These bits correspond to the values 1, 2, and 4, respectively. It indicates the state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are depressed.

Shift As Integer
An integer which acts as a bit field corresponding to the state of the SHIFT, CTRL, and ALT keys when they are depressed. The SHIFT key is bit 0, the CTRL key is bit 1, and the ALT key is bit 2. These bits correspond to the values 1 (**vbShiftMask**), 2 (**vbCtrlMask**), and 4 (**vbAltMask**), respectively. The shift parameter indicates the state of these keys; some, all, or none of the bits can be set, indicating that some, all, or none of the keys are depressed. For example, if both the CTRL and ALT keys were depressed, the value of shift would be 6.

X, Y As Single
These parameters specify the current location of the mouse pointer, in twips.

OLEDragMode Property

See Also Examples Applies to

Returns or sets whether the control can act as an OLE drag source, either automatically or under program control.

Syntax

[*form*!]*vsFlexGrid*.**OLEDragMode**[= *OLEDragModeSettings*]

Remarks

Valid settings for the **OleDragMode** property are:

flexOleDragManual
When **OLEDragMode** is set to *flexOleDragManual*, you must call the **OleDrag** method to start dragging, which then triggers the **OLEStartDrag** event.

flexOleDragAutomatic
When **OLEDragMode** is set to *flexOleDragAutomatic*, the control fills a **DataObject** object with the data it contains and sets the effects parameter before initiating the **OLEStartDrag** event when the user attempts to drag out of the control. This gives you control over the drag/drop operation and allows you to intercede by adding or modifying the data that is being dragged.

Note
If the **DragMode** property is set to *Automatic*, the setting of **OLEDragMode** is ignored, because regular Visual Basic drag-and-drop events take precedence.

Data Type

OleDragModeSettings (Enumeration)

Default Value

flexOleDragManual (0)

OLEDragOver Event

See Also Examples Applies to

Fired when a component is dragged over another.

Syntax

Private Sub *vsFlexGrid*_**OLEDragOver**(*Data* As vsDataObject, *Effect* As Long, ByVal *Button* As Integer, ByVal *Shift* As Integer, ByVal *X* As Single, ByVal *Y* As Single, *State* As Integer)

Remarks

The parameters for the **OLEDragOver** event are described below:

Data As vsDataObject
An object containing formats that the source will provide and (possibly) the data for those formats. If no data is contained in the object, it is provided when the control calls the **GetData** method. The **SetData** and **Clear** methods cannot be used here.

Effect As Long
A long integer initially set by the source object identifying all effects it supports. This parameter must be correctly set by the target component during this event. The value of effect is determined by logically Ording together all active effects. The target component should check these effects and other parameters to determine which actions are appropriate for it, and then set this parameter to one of the allowable effects (as specified by the source) to specify which actions will be performed if the user drops the selection on the component. The possible values are:

Constant	Description
vbDropEffectNone	Drop operation was cancelled.
vbDropEffectCopy	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	Drop moves the data from the source to the target. The original data should be deleted.

Button As Integer
An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0, the right button is bit 1, and the middle button is bit 2. These bits correspond to the values 1, 2, and 4, respectively. It indicates the state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are depressed.

Shift As Integer
An integer which acts as a bit field corresponding to the state of the SHIFT, CTRL, and ALT keys when they are depressed. The SHIFT key is bit 0, the CTRL key is bit 1, and the ALT key is bit 2. These bits correspond to the values 1, 2, and 4, respectively. The shift parameter indicates the state of these keys; some, all, or none of the bits can be set, indicating that some, all, or none of the keys are depressed. For example, if both the CTRL and ALT keys are depressed, the value of shift would be 6.

X, Y As Single
These parameters specify the current location of the mouse pointer, in twips.

State As Integer
An integer that corresponds to the transition state of the control being dragged in relation to a target form or control. The possible values are:

Constant	Description
vbEnter	Source component is being dragged within the range of a target.
vbLeave	Source component is being dragged out of the range of a target.
vbOver	Source component has moved from one position in the target to another.

OLEDropMode Property

See Also Examples Applies to

Returns or sets whether the control can act as an OLE drop target, either automatically or under program control.

Syntax

[*form*!]*vsFlexGrid*.**OLEDropMode**[= *OLEDropModeSettings*]

Remarks

The effect of the settings for the **OleDropMode** property are described below:

flexOleDropNone
The control does not accept OLE drops and displays the No Drop cursor.

flexOleDropManual
The target component triggers the OLE drop events, allowing the programmer to handle the OLE drop operation in code.

flexOleDropAutomatic
The control automatically accepts OLE drops if the **DataObject** object contains data in string or file formats.

Data Type

OleDropModeSettings (Enumeration)

Default Value

flexOleDropNone (0)

OLEGiveFeedback Event

See Also Examples Applies to

Fired after every OLEDragOver event to allow the source component to provide visual feedback to the user.

Syntax

Private Sub *vsFlexGrid*_**OLEGiveFeedback**(*Effect* As Long, *DefaultCursors* As Boolean)

Remarks

The parameters for the **OLEGiveFeedback** event are described below:

Effect As Long
A long integer set by the target component in the **OLEDragOver** event specifying the action to be performed if the user drops the selection on it. This allows the source to take the appropriate action (such as giving visual feedback). The possible values are:

Constant	Description
vbDropEffectNone	Drop operation was canceled.
vbDropEffectCopy	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	Drop moves the data from the source to the target. The original data should be deleted.

DefaultCursors As Boolean
A boolean value which determines whether Visual Basic uses the default or a user-defined mouse cursor. If you set this parameter to False, the mouse cursor must be set with the **MousePointer** property of the **Screen** object.

OLESetData Event

See Also Examples Applies to

Fired on the source component when a target component performs the GetData method on the source's DataObject object.

Syntax

Private Sub *vsFlexGrid*_**OLESetData**(*Data* As vsDataObject, *DataFormat* As Integer)

Remarks

In certain cases, you may wish to defer loading data into the **DataObject** object of a source component to save time, especially if the source component supports many formats. This event allows the source to respond to only one request for a given format of data. When this event is called, the source should check the format parameter to determine what needs to be loaded and then perform the **SetData** method on the **DataObject** object to load the data which is then passed back to the target component.

The parameters for the **OLESetData** event are described below:

Data As vsDataObject
An object in which to place the requested data. The component calls the **SetData** method to load the requested format.

DataFormat As Integer
An integer specifying the format of the data that the target component is requesting. The source component uses this value to determine what to load into the **DataObject** object.

OLEStartDrag Event

See Also Examples Applies to

Fired after an OLE drag operation is started (manually or automatically).

Syntax

Private Sub *vsFlexGrid*_**OLEStartDrag**(*Data* As vsDataObject, *AllowedEffects* As Long)

Remarks

This event is fired when the **OleDrag** method is invoked, or when the **OleDragMode** property is set to *flexOleDragAutomatic* and the user initiates an OLE drag/drop operation with the mouse.

This event specifies the data formats and drop effects that the control supports (by default, a string containing the current selection). It can also be used to insert data into the **vsDataObject** object.

The parameters for the **OLEStartDrag** event are described below:

Data As vsDataObject
An object containing formats that the source will provide. You may provide the values for this parameter in this event.

AllowedEffects As Long
A long integer containing the effects that the source component supports. The possible values are:

Constant	Description
vbDropEffectNone	Drop operation was canceled.
vbDropEffectCopy	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	Drop moves the data from the source to the target. The original data should be deleted.

Outline Method

See Also Examples Applies to

Sets an outline level for displaying subtotals.

Syntax

[*form*!]*vsFlexGrid*.**Outline** *Level* As Integer

Remarks

This method collapses or expands an outline to the level specified. If the *Level* parameter 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 the **IsSubtotal** property.

OutlineBar Property

See Also Examples Applies to

Returns or sets the type of outline bar that should be displayed.

Syntax

[*form*!]*vsFlexGrid*.**OutlineBar**[= *OutlineBarSettings*]

Remarks

This property determines whether the control should display an outline bar when it is used as an outliner. The outline bar contains a tree similar to the one in Windows Explorer. It shows the outline's structure and has buttons that can be used to collapse and expand parts of the outline.

Clicking on a collapsed branch (with a plus sign) expands it. Clicking on an expanded branch (with a minus sign) collapses it. Shift and shift-control clicking on a branch expands or collapses the entire outline to the level of the branch that was clicked.

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

By default, the outline bar is drawn on the first column of the control. You may choose to display it in a different column by setting the **OutlineCol** property. The color used to draw the outline tree is specified by the **TreeColor** property.

Valid settings for this property are:

Constant	Description
flexOutlineBarNone	No outline bar
flexOutlineBarComplete	Complete outline tree plus button row on top
flexOutlineBarSimple	Complete outline tree
flexOutlineBarSymbols	Outline symbols (no lines connecting + and - signs)

The *flexOutlineBarComplete* only displays the button row if the outline bar is displayed in a fixed column (i.e. OutlineCol < FixedCols).

Data Type

OutlineBarSettings (Enumeration)

Default Value

flexOutlineBarNone (0)

OutlineCol Property

See Also Examples Applies to

Returns or sets the column used to display the outline tree.

Syntax

[*form*!]*vsFlexGrid*.**OutlineCol**[= *value As Long*]

Remarks

The **OutlineCol** property works in conjunction with the **OutlineBar** property to control the appearance and behavior of the outline tree.

By default, the **OutlineCol** property is set to zero, so the outline bar (if present) is displayed on the first column of the control. You may use **OutlineCol** to place the outline tree in a different column. If you place the outline tree in a column that contains data, the entries will be indented to accommodate the tree.

Typically, you should use the **AutoSize** method after setting this property, to ensure that the tree and data on that column will be fully visible.

Data Type

Long

Default Value

0

OwnerDraw Property

See Also Examples Applies to

Returns or sets whether and when the control will fire the DrawCell event.

