CLIENTWIDTH PROPERTY ................................................................. 120
CLIP PROPERTY ................................................................................. 120
CLIPSEPARATORS PROPERTY .......................................................... 121
COL PROPERTY .................................................................................. 122
COLALIGNMENT PROPERTY .............................................................. 122
COLCOMBOLIST PROPERTY .............................................................. 123
COLDATA PROPERTY ......................................................................... 124
COLDATATYPE PROPERTY ............................................................... 125
COLEDITMASK PROPERTY ............................................................... 126
COLFORMAT PROPERTY ...................................................................... 126
COLHIDDEN PROPERTY ...................................................................... 128
COLIMAGELIST PROPERTY ............................................................... 128
COLINDEX PROPERTY ......................................................................... 130
COLISVISIBLE PROPERTY ................................................................... 131
COLKEY PROPERTY ............................................................................. 131
COLPOS PROPERTY ............................................................................. 132
COLPOSITION PROPERTY ................................................................. 132
COLS PROPERTY ................................................................................ 133
COLDROPDOWNPROPERTY ................................................................ 133
COLSORT PROPERTY .......................................................................... 134
COLWIDTH PROPERTY .......................................................................... 134
COLWIDTHMAX PROPERTY .................................................................. 135
COLWIDTHMIN PROPERTY .................................................................... 135
COMBOCOUNT PROPERTY ................................................................... 136
COMBODATA PROPERTY ..................................................................... 136
COMBOINDEX PROPERTY ..................................................................... 137
COMBOITEM PROPERTY ....................................................................... 137
COMBOSELECTPROPERTY ................................................................... 137
COMBOSELECTPROPERTY ................................................................... 138
COMBOSELECTPROPERTY ................................................................... 139
COMPARE EVENT ................................................................................ 140
DATAMEMBER PROPERTY .................................................................... 140
DATAMODE PROPERTY ........................................................................ 141
DATAREFRESH METHOD ....................................................................... 142
DATASOURCE PROPERTY ....................................................................... 142
DRAGROW METHOD .............................................................................. 143
DRAWCELL EVENT .............................................................................. 143
EDITABLE PROPERTY ........................................................................... 144
EDITCELL METHOD ............................................................................. 145
EDITMASK PROPERTY .......................................................................... 145
EDITMAXLENGTH PROPERTY ............................................................... 147
EDITSELECTLENGTH PROPERTY ........................................................ 147
EDITSELECTSTARTPROPERTY .............................................................. 148
EDITSELECTTEXT PROPERTY ............................................................... 148
EDITTEXT PROPERTY ........................................................................... 149
EDITWINDOW PROPERTY ....................................................................... 150
ELLIPSIS PROPERTY ............................................................................ 150
ENTERCELL EVENT ............................................................................. 151
ERROR EVENT (VSFLEXGRID CONTROL) .............................................. 151
EXPLORERBAR PROPERTY ................................................................... 151
EXTENDLASTCOL PROPERTY ............................................................. 152
FILLSTYLE PROPERTY .......................................................................... 153
FILTERDATA EVENT ............................................................................ 153
FINDROW PROPERTY ........................................................................... 154
FIXEDALIGNMENT PROPERTY ............................................................ 156
FIXEDDCOLS PROPERTY ....................................................................... 156
PRINTGRID METHOD ................................................................. 191
REDRAW PROPERTY ................................................................. 193
REMOVEITEM METHOD ............................................................ 193
RIGHTCOL PROPERTY ............................................................... 194
ROW PROPERTY (VSFLEXGRID CONTROL) ................................. 194
ROWCOLCHANGE EVENT .......................................................... 195
ROWDATA PROPERTY ............................................................... 195
ROWHEIGHT PROPERTY ........................................................... 196
ROWHEIGHTMAX PROPERTY ..................................................... 196
ROWHEIGHTMIN PROPERTY ...................................................... 196
ROWHIDDEN PROPERTY ............................................................ 197
ROWISVISIBLE PROPERTY ......................................................... 197
ROWOUTLINELEVEL PROPERTY .................................................. 198
ROWPOS PROPERTY ................................................................. 198
ROWPOSITION PROPERTY ......................................................... 198
ROWS PROPERTY ........................................................................ 199
ROWSEL PROPERTY ................................................................. 199
ROWSTATUS PROPERTY ............................................................ 200
SAVEGRID METHOD ................................................................. 200
SCROLLBARS PROPERTY ............................................................ 201
SCROLLTIPS PROPERTY ............................................................. 202
SCROLLTIPTTEXT PROPERTY ...................................................... 203
SCROLLTRACK PROPERTY .......................................................... 203
SELCHANGE EVENT ................................................................. 203
SELECT METHOD (VSFLEXGRID CONTROL) ............................... 204
SELECTEDROW PROPERTY ........................................................ 204
SELECTEDROWS PROPERTY ....................................................... 205
SELECTIONMODE PROPERTY ...................................................... 205
SHEETBORDER PROPERTY ......................................................... 206
SHOWCOMBOBUTTON PROPERTY .............................................. 207
SORT PROPERTY (VSFLEXGRID CONTROL) .................................. 207
STARTEDIT EVENT ................................................................. 210
STARTPAGE EVENT ................................................................. 211
SUBTOTAL METHOD ................................................................. 212
SUBTOTALPOSITION PROPERTY ............................................... 214
TABBEHAVIOR PROPERTY ......................................................... 215
TEXT PROPERTY (VSFLEXGRID CONTROL) .................................. 216
TEXTARRAY PROPERTY ............................................................. 216
TEXTMATRIX PROPERTY .......................................................... 217
TEXTSTYLE PROPERTY ............................................................. 217
TEXTSTYLEFIXED PROPERTY ..................................................... 218
TOPROW PROPERTY ............................................................... 218
TREECOLOR PROPERTY ............................................................ 219
VALIDATEDEEDIT EVENT .......................................................... 219
VALUE PROPERTY ................................................................. 219
VALUEMATRIX PROPERTY ......................................................... 220
VERSION PROPERTY (VSFLEXGRID CONTROL) ......................... 220
VIRTUALDATA PROPERTY ......................................................... 221
WALLPAPER PROPERTY ............................................................ 222
WALLPAPERALIGNMENT PROPERTY ......................................... 223
VSDATAOBJECT CONTROL ....................................................... 224
CLEAR METHOD (VSDATAOBJECT CONTROL) ......................... 224
FILECOUNT PROPERTY ........................................................... 224
FILES PROPERTY ................................................................. 224
GETDATA PROPERTY ............................................................. 224
Introduction

Welcome to VideoSoft VSFlexGrid® Pro 7.0®.

VSFlexGrid Pro 7.0 includes VSFlexGrid, a full-featured grid control and VSFlexString, a powerful regular expression engine.

VSFlexGrid Pro incorporates the latest data-binding technologies -- ADO 2.1 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 7.0 section of the documentation.

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

We want every programmer to get copies of our products and try them for as long as it is necessary to ensure the product meets his or her needs. 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 7.0. If you have any suggestions or ideas for new features that you would like to see included in a future version, or ideas for new controls, please call or write us at:

VideoSoft
5900-T Hollis Street
Emeryville, CA 94608
510-595-2400 (phone)
888-ACTIVEX (888-228-4839) - Toll-Free US & Canada)
510-595-2424 (fax)
http://www.videosoft.com
Overview

The VideoSoft VSFlexGrid Pro 7.0 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. VSFlexGrid gives you several choices of data binding: ADO/OLEDB, DAO, bind to 2-D or 3-D arrays, to other VSFlexGrid controls, create your own data source classes, or use the grid in unbound mode. It will read and write grids to a compressed binary file or to a text file (compatible with Microsoft Access and Excel). VSFlexGrid provides all the basics plus advanced features such as outline trees, sorting, cell merging, masked editing, translated combo and image lists, and automatic data aggregation.

**VSFlexString**
A flexible regular expression engine. It features pattern matching as well as regular expression text matching. VSFlexString's automatic replace capability immediately replaces all matches with the new assigned string. Tag matching capabilities determine which parts of the string matched what parts of the pattern.
Installing VSFlexGrid Pro 7.0

To install VideoSoft VSFlexGrid Pro 7.0, use the SETUP.EXE utility that you obtained electronically, or that was 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, or provided to you if you purchased the product on-line) 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.

Note: Always use the SETUP.EXE utility to install VSFlexGrid Pro 7.0 on new computers. If you simply copy the OCX files and register them, any applications you create on the new computer will be unlicensed, and will display a VideoSoft banner when you run them.

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

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSFLEX7.HLP</td>
<td>This file contains the VSFlexGrid Pro 7.0 online help topics.</td>
</tr>
<tr>
<td>VSFLEX7.CNT</td>
<td>This file contains the VSFlexGrid Pro 7.0 online help contents.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSFLEX7.OCX</td>
<td>This file contains the VSFlexGrid 7.0 control with ADO/OLEDB data-binding.</td>
</tr>
<tr>
<td>VSFLEX7D.OCX</td>
<td>This file contains the VSFlexGrid 7.0 control with DAO data-binding.</td>
</tr>
<tr>
<td>VSFLEX7L.OCX</td>
<td>This file contains the VSFlexGrid 7.0 control with no data-binding support.</td>
</tr>
<tr>
<td>VSSTR7.OCX</td>
<td>This file contains the VSFlexString 7.0 control (this control used to be part of the VSFLEX6.OCX file).</td>
</tr>
</tbody>
</table>

The following folders will be created by the setup utility:

<table>
<thead>
<tr>
<th>Folder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS</td>
<td>Main VideoSoft folder to store VideoSoft control information.</td>
</tr>
<tr>
<td>VS\VSFLEX</td>
<td>Contains sample Visual Basic projects, utilities, and the README.TXT file which discusses version specific information.</td>
</tr>
</tbody>
</table>

Installing a demo version

If you wish to try VideoSoft VSFlexGrid Pro 7.0 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 7.0, use the UNSETUP.EXE utility provided on the installation CD or diskettes. The UNSETUP.EXE utility will remove all VSFLEX7 files from your \Windows\System directory.
Upgrading From Previous Versions

Visual Basic projects that use VSFlexGrid 6.0 may be upgraded to VSFlexGrid 7.0 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.

CONVERT 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 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 VSDDataObject. These changes were made to avoid conflicts with VSFLEX6 projects. Both versions of the control may coexist on the same computer.

**Some Event names and parameter lists have changed**
The old Scroll event has been replaced by BeforeScroll and AfterScroll events.
New Features in VSFlexGrid Pro 7.0

This section summarizes the new features in VideoSoft VSFlexGrid Pro 7.0. 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.

**Light Version**

VSFlexGrid 7.0 control ships in three versions that support different types of data-binding. The choice is yours: Use the ADO/OLEDB version for compatibility with the latest database standards, the DAO version for compatibility with existing systems and projects, or the new Light version if you don't need database support and want to minimize the dependencies on your projects. Also, the VSFlexString control has been repackaged in a separate file, VSSTR7.OCX. This reduces the footprint of projects that use only one of the controls.

**Stand-Alone Printing**

The new PrintGrid method allows you to print the grid with a single statement. You have control over paper orientation, margins, and footer text, or show a dialog to let the user select and set up the printer. Printing events allow you to control page breaks, add repeating header rows, or add custom elements to each page.

**Frozen Rows and Columns**

The new FrozenRows and FrozenCols properties allow you to "freeze" rows and columns on the grid so they remain visible as the user scrolls the control, yet can be selected and edited. The appearance of the frozen cells is customizable, and the AllowUserFreezing property allows users to expand and shrink the frozen area with the mouse.

**WallPaper**

The new WallPaper property allows you to use any picture as a backdrop for your grids. Add logos or textures to enhance the appearance of your applications.

**Custom Data Sources**

The new FlexDataSource property allows you to bind the grid to your own data source objects, which you can create with Visual Basic or Visual C++. Creating your own data sources is easy and gives you the ultimate flexibility. Bind grids to existing data structures, use calculated "virtual" fields, perform data validation, or load data on-demand.

**Translated ImageLists**

The new ColImageList property allows you to bind grid columns to image lists. The value on each cell is used as an index into the image list, and the corresponding picture is displayed. This is an easy and efficient way to display database information graphically.

**Refer to Columns by Key**

The new ColKey and ColIndex properties allow you to refer to columns by name instead of position. Column keys are automatically assigned to field names when the grid is data-bound, or you may assign them with code. You can later refer to a column using a ColIndex(ColKey) syntax, which will retrieve the column you want even if the user has moved it to a different position on the grid.

**Range Aggregates**

The new Aggregate property allows you to calculate totals, averages, and other statistics for ranges of cells. It is quick and takes only a single statement.

**Easy ComboLists**

The new BuildComboList method allows you to build strings for use with the ColComboList property. Automatically create translated lists with multiple fields directly from any ADO or DAO recordset.

**Better User-Interface**

Several properties and settings have been added to make the VSFlexGrid control even easier to use:
- The new **ComboSearch** property makes combos and lists boxes smart, giving them select-as-you-type capability.
- The new **AutoSizeMouse** property makes columns resize to fit when the user double-clicks on the column resize area.
- The new **CellButtonPicture** property allows you to show images on cell buttons.
- The new **AutoSearchDelay** property allows you to customize the reset time for **AutoSearch** operations.
- The new **NodeOpenPicture** and **NodeClosedPicture** properties allow you to customize the look of outline trees.
- The **ExplorerBar** property has new settings that show sort icons on column headers, and better feedback while moving columns. The new ExplorerBar is easier to use: when sorting, it shows sorting triangle glyphs; when moving columns, it clearly shows the new column position and scrolls the control to allow the column to be moved anywhere.
- The **Editable** property has a new setting that allows users to start editing by double-clicking a cell.
- The **Redraw** property has a new setting that allows buffered painting for an even smoother appearance.
- The **GridLinesFixed** property has a new setting that makes fixed cells highlight the current selection (Excel-style).
- The **AllowUserResizing** property has a new setting that resizes all rows simultaneously (also resize groups of columns and rows).
- The **ComboList** syntax has been extended to provide control over the appearance of the default field.
- The **OLEDragDrop Data** object has been enhanced to the grid to be used as a source for files.

**More Control through New Events**
Several new events have been added to give you total control over the behavior of the **VSFlexGrid** control:
- The new **FilterData** event allows you to perform data formatting/conversions as data is being read from/written to bound recordsets.
- The new **CellChanged** event allows you to provide conditional formatting (based on cell contents) and display dynamic data summaries.
- The new **Before/AfterCollapse** events allow you to control outline trees and even build them asynchronously.
- The new **Before/AfterSelChange** events allow you to create protected ranges that cannot be selected.
- The new **Before/AfterScroll** events give you control over scrolling.

**Other new Properties and Methods**
- The new **EditWindow** property determines whether the grid is in edit mode and gives you access to the editor window handle.
- The new **ClipSeparators** property specifies which the separator characters used with the **Clip** property, **AddItem**, **LoadGrid**, and **SaveGrid** methods.
- The new **LoadArray** method loads data from Variant arrays or other VSFlexGrid controls without binding.
- The new **GetOutlineNode** method gets a row's parent, first, or last child nodes in a tree outline.
- The new **ShowCell** method brings arbitrary cells into view.
- The new **GetSelection** method returns the selection in order for use as loop limits.

**Better Design-Time Interface**
New and improved property pages allow you to set properties with ease. Specify fixed-cell contents, column widths and alignments, data types, edit masks and combo lists visually at design time. Alternatively, use the Visual Basic property browser to see a categorized view of the **VSFlexGrid** properties.

**Better Documentation**
We have invested a lot of time and effort to make the **VSFlexGrid** documentation complete and comprehensive, including more sample code and details on the use of each property,
event and method. We have also developed several new samples in Visual Basic and in Visual C++ to illustrate the main aspects of the control. The samples include a generic, categorized property page object written in Visual C++ that we use for our own controls and you can use as-is or customize to fit your needs.
Product Support

Product support for VideoSoft VSFlexGrid Pro 7.0 is available to licensed users through the following channels:

Home Page: http://www.videosoft.com
E-Mail support: support.vsflex@videosoft.com
Regular 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 7.0 on our Web page (http://www.videosoft.com). Before calling for technical support, please identify the version of VideoSoft VSFlexGrid Pro 7.0 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 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.
VSFlexGrid Introduction

The VSFlexGrid control is a powerful, full-featured grid. It provides new ways to display, edit, format, organize, summarize, and print tabular data.

The VSFlexGrid control was designed to be used either in unbound mode, where the grid "owns" the data, or in data-bound mode, where the grid is used to view and edit data that belongs to a database.