Syntax

[*form*!]*vsFlexGrid*.**OwnerDraw**[= *OwnerDrawSettings*]

Remarks

The **OwnerDraw** property allows the application to add custom graphics or text to cells. It determines whether the control should fire the **DrawCell** event to allow the application to perform custom drawing.

The effect of the settings for the **OwnerDraw** property are described below:

flexODNone
This setting indicates that the control should perform all drawing itself. The **DrawCell** event does not get fired at all. This is the default setting.

flexODOver
This setting indicates that the control should draw the cell as normal, then fire the **DrawCell** event so the application can add text or graphics to the default cell contents.

flexODContent
This setting indicates that the control should draw the cell background, including any pictures, but no text. The control will fire the **DrawCell** event so the application can draw the text itself.

flexODComplete
This setting indicates that the control should draw nothing at all in the cell. The control will fire the **DrawCell** event and the application becomes responsible for drawing the entire cell.

flexODOverFixed, flexODContentFixed, flexODCompleteFixed
These settings are similar to the ones described above, but they indicate the application will only perform custom drawing on fixed cells. This allows some optimization, because the **DrawCell** event is fired only for the fixed cells.

For more details, see the **DrawCell** event.

Data Type

OwnerDrawSettings (Enumeration)

Default Value

flexODNone (0)

Picture* Property

See Also Examples Applies to

Returns a picture of the entire control.

Syntax

val% = [*form*!]*vsFlexGrid*.**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 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 example below shows a routine that creates a picture of the the current selection. It traps out-of-memory errors and automatically switches to monochrome mode if required.

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

  ' save current settings
  With fa
    hl = .HighLight
    tr = fa.TopRow
    lc = fa.LeftCol

    ' hide non-selected rows and columns
    .HighLight = 0
    For i = .FixedRows To .Rows - 1
      If i < .Row Or i > .RowSel Then .RowHidden(i) = True
    Next
    For i = .FixedCols To .Cols - 1
      If i < .Col Or i > .ColSel Then .ColHidden(i) = True
    Next

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

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

    ' restore control
    For i = .FixedRows To .Rows - 1
      If i < .Row Or i > .RowSel Then .RowHidden(i) = False
    Next
    For i = .FixedCols To .Cols - 1
      If i < .Col Or i > .ColSel Then .ColHidden(i) = False
    Next
    .TopRow = tr
    .LeftCol = lc
    .Highlight = hl
  End With
End Sub
```

Data Type
Picture

PicturesOver Property

See Also Examples Applies to

Returns or sets whether text and pictures should be overlaid in cells.

Syntax

[*form*!]*vsFlexGrid*.**PicturesOver**[= {**True | False**}]

Remarks

If the **PicturesOver** property is set to True, pictures and text overlap within cells. This setting is useful for displaying pictures that look like button frames or other elements on which text should be overlaid.

If the **PicturesOver** property is set to False, pictures are drawn next to the text. This setting is useful for displaying icons next to text.

Data Type

Boolean

Default Value

False

PictureType Property

See Also Examples Applies to

Returns or sets the type of picture returned by the Picture property.

Syntax

[*form*!]*vsFlexGrid*.**PictureType**[= *PictureTypeSettings*]

Remarks

The effect of the settings for the **PictureType** property are described below:

flexPictureColor
This setting causes the **Picture** property to generate a color bitmap of the control. This mode creates high quality pictures, but they may be quite large and slow to manipulate. Use this setting if the control has only a few rows, or if it contains pictures that should be rendered faithfully.

flexPictureMonochrome
This setting causes the **Picture** property to generate a monochrome bitmap of the control. This mode creates lower quality pictures which consume less memory and are faster to manipulate. Use this mode if the control is large or if a lower quality picture is acceptable.

For more details and sample code, see the **Picture** property.

Data Type

PictureTypeSettings (Enumeration)

Default Value

flexPictureColor (0)

Redraw Property

See Also Examples Applies to

Enables or disables redrawing of the VSFlexGrid control.

Syntax

[*form*!]*vsFlexGrid*.**Redraw**[= {**True | False**}]

Remarks

Use **Redraw** property to reduce flicker and increase speed while making extensive updates to the contents of the control. Set **Redraw** to False before making the changes, make the changes, then set **Redraw** property back to True.

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

```
Sub UpdateGrid()
  Dim i As Long

  ' avoid flicker
  fa.Redraw = False

  ' update control contents
  fa.Rows = fa.FixedRows
  For i = 0 To NRECORDS - 1
    fa.AddItem RECORD(i)
  Next

  ' show results
  fa.Redraw = True
End Sub
```

Note

Using the **Redraw** property is especially important when adding large numbers of rows to the grid, because each time a row is added, the control needs to recalculate the scroll ranges even if the new row is not visible. By using the **Redraw** property, you may increase speed by an order of magnitude when populating a grid.

Data Type

Boolean

Default Value

True

RemoveItem Method

See Also Examples Applies to

Removes a row from the control.

Syntax

[*form*!]*vsFlexGrid*.**RemoveItem** [*Row* As Variant]

Remarks

The *Row* parameter determines which row should be removed from the control. The parameter is zero-based -- it must be in the range between 0 and **Rows**-1, or an Invalid Index error will be triggered.

RightCol Property

See Also Examples Applies to

Returns the zero-based index of the last column displayed in the control

Syntax

val& = [*form*!]*vsFlexGrid*.**RightCol**

Remarks

The right column returned may be only partially visible.

You cannot set this property. To scroll throguh the contents of the control through code, set the **TopRow** and **LeftCol** properties instead. Or you may bring a cell into view by reading the **CellTop** property.

Data Type

Long

Row Property

See Also Examples Applies to

Returns or sets the zero-based index of the current row.

Syntax

[*form*!]*vsFlexGrid*.**Row**[= *value As Long*]

Remarks

Use the **Row** and **Col** 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.

Setting the **Row** and **Col** properties automatically resets **RowSel** and **ColSel** properties, 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**. Alternatively, you may use the **Select** method to do it all with a single statement.

Data Type

Long

RowColChange Event

See Also Examples Applies to

Fired when the current cell changes to a different cell.

Syntax

Private Sub *vsFlexGrid*_**RowColChange**()

Remarks

This event is fired when the **Row** or **Col** properties change, either as a result of user actions (mouse or keyboard) or through code.

This event is not fired when the selection changes (**RowSel** or **ColSel** properties) but the active cell (**Row**, **Col**) remains the same. In this case, the **SelChange** event is fired instead.

RowData Property

See Also Examples Applies to

Returns or sets a user-defined variant associated with the given row.

Syntax

[*form*!]*vsFlexGrid*.**RowData**(*Row* As Long)[= *value As Variant*]

Remarks

The **RowData** and **ColData** properties allow you to associate values with each row or column on the control.

A typical use for these properties is to keep indices into an array of data structures associated with each row, or pointers to objects represented by the data in the row or column. The values assigned will remain current even if you sort the control or move its columns.

You may also associate values to individual cells using the **Cell** property.

Because these properties hold Variants, you have extreme flexibility in the types of information you may associate with each row, column, or cell. The examples below shows some valid uses for these properties.

Store a long that represents a unique ID:

```
fa.RowData(i) = 212
```

Store a string that holds non-numeric information:

```
fa.RowData(i) = "Hello"
```

Store a pointer to another control:

```
fa.RowData(i) = ListBox1
Debug.Print fa.RowData(i).List(0), fa.RowData(i).List(1)
First Item        Second Item
```

Store a pointer to an object:

```
Dim x As Collection
Set x = New Collection
x.Add "Arnold"
x.Add "Billy"
x.Add "Cedric"
fa.RowData(i) = x
Debug.Print fa.RowData(i).Item(2)
  Billy
```

Data Type

Variant

RowHeight Property

See Also Examples Applies to

Returns or sets the height of the specified row in twips.

Syntax

[*form*!]*vsFlexGrid*.**RowHeight**(*Row* As Long)[= *value As Long*]

Remarks

Use this property to set the height of a row at runtime. To set height limits for all rows, use the **RowHeightMin** and **RowHeightMax** properties.

If *Row* is -1, then the specified height is applied to all rows.

If you specify a height of -1, the row height is reset to its default value, which depends on size and type of the control's current font.

To set row heights automatically, based on the contents of the control, use the **AutoSizeMode** property and the **AutoSize** method.

If you specify a height of 0, the column becomes invisible. If you want to hide a row, however, consider using the **RowHidden** property instead. This allows you to make the row visible again with the same height it had before it was hidden. Also, hidden rows are ignored by the **AutoSize** method.

Data Type

Long

RowHeightMax Property

See Also Examples Applies to

Returns or sets the maximum row height, in twips.

Syntax

[*form*!]*vsFlexGrid*.**RowHeightMax**[= *value As Long*]

Remarks

Set this property to a non-zero value to set a maximum limit to row heights. This is often useful when you use the **AutoSize** method to automatically set row heights, to prevent some rows from becoming too large.

See also the **ColWidthMin**, **ColWidthMax**, and **RowHeightMin** properties.

Data Type

Long

Default Value

0

RowHeightMin Property

See Also Examples Applies to

Returns or sets the minimum row height, in twips.

Syntax

[*form*!]*vsFlexGrid*.**RowHeightMin**[= *value As Long*]

Remarks

Set this property to a non-zero value to set a minimum limit to row heights. This is often useful when you use the **AutoSize** method to automatically set row heights, to prevent some rows from becoming too short. This may also be useful when you want to use small fonts, but don't want the rows to become short.

See also the **ColWidthMin**, **ColWidthMax**, and **RowHeightMax** properties.

Data Type

Long

Default Value

0

RowHidden Property

See Also Examples Applies to

Returns or sets whether a row is hidden.

Syntax

[*form*!]*vsFlexGrid*.**RowHidden**(*Row* As Long)[= {**True | False**}]

Remarks

Use the **RowHidden** property to hide and display rows. This is a better approach than setting the row's **RowHeight** property to zero, because you may later show the row without restoring its original height.

When the control collapses or expands an outline branch, either as a result of user mouse action or programmatically (see the **Subtotal** method and **IsCollapsed** property), it sets the **RowHidden** property accordingly.

Hidden rows are ignored by the **AutoSize** method.

Data Type

Boolean

RowIsVisible Property

See Also Examples Applies to

Returns whether a given row is currently within view.

Syntax

val% = [*form*!]*vsFlexGrid*.**RowIsVisible**(*Row* As Long)

Remarks

The **ColIsVisible** and **RowIsVisible** properties are used to determine whether the specified column or row is within the visible area of the control or whether it has been scrolled off the visible part of the control.

If a row has zero height or is hidden but is within the scrollable area, **RowIsVisible** will return True.

Data Type

Boolean

RowOutlineLevel Property

See Also Examples Applies to

Returns or sets the outline level for a subtotal row.

Syntax

[*form*!]*vsFlexGrid*.**RowOutlineLevel**(*Row* As Long)[= *value As Integer*]

Remarks

Each subtotal row has a *level* that is used to indicate which column is being grouped. The subtotal level is also used for outlining. When you create subtotals using the **Subtotal** method, the *level* is set automatically based on the *GroupOn* parameter. When you create an outline manually, use the **RowOutlineLevel** property to set the outline level for each subtotal row.

For more details and an example, see the **Subtotal** method.

Data Type

Integer

RowPos Property

See Also Examples Applies to

Returns the top (y) coordinate of a row relative to the edge of the control, in twips.

Syntax

val& = [*form*!]*vsFlexGrid*.**RowPos**(*Row* As Long)

Remarks

This property is similar to the **CellTop** property, except **RowPos** applies to an arbitrary row and will not cause the control to scroll. The **CellTop** property applies to the current selection and reading it will make the current cell visible, scrolling the contents of the control if necessary.

Data Type

Long

RowPosition Property

See Also Examples Applies to

Moves a given row into a new position.

Syntax

[*form*!]*vsFlexGrid*.**RowPosition**(*Row* As Long)[= *NewPosition As Long*]

Remarks

The *Row* and *NewPosition* must be valid row numbers (in the range 0 to **Rows** - 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.

See the **ColPosition** property for an example.

Data Type

Long

Rows Property

See Also Examples Applies to

Returns or sets the total number of rows in the control.

Syntax

[*form*!]*vsFlexGrid*.**Rows**[= *value As Long*]

Remarks

Use the **Rows** and **Cols** properties to get the dimensions of the control or to resize the control dynamically at runtime.

The minimum number of rows and columns is 0. The maximum number is limited by the memory available on your computer.

If the control runs out of memory while trying to add rows, columns, or cell contents, it will trigger a Visual Basic error. To make sure your code works properly when dealing with large controls, you should add error-handling code to your programs.

Data Type

Long

RowSel Property

See Also Examples Applies to

Returns or sets the extent of a range of rows.

Syntax

[*form*!]*vsFlexGrid*.**RowSel**[= *value As Long*]

Remarks

Use the **RowSel** and **ColSel** 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 is 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**.

Note:

Whenever you set the **Row** and **Col** properties, **RowSel** and **ColSel** are automatically reset so the cursor becomes the current selection. Therefore, 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 to do it all with a single statement).

Data Type

Long

RowStatus Property

See Also Examples Applies to

Sets or returns a value that indicates whether a row has been added, deleted, or modified.

Syntax

[*form*!]*vsFlexGrid*.**RowStatus**(*Row* As Long)[= *RowStatusSettings*]

Remarks

The **RowStatus** property is set by the control to reflect the status of the row. This allows you to determine whether a row has just been created, whether it was modified by the program itself, or whether it was edited by the user.

The control automatically assigns the following values to each row:

Constant	Description
flexrsNew	When the row is created.
flexrsUpdated	When the program modifies a row by writing to it.
flexrsModified	When the user modifies a row by editing it.
flexrsDeleted	Not assigned by the control.

Each new action updates the row status and replaces the previous value. For example, if you create a new instance of the control, all rows will have **RowStatus** = *flexrsNew.* If you then assign values to one of the rows, its status will become *flexrsUpdated.* If the user then edits one or more values on this row, the status becomes *flexrsModified.*

The *flexrsDeleted* value is never really assigned to a row, but is the value returned when you ask for a row that does not exist (e.g. **RowState**(-1)).

The **RowStatus** property is read/write, so you may define and assign your own constants to it. If you do so, define your own enumeration and use values above 100 to avoid conflict with the control-defined constants and future values that may be added in future releases of the control.

Data Type

RowStatusSettings (Enumeration)

SaveGrid Method

See Also Examples Applies to

Saves grid contents and format to a file.

Syntax

[*form*!]*vsFlexGrid*.**SaveGrid** *FileName* As String, *SaveWhat* As SaveLoadSettings, [*FixedCells* As Variant]

Remarks

This method saves a grid to a binary or to a text file. The grid may be retrieved later with the **LoadGrid** method. Grids saved to text files may also be read by other programs, such as Microsoft Excel or Microsoft Word.

The parameters for the **SaveGrid** method are described below:

FileName As String
This parameter contains the name of the file, including the path.

SaveWhat As SaveLoadSettings
This parameter specifies what should be saved. Valid options are:

Constant	Description
flexFileAll	Save all formatting and data
flexFileData	Save only the data
flexFileFormat	Save only the formatting
flexFileCommaText	Save data to a comma-delimited text file
flexFileTabText	Save data to a tab-delimited text file

FixedCells As Variant (optional)
If this parameter is set to False (the default), then values in fixed cells are not saved to text files -- only the scrollable part of the grid is saved. If this parameter is set to True, then fixed rows and columns are also saved.

The *flexFileFormat* option saves global formatting only. It does not save any cell-specific information, not even the number of rows and columns. This allows you to use this setting to create formats that can be applied to existing grids even if they have different dimensions.

Because column widths and row heights are related to the number of rows and columns on the grid, they are not saved or restored if you use the *flexFileFormat* option.

The following is a list of the properties that do get saved and restored if you use the *flexFileFormat* option: BackColor, ForeColor, BackColorBkg, BackColorAlternate, BackColorFixed, ForeColorFixed, BackColorSel, ForeColorSel, TreeColor, SheetBorder, GridLines, GridLinesFixed, GridLineWidth, GridColor, GridColorFixed, TextStyle, TextStyleFixed, ScrollBars, SelectionMode, RowHeightMin, MergeCells, SubtotalPosition, OutlineBar, Font, and WordWrap.

Scroll Event

See Also Examples Applies to

Fired after the control scrolls.

Syntax

Private Sub *vsFlexGrid*_**Scroll**()

Remarks

This event is useful to synchronize the scrolling of multiple controls. You may do this by reading the **TopRow** and **LeftCol** properties, then assigning their values to other controls.

ScrollBars Property

See Also Examples Applies to

Returns or sets whether the control will display horizontal or vertical scroll bars.

Syntax

[*form*!]*vsFlexGrid*.**ScrollBars**[= *ScrollBarsSettings*]

Remarks

Valid settings for the **ScrollBars** property are:

Value	Constant
0	*flexScrollBarNone*
1	*flexScrollBarHorizontal*
2	*flexScrollBarVertical*
3	*flexScrollBarBoth*

Scroll bars are displayed only if the contents of the control extend beyond its borders. For example, a horizontal scroll bar appears when the control is not wide enough to display all columns at once.

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

ScrollBarsSettings (Enumeration)

Default Value

flexScrollBarBoth (3)

ScrollTips Property

See Also Examples Applies to

Returns or sets whether tool tips are shown while the user scrolls vertically.

Syntax

[*form*!]*vsFlexGrid*.**ScrollTips**[= {**True | False**}]

Remarks

Use this property to display a tooltip over the vertical scrollbar as the user moves the scroll thumb. This allows the user to see which row will become visible when he releases the scroll thumb.

This feature makes it easy for users to browse and find specific rows on large data sets. This feature is especially useful if the **ScrollTrack** property is set to False, because then the control will not scroll until the thumb track is released.

To implement this feature in your programs, you must do two things:

1) Set the **ScrollTips** property to True
2) Respond to the **BeforeScrollTip** event by setting the **ScrollTipText** property to text that describes the given row.

For example:

```
fa.ScrollTrack = False
fa.ScrollTips = True

Private Sub fa_BeforeScrollTip(ByVal Row As Long)