VSFlexGrid Pro 7.0 ships with four versions of the VSFlexGrid control. Each version has its own OCX file and supports a different type of data binding:

<table>
<thead>
<tr>
<th>OCX File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSFLEX7.OCX</td>
<td>This version provides support for ADO/OLEDB data binding, the latest data binding standard from Microsoft. It requires ADO to be installed on the computer.</td>
</tr>
<tr>
<td>VSFLEX7D.OCX</td>
<td>This version provides support for DAO data binding, the more traditional type of data binding supported by Visual Basic's built-in Data Control.</td>
</tr>
<tr>
<td>VSFLEX7L.OCX</td>
<td>This version has no support for data-binding. Use it when your application does not require data-binding and you want to reduce the number of components your application depends on.</td>
</tr>
<tr>
<td>VSFLEX7U.OCX</td>
<td>This is a Unicode version that provides support for ADO/OLEDB data binding. Note that this version can only be used on the Windows NT platform.</td>
</tr>
</tbody>
</table>

The following sections walk you through the main features in the VSFlexGrid control:

**Basic Operations**
Describes how to set up the grid dimensions and layout, and the concepts of "current cell" and "selection".

**Editing Cells**
Describes how to implement simple text editing, drop-down lists and combo lists, cell buttons, editing masks, and data validation.

**Formatting Cells**
Describes how to customize the appearance of the grid by formatting numbers, dates, and boolean values, or by changing fonts, colors, alignment, and pictures for individual cells or ranges.

**Merging Cells**
Describes how to change the grid display so that cells with similar contents are merged, creating "grouped" views that highlight relationships in the data.

**Outlining and Summarizing**
Describes how to add subtotals to grids and how to build outline trees.

**Data Binding (ADO and DAO)**
Discusses the basic aspects of ADO/OLEDB and DAO data-binding.

**Other types of Data Binding**
Discusses how you can bind the VSFlexGrid control to arrays of Variants, to other VSFlexGrid controls, or to custom data sources that you can implement yourself.

**Saving, Loading, and Printing**
Describes how you can save the contents or formatting of a grid and re-load it later, or exchange grid data with other applications such as Microsoft Access and Excel. This section also shows how you can print grids.

**Using VSFlexGrid in Visual C++**
Shows useful techniques and programming tips on how to use the VSFlexGrid control in Visual C++, both in MFC and in ATL projects.

**Using VSFlexGrid in Visual J++**
Shows programming tips on how to use the VSFlexGrid control in Visual J++.

**VSFlexGrid Property Groups**
 Presents a map of the main VSFlexGrid properties cross-referenced by function.
Basic Operations

The VSFlexGrid control has two properties that determine its dimensions: **Rows** and **Cols**. When used in bound mode, these properties are set automatically based on how much data is available on the data source. In unbound mode, you can set them to arbitrary values.

There are two basic types of rows and columns: **fixed** and **scrollable**. Fixed rows remain on the top of the grid when the user scrolls the grid vertically, and fixed columns remain on the left of the grid when the user scrolls the grid horizontally. Fixed cells are useful for displaying row and column header information. They cannot be selected or edited by the user. The number of fixed rows and columns is set by the **FixedRows** and **FixedCols** properties.

The **AllowUserResizing** property allows the user to resize rows and columns by dragging the edges of the fixed cells. The **ExplorerBar** property allows the user to move and sort columns by clicking and dragging the header rows.

### Cursor

The grid has a **cursor**, which is the cell defined by the **Row** and **Col** properties. The cursor displays a focus rectangle while the grid is active. The user may move the cursor with the keyboard or the mouse, and edit the contents of the cell if the grid is editable. Changing the **Row** and **Col** properties in code also moves the cursor.

The **Row** property may be set to values between zero and **Rows** - 1 to select a row, or to -1 to hide the cursor. The **Col** property may be set to values between zero and **Cols** - 1 to select a column, or to -1 to hide the cursor.

Setting the **Row** and **Col** properties does not ensure that the new cursor is visible. For that, use the **ShowCell** method.

### Selection

The grid also has a **selection**, which is a rectangular range of cells defined by two opposing corners: the cursor (**Row**, **Col** properties) and the cell defined by the **RowSel** and **ColSel** properties. The user may change the selection using the keyboard or the mouse. Changing the **RowSel** and **ColSel** properties in code also changes the selection.

When the **Row** or **Col** properties change, **RowSel** and **ColSel** are automatically reset and the selection collapses into a single cell, the cursor. To create selection in code, either set **RowSel** and **ColSel** after setting **Row** and **Col**, or use the **Select** method.

Several grid properties apply to the cursor or to the selection, depending on the setting of the **FillStyle** property. These include **Text** and all properties with names that start with **Cell** (**CellBackColor**, **CellForeColor**, etc.)

### Variations

For some applications, only a cursor makes sense, and no selection. In these cases, set the **AllowSelection** property to False. This will prevent the user from extending the selection with the keyboard or the mouse.

Some applications require selections to be made row by row, list-box style. In these cases, set the **SelectionMode** property to **flexSelectionListBox** (3) and retrieve the selected status for each row through the **IsSelected()** property.
Editing Cells

The main property related to editing is the Editable property. You must set it to a non-zero value to allow users to edit the contents of the grid. The flexEDKbd (1) setting allows users to start editing a cell by typing into it. The flexEDKbdMouse (2) setting also allows users to double-click a cell to start editing it.

Editing Text, Lists, and Combos

Once you set the Editable property to a non-zero value, users may edit text by selecting a cell and then typing into it. This is the most basic type of editing. Often, the user's choices will be limited to a list of possible values. In these cases, you can let them select the choice from a drop-down list. To do this, build a string containing all the choices separated by pipe characters (e.g. "True|False|Don't know") and assign it to the ColComboList() property of the column for which the choices apply. Each column may have a different list. After this, the grid will display an arrow next to the cell. When the user clicks the arrow or presses a key, the list will drop down and offer the choices available.

A third option is one where there are typical values for a cell, but the user should be allowed to type something else as well. This can be accomplished with drop-down combos, a combination of text box and drop-down list. To create combos, just start the choice list with a pipe character (e.g. ",|True|False|Don't know"), then assign it to the ColComboList as before.

In some cases, cells in the same column may need different lists. For example, a property list may show properties on the first column and their values on the second. The values depend on the property, so valid choices change from one row to the next. In these cases, you should trap the BeforeEdit event and set the ComboList property to the appropriate list for the current cell.

Cell Buttons

Certain types of cell may require sophisticated editors other than text boxes or choice lists. For example, if a cell contains a file name or a color, it should be edited with a dialog box. In these cases, set the ColComboList property to an ellipsis ("..."). The control will display a button next to the cell and will fire the CellButtonClick event when the user clicks on it. You can trap the event, show the dialog, and update the cell's contents with the user's selection.

Masks

The VSFlexGrid control supports masked editing, where an input mask is used to automatically validate the input as the user types. This is done through the ColEditMask property, which takes a string that defines what characters are valid for each input position. Masks may be used with regular text fields and with drop-down combo fields.

Mask strings have two types of characters: literal characters, which become part of the input, and template characters, which serve as placeholders for characters belonging to specific categories (e.g. digits or alphabetic). For example, you could use a mask like "(999) 999-9999" for entering phone numbers (the digit "9" is a placeholder that stands for any digit). For details on the syntax used to build the mask strings, see the EditMask property in the control reference section.

If different cells in the same column need different masks, trap the BeforeEdit event and set the EditMask property to an appropriate value for the current cell.

Validation

In many cases, edit masks are not enough to ensure that the data enters by the user was valid. For example, a mask won't let you specify a range of possible values, or validate the current cell based on the contents of another cell. In these cases, trap the ValidateEdit event and see if the value contained in the EditText property is a valid entry for the current
cell (at this point, the Text property still has the original value in it). If the input is invalid, set the Cancel parameter to True and the grid will remain in edit mode until the user types a valid entry.

**Controlling Edit Mode**

You can determine whether the grid is in edit mode by reading the value of the EditWindow property. If this property returns zero, the grid is not in edit mode. If it returns a non-zero value, the grid is in edit mode and the value returned is the handle of the edit window.

You can force the grid into edit mode at any time using the EditCell method. You can even allow the user to edit fixed cells by selecting them in code (using the Select method) and then invoking the EditCell method. You can cancel the edit mode by selecting the current cell (e.g. .Select .Row, .Col).
Formatting Cells

One of the main strengths of the VSFlexGrid control is the ability to customize almost every aspect of the appearance of the entire grid and individual cells. There are properties that affect the whole grid, such as Font, BackColor, ForeColor, and GridLines. Others are specific to rows and columns, such as RowHeight(), RowHidden(), ColWidth(), ColAlignment(), and ColFormat(). Finally, the Cell property allows you to format arbitrary ranges and individual cells.

Formatting cell contents

The ColFormat() property controls how cell contents are formatted for display. It takes as a parameter a format string similar to the one used by Visual Basic's Format function. For example, you may set ColFormat() to "#.###,#", "Currency", or "Long Date". This property does not affect the cell's contents, only the way it is displayed. You may also use check boxes to display boolean values. To do this, set the ColDataType() property to flexDTBoolean. The grid will automatically display check boxes and handle them if the grid is editable.

Formatting cell appearance

By default, the VSFlexGrid control will align strings to the left and numbers and dates to the right of each column. You may override this default using the ColAlignment() property.

For the ultimate in cell formatting control, use the Cell property. This property allows you to set or retrieve every aspect of a range's formatting. You may set a cell's contents, font, back color, fore color, alignment, and picture, among other options.

If you need even more control over the appearance of the cells, use the OwnerDraw property and the DrawCell event to paint the cell yourself, using Windows API calls.

Conditional formatting

To format cells based on their contents, trap the CellChanged event and apply the formatting using the Cell property. For example, you can make negative values red and bold or give values above a certain threshold a blue background.
Outlining and Summarizing

The VSFlexGrid control has methods and properties that allow you to summarize data and display it in a hierarchical manner. To summarize data, use the Subtotal method. To display hierarchical views of the data, use the OutlineBar and OutlineCol properties.

Creating Subtotals

The Subtotal method adds subtotal rows that contain aggregate data for the regular (data) rows.

Subtotal supports hierarchical aggregates. For example, you may call it several times in a row using different parameters to get sales figures by Product, Region, and Salesperson. You may also calculate aggregates other than sums (e.g. averages or percentages) and format the subtotal rows to highlight them.

For example, assuming you had a VSFlexGrid control named fg containing Product, Region, Salesperson, and Sales information, you could summarize it with the following code:

```
' clear existing subtotals
fg.Subtotal flexSTClear
' get an Grand total (use 1 instead of columns index)
fg.Subtotal flexSTSum, -1, 3, , 1, vbWhite, True
' total per Product (column 0)
fg.Subtotal flexSTSum, 0, 3, , vbRed, vbWhite, True
' total per Region (column 1)
fg.Subtotal flexSTSum, 1, 3, , vbBlue, vbWhite, True
' show an OutlineBar on column 0
fg.OutlineBar = flexOutlineBarSimple
```

After executing this code, the grid would look like this:

<table>
<thead>
<tr>
<th>Product</th>
<th>Region</th>
<th>Associate</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand Total</td>
<td></td>
<td></td>
<td>1,736,402</td>
</tr>
<tr>
<td></td>
<td>Total Drums</td>
<td></td>
<td>161,556</td>
</tr>
<tr>
<td></td>
<td>Drums</td>
<td>Total East</td>
<td>18,866</td>
</tr>
<tr>
<td></td>
<td>Drums</td>
<td>Total North</td>
<td>45,342</td>
</tr>
<tr>
<td></td>
<td>Drums</td>
<td>Total South</td>
<td>47,674</td>
</tr>
<tr>
<td></td>
<td>Drums</td>
<td>South</td>
<td>45,342</td>
</tr>
<tr>
<td></td>
<td></td>
<td>John</td>
<td>45,342</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mike</td>
<td>2,532</td>
</tr>
<tr>
<td></td>
<td>Total Flutes</td>
<td></td>
<td>262,511</td>
</tr>
<tr>
<td></td>
<td>Flutes</td>
<td>Total East</td>
<td>47,975</td>
</tr>
<tr>
<td></td>
<td>Flutes</td>
<td>East</td>
<td>4,543</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sylvia</td>
<td>43,432</td>
</tr>
<tr>
<td></td>
<td>Flutes</td>
<td>Total North</td>
<td>75,877</td>
</tr>
<tr>
<td></td>
<td>Flutes</td>
<td>North</td>
<td>75,877</td>
</tr>
</tbody>
</table>

The subtotal rows created by the Subtotal method differ from regular rows in three aspects:

1) Subtotal rows can be automatically removed by invoking the Subtotal method with the flexSTClear parameter. This is useful to provide dynamic views of the data, where the user may move columns and re-sort the data, making it necessary to recalculate the subtotals.

2) Subtotal rows can be used as nodes in an outline, allowing you to collapse and expand groups of rows to present an overview of the data or to reveal its details. To see the outline tree, you need to set the OutlineBar property to a non-zero value. Because the outline is a
Hierarchical structure, each row has a level that defines how deep into the outline the node is. This level can be set or retrieved through the `RowOutlineLevel()` property.

3) When the grid is bound to a data source, the subtotal rows do not correspond to actual data. Thus, if you navigate the recordset using the `MoveFirst` and `MoveNext` methods, the subtotal rows will be skipped.

The picture above shows the subtotals and the outline tree next to the data on the first column. The outline tree allows users to collapse and expand sections of the grid by clicking on the nodes, and can be very useful to display other types of data, not only aggregates.

**Creating Outline Trees**

To create outline trees without using the `Subtotal` method, you need to follow these steps:

1) Populate the grid.
2) Turn some rows into outline nodes by setting their `IsSubtotal()` property to True.
3) Set each node’s level in the hierarchy by setting their `RowOutlineLevel()` property. Higher values mean the node is deeper (more indented) into the outline tree.

For example, the code below creates a custom (and somewhat random) outline:

```vbnet
' initialize grid
fg.Rows = 1: fg.FixedRows = 1
fg.Cols = 3: fg.FixedCols = 0
fg.OutlineBar = flexOutlineBarSimpleLeaf
fg.GridLines = flexGridNone
fg.FormatString = "Heading |Date |Time"

' fill the control with data
Dim i%, j%
While fg.Rows < 150
    ' decide randomly whether to add a subtotal
    If fg.Rows <= 2 Or Rnd() < 0.4 Then
        ' add an item, make it a subtotal
        fg.AddItem "Branch Level " & i
        fg.IsSubtotal(fg.Rows - 1) = True
        fg.RowOutlineLevel(fg.Rows - 1) = i
        fg.Cell(flexcpPicture, fg.Rows - 1, 0) = imgFolder
        j = i
        ' decide whether to go deeper or shallower
        If Rnd() < 0.5 And i < 10 Then
            i = i + 1
        ElseIf Rnd() < 0.5 And i > 0 Then
            i = i - 1
        End If
    Else
        ' add a regular item
        fg.AddItem "Data on Level " & j & vbTab & _
                    Date & vbTab & Time
        fg.Cell(flexcpPicture, fg.Rows - 1, 0) = imgItem
    End If
Wend

' do an autosize
fg.AutoSize 0, 1, , 300
```
This code creates a grid that looks like this:

<table>
<thead>
<tr>
<th>Heading</th>
<th>Date</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Level 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data on Level 1</td>
<td>22/10/1999</td>
<td>16</td>
</tr>
<tr>
<td>Data on Level 1</td>
<td>22/10/1999</td>
<td>16</td>
</tr>
<tr>
<td>Data on Level 1</td>
<td>22/10/1999</td>
<td>16</td>
</tr>
<tr>
<td>Branch Level 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Level 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch Level 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data on Level 1</td>
<td>22/10/1999</td>
<td>16</td>
</tr>
<tr>
<td>Data on Level 1</td>
<td>22/10/1999</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram showing a hierarchy with dates and titles.
Merging Cells

The VSFlexGrid control allows you to merge cells, making them span multiple rows or columns. This capability can be used to enhance the appearance and clarity of the data displayed on the grid. The effect of these settings is similar to the HTML "<ROWSPAN>" and "<COLSPAN>" tags.

Cell merging is controlled by three properties: MergeCells, MergeRow(), and MergeCol(). MergeCells enables cell merging for the entire grid. After it is set to a non-zero value, rows that have MergeRow() set to True will be allowed to merge across and columns that have MergeCol() set to true will be allowed to merge down. The merging will occur if adjacent cells contain the same non-empty string. The MergeCells property has several settings that allow you to restrict cell merging in different ways. For details on how to use it, see the MergeCells property in the control reference section.

Note that there is no method to force a pair of cells to merge. The merging is done automatically based on the cell contents. This makes it easy to provide merged views of sorted data, where values in adjacent rows present repeated data.

Cell merging has several possible uses. For example, you can use it to create merged table headers, merged data views, or grids where the text spills into adjacent columns.

**Merged table headers**

To create merged table headers, you must start by setting the MergeCells property to flexMergeFixedOnly. Then, designate the rows and columns that you want to merge using the MergeRow(), and MergeCol() properties. Finally, assign the text to the header cells so that the cells you want to merge have the same contents.

The code below shows an example:

```vba
Private Sub Form_Load()
    Dim i%

    ' initialize control
    fg.WordWrap = True
    fg.Cols = 9
    fg.FixedRows = 2
    fg.MergeCells = flexMergeFixedOnly

    ' create row headers
    fg.MergeRow(0) = True
    fg.Cell(flexcpText, 0, 1, 0, 4) = "North" ' four cells, will merge
    fg.Cell(flexcpText, 0, 5, 0, 8) = "South" ' four cells, will merge
    For i = 1 To 4
        fg.Cell(flexcpText, 1, i, 1) = "Qtr " & i
        fg.Cell(flexcpText, 1, i + 3, 1) = "Qtr " & i
    Next

    ' create column header
    fg.MergeCol(0) = True
    fg.Cell(flexcpText, 0, 0, 1, 0) = "Sales by Product" ' two cells, will merge

    ' align and autozise the cells
    fg.Cell(flexcpAlignment, 0, 0, 1, fg.Cols - 1) = flexAlignCenterCenter
    fg.AutoSize 1, fg.Cols - 1, False, 300

    End Sub
```

This is the result:
Merged data views
To create merged data views, you start by populating the grid. For example, you may bind it to a database. Then, set the **MergeCells** property to **flexMergeRestrictAll** and **MergeCol(-1)** to **True**. Notice that the -1 index means "apply this setting to all columns".

The code below shows an example. It uses an ADO/OLEDB VSFlexGrid control named **fg** and a Microsoft ADO Data Control named **Adodc1**:

```
Private Sub Form_Load()
    ' create the data source
    Adodc1.ConnectionString = "DSN=NorthWind"
    Adodc1.CommandType = adCmdText
    Adodc1.RecordSource = "SELECT Country, City, CompanyName FROM Customers;"
    Adodc1.Refresh
    Adodc1.Recordset.Sort = "Country, City"

    ' populate the grid
    Set fg.DataSource = Adodc1

    ' activate merging for all columns
    fg.MergeCells = flexMergeRestrictAll
    fg.MergeCol(-1) = True

End Sub
```

This is the result:

<table>
<thead>
<tr>
<th>Country</th>
<th>City</th>
<th>CompanyName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argentina</td>
<td>Buenos Aires</td>
<td>Cactus Comidas para llevar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Oceano Atlantico Ltda.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rancho grande</td>
</tr>
<tr>
<td>Austria</td>
<td>Graz</td>
<td>Emit Handel</td>
</tr>
<tr>
<td></td>
<td>Salzburg</td>
<td>Piccolo und mehr</td>
</tr>
<tr>
<td>Belgium</td>
<td>Bruxelles</td>
<td>Maison Dewey</td>
</tr>
<tr>
<td></td>
<td>Charleroi</td>
<td>Supènes délices</td>
</tr>
<tr>
<td>Brazil</td>
<td>Campinas</td>
<td>Gourmet Lanchonettes</td>
</tr>
<tr>
<td></td>
<td>Resende</td>
<td>Wellington Importadora</td>
</tr>
<tr>
<td></td>
<td>Rio de Janeiro</td>
<td>Hanari Carres</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Que Delicia</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ricardo Adociados</td>
</tr>
<tr>
<td></td>
<td>São Paulo</td>
<td>Comério Mineiro</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Familia Aquibaldo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queen Cozinha</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Spilling Text
The **MergeCells** property has one setting that operates differently from the others. The `flexMergeSpill` setting causes text that is too long to fit in a cell to spill into empty adjacent cells. This setting does not require you to set the `MergeRows()` or `MergeCols()` properties.

For example, the picture below shows what a grid might look like when **MergeCells** is set to `flexMergeSpill` and the user types entries of varying lengths:
Saving, Loading, and Printing

The VSFlexGrid control has methods that allow you to read and write grids to disk files using the `LoadGrid` and `SaveGrid` methods, or print them using the `PrintGrid` method. When saving grids to files, you have several options: save the formatting, the data, or both; use text or binary files; or archive several grids into a single compressed file.

**Saving Grids**
The `SaveGrid` method allows you to save a grid to a disk file. The `SaveGrid` method is very fast and flexible. It has arguments that allow you to save the whole grid, only the data, or only the formatting. If you save the whole grid, you can restore it later and it will look exactly as it did when you saved it. Save only the data to get a more compact file. Or save only the formatting to get a template that you can load into other grids without affecting their data.

You may also save grids into comma or tab-delimited text files, which allows you to load them into other applications such as Microsoft Excel or Access. For added flexibility, you can use the `ClipSeparators` property to select arbitrary delimiters, such as pipes or semi-colons. Saving in text mode saves only the grid data.

**Loading Grids**
The `LoadGrid` method allows you to load grid files that were saved with the `SaveGrid` method. You can also use it to import data from text files created by other applications such as Microsoft Excel or Access.

**Creating Archives**
Many applications need to save several grids (or tables) rather than one. In these cases, the `Archive` method allows you to consolidate many files, optionally compressing them, into a single file. The `Archive` method has parameters that allow you to add, remove, or extract files from the archive. The `ArchiveInfo` property allows you to extract information from an archive file, such as the number of files in the archive, their names, sizes, compressed sizes, and dates. `Archive` and `ArchiveInfo` are not limited to files created with the `SaveGrid` method: they work with any file and could even be used to create a stand-alone compression utility.

Archive files created with the `Archive` method are not compatible with ZIP files, although their compression ratios are similar.

**Printing Grids**
You may print grids using the `PrintGrid` method. `PrintGrid` allows you to specify paper orientation, margins, and a footer, or show a printer setup dialog box and allow the user to select the printer, paper orientation, etc. While the document is printing, the `PrintGrid` method fires several events that allow you to cancel the printing and annotate each page as it is created (`StartPage`), control page breaks (`BeforePageBreak`), or select specific rows and use them as page headers (`GetHeaderRow`).

If you need more sophisticated printing capabilities, such as print previewing or the ability to render several grids and other text and graphical elements on a single document, you should use the VSPrinter control (available separately from VideoSoft). The VSPrinter control has a `RenderControl` property that allows you to render grids on documents along with other data. VSPrinter documents can be previewed, printed, or saved to files.
Data Binding (ADO and DAO)

Data-binding is a process that allows one or more data consumers to be connected to a data provider in a synchronized manner. For example, if you move the cursor on a data-bound grid, other controls connected to the same data source will change to reflect the new current record.

There are two main types of data binding: OLEDB/ADO and DAO. OLEDB is the latest Microsoft standard, and is becoming increasingly popular. It provides access to data from many different sources, and was designed to be OLE-compliant. DAO used to be the standard before OLEDB came along, and is still fairly popular.

The VSFlexGrid control has an OLEDB/ADO version (VSFLEX7.OCX) and a DAO version (VSFLEX7D.OCX), so it will work with your data regardless of your choice of database standard.

The VSFlexGrid control has three main properties and one event that control data binding: the DataSource, DataMode, and VirtualData properties, and the AfterDataRefresh event.

**DataSource property**
The DataSource property refers to a recordset which contains the data. In the OLEDB version, this property may be set at design time to refer to a data source such as the Microsoft ADO control, a DataEnvironment object, or a custom OLEDB data source class you create with Visual Basic. It may also be set at run time. In the DAO version, this property must be set at design time to refer to a DataControl (this control is built-into Visual Basic, rather than an OCX), and cannot be changed later. You may still change the data source, but only through the DataControl itself, and not through the DataSource property.

**DataMode property**
The DataMode property allows you to determine whether the control should simply read the data from the data source (flexDMFree setting) or whether it should be fully bound to the data source (flexDMBound setting). In the flexDMFree mode, the data is read from the data source and becomes property of the grid. If you change it, the changes will not be written back to the data source. If you move the cursor, other controls will not be affected. In the flexDMBound mode, the control is fully bound. Changes made to the grid will be reflected in the data source, and moving the cursor on the grid will cause other bound controls to synchronize and display the current record.

Note that in order to modify the data, the Editable property must be set to a non-zero value.

**VirtualData property**
The VirtualData property allows to determine whether the control should read the entire recordset at once (synchronously), or in small chunks, on an as-needed basis (asynchronously). If VirtualData is set to False, the entire recordset is read immediately. For large recordsets (over 1000 records or so), this can be time-consuming, so this setting is rarely used. If VirtualData is set to True, the data is read only when the control needs it (to display or edit, for example), in chunks of 100 rows at a time. The default value for the VirtualData property is True, and you should rarely have to change it.

**AfterDataRefresh event**
The AfterDataRefresh event is useful when the control is bound to a data source and you want to perform some operation on the data whenever it is refreshed. For example, you might want to display subtotals or add special formatting to certain columns or cells. Also, it is important to know that when the source recordset changes, all existing columns are destroyed and recreated from scratch. In this process, most column properties are reset.
to their default values. Thus, if you set up your columns using the ColEditMask, ColFormat, ColComboList, ColImageList, etc you should do it in response to the AfterDataRefresh event, not in the Form_Load event.
Other types of Data Binding

In addition to the traditional types of data binding described above, the VSFlexGrid control provides three new ways to connect the control to data sources. You may use the **BindToArray** method to connect the grid to a Variant array or to another VSFlexGrid control, or use the **FlexDataSource** property to connect the grid to a custom data source that you develop yourself.

**Binding to Variant arrays**

Binding to arrays is useful when the data you want to display is already stored in an array, so you don't have to copy it back and forth between the control and the array, when you want to connect several grids to a single array, or when you have three-dimensional arrays and want to use the VSFlexGrid to view the array "page by page". To bind the VSFlexGrid to an array, use the **BindToArray** method and pass a Variant array as the first parameter. The grid will display values from the array and automatically write any modifications back into the array. If you make changes to the array in code, however, you must call the grid's **Refresh** method to make them visible to the user.

The parameters on the **BindToArray** method allow you to control how the rows and columns map onto the array's dimensions, so you can easily transpose the array. The mapping always spans the entire array, however. 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 on the scrollable part of the control.

**Binding to other VSFlexGrid controls**

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, except the first parameter is a reference to another VSFlexGrid control.

When two controls are bound, changes made 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. Note that the binding only applies to the data, not to the formatting.

**Binding to a FlexDataSource**

For the ultimate in data-binding flexibility, the VSFlexGrid control allows you to create your own data source objects and assign them to the VSFlexGrid control, which will display the data and allow the user to interact with it.

The main advantages of data-binding through the **FlexDataSource** property are speed and flexibility. You should consider using the **FlexDataSource** property when you have large amounts of data stored in custom structures or objects (other than database recordsets). By using the **FlexDataSource** property, you may display and edit the data in-place. There is no need to copy it to the grid and save it back later. In fact, the data may even be mostly virtual, consisting of dynamically calculated values rather than static information. Plus, you have complete flexibility to format and filter the data in any way you want.

To qualify as a **FlexDataSource**, your object must implement the **IVSFlexDataSource** interface. This interface consists of five simple methods:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFieldCount()</td>
<td>Returns the number of fields in the data source.</td>
</tr>
<tr>
<td>GetRecordCount()</td>
<td>Returns the number of records in the data source.</td>
</tr>
<tr>
<td>GetFieldName(Fld&amp;)</td>
<td>Returns the name of field number Fld (ranging from zero to GetFieldCount() - 1).</td>
</tr>
<tr>
<td>GetData(Fld&amp;, Rec&amp;)</td>
<td>Returns the data in field Fld, record Rec (ranging from zero to GetRecordCount() - 1).</td>
</tr>
</tbody>
</table>
**SetData(Fld&, Rec&, Data$)**  Returns the data in field Fld, record Rec.

For more details and a sample implementation of a custom data source, refer to the FlexDataSource property in the control reference section.
Using VSFlexGrid in Visual C++

The VSFlexGrid control can be used in Visual C++ as well as in Visual Basic.

This part of the manual was written to help experienced C++ programmers get started using the VSFlexGrid control in VC++. If you don't know C++, MFC, or ATL, you may skip this section as it probably won't make much sense to you.

Until recently, using ActiveX controls in Visual C++ meant you had to use the MFC (Microsoft Foundation Classes) framework, because this was the only reasonable way to get the ActiveX hosting capabilities and Wizard support most programmers want. With the release of Microsoft Visual Studio 6, however, this situation has changed. You can now use ATL (Active Template Library) and native compiler support for COM to create projects that do not depend on MFC. In fact, even if your project is based on MFC you should take advantage of the native COM support to improve the performance of your applications.

Using VSFlexGrid in MFC projects

To use the VSFlexGrid control in MFC projects, you will normally follow these steps:

1) Create a new dialog-based MFC project.

2) Go to the resource editor, open the dialog, right-click on it, select "Insert ActiveX control", and pick the VSFlexGrid control from the list (if the grid is not on the list, it hasn't been registered on your computer).

3) Hold down the CTRL key and double-click on the grid. This will cause Developer Studio to generate wrapper classes through which you can interact with the control. When the wrapper classes are ready, select a name for the control (e.g. m_Grid).

4) From now on, things are pretty much the same as in VB. You can right-click on the control to implement event handlers, and access the controls properties and methods through the m_Grid variable. Most properties are exposed through GetPropertyName and SetPropertyName member functions. Unfortunately, enumerated values get translated into longs instead of their proper enumeration symbols, but that's a relatively minor inconvenience. (And one that can be avoided, read on).

If you take a look at the CVSFlexGrid wrapper generated by Developer Studio, you will see that the class is derived from CWnd. This means you can move, size, show, or hide the control as if it were a regular window. The wrapper class also has a handy Create function that lets you create new instances of the control. For example, if you add this declaration to the main dialog's header file:

    class CMyDlg : public Cdialog
    {
        // Construction
        public:
            CMyDlg(CWnd* pParent = NULL); // standard constructor
            CvsFlexGrid m_GridDynamic;     // new VSFlexGrid
    }

You can create the control by adding the following code to the dialog's OnInitDialog function:

    BOOL CMyDlg::OnInitDialog()
    {
        // Wizard-generated code
        . . .
        // TODO: Add extra initialization here
        // create a second instance of the VSFlexGrid control
    }
RECT rc;
GetClientRect(&rc);
InflateRect(&rc, -5, -5);
rc.left = (rc.left + rc.right) / 2;
m_GridDynamic.Create(NULL, WS_VISIBLE, rc, this, 100);

return TRUE;  // return TRUE
}

The problem with this second approach is that you have to hook up the event handlers manually. You can do this by copying the code created by the Wizard for the first control (it involves using several macros to define a "sink event"). Hooking up the events manually is not difficult, but it is a tedious and error-prone process. Unless you have a good reason to create the controls dynamically, you should stick to the resource editor and the Wizard.

This covers most of what you need to know about using ActiveX controls in MFC. There are a couple of issues that deserve additional explanation (at least our tech support department gets many questions on these): Handling optional parameters, Picture properties, and Dual interfaces.

Handling Optional Parameters in MFC
Optional parameters in COM interfaces are always of type VARIANT. To omit them in Visual Basic, you simply don't supply a value for them at all. The wrapper classes created by the MFC Wizard, however, require that you supply VARIANTS for all parameters, optional or not. In these cases, what you need to do is create a VARIANT of type VT_ERROR, and use that in place of the optional parameters.

For example, the VSFlexGrid control has an AutoSize method that takes three optional parameters. To invoke AutoSize omitting the optional parameters, you could write:

    COleVariant vtNone(0L, VT_ERROR);
m_Grid.AutoSize(1, vtNone, vtNone, vtNone);

Notice that the VARIANT is created with a 0L value instead of simply 0. This is required by the compiler to define whether the zero is a short or a long integer.

Handling Picture Properties in MFC
OLE Pictures are objects in their own right. They have methods that retrieve their size, type, and so on. As such, setting or retrieving Picture properties involves dealing with their IDispatch pointers (IDispatch is the basic type of Automation COM interface). The question is, how do I create one of these interfaces to give to the control? The easiest way is through an MFC helper class called CPictureHolder. This class, declared in the AFXCTL.H file, has methods that allow you to create and manage OLE pictures.

The code below shows how you can use the CPictureHolder class to set the VSFlexGrid control's CellPicture property:

```cpp
#include <afxctl.h> // declare CPictureHolder class
void CJnkDlg::OnButton1()
{
    // create CPictureHolder
    CPictureHolder pic;
    // load a picture from a bitmap resource
    pic.CreateFromBitmap(IDB_BITMAP1);
    // get the LPDISPATCH pointer
    LPDISPATCH pPic = pic.GetPictureDispatch();
    // assign LPDISPATCH to control
    m_Grid.SetCellPicture(pPic);
    // don't forget to release the LPDISPATCH
```
Besides setting the picture property, the code above illustrates an important point when dealing with COM interfaces. Notice how the LPDISPATCH pointer is obtained, used, and released. Failing to release COM pointers results in objects dangling in memory and wasting resources.