  ' the ScrollTip will show a string such as
  ' "Row 5: Accounts Receivable"
  fa.ScrollTipText = " Row " & Row & ": " & _
                     fa.Cell(flexcpTextDisplay, Row, 0) & " "
End Sub
```

Note that you may also implement regular tooltips in Visual Basic by trapping the **MouseMove** event and setting the **ToolTipText** property.

Data Type

Boolean

Default Value

False

ScrollTipText Property

See Also Examples Applies to

Returns or sets the tool tip text shown while the user scrolls vertically.

Syntax

[*form*!]*vsFlexGrid*.**ScrollTipText**[= *value As String*]

Remarks

Set this property in response to the **BeforeScrollTip** event to display information describing a given row as the user scrolls the contents of the control.

For more details, see the **ScrollTips** property.

Data Type

String

ScrollTrack Property

See Also Examples Applies to

Returns or sets scrolling should occur while the user moves the scroll thumb.

Syntax

[*form*!]*vsFlexGrid*.**ScrollTrack**[= {**True | False**}]

Remarks

This property is usually set to False to avoid excessive scrolling and flickering. Set it to True if you want to emulate other controls that have this behavior.

Either way, you may use the **ScrollTips** property to provide the user with additional information while he scrolls the contents of the control.

Data Type

Boolean

Default Value

False

SelChange Event

See Also Examples Applies to

Fired after the selected range changes.

Syntax

Private Sub *vsFlexGrid*_**SelChange**()

Remarks

This event is fired when the **Row**, **Col**, **RowSel** or **ColSel** properties change, either as a result of user actions (mouse or keyboard) or through code.

This event is also fired while the user extends the selection with the mouse.

Select Method

See Also Examples Applies to

Selects a range of cells.

Syntax

[*form*!]*vsFlexGrid*.**Select** *Row* As Long, *Col* As Long, [*RowSel* As Variant], [*ColSel* As Variant]

Remarks

The **Select** method allows you to select ranges or cells (by omitting the last two paramters) with a single command.

This method is more efficient than setting the **Row**, **Col**, **RowSel**, and **ColSel** properties separately and makes the code more readable.

SelectedRow Property

See Also Examples Applies to

Returns the position of a selected row when SelectionMode is set to flexSelectionListBox.

Syntax

val& = [*form*!]*vsFlexGrid*.**SelectedRow**(*Index* As Long)

Remarks

This property works in conjunction with the **SelectedRows** property to enumerate all selected rows in the control.

These properties are especially useful when the **SelectionMode** property is set to *flexSelectionListBox*, which allows the user to select multiple, non-adjacent rows.

Using the **SelectedRows** and **SelectedRow** properties to enumerate all selected rows is much faster than scanning the entire control for selected rows by reading the **IsSelected** property.

For example, write

```
For i = 0 to fa.SelectedRows - 1
  Debug.Print "Row "; fa.SelectedRow(i); " is selected"
Next
```

instead of

```
For i = 0 to fa.Rows
  If fa.IsSelected(i) Then Debug.Print "Row "; i; " is selected"
Next
```

Data Type

Long

SelectedRows Property

See Also Examples Applies to

Returns the number of selected rows when SelectionMode is set to flexSelectionListBox.

Syntax

val& = [*form*!]*vsFlexGrid*.**SelectedRows**

Remarks

This property works in conjunction with the **SelectedRow** property to enumerate all selected rows in the control.

These properties are especially useful when the **SelectionMode** property is set to *flexSelectionListBox*, which allows the user to select multiple, non-adjacent rows.

Using the **SelectedRows** and **SelectedRow** properties to enumerate all selected rows is much faster than scanning the entire control for selected rows by reading the **IsSelected** property.

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

Data Type

Long

SelectionMode Property

See Also Examples Applies to

Returns or sets whether the control will select cells in a free range, by row, by column, or like a listbox.

Syntax

[*form*!]*vsFlexGrid*.**SelectionMode**[= *SelModeSettings*]

Remarks

The settings for the **SelectionMode** property are described below:

flexSelectionFree
This setting allows selections to be made as usual, spreadsheet-style.

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

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

flexSelectionListBox
This setting forces selections to span entire rows and allows for extended selections spanning non-adjacent rows. CTRL-clicking with the mouse toggles the selection for an individual row. Dragging the mouse over a group of rows toggles their selected status.

The **IsSelected** property allows you to read and set the selected status of individual rows.

You may prevent selection by setting the **AllowSelection** property to false.

Data Type

SelModeSettings (Enumeration)

Default Value

flexSelectionFree (0)

SheetBorder Property

See Also Examples Applies to

Returns or sets the color used to draw the border around the sheet.

Syntax

[*form*!]*vsFlexGrid*.**SheetBorder**[= *colorref&*]

Remarks

This property is useful if you want to make a grid look like a page, with no border around the cells. To do this, set the **SheetBorder** and **BackColorBkg** properties to the same color as the grid background (**BackColor** property).

Data Type

Color

ShowComboButton Property

See Also Examples Applies to

Returns or sets whether drop-down buttons are shown when editable cells are selected.

Syntax

[*form*!]*vsFlexGrid*.**ShowComboButton**[= {**True | False**}]

Remarks

If the **ShowComboButton** property is set to True, VSFlexGrid will display drop-down buttons automatically when cells with associated combo boxes or drop-down lists are selected. The user may edit the cells directly, by clicking the button with the mouse.

If the **ShowComboButton** property is set to False, the drop-downs will only appear when the control enters edit mode (either as a result of keyboard action or when the **EditCell** method is used.

You should only set this property to False if the grid columns are narrow and you don't want the drop-down buttons to obscure cell contents while not editing.

Data Type

Boolean

Default Value

True

Sort Property

See Also Examples Applies to

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

Syntax

[*form*!]*vsFlexGrid*.**Sort** = *SortSettings*

Remarks

The **Sort** property allows you to sort a range or rows in ascending or descending order based on the values in one or more columns.

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 sorting algorithm used by the VSFlexGrid control is "stable": this means that the sorting keeps the relative order of records when the sorting key is the same. For example, if you sort a list of files by name, then by extension, file names will still be sorted within each extension group.

Valid settings for the **Sort** property are:

Value	Constant
0	*flexSortNone*
1	*flexSortGenericAscending*
2	*flexSortGenericDescending*
3	*flexSortNumericAscending*
4	*flexSortNumericDescending*
5	*flexSortStringNoCaseAscending*
6	*flexSortStringNoCaseDescending*
7	*flexSortStringAscending*
8	*flexSortStringDescending*
9	*flexSortCustom*
10	*flexSortUseColSort*

The method used to compare the rows is determined by the settings shown above. Most settings have names that are self-descriptive. The exceptions are *flexSortCustom*, *flexSortUseColSort*, and *flexSortNone*.

The *flexSortCustom* setting is the most flexible. It fires a **Compare** event that allows you to compare rows in any way you wish, using any columns in any order. However, *flexSortCustom* is also much slower that the others, typically by a factor of 10, so it should be used only when really necessary. If you want to sort based on arbitrary criteria (e.g. "Urgent", "High", "Medium", "Low"), use a hidden column with numerical values that correspond to the criteria you are using.

The *flexSortUseColSort* allows you to use different settings for each column, as determined by the **ColSort** property. Using this setting, you may sort some columns in ascending and others in descending order.

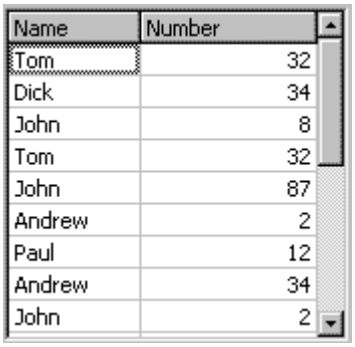
The *flexSortNone* setting is useful only if you assign it to the **ColSort** property and later sort the control with the *flexSortUseColSort* setting. In this case, *flexSortNone* allows you to specify columns that should be ignored by the sorting process.

To sort dates, make sure the column contianing dates has its **ColDataType** property set to *flexDTDate*. This will allow the control to sort them properly. For example:

```
fa.ColDataType(i) = flexDTDate
fa.Col = i
fa.Sort = flexSortGenericAscending
```

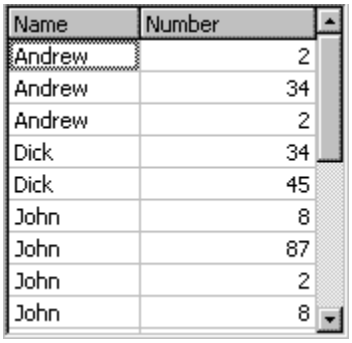
The example below shows how the **Sort** property is used:

```
' fill control with random data
fa.Cols = 2
fa.FixedCols = 0
FillColumn fa, 0, "Name|Andrew|John|Paul|Mary|Tom|Dick|Harry"
FillColumn fa, 1, "Number|12|32|45|2|65|8|87|34"
```



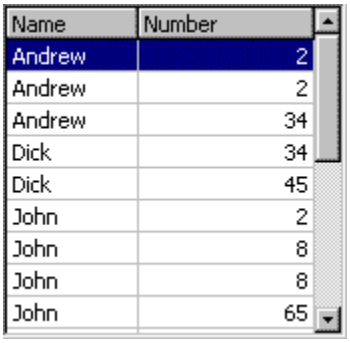
Name	Number
Tom	32
Dick	34
John	8
Tom	32
John	87
Andrew	2
Paul	12
Andrew	34
John	2

```
' sort by name
fa.Select 1, 0
fa.Sort = flexSortGenericAscending
```



Name	Number
Andrew	2
Andrew	34
Andrew	2
Dick	34
Dick	45
John	8
John	87
John	2
John	8

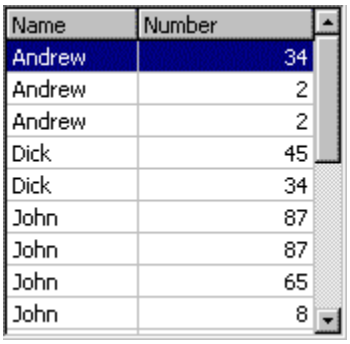
```
' sort by name and number
fa.Select 1, 0, 1, 1
fa.Sort = flexSortGenericAscending
```



Name	Number
Andrew	2
Andrew	2
Andrew	34
Dick	34
Dick	45
John	2
John	8
John	8
John	65

If you want to select different sorting orders for each column, either sort them one by one or use the **ColSort** property and the *flexSortUseColSort* setting. Here is an example that sorts the names in ascending order and the numbers in descending order:

```
fa.ColSort(0)=flexSortGenericAscending
fa.ColSort(1) = flexSortGenericDescending
fa.Select 1, 0, 1, 1
fa.Sort = flexSortUseColSort
```



Name	Number
Andrew	34
Andrew	2
Andrew	2
Dick	45
Dick	34
John	87
John	87
John	65
John	8

Data Type

SortSettings (Enumeration)

Subtotal Method

See Also Examples Applies to

Inserts rows with summary data.

Syntax

[*form*!]*vsFlexGrid*.**Subtotal** *Function* As SubtotalSettings, [*GroupOn* As Long], [*TotalOn* As Long], [*Format* As String], [*BackColor* As Color], [*ForeColor* As Color], [*FontBold* As Boolean], [*Caption* As String], [*MatchFrom* As Long], [*TotalOnly* As Boolean]

Remarks

The **Subtotal** method adds subtotal rows which summarize the data in the control.

Subtotal rows are used for summarizing data and for displaying outlines. You may use the **Subtotal** method to create subtotal rows automatically, or the **IsSubtotal** property to create them manually.

Each subtotal row has a *level* that is used to indicate which column is being grouped. The subtotal level is also used for outlining. When you created subtotals using the **Subtotal** method, the *level* is set automatically based on the *GroupOn* parameter. When you create an outline manually, use the **RowOutlineLevel** property to set the outline level for each subtotal row.

Subtotal rows may be added at the top or at the bottom of the values being summarized. This is determined by the **SubtotalPosition** property. When creating outlines, you will typically use the **SubtotalPosition** property is used to place the subtotals *above* the data. When creating reports, you will typically use the **SubtotalPosition** property to place the subtotals *below* the data.

The parameters for the **Subtotal** method are described below:

Function As SubtotalSettings
This parameter specifies the type of aggregate function to be used for the subtotals. Valid settings are:

Constant	Description
flexSTNone	Outline only, no aggregate values
flexSTClear	Clear all subtotals
flexSTSum	Sum
flexSTPercent	Percent of total sum
flexSTCount	Row count
flexSTAverage	Average
flexSTMax	Maximum
flexSTMin	Minimum
flexSTStd	Standard deviation
flexSTVar	Variance

G*roupOn* As Long (optional)
This parameter specifies the column that contains the categories for calculation of a subtotal. By default, the control assumes that all data is sorted from the leftmost column to the column specified as *GroupOn*. Consequently, a subtotaling break occurs whenever there is a change in any column from the leftmost one up to and including the column specified as *GroupOn*.

To create subtotals based on a column or range of columns that does not start with the leftmost column, use the *MatchFrom* parameter. If *MatchFrom* is specified, the control generates subtotal line only on a change of data in any column between and including column *MatchFrom* and *GroupOn*.

For example, to subtotal values in column 3 of the control whenever there are changes in column 2 only, use

```
.Subtotal flexSTSum, 2, 3, , , , , 2
```

TotalOn As Long (optional)
This parameter specifies the column that contains the values to use when calculating the total.

Format As String (optional)
This parameter 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. For a detailed description of the syntax used to specify formats, see the **ColFormat** property.

BackColor, ForeColor As Color (optional)
These parameters specify the colors to be used for the cells in the subtotal rows.

FontBold As Boolean (optional)
This parameter specifies whether text in the subtotal rows should be boldfaced.

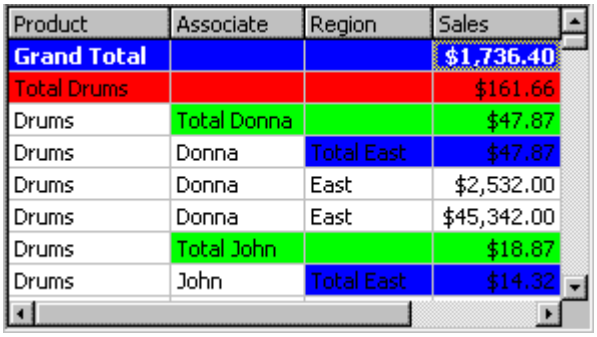
Caption As Variant (optional)
This parameter specifies the text that should be put in the subtotal rows. If omitted, the text used is the function name plus the category name (e.g. "Total Widgets"). If supplied, you may add a "%s" marker to indicate a place where the category name should be inserted (e.g. "The %s Count").

MatchFrom As Variant (optional)
When deciding whether to insert a subtotal row between two adjacent rows, the control compares the values in columns between *MatchFrom* and *GroupOn*. If any of these cells are different, a subtotal row is inserted. The default value for *MatchFrom* is **FixedCols**, which means all columns to the left of and including *GroupOn* must match, or a subtotal row will be inserted. If you set *MatchFrom* to the same value as *GroupOn*, then subtotal rows will be inserted whenever the contents of the *GroupOn* column change.

TotalOnly As Boolean (optional)
By default, the control will copy the contents of all columns between *MatchFrom* and *GroupOn* to the new subtotal row, and will place the calculated value on column *TotalOn*. If you set the *TotalOnly* parameter to True, the control will not copy the contents of the rows. The subtotal row will contain only the title and the calculated value.

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

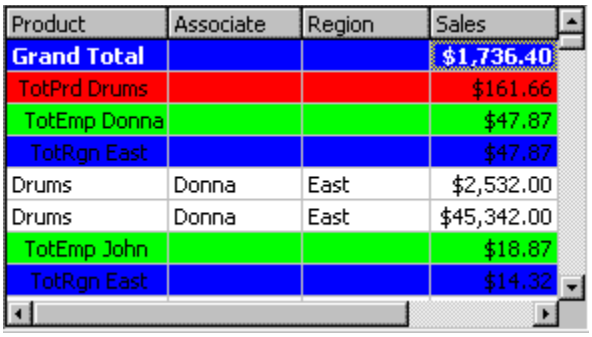
```
' this assumes we have a populated grid fa with
' 4 columns: product, employee, region, and sales
fa.ColFormat(3) = "$(#,###.00)" ' set format for calculated totals
fa.Subtotal flexSTClear         ' remove old values

' calculate subtotals (the order doesn't matter)
' (sales values to be added are in column 3)
' col 0: product
fa.Subtotal flexSTSum, 0, 3, , vbRed
' col 1: employee
fa.Subtotal flexSTSum, 1, 3, , vbGreen
' col 2: region
fa.Subtotal flexSTSum, 2, 3, , vbBlue

' total on a negative column to get a grand total
fa.Subtotal flexSTSum, -1, 3, , vbblue, vbwhite, true
```



Product	Associate	Region	Sales
Grand Total			$1,736.40
Total Drums			$161.66
Drums	Total Donna		$47.87
Drums	Donna	Total East	$47.87
Drums	Donna	East	$2,532.00
Drums	Donna	East	$45,342.00
Drums	Total John		$18.87
Drums	John	Total East	$14.32

The parameters in the **Subtotal** method allow a great deal of customization. The example below shows how the *Caption* and *TotalOnly* parameters can be used to generate report-type subtotals:

```
fa.ColFormat(3) = "$(#,###.00)" ' set format for calculated totals
fa.Subtotal flexSTClear         ' remove old values

' calculate subtotals (the order doesn't matter)
' (sales values to be added are in column 3)
' col 0: product
fa.Subtotal flexSTSum, 0, 3, , vbRed  ,,," TotPrd %s",,True
' col 1: employee
fa.Subtotal flexSTSum, 1, 3, , vbGreen,,,"  TotEmp %s",,True
' col 2: region
fa.Subtotal flexSTSum, 2, 3, , vbBlue ,,,"   TotRgn %s",,True

' total on a negative column to get a grand total
fa.Subtotal flexSTSum, -1, 3, , vbblue, vbwhite, true
```



Product	Associate	Region	Sales
Grand Total			**$1,736.40**
TotPrd Drums			$161.66
TotEmp Donna			$47.87
TotRgn East			$47.87
Drums	Donna	East	$2,532.00
Drums	Donna	East	$45,342.00
TotEmp John			$18.87
TotRgn East			$14.32

SubtotalPosition Property

See Also Examples Applies to

Returns or sets whether subtotals should be inserted above or below the totaled data.

Syntax

[*form*!]*vsFlexGrid*.**SubtotalPosition**[= *SubtotalPositionSettings*]

Remarks

Valid settings for the **SubtotalPosition** property are:

Value	Constant
0	*flexSTBelow*
1	*flexSTAbove*

Setting *flexSTAbove* is typically used to create outlines. This way, the subtotal rows (which correspond to outline nodes) appear above the data to which they refer. Setting *flexSTBelow* is typically used to create reports. This way, the subtotal rows appear below the data to which they refer.

Data Type

SubtotalPositionSettings (Enumeration)

Default Value

flexSTAbove (1)

TabBehavior Property

See Also Examples Applies to

Returns or sets whether the tab key will move focus between controls (VB default) or between grid cells.

Syntax

[*form*!]*vsFlexGrid*.**TabBehavior**[= *TabBehaviorSettings*]

Remarks

Valid settings for the **TabBehavior** property are:

Constant	Description
flexTabControls	Tab key is used to move to the next or previous control on the form.
flexTabCells	Tab key is used to move to the next or previous cell on the control.

Data Type

TabBehaviorSettings (Enumeration)

Default Value

flexTabControls (0)

Text Property (vsFlexGrid Object)

See Also Examples Applies to

Returns or sets the contents of the selected cell or range.

Syntax

[*form*!]*vsFlexGrid*.**Text**[= *value As String*]

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 **Cell**(*flexcpText*) or the **TextMatrix** properties.

You may read the formatted contents of a cell using the **Cell**(*flexcpTextDisplay*) property.

You may read the value of a cell using the **Cell**(*flexcpValue*), **Value**, and **ValueMatrix** properties. This is useful when the cell contains text that is formatted with thousand separators, which are not recognized by the Visual Basic **Val** function.

Data Type

String

TextArray Property

See Also Examples Applies to

Returns or sets the contents of a cell identified by a single index.

Syntax

[*form*!]*vsFlexGrid*.**TextArray**(*Index* As Long)[= *value As String*]

Remarks

This property is provided for backward compatibility with earlier versions of this control. New applications should use the **Cell**(*flexcpText*) or **TextMatrix** properties.

Data Type

String

TextMatrix Property

See Also Examples Applies to

Returns or sets the contents of a cell identified by its row and column coordinates.

Syntax

[*form*!]*vsFlexGrid*.**TextMatrix**(*Row* As Long, *Col* As Long)[= *value As String*]

Remarks

The **TextMatrix** property allows you to set or retrieve the contents of a cell without changing the **Row** property and **Col** property.

See also the **Cell** property, which allows you to set or retrieve text, pictures and formatting information for a cell or range of cells.

Data Type

String

TextStyle Property

See Also Examples Applies to

Returns or sets 3D effects for displaying text in non-fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**TextStyle**[= *TextStyleSettings*]

Remarks

Valid settings for the **TextStyle** property are:

Value	Constant
0	*flexTextFlat*
1	*flexTextRaised*
2	*flexTextInset*
3	*flexTextRaisedLight*
4	*flexTextInsetLight*

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

You may set the text style for the fixed cell using the **TextStyleFixed** property, or set the text style for individual cells and ranges using the **Cell**(*flexcpTextStyle*) property.

Data Type

TextStyleSettings (Enumeration)

Default Value

flexTextFlat (0)

TextStyleFixed Property

See Also Examples Applies to

Returns or sets 3D effects for displaying text in fixed cells.

Syntax

[*form*!]*vsFlexGrid*.**TextStyleFixed**[= *TextStyleSettings*]

Remarks

Valid settings for this property are the same as those for the **TextStyle** property.

Data Type

TextStyleSettings (Enumeration)

Default Value

flexTextFlat (0)

TopRow Property

See Also Examples Applies to

Returns or sets the zero-based index of the topmost non-fixed row displayed in the control.

Syntax

[*form*!]*vsFlexGrid*.**TopRow**[= *value As Long*]

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 (no error will be generated).

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

Data Type

Long

TreeColor Property

See Also Examples Applies to

Returns or sets the color used to draw the outline tree.

Syntax

[*form*!]*vsFlexGrid*.**TreeColor**[= *colorref&*]

Remarks

The outline tree is drawn only when the **OutlineBar** property is set to a non-zero value and the control contains subtotal rows. It allows users to collapse and expand the outline.

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

Data Type

Color

ValidateEdit Event

See Also Examples Applies to

Fired before the control exits cell edit mode.

Syntax

Private Sub *vsFlexGrid*_**ValidateEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *Cancel* As Boolean)

Remarks

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

You may trap this event to read the contents of the cell editor with the **EditText** property and to make sure the entry is valid for the given cell (*Row*, *Col*). If the entry is invalid set the *Cancel* parameter to True. The changes will be discarded and the control will remain in edit mode.

If you want to validate keys as they are typed into the editor, use the **KeyPressEdit** or the **ChangeEdit** events.

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

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