Some properties and methods require that you pass pictures in VARIANT parameters (e.g. the Cell property). To do this, initialize a ColeVariant as follows:

```cpp
COleVariant vPic;
V_VT(&vPic) = VT_DISPATCH;
V_DISPATCH(&vPic) = pic.GetPictureDispatch();
```

Notice that in this case you must not release the LPDISPATCH pointer, because the COleVariant destructor will do that automatically when vPic goes out of scope.

**Dual Interfaces in MFC**

The wrapper classes created by the MFC wizard are very helpful, and for a while they were the best you could get. With Visual Studio 6, however, the compiler has built-in COM support, including a different way to create wrapper classes for COM objects through the new #import compiler directive. These “native” wrapper classes are faster and more flexible than the ones generated by MFC:

The MFC wrappers are based on the IDispatch interface, so every method or property you access needs to pack its parameters into VARIANTS and go through a call to the Invoke method. The native wrappers, by contrast, take advantage of dual interfaces to access properties and methods via direct calls, which is much faster. The native wrappers are more complete and configurable. They include object-defined enumerations, default values for optional parameters, and an optional VB-like syntax for accessing properties. These new features allow you to write

```cpp
m_spGrid->MousePointer = flexHourglass;
m_spGrid->AutoSize(1);
```

instead of

```cpp
m_Grid.SetMousePointer(11);
COleVariant vtNone(0L, VT_ERROR);
m_Grid.AutoSize(1, vtNone, vtNone, vtNone);
```

Taking advantage of dual interfaces in existing MFC projects is easy. All you have to do is include the appropriate #import statement in your StdAfx.h file, then create a pointer and assign it to the existing control. For example, assuming you have an MFC-based m_Grid control, all the extra code you would need would be this:

```cpp
// include this statement in the StdAfx.h file
#import "c:\windows\system\vsflex7l.ocx" no_namespace

// note: to use the OLEDB/ADO version of the control,
// you need to #import "msdatsrc.tlb" as well.
#endif #import "c:\windows\system\msdatsrc.tlb" no_namespace
#endif #import "c:\windows\system\vsflex7l.ocx" no_namespace
```

Then, instead of using the control in the usual way, declare a variable of type IVSFlexGridPtr, initialize it by setting it to m_Grid.GetControlUnknown(), and use it instead of m_Grid. The two routines listed below illustrate the difference between the two approaches (both routines fill a grid 1000 times with a string and report how long it took them to do it):
// Using the MFC-generated wrapper class
void CMyDlg::BenchDispatchClick()
{
    CString strText = "Hello.;
    DWORD tStart = GetTickCount();
    for (long i = 0; i < 1000; i++)
        for (long r = m_Grid.GetFixedRows(); r < m_Grid.GetRows(); r++)
            for (long c = m_Grid.GetFixedCols(); c < m_Grid.GetCols(); c++)
                m_Grid.SetTextMatrix(r, c, (LPCTSTR)strText);
    DWORD tElapsed = GetTickCount() - tStart;
    CString str;
    str.Format("Done in %d seconds using dispatch interface.",
               (int)(tElapsed / 1000));
    MessageBox(str);
}

// Using the native wrapper class (#import-based)
void CMyDlg::BenchDualClick()
{
    IVSFlexGridPtr spGrid = m_Grid.GetControlUnknown();
    _bstr_t _strText = "Hello.;
    DWORD tStart = GetTickCount();
    for (long i = 0; i < 1000; i++)
        for (long r = spGrid->FixedRows; r < spGrid->Rows; r++)
            for (long c = spGrid->FixedCols; c < spGrid->Cols; c++)
                spGrid->PutTextMatrix(r, c, strText);
    DWORD tElapsed = GetTickCount() - tStart;
    CString str;
    str.Format("Done in %d seconds using dual interface.",
               (int)(tElapsed / 1000));
    MessageBox(str);
}

The code looks very similar, except for the dot notation used with the m_Grid variable and the arrow used with the spGrid variable. The big difference is in execution speed. The MFC/Dispatch version takes 30 seconds to fill the grid 1000 times, while the native/dual version takes only 8 seconds. The dual version is over three times faster than the traditional MFC/Dispatch version.

In functions that only set a few properties, it probably doesn't matter much which type of wrapper class you choose. But in functions with lengthy for statements that access properties or methods several hundred times, you should definitely consider using the #import statement/ dual interface approach.

Using VSFlexGrid in ATL projects
Using the VSFlexGrid control in ATL is not much different than using it in MFC projects. You can still use Wizards and a rich set of low-level support classes. What you don't get is the higher-level classes, document/view architecture, and other amenities provided by MFC.

To use the VSFlexGrid control in ATL projects, you will normally follow these steps:

1) Create a new ATL COM project of type "Executable".

2) Select the "Insert|New ATL Object..." menu option, select the "Miscellaneous" object type, then choose "Dialog". Pick any name for the dialog.

3) Go to the resource editor, open the dialog, right-click on it, select "Insert ActiveX control", and pick the VSFlexGrid control from the list (if the grid is not on the list, it hasn't been registered on your computer). You may also want to set the dialog's ClipChildren property
to True to make it repaint more smoothly.

4) Right-click on the control and select the "Events..." option. Then select the grid control from the list on the right and the list on the left will show all the events available for the control. Select the ones you want to handle by double-clicking them, and click OK when you are done. This will automatically insert an #import statement into the dialog header file.

5) Edit the #import statement and remove all qualifiers except for no_namespace. If you are using the OLEDB/ADO version of the VSFlexGrid control, then you need two #import statements instead of one (this is because the OLEDB/ADO version of the grid relies on the Microsoft DataSource object, which needs to be imported as well):

    #import "c:\windows\system\msdatsrc.tlb" no_namespace
    #import "c:\windows\system\vsflex7l.ocx" no_namespace

6) Now the control is on the form, but you can't talk to it yet. To get a pointer to the control, open the dialog header file and edit the OnInitDialog function so it looks like this:

```cpp
IVSFlexGridPtr m_spGrid;   // pointer to the control
CAxWindow m_wndGrid;  // pointer to the host window
LRESULT OnInitDialog(UINT uMsg, WPARAM wParam,
 LPARAM lParam, BOOL& bHandled)
{
    m_wndGrid = GetDlgItem(IDC_VSFLEXGRID1);  // get host window
    m_wndGrid.QueryControl(&m_spGrid);        // get control
    m_wndGrid.SetFocus();           // activate the control
    AtlAdviseSinkMap(this, true);   // hook up events
    return 1;  // let the system set the focus
}
```

This code declares a pointer to the control and one to the control's host window. When the dialog initiates, the code makes m_wndGrid point to the host window and queries it for the contained control, which is stored in the m_spControl variable. From then on, you may move and resize the control through its host window (m_wndGrid) and access the control's properties and methods through the control pointer (m_spControl).

7) The only thing missing is the code that displays the dialog. That needs to be added to the project's main .cpp file. Here's the code that you will need:

```cpp
#include "MyProject_i.c"
#include "MyDlg.h"   // add this line

// wizard-generated code . . .

////////////////////////////////////////////////////////////////////////
// extern "C" int WINAPI _tWinMain(HINSTANCE hInstance,
//      HINSTANCE /*hPrevInstance*/, LPTSTR lpCmdLine, int /*nShowCmd*/)
{
    // wizard-generated code . . .

    CMyDlg dlg;   // create the dialog
dlg.DoModal(); // show the dialog

    //MSG msg;       // comment these lines out
    //while (GetMessage(&msg, 0, 0, 0))
    //  DispatchMessage(&msg);

    // wizard-generated code . . .
}
```
That's about it. You could clean up this project by removing the references to the \texttt{idl} and \texttt{rgs} files, which it doesn't need (it is just an EXE, not a COM server). See the samples on the distribution CD to find out what changes are necessary or refer to the ATL documentation for more details.

**Handling Pictures in ATL projects**
If you are not using MFC, you can't use the MFC \texttt{CPictureHolder} class. But you can use the \texttt{CPicHolder} class provided in the samples on the distribution CD. \texttt{CPicHolder} is actually more powerful than \texttt{CPictureHolder} because it provides methods for loading pictures from disk files and from the clipboard. Once you include the \texttt{CPicHolder} class in your project, you can handle pictures just as we discussed earlier, when dealing with MFC projects.

The samples on the distribution CD include other useful classes, such as \texttt{CToolTip}, and a file named \texttt{AtlControls.h} that is part of a Visual Studio sample and defines ATL wrapper classes for all Windows common controls.

**Creating Controls Dynamically in ATL**
As in MFC, you can create controls dynamically with ATL. These are the steps required:

1) Insert the appropriate \texttt{#import} statement in the dialog header file or in the \texttt{StdAfx.h} file.

2) Add two member variables to your dialog or window class:

   \begin{verbatim}
   CAxWindow m_wndGrid;      // host window
   IVSFlexGridPtr m_spGrid;  // pointer to control
   \end{verbatim}

3) When the dialog or window is created, create the control and its host window, and attach the control to the window:

   \begin{verbatim}
   // initialize ATL ActiveX hosting
   AtlAxWinInit();

   // create the control (will fail if not registered)
   m_spGrid.CreateInstance(__uuidof(VSFlexGrid));
   ATLASSERT(m_spGrid != NULL);

   // create the host window (nID = IDC_GRID1)
   // the nID is needed if you want to sink events.
   RECT rc;
   GetClientRect(&rc);
   m_wndGrid.Create(m_hWnd, rc, NULL,
                    WS_CHILD | WS_VISIBLE, 0, IDC_GRID1);

   // attach the control to the host window
   CComPtr<IAxWinHostWindow> spHost;
   m_wndGrid.QueryHost(&spHost1);
   spHost->AttachControl(m_spGrid, m_wndGrid);
   \end{verbatim}

As in MFC, this approach requires you to hook up the event handlers manually. You can also copy wizard-generated code, which does two things: it adds an \texttt{IDispEventImpl} declaration to your class so it inherits the ATL event handling mechanisms, and adds an event sink map to your class using \texttt{BEGIN_SINK_MAP}, \texttt{SINK_ENTRY}, and \texttt{END_SINK_MAP} macros. You still need to add the call to \texttt{AtlAdviseSinkMap} in order to start getting the events.

Also in ATL, unless you have a good reason to create the controls dynamically, you should still stick to the resource editor and the Wizard.
Using VSFlexGrid in Visual J++

Using the VSFlexGrid control in Microsoft Visual J++ is a lot like using it in Visual Basic. The visual interface is almost the same (unlike VC++), despite the different syntax.

There are a few caveats, however, most of which apply to all ActiveX controls (they are not specific to the VSFlexGrid control). They are described below:

**Using the Text Property**

When using the VSFlexGrid control in VJ++, you should not access the Text property using the wrapper-generated `setText()` and `getText()` methods. Use `setCtlText()` and `getCtlText()` instead.

This is because when VJ++ generates the wrapper class for the control, it includes `getText()` and `setText()` methods that access the control’s window text, which is not related to any properties. If the control happens to have a Text property (which the VSFlexGrid control does), it gets mapped into `getcTText()` and `setCtlText()` instead. Note that this happens for all ActiveX controls, not just the VSFlexGrid.

To use the Text property in VJ++ you would write code such as this:

```vbnet
private void button1_click(Object source, Event e)
{
    VSFlexGrid1.setCtlText("Hello World");  // this works
    //VSFlexGrid1.setText("Hello World");   // this doesn't work!
}
```

**Handling Pictures**

The VJ++ wrappers generated for picture properties work correctly, and you can use the `setCellPicture` and `getCellPicture` methods normally. However, the wrappers fail when the picture is contained in a Variant, which prevents you from using the `SetCell` method to set cell pictures directly.

There is an easy workaround for this. By inspecting the wrapper code, you can see that J++ converts its native image type to an IPictureDisp interface with a call to the `com.ms.wfc.ui.AxHost.getIPictureDispFromPicture(img)` support function.

By using this in your calls to `setCell`, you can set pictures with no problems. For example:

```vbnet
// get Image object
Image img = pictureBox1.getImage();

// assign to current cell
VSFlexGrid1.setCellPicture(img);

// assign to a range
VSFlexGrid1.setCell(3, new Variant(3), new Variant(3),
                     new Variant(4), new Variant(4),
                     new Variant(com.ms.wfc.ui.AxHost.getIPictureDispFromPicture(img)));

// this does *not* work
//VSFlexGrid1.setCell(3, new Variant(3), new Variant(3),
//                    new Variant(4), new Variant(4),new Variant(img));
```

**Clearing Pictures**

You can remove pictures from cells using three techniques:
1) Assign an empty picture to a cell or range:

```java
VSFlexGrid1.setCellPicture(new Bitmap(0,0));
VSFlexGrid1.setCell(3, new Variant(3), new Variant(3),
    new Variant(4), new Variant(4),
    new Variant(com.ms.wfc.ui.AxHost.getIPictureDispFromPicture(new Bitmap(0,0))));
```

2) Use the `Clear` method to remove all cell formatting:

```java
VSFlexGrid1.Clear(new Variant(1), new Variant(2)); // scrollable area/format
```

3) Use the `setCell` property and set `flexcpCustomFormat` (21) to false. This removes all formatting from a range:

```java
VSFlexGrid1.setCell(21, new Variant(1), new Variant(1),
    new Variant(10), new Variant(10), new Variant(0));
```

**Data Binding**

In Visual Basic, you may select the `DataSource` property at design time and select from a list of available data sources. This is not done by the controls, but by VB itself. In Visual J++, the `DataSource` property is not displayed in the property window. This is because J++ uses a different type of design time mechanism, and that is why you don’t see the `DataSource` property at design time (not on the VSFlexGrid, MSDataGrid, or any other ActiveX OLEDB control).

To implement data binding in J++, you have two options: use the native WFC `DataSource` control, or use the ADO classes and create the data source yourself. Either way, you need to connect the grid to the data source using code.

**a)** To use the native WFC `DataSource` control, follow these steps:

- Put a WFC `DataSource` control on the form.
- Set the `connectionString` and `commandText` properties to the data you want.
- Put a VSFlexGrid control on the form.
- Set the `Editable`, `DataMode`, and `VirtualData` properties to the values you want.
- Bind the VSFlexGrid control to the `DataSource` control using the following code:

```java
public class Form1 extends Form {
    public Form1() {
        // Required for Visual J++ Form Designer support
        initForm();

        // Set FlexGrid properties.
        fg.setEditable(true);
        fg.setDataMode(vsflex7.DataModeSettings.flexDMBound);

        // Bind FlexGrid to dataSource control.
        msdatsrc.DataSource ds = (msdatsrc.DataSource) dataSource1.getRecordset().getDataSource();
        fg.setDataSource(ds);
    }
}
```

**b)** To use the VSFlexGrid control with ADO data objects, follow these steps:

- Import the WFC ADO package with the following import statement:

  ```java
  import com.ms.wfc.data.*;
  ```

- Declare a Recordset variable and initialize it when the form loads
- Bind the Recordset to the VSFlexGrid control as before:

```java
public class Form1 extends Form {
    private Recordset m_rs;
    public Form1() {
        // Required for Visual J++ Form Designer support
        initForm();

        // create ADO recordset
        m_rs = new Recordset();
        m_rs.open("SELECT * FROM CUSTOMERS", "NORTHWIND",
                   AdoEnums.CursorType.STATIC,
                   AdoEnums.LockType.BATCHOPTIMISTIC);

        // Bind FlexGrid to ADO recordset
        msdatsrc.DataSource ds = (msdatsrc.DataSource)m_rs.getDataSource();
        fg.setDataSource(ds);
    }
}
```
VSFlexGrid Property Groups

The VSFlexGrid control has a rich set of properties, methods, and events. You do not have to know all of them in order to use the control effectively.

The reference below shows the most important properties, methods, and events, grouped by type of usage. Some elements appear in more than one group. For details on specific properties, events, or methods, check the reference part of this document.

1) Grid Layout

- **Rows, Cols, FixedRows, FixedCols, FrozenRows, FrozenCols, RightToLeft**
- **RowHeight, ColWidth, AutoSize, AutoSizeMode, Sort, ColSort**
- **RowHeightMin, RowHeightMax, ColWidthMin, ColWidthMax**
- **LeftCol, TopRow, RightCol, BottomRow, ClientWidth, ClientHeight**
- **CellLeft, CellTop, CellWidth, CellHeight**
- **RowIsVisible, ColIsVisible, RowPos, ColPos, RowPosition, ColPosition**

2) Cursor and Selection

- **Row, Col, RowSel, ColSel, Select, GetSelection, ShowCell**
- **AllowSelection, AllowBigSelection, FocusRect, Highlight**
- **SelectionMode, IsSelected, SelectedRow, SelectedRows**
- **BeforeMouseDown, MouseRow, MouseCol**
- **BeforeRowColChange, AfterRowColChange, BeforeSelChange, AfterSelChange**

3) Editing

- **Editable, EditCell, EditWindow, ShowComboBox, FillStyle, TabBehavior**
- **BeforeEdit, StartEdit, ValidateEdit, AfterEdit, CellButtonClick**
- **ComboList, ColComboList, BuildComboList**
- **EditMask, ColEditMask, EditMaxLength**
- **EditText, EditSelStart, EditSelLength, EditSelText**
- **ComboCount, ComboData, ComboIndex, ComboCount, ComboSearch**
- **KeyDownEdit, KeyPressEdit, KeyUpEdit, ChangeEdit**

4) Getting and Setting Values

- **Cell, Text, TextMatrix, Clip, ClipSeparators, FillStyle**
- **AddItem, RemoveItem, Clear, FindRow, Aggregate, Value, ValueMatrix**

5) User Interface

- **ExplorerBar, BeforeMoveColumn, AfterMoveColumn, BeforeSort, AfterSort**
- **AllowUserResizing, AutoSizeMouse, BeforeUserResize, AfterUserResize**
- **FrozenRows, FrozenCols, AllowUserFreezing, AfterUserFreeze**
- **AllowSelection, AllowBigSelection, Editable, TabBehavior**
- **ScrollBars, ScrollTrack, BeforeScroll, AfterScroll, ScrollTips, ScrollTipText, BeforeScrollTip**
- **AutoSearch, AutoSearchDelay**

6) Outlining and Summarizing

- **OutlineBar, OutlineCol, TreeColor, NodeOpenPicture, NodeClosedPicture**
- **IsSubtotal, RowOutlineLevel, IsCollapsed, BeforeCollapse, AfterCollapse**
- **Subtotal, SubtotalPosition, MultiTotals, Outline, GetOutlineNode**

7) Merging Cells

- **MergeCells, MergeRow, MergeCol, MergeCompare, GetMergedRange**
8) Data Binding
- DataSource, DataMember, DataMode, VirtualData
- DataRefresh, BeforeDataRefresh, AfterDataRefresh, AutoResize, Error
- BindTo Array, LoadArray, FlexDataSource
- AddItem, RemoveItem, FilterData

9) Saving, Loading, and Printing Grids
- SaveGrid, LoadGrid, ClipSeparators, Archive, ArchiveInfo
- PrintGrid, StartPage, BeforePageBreak, GetHeaderRow

10) OLE Drag Drop
- OLEDragMode, OLEDropMode, BeforeMouseDown, OLEDrag, OLEStartDrag
- OLECompleteDrag, OLEDragDrop, OLEDragOver, OLEGiveFeedback, OLESetData
**VSFlexGrid Tutorials**

The following sections contain tutorials that illustrate some of the main features in the VSFlexGrid control. The tutorials walk you through the creation of several simple projects, describing each step in detail. After walking through the tutorials, you should be able to start programming the VSFlexGrid grid on your own.

The tutorials are:

**Edit Demo**
Starting with a basic data-entry grid, this tutorial shows how to implement data formatting, check boxes, drop-down lists, input masks, data validation, and clipboard support.

**Outline Demo**
Shows how you can use the VSFlexGrid as an outliner to display structured (or hierarchical) data.

**Data Analysis Demo**
Starting with a grid containing sales data for different products, regions, and salespeople, this tutorial show how to implement dynamic layout (column order), automatic sorting, cell merging, automatic subtotals, and outlining.

**Cell Flooding Demo**
This tutorial shows how to use the Cell property to format individual cells. The demo uses flooding to create a display combining numbers and bars.

**ToolTip Demo**
This tutorial shows how you can implement tooltips with different text for each cell.

**Printing Demo**
This tutorial shows how you can print a grid with control over page breaks "header" rows which appear at the top of every page.

**OLE Drag and Drop Demo**
This tutorial shows how to implement automatic and manual OLE drag and drop.

**Visual C++ MFC Demo**
This tutorial contains another version of the Outline Demo written in Visual C++ using the MFC application framework.

The distribution CD contains more sophisticated samples that can be used as a reference.
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:

![Form1](image)

**Step 1: Create the Control**

Start a new Visual Basic project including **VSFlexGrid Pro 7.0** (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) = fg
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:

```vbnet
Private Sub Form_Load()
    ' format column 3 (Amount Sold) to display currency
```
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
    fg.ColFormat(3) = "$#,###.00"
    ' make column 4 (Bonus) a boolean column
    fg.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 ColComboBox property. The list consists of a string with choices, separated by pipe characters ("|").

The ColComboBox 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
    fg.ColFormat(3) = "$#,###.00"
End Sub
' make column 4 (Bonus) a boolean column
fg.ColDataType(4) = flexdtBoolean

' assign combo lists to each column
fg.ColComboList(0) = "Applets|Wahoos|Gadgets"
fg.ColComboList(1) = "North|South|East|West"
fg.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 cursor 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
fg.ColFormat(3) = "$#,###.00"

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

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

' assign combo lists to each column
fg.ColComboList(0) = "Applets|Wahoos|Gadgets"
fg.ColComboList(1) = "North|South|East|West"
fg.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:

```vba
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 = fg.EditText
            rgn = fg.TextMatrix(Row, 1)
        Case 1
            prd = fg.TextMatrix(Row, 0)
            rgn = fg.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 transferring information between applications. Adding clipboard support to **VSFlexGrid** projects is very easy. All it takes is the following code:

```vba
Private Sub fg_KeyDown(KeyCode%, Shift%)
    Dim Cpy As Boolean, Pst As Boolean
    ' copy: ctrl-C, ctrl-X, ctrl-ins
    If KeyCode = vbKeyC And Shift = 2 Then Cpy = True
    If KeyCode = vbKeyX And Shift = 2 Then Cpy = True
    If KeyCode = vbKeyInsert And Shift = 2 Then Cpy = True
    If KeyCode = vbKeyInsert And Shift = 1 Then Pst = True
    ' paste: ctrl-V, shift-ins
    If KeyCode = vbKeyV And Shift = 2 Then Pst = True
    If KeyCode = vbKeyInsert And Shift = 1 Then Pst = True
End Sub
```
' do it
If Cpy Then
    Clipboard.Clear
    Clipboard.SetText fa.Clip
ElseIf Pst Then
    fg.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:

![Image of VSFlexGrid as an outliner]

**Step 1: Create the Control**

Start a new Visual Basic project including **VSFlexGrid Pro 7.0** (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)** = fg
- **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:

```vba
Private Sub Form_Load()
    ' suspend repainting to increase speed
    fg.Redraw = False

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

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

    ' repainting is back on
    fg.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:

```vba
Sub AddNode(iniFile As String)
    Dim In As String, p As Integer
    With fg

        ' 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, In
            If Left(In, 1) = "[" Then
```

44
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 BeforeCollapse and AfterCollapse events 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:

```vbnet
Private Sub fg_DblClick()
    Dim r As Long
    With fg
        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 **DbIcIck** event handler from the **KeyPress** handler:

```vba
Private Sub fg_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:

![Grid with data](image)

**Step 1: Create the Control**

Start a new Visual Basic project including VSFlexGrid Pro 7.0 (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 `fg`.

**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:

```vbnet
Private Sub Form_Load()
    ' initialize the control
    fg.Cols = 4
    fg.FixedCols = 0
    fg.GridLinesFixed = flexGridExplorer
```
fg.AllowUserResizing = flexResizeBoth
fg.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 fg, 0, slProduct
FillColumn fg, 1, slAssociate
FillColumn fg, 2, slRegion
FillColumn fg, 3, slSales
fg.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(fg 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
    fg.Cell(flexcpText, 0, c) = lst(0)
    For r = fg.FixedRows To fg.Rows - 1
        i = (Rnd() * 1000) Mod cnt + 1
        fg.Cell(flexcpText, r, c) = lst(i)
    Next

' do an autosize on the column we just filled
    fg.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 **flexDMLFree** to be able to sort using the **Sort** property.)

Here is the code:

```vbscript
Private Sub fg_AfterMoveColumn(ByVal Col As Long, Position As Long)
    ' sort the data from first to last column
    fg.Select 1, 0, 1, fa.Cols - 1
    fg.Sort = flexSortGenericAscending
    fg.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.

```vbscript
Private Sub Form_Load()
    ' initialize the control
    ' ...
    ' define some sample data
    ' ...
    ' populate the control with the data
    ' ...

    ' organize the data
    fg_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:

```vbscript
Private Sub Form_Load()
    ' initialize the control
    ' ...
    ' define some sample data
    ' ...
    ' populate the control with the data
    ' ...
```

49
' set up cell merging (all columns)
fg.MergeCells = flexMergeRestrictAll
fg.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 fg_AfterMoveColumn(ByVal Col As Long, Position As Long)
    ' suspend repainting to get more speed
    fg.Redraw = False
    ' sort the data from first to last column
    fg.Select 1, 0, 1, fa.Cols - 1
    fg.Sort = flexSortGenericAscending
    fg.Select 1, 0
    ' calculate subtotals
    fg.Subtotal flexSTClear
    fg.Subtotal flexSTSum, -1, 3, , 1, vbWhite, True
    fg.Subtotal flexSTSum, 0, 3, , vbRed, vbWhite, True
    fg.Subtotal flexSTSum, 1, 3, , vbBlue, vbWhite, True
    ' autosize
    fg.AutoSize 0, fa.Cols - 1, , 300
    ' turn repainting back on
    fg.Redraw = True
End Sub

This code starts by setting the Redraw 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:

```vba
Private Sub fg_BeforeMoveColumn(ByVal Col As Long, Position As Long)
    ' don't move sales figures
    If Col = fg.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:

```vba
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
   fg.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:

![Cell Flooding Table](image)

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

```vbscript
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
    fg.Cols = 3
    fg.Rows = 9
    fg.FloodColor = RGB(100, 255, 100)
    fg.ColAlignment(0) = flexAlignCenterCenter
    fg.ColAlignment(1) = flexAlignRightCenter
    fg.ColAlignment(2) = flexAlignLeftCenter
    fg.Cell(flexcpText, 0, 0) = "Age Range"
    fg.Cell(flexcpText, 0, 1) = "Females"
    fg.Cell(flexcpText, 0, 2) = "Males"
    fg.ColFormat(-1) = ".#,##"
    ' make data bold
    fg.Cell(flexcpFontBold, 1, 1, _
        fg.Rows - 1, fa.Cols - 1) = True
    ' place text in cells, keep track of maximum
    For i = 0 To 7
        fg.Cell(flexcpText, i + 1, 0) = _
            10 * i & " - " & (10 * (i + 1) - 1)
        fg.Cell(flexcpText, i + 1, 1) = count(0, i)
        fg.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
```

53
' 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
  fg.Cell(flexcpFloodPercent, i + 1, 1) = _
    -100 * count(0, i) / max
  fg.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.
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 fg_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 = fg.MouseRow
    nc = fg.MouseCol

    ' update tooltip text
    If c <> nc Or r <> nr Then
        r = nr
        c = nc
        fg.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.
Printing Demo

The **PrintGrid** method is part of the **VSFlexGrid** control, and you may use to print the grid contents directly to the printer.

If you need more sophisticated printing, with print preview and the ability to combine several grids and other elements such as tables and text in a single document, then you should use Videosoft's **VSPrinter** control, a separate VideoSoft product that is part of the **VSVIEW** product.

This sample shows how you can print a grid using either method. Both methods allow you to control page breaks and to designate certain rows as "header" rows, which appear at the top of every page.

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

```vbscript
Sub PrintFlexGridBuiltIn()
    fg.PrintGrid "My Grid"
End Sub

Sub PrintFlexGridOnVSPrinter()
    vp.StartDoc
    vp.RenderControl = fg hWnd
    vp.EndDoc
End Sub
```

The routines above are all you need in order to print simple reports. The first uses the built-in **PrintGrid** method, and the second uses the **VSPrinter** control. The second method allows you to show the report on a preview window, render it on the printer, or save it 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:

```vbscript
' BeforePageBreak: controls page breaks
' we assume we have subtotals above details, and prevent subtotal rows from
' being the last on a page
Private Sub fg_BeforePageBreak(ByVal Row As Long, BreakOK As Boolean)
    ' if this row is a subtotal heading, we can't break here
    If fg.IsSubtotal(Row) Then BreakOK = False
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 fg_GetHeaderRow(ByVal Row As Long, HeaderRow As Long)
    Dim r As Long

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

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

This sample shows how to implement automatic and manual OLE drag and drop using **VSFlexGrid Pro 7.0**.

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:

![Diagram of the final application](image)

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 7.0** (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 `fgDManual` and `fgDDAuto`.

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 `fgDDManual` and `fgDDAuto` 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:

```vbscript
Private Sub Form_Load()
    ' initialize manual control
    With fgDManual
        ' code...
    End With

    ' initialize auto control
    With fgDDAuto
        ' code...
    End With

    ' code...
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 **fgDDAuto** 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 **fgDDAuto** 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 **fgDDManual** control.

**Step 3: Manual OLE Drag**

We will customize the behavior of the **fgDDManual** 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 **fgDDManual** 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:

```vbnet
Private Sub fgDDManual_BeforeMouseDown(ByVal Button As Integer, _
    ByVal Shift As Integer, _
    ByVal X As Single, ByVal Y As Single, _
    Cancel As Boolean)
    With fgDDManual
        ' 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.
            .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:

```vbnet
Private Sub fgDDManual_OLEStartDrag(Data As VSFlex6Ctl.vsDataObject, _
    AllowedEffects As Long)
    ' set contents of data object for manual drag
    Dim s$ = fgDDManual.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 fgDDManual 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 fgDDManual 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:

```vbscript
Private Sub fgDDManual_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 fgDDManual
        ' 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++ MFC Demo

The VSFlexGrid Pro 7.0 documentation is geared toward Visual Basic users. However, many other development environments are capable of hosting ActiveX controls, including Microsoft Visual C++, Visual J++, Internet Explorer, Microsoft Access, and others.

This demo shows the basic techniques you will need to use VSFlexGrid with Visual C++ in MFC projects. (The Using VSFlexGrid in Visual C++ section of the documentation shows how to use VSFlexGrid in non-MCF applications.) To follow this demo, you must know how to use the Visual C++ development environment and you must also know 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. Click OK.

Step 2: Add the VSFlexGrid Control to the Project

Now that the project has been created, 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_fg` 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_fg` 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:

```cpp
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_fg.SetRedraw(FALSE);

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

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

    // repainting is back on
    m_fg.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 CO outline CDlg::AddNode(LPSTR inifile)
{
    long row;

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

    // create file node
    m_fg.AddItem(inifile, varDefault);
    row = m_fg.GetRows() - 1;
    m_fg.SetIsSubtotal(row, TRUE);
    m_fg.Select(row, 0, varDefault, varDefault);
    m_fg.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_fg.AddItem(ln + 1, varDefault);
            row = m_fg.GetRows() - 1;
            m_fg.SetIsSubtotal(row, TRUE);
            m_fg.SetRowOutlineLevel(row, 1);
            m_fg.Select(row, 0, varDefault, varDefault);
            m_fg.SetCellFontBold(TRUE);
        } else if (strchr(ln, '=')) {
            char* p = strchr(ln, '=');
            *p = 0;
            CString str = (CString)\t" + (CString)ln +
                            (CString)\t" + (CString)(p + 1);
            m_fg.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 outline nodes.

**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:

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

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

```c++
#include <afxctl.h>
void COutlineCDlg::AddNode(LPSTR inifile)
```
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 Introduction

The **VSFlexString** control allows you to incorporate regular-expression text matching into your applications. This allows you to parse complex text 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 a regular expression syntax similar to the ones used in Unix systems.

The usual sequence of steps involved in using the **VSFlexString** controls is this:

1. Assign the string containing the text you want to work on to the **Text** property.
2. Assign a string containing a regular expression to the **Pattern** property. At this point, the **VSFlexString** control will automatically find all the matches it can.
3. Loop through the matches found, from zero to **MatchIndex** - 1.
4. For each match, perform some operation using the **MatchString**, **MatchStart**, and **MatchLength** properties.

For example, suppose you want to scan an HTML string and locate all the HTML tags in it. A good pattern to use in this case would be the string "<[^>]*>" , which means "a less-than sign, followed by any number of characters different than a greater-than sign, followed by a greater-than sign". The syntax for the regular expressions that describe patterns is explained in detail in later sections.

After assigning the HTML text to the **Text** property and the "<[^>]*>" string to the **Pattern** property, the **VSFlexString** control will automatically find as many matches as it can, and expose them to your application through the **Match** properties. The picture below illustrates this:

```
Pattern: <[^>]*>

Text: <p>Welcome to the world of <i>easy</i> <b>Pattern Matching</b>.</p>
```

```
MatchString(i): <p> Welcome to the world of <i>easy</i> <b>Pattern Matching</b>. </p>
MatchIndex: 0   1   2   3   4   5
MatchStart:  27  34  39  58  63
MatchLength: 3   4   3
```

The picture shows the **Text** and **Pattern** properties. Below the strings, you can see the six matches that were found (the number of matches found is returned in the **MatchCount** property). The **MatchString** property returns an array of strings, indexed from zero to **MatchCount** - 1, containing the text in each match. The **MatchStart** and **MatchLength** properties return the position and length of each match.