```
Sub fa_ValidateEdit(ByVal Row As Long, ByVal Col As Long, cancel As Boolean)
  Dim c$

  ' different validation rules for each column
  Select Case col

      ' column 1 only accepts strings
      Case 1
        c = Left$(fa.EditText, 1)
        If UCaseS(c) < "A" And UCase$(c) > "Z" Then Beep: Cancel = True

      ' column 2 only accepts numbers > 0
      Case 2
        If Val(fa.EditText) <= 0 Then Beep: Cancel = True

  End Select
End Sub
```

Note:

In previous versions of this control, this event was called **Validate**. The name of the event was changed to avoid conflicts with Visual Basic 6.0's new **Validate** event.

Value Property

See Also Examples Applies to

Returns the numeric value of the current cell.

Syntax

val# = [*form*!]*vsFlexGrid*.**Value**

Remarks

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

For example, if the current cell contains the string "$(1,234.56)", the **Value** property will return the value -1234.56.

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

Note

This property is not an expression evaluator. If the current cell contains the string "2+2", for example, the **Value** property will return 2 instead of 4. The Visual Basic statement **Val**("2+2") also returns 2.

Data Type

Double

ValueMatrix Property

See Also Examples Applies to

Returns the numeric value of a cell identified by its row and column coordinates.

Syntax

val# = [*form*!]*vsFlexGrid*.**ValueMatrix**(*Row* As Long, *Col* As Long)

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 (vsFlexGrid Object)

See Also Examples Applies to

Returns the version of vsFlex currently loaded in memory.

Syntax

val% = [*form*!]*vsFlexGrid*.**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 6.00 returns 600.

This property is read-only.

Data Type

Integer

Default Value

600

VirtualData Property

See Also Examples Applies to

Returns or sets whether all data is fetched from the data source at once or as needed.

Syntax

[*form*!]*vsFlexGrid*.**VirtualData**[= {**True | False**}]

Remarks

The **VirtualData** property works when the control is data-bound.

If **VirtualData** is set to True, data is retrieved from the data source only when it is needed (for displaying or reading its value, for example). This saves time and memory.

If **VirtualData** is set to False, the entire dataset is read from the data source into memory, all at once. This process may be slow, especially if the data source is large (over about 5,000 records).

See also the **DataSource** and **DataMode** properties.

Data Type

Boolean

Default Value

True

WordWrap Property

See Also Examples Applies to

Returns or sets whether text wider that its cell will wrap.

Syntax

[*form*!]*vsFlexGrid*.**WordWrap**[= {**True** | **False**}]

Data Type

Boolean

vsFlexString Object

Properties Methods Events

Object Name: VSFlexString
Description: :-) VideoSoft VSFlexString 6.0
Properties: 17
Events: 0
Methods: 0

Before you can use a **VSFlexString** object in your application, you must add the **VSFLEX6.OCX** file to your project.

To distribute applications you create with the **VSFlexString** object, you must install and register it on the user's computer. The Setup Wizard provided with Visual Basic provides tools to help you do this. Please refer to the Visual Basic manual for details.

The **VSFlexString** control allows you to incorporate regular-expression text matching into your Visual Basic 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 like Microsoft Word, Visual C++, and Visual Basic.

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

CaseSensitive Property

See Also Examples Applies to

Returns or sets whether matching is case-sensitive.

Syntax

[*form*!]*vsFlexString*.**CaseSensitive**[= {**True | False**}]

Remarks

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

Data Type

Boolean

Error Property (vsFlexString Object)

See Also Examples Applies to

Returns status information after setting the Pattern or Text properties.

Syntax

val% = [*form*!]*vsFlexString*.**Error**

Remarks

You should always check the **Error** property when a match fails. Possible values for this property are:

Value	Constant
0	*flexErrNone*
1	*flexErrOutOfMemory*
2	*flexErrSquareB*
3	*flexErrCurlyB*
4	*flexErrBadPattern*
5	*flexErrBadTagIndex*
6	*flexErrNoMatch*
7	*flexErrInvalidMatchIndex*

flexErrOutOfMemory
Occurs if you assign a string that is too long for the **Text** property or a pattern that is too complex for the **Pattern** property.

flexErrSquareB, flexErrCurlyB
Occurs when you assign a pattern with unbalanced square 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
Occurs when you try to retrieve the results of a match and the **Pattern** or **Text** properties are empty.

flexErrBadTagIndex
Occurs when you use a tag in a replacement string for which there is no match (e.g. **Pattern** = "{[a-z]*} ", **Replace** = "{0} {1}": the **Pattern** defines one tag only, and the replacement string references two).

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

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

Data Type

StringErrorSettings (Enumeration)

MatchCount Property

See Also Examples Applies to

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

Syntax

val& = [*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

Long

MatchIndex Property

See Also Examples Applies to

Returns or sets the zero-based index of the current match when there are multiple matches.

Syntax

[*form*!]*vsFlexString*.**MatchIndex**[= *value As Long*]

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.

Alternatively, you may specify the **MatchIndex** as an index when you read the **MatchLength**, **MatchStart**, and **MatchString** properties. This is a new feature in version 6 of the control. For example:

```
' show all matches
Dim i As Long
For i = 0 to fs.MatchCount - 1
    Debug.Print "[" & fs.MatchString(i) & "]"
Next
```

Data Type

Long

MatchLength Property

See Also Examples Applies to

Returns the length of the current match, in characters.

Syntax

val& = [*form*!]*vsFlexString*.**MatchLength**([*MatchIndex* As Variant])

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

Long

MatchStart Property

See Also Examples Applies to

Returns the zero-based position of the current match within the Text string.

Syntax

val& = [*form*!]*vsFlexString*.**MatchStart**([*MatchIndex* As Variant])

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

Long

MatchString Property

See Also Examples Applies to

Returns or sets the string corresponding to the current match.

Syntax

[*form*!]*vsFlexString*.**MatchString**([*MatchIndex* As Variant])[= *value As String*]

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

See Also Examples Applies to

Returns or sets the regular expression used for matching against the Text string.

Syntax

[*form*!]*vsFlexString*.**Pattern**[= *value As String*]

Remarks

The regular expression syntax recognized by **vsFlexString** is based on the following special characters:

Char	Description
^	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.

For example,

```
fs.Pattern = "^stuff" ' any string starting with "stuff"
fs.Pattern = "stuff$" ' any string ending with "stuff"
fs.Pattern = "o.d"    ' "old", "odd", "ord", 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 (needs escape character)
```

Data Type

String

Replace Property

See Also Examples Applies to

Sets a string to replace all matches.

Syntax

[*form*!]*vsFlexString*.**Replace** = *value As String*

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:

```
' set up a pattern to search for a filename and extension:
' the curly brackets define two tags
' (note how the period is escaped with a backslash)
fs.Pattern = "{[A-Za-z0-9_]+}\.{...}"

' assign a string to be matched against the pattern
' tag 0 will match the filename, tag 1 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.Print fs.Text
  File AUTOEXEC.BAT, Name: AUTOEXEC, Ext: BAT
```

Data Type

String

Soundex Property

See Also Examples Applies to

Returns a phonetic code representing the current Text string.

Syntax

val$ = [*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

See Also Examples Applies to

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

Syntax

val& = [*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.