From this point on, you can work on the string either by modifying it directly or by changing the text in the **TagString** property.

The **VSFlexString** control extends the concept of matching patterns by allowing you to specify **tags** in each pattern. The tags are created by enclosing parts of the **Pattern** string in curly brackets ("{}") . This is useful when the pattern contains parts that need special processing.

For example, still using the same HTML text, imagine you want to extract all text that is bold or italic. You could use a pattern such as "<b><i>{[^>]*}</i></b>", which means "a less-than sign followed by a B or an I, followed by a greater-than sign, followed by any number of characters different than a greater-than sign, followed by a less-than sign,"
followed by a slash, followed by either a B or an I, followed by a greater-than sign". The important thing to notice here is that the actual text (excluding the tags) is enclosed in curly brackets and is thus recognized by the VSFlexString control as a tag. Tags can be accessed using the TagString, TagStart, and TagLength properties. A pattern may contain any number of tags.

The picture below shows how the tags would be retrieved.

![Pattern and Text Example](image)

Note that the tag index is relative to the current match. Thus, the first tag ("easy") would be retrieved by setting MatchIndex to zero and TagIndex to zero. The second tag ("Pattern Matching") would be retrieved by setting MatchIndex to one and TagIndex to zero. You would set TagIndex to a number greater than zero only if your pattern contained multiple tags.

The most important aspect of learning how to use the VSFlexString control is understanding and becoming familiar with the regular expression syntax. The following topics describe VSFlexString Regular Expressions and present a few possible uses for the control.
Regular Expressions

The regular expression syntax recognized by **VSFlexString** is based on the following special characters:

<table>
<thead>
<tr>
<th>Char</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Beginning of a string.</td>
</tr>
<tr>
<td>$</td>
<td>End of a string.</td>
</tr>
<tr>
<td>\</td>
<td>Any character.</td>
</tr>
<tr>
<td>[list]</td>
<td>Any character in list. For example, &quot;[AEIOU]&quot; matches any single uppercase vowel.</td>
</tr>
<tr>
<td>[^list]</td>
<td>Any character not in list. For example, &quot;[^ ]&quot; matches any character except a space.</td>
</tr>
<tr>
<td>[A-Z]</td>
<td>Any character between 'A' and 'Z'. For example, &quot;[0-9]&quot; matches any single digit.</td>
</tr>
<tr>
<td>?</td>
<td>Repeat previous character zero or one time.</td>
</tr>
<tr>
<td>*</td>
<td>Repeat previous character zero or more times.</td>
</tr>
<tr>
<td>+</td>
<td>Repeat previous character one or more times.</td>
</tr>
<tr>
<td>\</td>
<td>Escape next character. This is required to any of the special characters that are part of the syntax. For example &quot;.<em>&quot; matches &quot;..</em>&quot;. It is also required to encode some special non-printable characters (such as tabs) listed below.</td>
</tr>
<tr>
<td>{tag}</td>
<td>Tag this part of the match so you can refer to it later using the TagString property.</td>
</tr>
</tbody>
</table>

In addition to the characters listed above, there are seven special characters encoded using the backslash. These are listed below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Bell (Chr(7))</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace (Chr(8))</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed (Chr(12))</td>
</tr>
<tr>
<td>\n</td>
<td>New line (Chr(10), vbLf)</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return (Chr(13), vbCr)</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab (Chr(9), vbTab)</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab (Chr(11))</td>
</tr>
</tbody>
</table>

For example,

"\^stuff"  " any string starting with "stuff"
"stuff$"    " any string ending with "stuff"
"o.d"      "old", "odd", "ord", etc
"o[ld]d"    "old" or "odd" only
"o[^l]d"    "odd", "ord", but not "old"
"od?"       "o" or "od"
"od*"       "o", "od", "odd"
"od+"       "od", "odd", etc
"\."        decimal point (needs escape character)
"[A-Z][a-zA-Z]*"  any uppercase word
"[0-9]+"     any stream of digits
"[1-9]+[1-9]*"  any stream of digits not starting with zero
"[+\-]?[0-9]*[\.]?[0-9]*" any number with optional sign and decimal point (needs two escape characters)

One of the best ways to develop and quickly test your patterns is using the Visual Basic Properties window. If you place a VSFlexString control on a form, then click on it and press F4, the Properties window will appear. You can then type directly into the Text and Pattern
properties and watch the **MatchCount** property change. For example, if you type "]a-z"]" into the **Pattern** property and "Hello World" into the **Text** property, **MatchCount** will be set to 8 (the number of lower case character in the text). If you change the **Pattern** to "[A-Za-z]*", the **MatchCount** property will change to 2 (the number of words in the text).

Note that if a pattern can match the string in more than one way, the longest match will prevail. For example,

```vbnet
fs.Text = "testing, 1, 2, 3. Done testing."
fs.Pattern = ".*"  
Debug.Print fs.MatchCount; ";"; fs.MatchString; "]"
```

This pattern means "any sequence of characters terminating in a comma". It is ambiguous, because the control could break up the string in the following ways:

- **testing, 1, 2, 3. Done testing.** (Three matches)
- **testing, 1, 2, 3. Done testing.** (Two matches)
- **testing, 1, 2, 3. Done testing.** (Two matches)
- **testing, 1, 2, 3. Done testing.** (One match)

In such cases, the control will extend the match as far as it can, thus finding the longest match. The output will be:

1 [testing, 1, 2,]
Matching Demo

This demo shows one of the simplest uses for the VFlexString control: retrieving information from text based on patterns.

Suppose we have a text file containing client names and phone numbers, and want to retrieve the names of all clients who are in the 415 area code.

Here is what the list looks like:

```
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"
```

The most important part of the task is developing the Pattern string. It would definitely contain the string "(415)", which is our criterion for filtering the data. But using "(415)" would produce a number of matches consisting of the string "(415)", which is not very useful. So we extend the pattern to include everything before and after the "(415)" up to the record delimiter, in this case a comma. The Pattern string would look like this: "[^,]*(415)[^,]*".

Here is the code (fs is a VFlexString control):

```
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
```

And here is the result:

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

You could easily extend the Pattern string to take other delimiters into account. For example, to recognize commas, tabs, and line breaks you would use "[^,\t\n\l]*(415)[^,\t\n\l]*".
Replacing Demo

This demo builds on the previous one by showing how you can modify the data after retrieving it.

Suppose the phone company decided to change all the 415 prefixes into 414, and we want to update our client list to reflect this change.

To accomplish this using the **VSFlexString** control, we start by locating the information we want to change, and then change it into the new value using the **Replace** property. Here is the code:

```vbs
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 " matches found."
fs.Replace = "(414)"
Debug.Print fs.MatchCount " matches found."
Debug.Print fs
```

And here is the result:

3 matches found.
0 matches found.

John Doe: **414** 555-1212, Mary Smith: (212) 555-1212,
Dick Tracy: (412) 555-1212, Martin Long: **414** 555-1212,
Leo Getz: (510) 555-1212, Homer Simpson: **414** 555-1212

Notice that the code prints the number of matches found twice. The first time, it reports three matches. The second time, after the **Replace** statement, it reports no matches. This is because as soon as the replacement takes place, all the "(415)" strings are gone, and thus there's nothing to match.
Data-Cleaning Demo

This demo shows a fairly advanced application of the VSFlexString control. It uses a pattern with tags to retrieve information and automatically parse each match.

Tags are created by enclosing parts of the Pattern string in curly braces. By tagging the matches, you can determine which parts of the string matched what parts of the pattern.

For example, say we have a database that contains customer's names. But the same name may be stored as "John Doe", "John Francis Doe", "John F. Doe", or "Doe, John", depending on who did the data entry. We want to clean the data, converting all variations to the latter type ("Last, First").

Here is a small function that will accomplish this task:

```vbnet
Private Function CleanName(name$) As String
    fs.Text = name
    fs.Pattern = "^[{A-Za-z}]+[^,]*[{A-Za-z}]+$"
    If fs.MatchCount > 0 Then
        CleanName = fs.TagString(1) & ", " & fs.TagString(0)
    Else
        CleanName = name
    End If
End Function
```

The Pattern string needs some explanation. The first part, "^[{A-Za-z}]+", will match sequences of letters that start at the beginning of the string. This will be a first name or a last name. The second part, "[^,]*" will match any sequence of characters not including a comma and followed by a space. This will match the space between a first and a last name, and also the optional middle name. However, the pattern will not match names already in the "Last, First" format because of the comma. Finally, the "[{A-Za-z}]+$" part will match the last name.

Notice how the parts of the Pattern that match the first and last names are enclosed in curly brackets. This allows us to retrieve their values and replace names in "First Last" and "First Middle Last" format with the "Last, First" format we want. This is accomplished using the TagString property.

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.
```

The function works as expected. Note that the last try fails, because the last name is not supposed to contain periods. In this case, the function just returns the original string, which seems like a reasonable thing to do.

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

This demo shows how the **VSFlexString** control 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:

![Calculator Demo Image](image)

**Step 1: Create the Controls**

Start a new Visual Basic project. Right-click on the toolbox and select "Components...", then pick the **VSFlexString** control from the list. The **VSFlexString** icon will be added to the Visual Basic toolbox. (If the control does not appear on the list, it hasn't been registered on your computer. In this case, click the "Browse" button and select the **VSSTR7.OCX** file to register it.)

Create a **VSFlexString** object on the form by clicking the icon on the toolbox, then 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 mainly 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 called when the user clicks the button. Double click the button and type the following into the event handler:

```vbnet
' Evaluate the expression in Text1 and show the result in Text2
Sub Command1_Click()
    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:

```vbnet
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.
The **Eval** function consists of a pattern that repeats itself. The **VSFlexString** is used to parse each expression into its components, according to operator priority rules, and **Eval** is called recursively to evaluate each component. This process continues until a number is found and evaluated using VB’s built-in **Val** function.

The typical pattern has this format: 
```
{.+}{[*/]}{.+}
```

The "{.+}" at the start and end of the pattern match runs of one or more characters. The "{[*/]}" matches an operator that separates the left and right parts of the expression.

### Step 3: Trying it out

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.

### Step 4: Extending the Evaluator

As an exercise for the reader, try adding support for user-defined variables and functions such as **Sin** and **Cos**.
VSFlexGrid Control

Control Name: VSFlexGrid
Description: :-) VideoSoft VSFlexGrid 7.0
File Name: VSFLEX7.OCX
Help File: VSFLEX7.HLP
GUID: {d76d7128-4a96-11d3-bd95-d296dc2dd072}
Properties: 183
Events: 52
Methods: 22

Before you can use a VSFlexGrid control in your application, you must add the VSFLEX7.OCX file to your project. In Visual Basic, right-click the toolbox and select the VSFlexGrid Control from the list. In Visual C++, right-click on the dialog box and select the VSFlexGrid Control from the list, or use the #import statement to import the VSFLEX7.OCX file into the project.

To distribute applications you create with the VSFlexGrid control, 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

Adds a row to the control.

Syntax

\[ \text{[form!]VSFlexGrid.AddItem Item As String, [ Row As Long ]} \]

Remarks

The parameters for the AddItem method are described below:

Item As String
String expression to add to the control. The string contains entries for each column on the new row, separated by tabs (vbTab or Chr(9)). You may change the column separator character by assigning a new value to the ClipSeparators property.

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 added at the bottom of the grid.

For example:

```
fog.FormatString = "=Rec#|Name          |Phone|Hired"
fog.Cols = 4
fog.Rows = 1
fog.AddItem fog.Rows & vbTab & "Paul" & vbTab & "555-1212" & vbTab & "12/10/1997"
fog.AddItem fog.Rows & vbTab & "John" & vbTab & "555-1313" & vbTab & "12/10/1997",
```

This code adds a row, then inserts another one above the first.

To remove rows, use the RemoveItem method. Alternatively, you may add or remove rows at the bottom of the grid by setting the Rows property.
AfterCollapse Event

Fired after the user expands or collapses one or more rows in an outline.

Syntax
Private Sub VSFlexGrid_AfterCollapse( ByVal Row As Long, ByVal State As Integer)

Remarks
This event is fired when the grid is used in outline mode and the user expands or collapses one or more nodes.

The parameters for the AfterCollapse event are described below:

Row As Long
If a single node is being expanded or collapsed, this parameter has the row number of the node being expanded or collapsed. If multiple nodes are being collapsed or expanded, this parameter is set to -1.

State As Integer
If a single node is being expanded or collapsed, this parameter has the new state of the node. If multiple nodes are being collapsed or expanded, this parameter is set to the new outline level being displayed.

Possible node states are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexOutlineExpanded</td>
<td>0</td>
<td>The node is expanded: all subordinate nodes are visible.</td>
</tr>
<tr>
<td>flexOutlineSubtotals</td>
<td>1</td>
<td>The node is partially expanded: subordinates nodes are visible but collapsed.</td>
</tr>
<tr>
<td>flexOutlineCollapsed</td>
<td>2</td>
<td>The node is collapsed: all subordinate nodes are hidden.</td>
</tr>
</tbody>
</table>

Single nodes are expanded or collapsed when the user clicks the outline tree (see the OutlineBar property) or when a new node state is assigned to the IsCollapsed property of a specific row. Multiple nodes are expanded or collapsed when the user clicks the outline buttons across the top of the outline tree or when the Outline method is invoked by the host application.

See also the BeforeCollapse event.

AfterDataRefresh Event

Fired after reading data from the record source.

Syntax
Private Sub VSFlexGrid_AfterDataRefresh()

Remarks
The AfterDataRefresh event is useful when the control is bound to a recordset and you to perform certain operations on the data whenever it is refreshed. For example, you might want to display subtotals or add special formatting to certain columns or cells.

When the source recordset changes, all existing columns are destroyed and recreated from scratch. In this process, most column properties are reset to their default values. Thus, if you set up your columns using the ColEditMask, ColFormat, ColComboList, ColImageList, etc you should do it in response to the AfterDataRefresh event.

The AfterDataRefresh event fires when a new recordset is assigned to the controls.
**DataSource** property, or when the current source recordset is modified (e.g. records are added or deleted). This event only fires when the **DataMode** property is set to a non-zero value.

### AfterEdit Event
Fired after the control exits cell edit mode.

**Syntax**

```vbscript
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. It is useful for performing actions such as re-sorting the data or calculating subtotals.

The **AfterEdit** event is not adequate for performing data 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
Fired after a column is moved by dragging on the ExplorerBar.

**Syntax**

```vbscript
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 using the **ExplorerBar**. It is not fired if the column was moved with the **ColPosition** property.

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

- **Col As Long**
  This parameter holds the index of the column that was moved.

- **Position As Long**
  This parameter holds the new position of the column.

This event is useful if you want to synchronize some other user-interface element to the columns on a grid, or to keep track of a column's position. To prevent the user from moving certain columns to certain positions, use the **BeforeMoveColumn** event instead.

### AfterMoveRow Event
Fired after a row is moved by dragging on the ExplorerBar or using the DragRow method.

**Syntax**

```vbscript
Private Sub VSFlexGrid_AfterMoveRow(ByVal Row As Long, Position As Long)
```
AfterRowColChange Event
Fired after the current cell (Row, Col) changes to a different cell.

Syntax
Private Sub VSFlexGrid_AfterRowColChange(ByVal OldRow As Long, ByVal OldCol As Long, ByVal NewRow As Long, ByVal NewCol As Long)

Remarks
This event is fired after the Row or Col properties change, either as a result of user actions (mouse or keyboard) or through code.

This event is useful if you want to display additional information about the currently selected row, column, or cell. To perform validation or prevent certain cells from being selected, use the BeforeRowColChange and BeforeSelChange events instead.

AfterScroll Event
Fired after the control scrolls.

Syntax
Private Sub VSFlexGrid_AfterScroll(ByVal OldTopRow As Long, ByVal OldLeftCol As Long, ByVal NewTopRow As Long, ByVal NewLeftCol As Long)

Remarks
This event is fired whenever the TopRow or LeftCol properties change, either as a result of user actions (keyboard or mouse) or through code.

For example, the following code ensures that a cell remains visible while it is being edited:

Private Sub fg_AfterScroll(ByVal OldTopRow As Long, _
ByVal OldLeftCol As Long, _
ByVal NewTopRow As Long, _
ByVal NewLeftCol As Long)
If fg.EditWindow <> 0 Then fg.ShowCell fg.Row, fg.Col
End Sub

You can prevent the user from scrolling the control by handling the BeforeScroll event and setting the Cancel parameter to True. You can control the visibility of the scrollbars using the ScrollBars property.

AfterSelChange Event
Fired after the selected range (RowSel, ColSel) changes.

Syntax
Private Sub VSFlexGrid_AfterSelChange(ByVal OldRowSel As Long, ByVal OldColSel As Long, ByVal NewRowSel As Long, ByVal NewColSel As Long)

Remarks
This event is fired after the RowSel or ColSel properties change, either as a result of user actions (mouse or keyboard) or through code.
This event is useful if you want to display additional information about the current selection. To perform validation or prevent certain cells from being selected, use the \texttt{BeforeRowColChange} and \texttt{BeforeSelChange} events instead.

### AfterSort Event

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

**Syntax**

\begin{verbatim}
Private Sub VSFlexGrid\_AfterSort( ByVal Col As Long, Order As Integer)
\end{verbatim}

**Remarks**

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

This event is useful if you want to update user interface elements to reflect the new sorting. Note that the \texttt{ExplorerBar} property has new settings that cause the sorting order to be displayed automatically as arrow icons on the header rows.

To prevent certain columns from being sorted, or to alter their default sorting order, use the \texttt{BeforeSort} event instead.

### AfterUserFreeze Event

Fired after the user changes the number of frozen rows or columns.

**Syntax**

\begin{verbatim}
Private Sub VSFlexGrid\_AfterUserFreeze()
\end{verbatim}

**Remarks**

This event is fired whenever the \texttt{FrozenRows} or \texttt{FrozenCols} properties change, either as a result of user action or through code.

The user may change the number of frozen rows and columns by dragging the solid line that divides the frozen and scrollable areas of the grid, depending on the setting of the \texttt{AllowUserFreezing} property.

This event is useful if you want to synchronize user interface elements with the number of frozen rows or columns.

### AfterUserResize Event

Fired after the user resizes a row or a column.

**Syntax**

\begin{verbatim}
Private Sub VSFlexGrid\_AfterUserResize( ByVal Row As Long, ByVal Col As Long)
\end{verbatim}

**Remarks**

The user may resize rows and columns by dragging the edges of fixed rows and columns, depending on the setting of the \texttt{AllowUserResizing} property. The user may also double-click the edges of fixed rows and columns to automatically resize columns to fit the widest entries, depending on the setting of the \texttt{AutoSizeMouse} 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.

You may prevent specific rows and columns from being resized using the **BeforeUserResize** event.

### Aggregate Property

Returns an aggregate function (sum, average, etc.) for a given range.

**Syntax**

```vba
val# = [form!]VSFlexGrid.Aggregate(Aggregate As SubtotalSettings, Row1 As Long, Col1 As Long, Row2 As Long, Col2 As Long)
```

**Remarks**

This property is used to quickly calculate totals, averages and other aggregates over a range of cells.

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

**Aggregate** As SubtotalSettings

This parameter defines the type of aggregate function to use. Valid settings for this parameter are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexSTSum</td>
<td>2</td>
<td>Returns the sum of all numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTCount</td>
<td>4</td>
<td>Returns the number of numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTAverage</td>
<td>5</td>
<td>Returns the average of all numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTMax</td>
<td>6</td>
<td>Returns the maximum of all numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTMin</td>
<td>7</td>
<td>Returns the minimum of all numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTStd</td>
<td>8</td>
<td>Returns the standard deviation of all numeric entries in the given range.</td>
</tr>
<tr>
<td>flexSTVar</td>
<td>9</td>
<td>Returns the variance of all numeric entries in the given range.</td>
</tr>
</tbody>
</table>

**Row1 As Long, Col1 As Long, Row2 As Long, Col2 As Long**

These parameters define the range over which the aggregate is to be calculated. If you set **Row1** and **Row2** to -1, all selected rows are used (the selection does not have to be continuous).

For example,

```vba
Debug.Print fg.Aggregate(flexSTCount, fg.FixedRows, 2, fg.Rows - 1, 2)
Debug.Print fg.Aggregate(flexSTSum, fg.FixedRows, 2, fg.Rows - 1, 2)
Debug.Print fg.Aggregate(flexSTSum, -1, 2, -1, 2)
```

The first statement would print the number of numeric entries in column 2, and the second their sum.

The last statement would print the sum of all numeric entries in column 2 on currently selected rows. This type of calculation is especially useful when the **SelectionMode** property is set to **flexSelectionListBox** (3).

**Data Type**

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

Syntax

\[\text{form}!\text{VSFlexGrid.AllowBigSelection} = \{\text{True} \mid \text{False}\}\]

Remarks
If AllowBigSelection is set to True, clicking on the top left fixed cell selects the entire grid.

Data Type
Boolean

Default Value
True

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

Syntax

\[\text{form}!\text{VSFlexGrid.AllowSelection} = \{\text{True} \mid \text{False}\}\]

Remarks
Set this property to False to prevent users from extending the selection by clicking and dragging or using the shift plus cursor keys. In this case, clicking and dragging the mouse will move the current cell, but will not extend the selection.

This capability is useful when using the VSFlexGrid control to implement certain user interface elements such as menus, property sheets, or explorer-style trees.

Data Type
Boolean

Default Value
True

AllowUserFreezing Property
Returns or sets whether the user is allowed to freeze rows and columns with the mouse.

Syntax

\[\text{form}!\text{VSFlexGrid.AllowUserFreezing} = \text{AllowUserFreezeSettings}\]

Remarks
Frozen cells can be selected and edited, but they remain visible when the user scrolls the contents of the control. The AllowUserFreezing property determines whether the user can change the number of frozen rows and columns by dragging the solid line between the frozen and scrollable areas of the grid.

The settings for the AllowUserFreezing property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexFreezeNone</td>
<td>0</td>
<td>The user cannot change the number of frozen rows or</td>
</tr>
</tbody>
</table>
The user can change the number of frozen columns.

flexFreezeRows 2 The user can change the number of frozen rows.

flexFreezeBoth 3 The user can change the number of frozen rows and columns.

This property is especially useful when the grid is used as a data browser. It allows users to freeze the leftmost columns of the data while they scroll the control to view the remaining columns.

The number of frozen rows and columns can be set or retrieved through the `FrozenRows` property and `FrozenCols` properties. You may customize the appearance of the frozen areas of the grid using the `BackColorFrozen` and `ForeColorFrozen` properties. The solid line between the frozen and scrollable areas of the grid is drawn using the color specified by the `SheetBorder` property.

**Data Type**

Default Value

`flexFreezeNone (0)`

**AllowUserResizing Property**

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flexResizeNone</code></td>
<td>0</td>
<td>The user may not resize rows or columns.</td>
</tr>
<tr>
<td><code>flexResizeColumns</code></td>
<td>1</td>
<td>The user may resize column widths.</td>
</tr>
<tr>
<td><code>flexResizeRows</code></td>
<td>2</td>
<td>The user may resize row heights.</td>
</tr>
<tr>
<td><code>flexResizeBoth</code></td>
<td>3</td>
<td>The user may resize column widths and row heights.</td>
</tr>
<tr>
<td><code>flexResizeBothUniform</code></td>
<td>4</td>
<td>The user may resize column widths and row heights. When a row height is resized, the new height is applied to all rows.</td>
</tr>
</tbody>
</table>

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.

A group of columns is selected (from first to last row) and the user resizes one of them, all selected columns are resized. The same applies to rows. To allow users to select entire rows and columns, set the `AllowBigSelection` property to True.

If column sizing is allowed and the `AutoSizeMouse` property is set to True, users may double-click the resizing area to resize a column so it will automatically fit the longest entry.

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:

```delphi
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)

**Appearance** Property

Returns or sets the paint style of the control on an MDIForm or Form object.

**Syntax**

```vbnet
(form!)VSFlexGrid.Appearance[ = AppearanceSettings ]
```

**Remarks**

The Appearance Property has a new setting:

- **flex3DLight** creates a thin 3-D border around the control.

**Data Type**

AppearanceSettings (Enumeration)

**Archive Method**

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

**Syntax**

```vbnet
(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 specifies the action to perform on the archive. Valid settings are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arcAdd</td>
<td>0</td>
<td>Adds the file <strong>FileName</strong> to the archive <strong>ArcFileName</strong>, compressing it. If the</td>
</tr>
</tbody>
</table>
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 1 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 2 Removes the file FileName from the archive ArcFileName.

arcExtract 3 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 specified.

For example, the code below creates or opens an archive file named "c:\arc.fg", then adds
two files to the archive:

    fg.Archive "c:\arc.fg", "c:\autoexec.bat", arcAdd
    fg.Archive "c:\arc.fg", "c:\config.sys", arcAdd

ArchiveInfo Property

Returns information from a VSFilex 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
The type of information to retrieve from the archive. Valid settings are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arcFileCount</td>
<td>0</td>
<td>Returns the number of files in the archive.</td>
</tr>
<tr>
<td>arcFileName</td>
<td>1</td>
<td>Returns the name of a file in the archive.</td>
</tr>
<tr>
<td>arcFileSize</td>
<td>2</td>
<td>Returns the original size of a file in the archive.</td>
</tr>
<tr>
<td>arcFileCompressedSize</td>
<td>3</td>
<td>Returns the compressed size of a file in the archive.</td>
</tr>
<tr>
<td>arcFileDate</td>
<td>4</td>
<td>The date of the last modification made to a file in the archive.</td>
</tr>
</tbody>
</table>

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. It may be omitted only when retrieving the file
count.

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

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

Data Type
Variant

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

Syntax
[form!]VSFlexGrid.AutoSearch[ = AutoSearchSettings ]

Remarks
The settings for the AutoSearch property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexSearchNone</td>
<td>0</td>
<td>No auto search.</td>
</tr>
<tr>
<td>flexSearchFromTop</td>
<td>1</td>
<td>When the user types, start searching from the first row.</td>
</tr>
<tr>
<td>flexSearchFromCursor</td>
<td>2</td>
<td>When the user types, start searching from the current row.</td>
</tr>
</tbody>
</table>

If AutoSearch is on, the control will search the current column as the user types, automatically moving the cursor and highlighting partial matches. The search is
case-insensitive. The search is canceled when the user presses the Escape key or moves the selection with the mouse or cursor keys.

When the user stops typing for about two seconds, the search buffer is reset. This amount of time can be changed by setting the \texttt{AutoSearchDelay} property.

If \texttt{AutoSearch} is on and the \texttt{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.

This property only affects the behavior of the grid itself. To automatically select options as the user types into a combo list or list box, use the \texttt{ComboSearch} property.

\textbf{Data Type}

AutoSearchSettings (Enumeration)

\textbf{Default Value}

flexSearchNone (0)

\textbf{AutoSearchDelay Property}

Returns or sets the delay, in seconds, before the AutoSearch buffer is reset.

\textbf{Syntax}

\begin{verbatim}
[form!]VSFlexGrid.AutoSearchDelay[ = value As Single ]
\end{verbatim}

\textbf{Remarks}

This property is only used when the \texttt{AutoSearch} property is set to a non-zero value.

\textbf{Data Type}

Single

\textbf{Default Value}

2

\textbf{AutoSize Method}

Resizes column widths or row heights to fit cell contents.

\textbf{Syntax}

\begin{verbatim}
[form!]VSFlexGrid.AutoSize Col1 As Long, [ Col2 As Long ], [ Equal As Boolean ],
[ ExtraSpace As Single ]
\end{verbatim}

\textbf{Remarks}

The parameters for the \texttt{AutoSize} method are described below:

\texttt{Col1} As Long, \texttt{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 \texttt{Cols} -1. \texttt{Col2} is optional. If it is omitted, only \texttt{Col1} is resized.

\texttt{Equal} As Boolean (optional)

If True, all columns between \texttt{Col1} and \texttt{Col2} are set to the same width. If False, then each column is resized independently. This parameter is optional and defaults to False.

\texttt{ExtraSpace} As Single (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. This parameter is optional and defaults to zero.

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

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexAutoSizeColWidth</td>
<td>0</td>
<td>Adjust column widths to accommodate the widest entry in each column.</td>
</tr>
<tr>
<td>flexAutoSizeRowHeight</td>
<td>1</td>
<td>Adjust row heights to accommodate the longest entry in each row.</td>
</tr>
</tbody>
</table>

The **flexAutoSizeRowHeight** setting 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)

### AutoSizeMouse Property

Returns or sets whether columns should be resized to fit when the user double-clicks on the header row.

**Syntax**

```
[form!]VSFlexGrid.AutoSizeMouse[ = {True | False} ]
```

**Remarks**

If **AllowUserResizing** is set to a value that allows columns resizing, the mouse cursor changes as the user moves the mouse near the edge of fixed header cells to indicate resizing is possible. If **AutoSizeMouse** is set to True and the user double-clicks while the resize mouse cursor is displayed, the column will be resized automatically to fit the widest entry on the column.

**Data Type**
Boolean

**Default Value**
True

**BackColorAlternate Property**
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 non-zero value, 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**
0 (Disabled)

**BackColorBkg Property**
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 that shows which colors are used to paint which areas of the grid.

**Data Type**
Color

**BackColorFixed Property**
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 that shows which colors are used to paint which areas of the grid.
Data Type
Color

BackColorFrozen Property
Returns or sets the background color of the frozen rows and columns.

Syntax
[form!]VSFlexGrid.BackColorFrozen[ = colorref& ]

Remarks
If you set this property to zero, the frozen areas of the grid are painted using the color specified by the BackColor property. In this case, the boundary between frozen and scrollable cells is still visible as a solid line painted in the color specified by the SheetBorder property.

See the BackColor property for a diagram that shows which colors are used to paint which areas of the grid.

Data Type
Color

Default Value
0 (Disabled)

BackColorSel Property
Returns or sets the background color of the selected cells.

Syntax
[form!]VSFlexGrid.BackColorSel[ = colorref& ]

Remarks
See the BackColor property for a diagram that shows which colors are used to paint which areas of the grid.

Data Type
Color

BeforeCollapse Event
Fired before the user expands or collapses one or more rows in an outline.

Syntax
Private Sub VSFlexGrid_BeforeCollapse( ByVal Row As Long, ByVal State As Integer, Cancel As Boolean)

Remarks
This event is fired when the grid is in outline mode, before a node is expanded or collapsed.
either through user action (clicking on the **OutlineBar**) or through code (setting the **IsCollapsed** property or invoking the **Outline** method).

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

**Row As Long**
If a single node is being expanded or collapsed, this parameter has the row number of the node being expanded or collapsed. If multiple nodes are being collapsed or expanded, this parameter is set to -1.

**State As Integer**
If a single node is being expanded or collapsed, this parameter has the new state of the node. If multiple node are being collapsed or expanded, this parameter is set to the new outline level being displayed. Possible node states are described below.

**Cancel As Boolean**
Set this parameter to True to prevent the node from being collapsed or expanded.

Possible node states are:

<table>
<thead>
<tr>
<th><strong>Constant</strong></th>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>flexOutlineExpanded</td>
<td>0</td>
<td>The node is expanded: all subordinate nodes are visible.</td>
</tr>
<tr>
<td>flexOutlineSubtotals</td>
<td>1</td>
<td>The node is partially expanded: subordinates nodes are visible but collapsed.</td>
</tr>
<tr>
<td>flexOutlineCollapsed</td>
<td>2</td>
<td>The node is collapsed: all subordinate nodes are hidden.</td>
</tr>
</tbody>
</table>

Single nodes are expanded or collapsed when the user clicks the outline tree (see the **OutlineBar** property) or when a new node state is assigned to the **IsCollapsed** property of a specific row. Multiple nodes are expanded or collapsed when the user clicks the outline buttons across the top of the outline tree or when the **Outline** method is invoked by the host application.

This event is especially useful for building outlines asynchronously. One of the samples provided on the distribution CD uses this technique to build a directory tree display. Branches are added to the tree when the user expands each directory. In this case, building the entire tree when the control loads would take too long.

The sample code below creates an outline with five nodes, each of which contains an empty "dummy" child node. Before a parent node is expanded, the dummy child is removed and five new parent nodes are added, each with a dummy child. Thus the outline tree is build dynamically and can grow indefinitely.