```
fs.Text = "Mary had a little lamb"
fs.Pattern = "Mary had {.*}"

Debug.Print fs.TagCount; fs.TagIndex; "[" & fs.TagString & "]"
  1  0 [a little lamb]
```

Data Type

Long

TagIndex Property

See Also Examples Applies to

Returns or sets the index of the current tag when there are multiple tags in the Pattern string.

Syntax

[*form*!]*vsFlexString*.**TagIndex**[= *value As Long*]

Remarks

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

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

The example below shows how the **TagIndex** property works:

```
' set some text
fs.Text = "Mary had a little lamb"

' define a patern with two tags
fs.Pattern = "{[^ ]*} had {.*}"

' show tag 0
fs.TagIndex = 0
Debug.Print "[" & fs.TagString & "]"
  [Mary]

' show tag 1
fs.TagIndex = 1
Debug.Print "[" & fs.TagString & "]"
  [a little lamb]
```

Alternatively, you may specify the **TagIndex** as an index when you read the **TagString** property. This is a new feature in version 6 of the control. For example:

```
' show tag 0
Debug.Print "[" & fs.TagString(0) & "]"
  [Mary]

' show tag 1
fs.TagIndex = 1
Debug.Print "[" & fs.TagString(1) & "]"
  [a little lamb]
```

Data Type

Long

TagLength Property

See Also Examples Applies to

Returns the length of the current tag, in characters.

Syntax

val& = [*form*!]*vsFlexString*.**TagLength**([*TagIndex* As Variant])

Remarks

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

Data Type

Long

TagStart Property

See Also Examples Applies to

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

Syntax

val& = [*form*!]*vsFlexString*.**TagStart**([*TagIndex* As Variant])

Remarks

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

Data Type

Long

TagString Property

See Also Examples Applies to

Returns or sets the string corresponding to the current tag.

Syntax

[*form*!]*vsFlexString*.**TagString**([*TagIndex* As Variant])[= *value As String*]

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 (vsFlexString Object)

See Also Examples Applies to

Returns or sets the text to be scanned searching for the Pattern string.

Syntax

[*form*!]*vsFlexString*.**Text**[= *value As String*]

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 (vsFlexString Object)

See Also Examples Applies to

Returns the version of vsFlex currently loaded in memory.

Syntax

val% = [*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 6.00 returns 600.

This property is read-only.

Data Type

Integer

Frequently Asked Questions

This section contains answers to the most common questions people ask our technical support staff. You should read this section even if you have not experienced any problems, especially if you are using Visual C++. You may find some useful tips here.

How do I update a project file that uses VSFLEX3 to VSFlexGrid Pro 6.0?

Use CONVERT, the conversion utility provided with **VSFlexGrid Pro 6.0**. The CONVERT utility also allows you to convert between the ADO/RDO and OLEDB/ADO versions of **VSFlexGrid Pro 6.0**.

The **CONVERT** utility handles practically all aspects of the conversion automatically. There are only three areas that require your attention:

1) If your VSFlex3 project builds custom outlines using the **IsSubtotal** and **RowData** properties, you will need to manually modify it to use **RowOutlineLevel** instead of **RowData**. This is because VSFlexGrid reserves the **RowData** property for exclusive use by the programmer.

2) If your VSFlex3 project uses cell merging, you may want to set the **HighLight** property to 0 (*flexHighlightNever*) to keep the same behavior you had before. This is because VSFlexGrid, unlike VSFlex3, allows highlighting of selected ranges even when cells are merged.

3) If your VSFlex3 project assigns Variant values to the **RowData** or **ColData** properties, you should add code to convert the Variant value into a Long before the assignment (you may use VB's **CLNG**() function). This is because these properties used to be of type Long, and are now Variants.

The **CONVERT** utility is supplied in source code format, so you may modify it if you need to.

What is difference between VSFLEX6.OCX and VSFLEX6D.OCX?

The VSFlexGrid Pro package includes two OCX files:

VSFLEX6.OCX contains the version with OLEDB/ADO data-binding. You may bind the control to any ADO data source, including the ADO data control that ships with VB6.

VSFLEX6D.OCX contains the version with DAO/RDO data-binding. You may bind the control to the traditional data sources (built-in DAO data control, RDO data control).

The controls included in each file are functionally identical, but have slightly different names and GUIDs. This allows programs using both versions to run simultaneously on the same computer without conflict.
Before starting a new project or migrating an existing project to VSFlexGrid Pro 6.0, you should decide which version to use. The following information will help you make the decision:

1) If you are planning to migrate DAO database applications to OLEDB/ADO, use VSFLEX6.OCX.

2) If you are planning to migrate DAO database applications to VB6, but still want to use DAO or RDO, use VSFLEX6D.OCX.

3) If you are using the VSFlexGrid in unbound mode (i.e., not bound to any databases at all), then you should use VSFLEX6D.OCX (the DAO/RDO version). This is because ADO-based controls **will not load** on computers that don't have ADO installed (even if the control does not use its ADO features.)

You cannot use VSFLEX6.OCX on a computer that does not have ADO installed. VB6 installs ADO automatically. VB5 and VB4 do not. In the future, most computers will have ADO available as part of the system. Until then, VSFLEX6D.OCX is a better choice for unbound-mode development because it has no dependencies.

Whichever version you decide to use, you may easily switch later using the CONVERT utility supplied with VSFlexGrid Pro 6.0.

Does VSFlexGrid 6.0 work with VB4-16 or any other 16-bit environments?

Sorry, it does not. VSFlexGrid Pro 6.0 is a 32-bit-only product.

If your application needs to run under 16-bits, you may still use VSFlex3, which VideoSoft still supports.

When adding VSFLEX6.OCX to my VB4 or VB5 project, I get the following error message: "Error loading DLL". What is wrong?

VSFLEX6.OCX contains the OLEDB/ADO version of the VSFlexGrid control. Because of that, it requires the ADO system DLL's in order to run (the same is true for the OLEDB controls that ship with VB6).

To use VSFLEX6.OCX on a computer that only has VB4 or VB5 installed, you will need to install the ADO system DLL's.

If you are not using OLEDB/ADO, consider using the VSFLEX6D.OCX version of the controls, which is not subject to this limitation.

My VSFLEX 3.0 project works fine in VB 5.0, but in VB 6.0 I get the following error message: "Procedure declaration does not match description of event or procedure having the same name". What is wrong?

Visual Basic 6.0 introduced a Validate event (with one argument) that is supplied and managed by VB itself. This causes a conflict with the VSFlexArray 3.0 control's built-in **Validate** event which has three arguments. (If you don't place any code in the Validate event, the problem does not arise since the event is never handled.)

In VSFlexGrid Pro, this problem has been solved by renaming the built-in event to **ValidateEdit**. The upgrading utility that ships with the new control performs this translation automatically, so upgrading code from VSFlexArray is not a problem.

If you are upgrading from VB 5.0 to VB 6.0 but not from VSFlexArray to VSFlexGrid Pro, there is an easy workaround. The code below shows how this can be done using the **BeforeEdit** and **AfterEdit** events instead of **Validate**:

```
Private Sub fa_BeforeEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)

  ' save original content in Tag property
  fa.Tag = fa.TextMatrix(Row, Col)

End Sub

Private Sub fa_AfterEdit(ByVal Row As Long, ByVal Col As Long)

  ' no change? then don't validate
  If fa.Tag = fa.TextMatrix(Row, Col) Then Exit Sub

  ' valid new value? then we're done
  ' (in this example value has to be up to 100)
  If Val(fa.TextMatrix(Row, Col)) <= 100 Then Exit Sub

  ' beep to warn user the input was rejected
  Beep

  ' Option 1: light validation
  ' restore original cell content
  fa.TextMatrix(Row, Col) = fa.Tag

  ' Option 2: robust validation
  ' use Timer to restore cursor and get into edit mode
  fa.Tag = Row & "," & Col
  Timer1.Enabled = True

End Sub
```

```
Private Sub Timer1_Timer()

  ' do this once per call
  Timer1.Enabled = False

  ' parse row and col where validation failed
  Dim r&, c&
  r = Val(fa.Tag)
  c = InStr(fa.Tag, ",")
  If c  0 Then c = Val(Mid(fa.Tag, c + 1))

  ' select the cell and try again
  fa.Select r, c
  fa.EditCell

End Sub
```

Does VSFlexGrid 6.0 work with VB4, VB5 and VB6?

VSFlexGrid Pro 6.0 works with any 32-bit version of Visual Basic. Ideally, however, you should use it with VB5 or later.

When used with VB4, the optional parameters in some properties are not interpreted as optional by VB. The most important property affected by this is the **Cell** property, which has the following syntax:

```
[v =] fa.Cell(iProp, [Row1], [Col1], [Row2], [Col2])
```

In VB5 or VB6, you may omit all or some of the last four parameters. In VB4, you must supply all five.

This limitation does not apply to optional parameters in methods, only to optional parameters in properties.

How do I limit the length of text entries in a column?

Set the **EditMaxLength** property in response to the **BeforeEdit** event.

There are several ways to add data to a VSFlexGrid control. Which one is the fastest?

The fastest way to add data is using the **TextMatrix** or the **Cell** properties. The slowest way is using the **AddItem** method, because it adds rows in addition to data.

If the data is already loaded in an array of Variants, then the **BindToArray** method is even faster. (**BindToArray** does not actually load the data, it just tells the control where the data is).

Whatever method you choose, make sure you set the **Redraw** property to False before you start populating the grid, and set it back to True when you are done. This may increase speed by an order of magnitude, especially when using **AddItem**.

How can I add or delete a column at a given position?

To add a column at a specific position: create the new column by incrementing the **Cols** property, then move it to the desired position using the **ColPosition** property.

To delete a column at a specific position: move the column to the right using the **ColPosition** property, then delete it by decrementing the **Cols** property.

The following VB code shows how to do it: it deletes the current column or inserts a new column to the left of the current column, depending on which button was clicked.

```
Private Sub Command1_Click(Index As Integer)
  With fa
```

```
    ' insert column
    If Index = 0 Then
      .Cols = .Cols + 1               ' add column
      .ColPosition(.Cols - 1) = .Col  ' move into place

    ' delete column
    Else
      .ColPosition(.Col) = .Cols - 1  ' move to right
      .Cols = .Cols - 1               ' delete column
    End If
  End With

End Sub
```

How can I implement OLE Drag and Drop?

To implement automatic OLE Drag and Drop, set the **OLEDragMode** or **OLEDropMode** properties to the automatic settings, and you are done.

To implement manual OLE Drag and Drop, you will need to write some code. See the **OLE Drag and Drop Demo** for an example that implements both manual and automatic OLE Drag and Drop.

How can I print the contents of a VSFlexGrid control?

VSFlexGrid has a **Picture** property that may be assigned to Visual Basic's **Printer** object. This method works well for grids that will fit on a single page.

For grids that span multiple pages, you should consider using VideoSoft's **VSPrinter** control (part of the VSVIEW product). The **VSPrinter** control has a **RenderControl** property that you can use to print grids of any size. This method will also allow you to control page breaks, create repeating headings, and preview the document.

How do I handle optional parameters in VSFlexGrid using C++?

Optional parameters are always variants. If you want to make the parameter an integer then edit the **vsFlexGrid** wrapper files that Visual C++® creates for you and overload the appropriate method or property.

For example, if you add the functions below to the vsFlexGrid.cpp file (and the corresponding declarations to vsFlexGrid.h), then you can use the **AddItem** method with a nice clean syntax:

```
void CvsFlexGrid::AddItem(LPCTSTR Item) {
  VARIANT v;
  V_VT(&v) = VT_ERROR;
  AddItem(Item, v);
}

void CvsFlexArray::AddItem(LPCTSTR Item, int i) {
  VARIANT v;
  V_VT(&v) = VT_I2;
  V_I2(&v) = (short)i;
  AddItem(Item, v);
}
```

This lets you write the following:

```
fa.AddItem("hello\tmy friend"); // append as last row and
for (i = 1; i < 10; i++)
    fa.AddItem("hello\tmy friend", i); // insert as ith row
```

This solution applies to all properties and methods that take optional parameters.

How do I set a picture in VSFlexGrid using C++?

Here's some code that shows how you can set the VSFlexGrid's **CellPicture** property (or any other ActiveX picture property, actually) from C++ when the user clicks on the control.

Using the AppWizard, generate a new project with as a dialog-based app with the OLE controls option set to True. After creating the project, add a VSFlexGrid control to the form and connect it to the **m_flex** member variable.

Add a bitmap resource and set its id to IDB_ARROWPIC.

Add the following handler for the **m_flex** Click event:

```
// include MFC header that declares the CPictureHolder class, which
// is the easiest way to deal with OLE-based pictures
#include "afxctl.h"

// this is the click event handler, and also the only custom function in this project
void CTestCDlg::OnClickFlex()
{
  // Create a CPictureHolder variable that will hold the picture.
  // (For details, see the ctlPict.cpp file in your MFC\SRC directory.)
  CPictureHolder pic;

  // Initialize the picture holder by giving it a picture to hold.
  // In this case, we're giving it the resource ID of a bitmap, but
  // CPictureHolder can also handle icons and metafiles.
  pic.CreateFromBitmap(IDB_ARROWPIC);

  // Tell the control to show the picture. Because we're handling
  // a click event, the row and column have already been selected.
  m_flex.SetCellPicture(pic.GetPictureDispatch());
}
```

VideoSoft Products

VideoSoft VSDATA™
A very small, very fast database engine.

VSData
A complete database engine in one ActiveX Control. Along with it's speed and small size, **VideoSoft VSDATA** also provides the developer with full support for multimedia applications.

VideoSoft VSDIRECT™ 1.0
Active X control package that allows Visual Basic developers to exploit the power and speed of Microsoft DirectX technology.

VSDirectDraw
Accesses DirectDraw which works with various creative interfaces that accelerate animation techniques through direct access to video memory and hardware.

VSDirectSound
Accesses DirectSound which enables hardware and software sound mixing and playback.

VSDirectPlay
Accesses DirectPlay which provides connectivity of games over a modem link or network.

VideoSoft VSDOCX™ 1.0
VideoSoft VSDOCX automatically creates documentation for any ActiveX component. The core documentation is extracted directly from the component, so it is always 100% up-to-date and accurate. The user does not need access to the component source code. The documentation may be extended and customized as it is created, and revised later. Output is in the form of Word documents, Access databases, HTML files and help files. Custom reports may be generated in user-defined styles.

VideoSoft VSFLEX® 3.0
A set of two custom controls for analyzing, formatting, and displaying information.

VSFlexArray
A new way to display and operate on tabular data. **VSFlexArray** gives you total flexibility to display, sort, merge and format tables containing strings and pictures.

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

VideoSoft VS-OCX®/VSVBX™ (version 6)
A set of three custom controls for interface design and text parsing.

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.

VideoSoft VSSPELL™ 1.0
A set of two custom controls that allow you to easily add spell checking and thesaurus functionality to any Windows application.

VSSpell
A control that allows you to instantly access an extensive American English dictionary with more than 50,000 entries. Add full fledged spell checking to your apps with no code and quickly customize functionality like automatic or manual correction, generation of suggestions for specific words, etc. A utility is included to easily build and maintain custom dictionaries. **VSSpell** is also compatible with dictionary files created by Microsoft Word.

VSThesaurus
A control that allows you to access an American English thesaurus with more than 30,000 entries. **VSThesaurus** has properties and methods that allow you to both build and maintain thesaurus files.

VideoSoft VSVIEW® version 3.0
A set of four custom controls for creating, viewing, and printing text and graphics.

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. **VSInForm** also allows you to monitor the clipboard, drag and drop files from File manager, and more.

Order Form

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

To: VideoSoft
5900-T Hollis Street, Emeryville, California 94608
Toll-Free: 1(888) ACTIVEX * Tel: (510) 595-2400 * Fax: (510) 595-2424

CUSTOMER INFORMATION

Name:_____

Company:_____

Address:_____

City:_____State:_____

Zip:_____Country:_____

Phone:_____ Fax:_____

Email:_____

PRODUCT INFORMATION

PRODUCT	QUANTITY (in units)	PRICE (US$) (per unit)	TOTAL
ElasticLight 6.0 (ActiveX-32 Only)	_____	$49	_____
VS-OCX 6.0 (includes ElasticLight)	_____	$149	_____
VS-OCX 6.0 Upgrade from 5.0	_____	$99	_____
VSVIEW 3.0	_____	$249	_____
VSVIEW 3.0 Upgrade from 2.0	_____	$129	_____
VSVIEW 3.0 Upgrade from 1.0	_____	$198	_____
VSFLEX 3.0	_____	$249	_____
VSFLEX 3.0 Upgrade from 2.0	_____	$129	_____
VSFLEX 3.0 Upgrade from 1.0	_____	$198	_____
VSFlexGrid Pro 6.0	_____	$299	_____
VSFlexGrid Pro 6.0 Upgrade from 3.0	_____	$159	_____
VSFLEX 3.0 Upgrade from 2.0, 1.0 or competitive* grid	_____	$249	_____
VSFlexGrid Pro 6.0 plus Subscription	_____	$398	_____
VSFlexGrid Pro 6.0 Upgrade from 3.0 plus Subscription	_____	$258	_____
VSFLEX 3.0 Upgrade from 2.0, 1.0 or competitive* grid plus Subscription	_____	$348	_____
VSDATA 1.0 (ActiveX-32 Only)	_____	$199	_____
VSREPORTS 1.0 (ActiveX-32 Only)	_____	$149	_____
VSDOCX 1.0	_____	$249	_____
VSDIRECT 1.0 (ActiveX-32 Only)	_____	$189	_____
VSSPELL 1.0	_____	$149	_____
VSFORUM/ASP	_____	$499	_____

Total units: _____

Shipping and Handling Charges (see below): _____

Subtotal: _____

CA Sales Tax 8.50% (California customers only): _____

Total: _____

SHIPPING AND HANDLING CHARGES:

Regular Mail: $7 for the first unit, $3 for each additional unit
2 day (UPS): $15 for the first unit, $7 for each additional unit
Overnight (UPS): $25 for the first unit, $7 for each additional unit
Canada: $15 for the first unit, $3 for each additional unit
Other International: $15 for the first unit, $9 for each additional unit

**** Please note:** *UPS and Federal Express charges vary depending on weight and destination. You can prevent shipment delays by preapproving courier charges (up to $50) when you initial here:* _____.

PAYMENT METHOD

[ ] Here's my check for US$ _____.
NOTE: Checks must be drawn on U.S. Banks only.

[ ] Charge my...
[ ] MasterCard [ ] American Express
[ ] Visa [ ] Discover

Card #:_____ Expires:_____

Signature:_____

** **US Customers only:** *For your security and to help eliminate credit card fraud, if you're paying via credit card, please provide the street address number and zip code of your credit card billing address.*
For example, if your credit card bill is mailed to 5900 Hollis Street, the street address number is 5900.

Street Address Number: _____ Zip Code: _____

*** COMPETITIVE UPGRADES FOR VSFlexGrid Pro 6.0**
You are eligible to upgrade to VSFlexGrid Pro 6.0 for $249 if you are a licensed user of one of the following products: VSFLEX 2.0, VSFLEX 1.0, Visual Basic 5.0, Visual Basic 6.0, True DBGrid, DataWidgets, Spread, GridEx, Formula One, or Data Table.