```vba
Private Sub Form_Load()
  fg.Rows = 1
  fg.Cols = 1
  fg.FixedCols = 0
  fg.ExtendLastCol = True
  fg.OutlineBar = flexOutlineBarSimpleLeaf
  AddNodeGroup 1, 0
End Sub

Private Sub fg_BeforeCollapse(ByVal Row As Long, ByVal State As Integer, Cancel As Boolean)
  If Row < 0 Then Cancel = True: Exit Sub
  If State = flexOutlineCollapsed Then Exit Sub
  If fg.TextMatrix(Row + 1, 0) <> "Dummy" Then Exit Sub
  fg.RemoveItem Row + 1
  AddNodeGroup Row + 1, fg.RowOutlineLevel(Row) + 1
End Sub
```
End Sub

Sub AddNodeGroup(r&, level&)
    Dim i%
    For i = 1 To 5
        fg.AddItem "Row " & fg.Rows, r
        fg.AddItem "Dummy", r + 1
        fg.IsSubtotal(r) = True
        fg.RowOutlineLevel(r) = level
        fg.IsCollapsed(r) = flexOutlineCollapsed
        r = r + 2
    Next
End Sub

The control is initialized in the Form_Load event. The OutlineBar property is used to display an outline tree, and the AddNode routine is called to add a block of five nodes with dummy children.

Before a node is expanded or collapsed, the BeforeCollapse event is fired. If the Row parameter is negative, the operation is canceled (this happens when the user shift-clicks a node to expand or collapse all nodes at that level). If the node is being collapsed or the child node is not a dummy, the function exits without further processing. Otherwise, the dummy is removed and replaced with a new group of collapsed nodes.

BeforeDataRefresh Event

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 to a recordset, immediately before a batch of data is loaded.

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

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 before the control enters edit mode and before an editable cell is repainted when it has the focus. It allows you to prevent editing by setting the Cancel parameter to True, to supply a list of choices for a combo list with the ComboList property, or to specify an edit mask with the EditMask property.

If the choices or the mask are the same for a whole column, you may set them with the ColComboList and ColEditMask properties, and you don't need to handle the BeforeEdit
The parameters for the **BeforeEdit** event are described below:

- **Row** As Long,  **Col** As Long
  Indicate which cell is about to be edited or repainted.

- **Cancel** As Boolean
  Allows you to cancel the editing operation.

Because **BeforeEdit** is fired before each repaint, it does not guarantee that the control is really about to enter edit mode. For that, use the **StartEdit** event instead. If the user starts editing by pressing a key, several events get fired in the following sequence:

```vbs
KeyDown 65       ' user pressed the 'a' key (65 = 'A')
BeforeEdit 1 1   ' allows you to cancel the editing
StartEdit 1 1    ' not canceled, edit is about to start
KeyPressEdit 97  ' 'a' key passed to editor (97 = 'a')
KeyUpEdit 65     ' user released the key
KeyDownEdit 13   ' user pressed Enter
ValidateEdit     ' allows you to validate the edit
AfterEdit 1 1    ' editing done
BeforeEdit 1 1   ' repainting cell
KeyUp 13         ' user released Enter key
```

---

### BeforeMouseDown Event

Fired before the control processes the MouseDown event.

**Syntax**

```vbs
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. The default mouse processing is canceled so the control does not modify the selection as the user drags the mouse.

```vbs
Private Sub fa_BeforeMouseDown(ByVal Button As Integer, ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single, Cancel As Boolean)
    If Shift <> 1 Then Exit Sub ' use shift to drag (ctrl selects)
    Cancel = True               ' cancel remaining mouse events
    Dim i As Long               ' build a list of what we'll be dragging
    fa.Tag = ""
    For i = 0 To fa.SelectedRows - 1
        fa.Tag = fa.Tag & vbCrLf & vbTab & fa.Cell(flexcpText, fa.SelectedRow(i), 0)
    Next
    fa.Drag                     ' start dragging
End Sub
```
BeforeMoveColumn Event
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, the following code prevents Column 1 from being moved to another position and other columns from being moved to its position:

Private Sub VSFlexGrid1_BeforeMoveColumn(ByVal Col As Long, Position As Long)
If Col = 1 Then Position = 1
If Col <> 1 And Position = 1 Then Position = Col
End Sub

BeforeMoveRow Event
Fired before a row is moved by dragging on the ExplorerBar or using the DragRow method.

Syntax
Private Sub VSFlexGrid_BeforeMoveRow(ByVal Row As Long, Position As Long)

BeforePageBreak Event
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 printed to allow control over page breaks.

Set the BreakOK parameter to True to indicate that row number Row should be allowed to print at the top of a page, or set it to False to indicate otherwise. For example, you would set BreakOK to True if Row is a subtotal or a heading row.

The control may be printed with the PrintGrid method or with a VSPrinter control. The advantage of using the VSPrinter control is that it provides print previewing, the ability to integrate many grids and other graphical elements on a single document, and complete control over the printer. The VSPrinter control is available from VideoSoft as a separate product.

BeforeRowColChange Event
Fired before the current cell (Row, Col) changes to a different cell.
Private Sub VSFlexGrid_BeforeRowColChange(ByVal OldRow As Long, ByVal OldCol As Long, ByVal NewRow As Long, ByVal NewCol As Long, Cancel As Boolean)

Remarks
This event gets fired before the Row and Col properties change, either as a result of user actions or through code. It allows you to prevent the selection of certain cells, thus creating "protected" ranges on a grid.

BeforeRowColChange It is fired only when the Row and Col property are about to change. To prevent the extended selection of a range, you also need to handle the BeforeSelChange event.

For example, the following code creates a protected range with a green background and prevents the user from selecting any cells on the protected range and from extending any selections into the protected area:

' highlight protected range
Private Sub Form_Load()
    fg.Cell(flexcpBackColor, 2, 2, 8, 4) = RGB(200, 250, 200)
End Sub

' cancel if new cell is in protected area
Private Sub fg_BeforeRowColChange(ByVal OldRow As Long, ByVal OldCol As Long, ByVal NewRow As Long, ByVal NewCol As Long, Cancel As Boolean)
    If NewRow >= 2 And NewRow <= 8 And NewCol >= 2 And NewCol <= 4 Then
        Cancel = True
    End Sub

' cancel if new selection is on protected area
Private Sub fg_BeforeSelChange(ByVal OldRowSel As Long, ByVal OldColSel As Long, ByVal NewRowSel As Long, ByVal NewColSel As Long, Cancel As Boolean)
    If (fg.Row < 2 And NewRowSel < 2) Or (fg.Col < 2 And NewColSel < 2) Then Exit Sub
    If (fg.Row > 8 And NewRowSel > 8) Or (fg.Col > 4 And NewColSel > 4) Then Exit Sub
    Cancel = True
End Sub

BeforeScroll Event
Fired before the control scrolls.

Syntax
Private Sub VSFlexGrid_BeforeScroll(ByVal OldTopRow As Long, ByVal OldLeftCol As Long, ByVal NewTopRow As Long, ByVal NewLeftCol As Long, Cancel As Boolean)
Cancel As Boolean)
    If fg.EditWindow <> 0 And OldTopRow <> NewTopRow Then Cancel = True
End Sub

BeforeScrollTip Event
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 example the following code displays a tool tip as the user drags the vertical scroll bar thumb. The tooltip displays information about the new top row that will be visible when the user stops scrolling:

Private Sub Form_Load()
    fg.ScrollTips = True
    fg.Rows = 1
    While fg.Rows < 200
        fg.AddItem vbTab & fg.Rows
    Wend
End Sub

Private Sub fg_BeforeScrollTip(ByVal Row As Long)
    fg.ScrollTipText = "New Top Row: " & fg.TextMatrix(Row, 1)
End Sub

BeforeSelChange Event
Fired before the selected range (RowSel, ColSel) changes.

Syntax
Private Sub VSFlexGrid_BeforeSelChange(ByVal OldRowSel As Long, ByVal OldColSel As Long, ByVal NewRowSel As Long, ByVal NewColSel As Long, Cancel As Boolean)

Remarks
This event is fired before the RowSel and ColSel properties change, either as a result of user actions or through code. It allows you to prevent the the selection of certain cells, thus creating "protected" ranges on a grid.

To prevent the selection of a range, you also need to handle the BeforeRowColChange event, which is fired before the Row and Col properties change.

For example, the following code creates a protected range with a green background and prevents the user from selecting any cells on the protected range and from extending any selections into the protected area:

' highlight protected range
Private Sub Form_Load()
    fg.Cell(flexcpBackColor, 2, 2, 8, 4) = RGB(200, 250, 200)
End Sub
Private Sub fg_BeforeRowColChange(OldRow As Long, OldCol As Long, NewRow As Long, NewCol As Long, Cancel As Boolean)
    If NewRow >= 2 And NewRow <= 8 And NewCol >= 2 And NewCol <= 4 Then Cancel = True
End Sub

' cancel if new selection is on protected area
Private Sub fg_BeforeSelChange(OldRowSel As Long, OldColSel As Long, NewRowSel As Long, NewColSel As Long, Cancel As Boolean)
    If (fg.Row < 2 And NewRowSel < 2) Or (fg.Col < 2 And NewColSel < 2) Then Exit Sub
    If (fg.Row > 8 And NewRowSel > 8) Or (fg.Col > 4 And NewColSel > 4) Then Exit Sub
    Cancel = True
End Sub

BeforeSort Event
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
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 using the mouse, 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.
For example, the code below prevents the user from resizing columns 0 and 1:

```vba
Private Sub Form_Load()
    fg.AllowUserResizing = flexResizeColumns
End Sub

Private Sub fg_BeforeUserResize(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
    If Col = 0 Or Col = 1 Then
        Cancel = True
    End If
End Sub
```

### BindToArray Method

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

**Syntax**

```vba
form!VSFlexGrid.BindToArray [ VariantArray As Variant ], [ RowDim As Long ], [ ColDim As Long ], [ PageDim As Long ], [ CurrentPage As Long ]
```

**Remarks**

This method allows you to bind the `VSFlexGrid` control to a Visual Basic Variant Array or to another `VSFlexGrid` control.

When bound to an array, the grid displays values retrieved from the array and automatically writes any modifications back into the array. 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 with the `GetRows` method. 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.

If you change the contents or dimensions of the array, 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:

```
fg.BindToArray Null
```

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

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

' Default binding:
fg.BindToArray arr
```
' fg now has 5 non-fixed columns (0-4) and 9 non-fixed rows (0-8).
' the mapping is: arr(i, j) = fg.TextArray(j - fg.FixedRows, i - fg.FixedCols)

' Transposed binding:
fg.BindToArray arr, 0, 1
' fg now has 9 non-fixed columns (0-8) and 5 non-fixed rows (0-4).
' the mapping is: arr(i, j) = fg.TextArray(i - fg.FixedRows, j - fg.FixedCols)

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

' Default binding:
fg.BindToArray arr
' 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).
' the mapping is: arr(i, j, 0) = fg.TextArray(j - fg.FixedRows, i - fg.FixedCols)

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

fg.BindToArray fgSource

In this case, the fg control will display the data stored in the fgSource 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.

To load data from an array or another VSFlexGrid control without binding, use the LoadArray method instead.

**Bookmark Property**

Returns the Bookmark associated with a given row when the grid is bound to an ADO recordset.

**Syntax**

```
[form!] VSFlexGrid.Bookmark (Row As Long) [ = value As Variant ]
```

**Data Type**

Variant

**BottomRow Property**

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. To ensure that a given cell is visible, use the ShowCell method.

The following line prints the number of the bottom most row currently visible:

    Debug.Print fg.BottomRow

Data Type
Long

BuildComboBox Method
Returns a ColComboBox string from data in a recordset.

Syntax
[form!]VSFlexGrid.BuildComboBox rs As Object, FieldList As String, [ KeyField As Variant ], [ BackColor As Variant ]

Remarks
The VSFlexGrid control has extensive support for drop-down lists and combo-lists when editing cells. This support includes multi-column drop-down lists, automatic value translation, and default field highlighting and relies on two properties, ComboList and ColComboBox.

The syntax for creating the lists is simple, but creating the lists can be tedious. The BuildComboBox method builds a ColComboBox automatically from a recordset, reducing the amount of programming required.

The parameters for the BuildComboBox method are described below:

rs As Object
An ADO or DAO recordset containing the values to be displayed on the grid.

FieldList As String
A string containing a comma-separated list of field names to be displayed. If the list contains more than one field, a multi-column drop-down list is created. You may define a default field (displayed in the cell when the list is closed) by preceding its name with an asterisk.

KeyField As String (optional)
The name of the field to be used as a key. This field must be numeric, and is usually the recordset's ID field.

BackColor As OLE_COLOR (optional)
The color used to paint the background of the default field while the list is displayed. If omitted, the whole list is painted using the current cells background color.

For example, the code below displays a list of products from the NorthWind database. The "SupplierID" and "CategoryID" columns use translated lists built with the BuildComboBox method to convert the numeric IDs into company and category names. The code assumes you have created the following controls on the form:
Control Name | Description
---|---
fg | VSFlexGrid control, OLEDB version.
dataProducts | ADO Data Control bound to the NorthWind database, Products table.
dataSuppliers | ADO Data Control bound to the NorthWind database, Suppliers table.
dataCategories | ADO Data Control bound to the NorthWind database, Categories table.

' bind grid to Products table
Private Sub Form_Load()
    fg.Editable = flexEDKbdMouse
    Set fg.DataSource = dataProducts
End Sub

Private Sub fg_AfterDataRefresh()
    ' map Suppliers column to display supplier info
    Dim s$
    dataSuppliers.Refresh
    s = fg.BuildComboList(dataSuppliers.Recordset, "*CompanyName,ContactName", "SupplierID", vbYellow)
    fg.ColComboList(fg.ColIndex("SupplierID")) = s
    
    ' map Category column to display category info
    dataCategories.Refresh
    s = fg.BuildComboList(dataCategories.Recordset, "*CategoryName,Description", "CategoryID", vbYellow)
    fg.ColComboList(fg.ColIndex("CategoryID")) = s
    
    ' make sure all columns are visible
    fg.AutoSize 1, fg.Cols - 1
End Sub

Cell Property
Returns or sets 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 you to read or set cell properties directly to individual cells or ranges (without selecting them). The Cell property has a new flexcpSort setting that allows you to sort a range without changing the selection.

The parameters for the Cell property are described below:

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

Row1, Col1, Row2, and Col2 As Long (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.

Valid settings for the Setting parameter are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexcpText</td>
<td>0</td>
<td>Returns or sets the cell's text (or clip string for selections).</td>
</tr>
<tr>
<td>flexcpTextStyle</td>
<td>1</td>
<td>Returns or sets the cell's text style (see CellTextStyle).</td>
</tr>
<tr>
<td>flexcpAlignment</td>
<td>2</td>
<td>Returns or sets the cell's text alignment (see CellAlignment).</td>
</tr>
<tr>
<td>flexcpPicture</td>
<td>3</td>
<td>Returns or sets the cell's picture (see CellPicture).</td>
</tr>
<tr>
<td>flexcpPictureAlignment</td>
<td>4</td>
<td>Returns or sets the cell's picture alignment (see CellPictureAlignment).</td>
</tr>
<tr>
<td>flexcpChecked</td>
<td>5</td>
<td>Returns or sets the state of the cell's check box (see CellChecked).</td>
</tr>
<tr>
<td>flexcpBackColor</td>
<td>6</td>
<td>Returns or sets the cell's back color (see CellBackColor).</td>
</tr>
<tr>
<td>flexcpForeColor</td>
<td>7</td>
<td>Returns or sets the cell's fore color (see CellForeColor).</td>
</tr>
<tr>
<td>flexcpFloodPercent</td>
<td>8</td>
<td>Returns or sets the cell's flood percent (see CellFloodPercent).</td>
</tr>
<tr>
<td>flexcpFloodColor</td>
<td>9</td>
<td>Returns or sets the cell's flood color (see CellFloodColor).</td>
</tr>
<tr>
<td>flexcpFont</td>
<td>10</td>
<td>Returns or sets the cell's font.</td>
</tr>
<tr>
<td>flexcpFont*</td>
<td>11-17</td>
<td>Returns or sets properties of the cell’s font (see CellFontName etc).</td>
</tr>
<tr>
<td>flexcpValue</td>
<td>18</td>
<td>Returns the value of the cell's text (read-only).</td>
</tr>
<tr>
<td>flexcpTextDisplay</td>
<td>19</td>
<td>Returns the cell's formatted text (read only).</td>
</tr>
<tr>
<td>flexcpData</td>
<td>20</td>
<td>Returns or sets a Variant attached to the cell.</td>
</tr>
<tr>
<td>flexcpCustomFormat</td>
<td>21</td>
<td>Returns True if the cell has custom formatting (set to False to remove all custom formatting).</td>
</tr>
<tr>
<td>flexcpLeft</td>
<td>22</td>
<td>Returns a cell's left coordinate, in twips, taking merging into account (read-only).</td>
</tr>
<tr>
<td>flexcpTop</td>
<td>23</td>
<td>Returns a cell's top coordinate, in twips, taking merging into account (read-only).</td>
</tr>
<tr>
<td>flexcpWidth</td>
<td>24</td>
<td>Returns a cell's width, in twips, taking merging into account (read-only).</td>
</tr>
<tr>
<td>flexcpHeight</td>
<td>25</td>
<td>Returns a cell's height, in twips, taking merging into account (read-only).</td>
</tr>
<tr>
<td>flexcpVariantValue</td>
<td>26</td>
<td>Returns a double if the cell contains a numeric value or a string otherwise (read-only).</td>
</tr>
<tr>
<td>flexcpRefresh</td>
<td>27</td>
<td>Set to True to force a cell or range to be repainted.</td>
</tr>
</tbody>
</table>

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. These values are Variants, which means you may associate virtually any type of data with 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:

```vba
fg.CellFontName = "Arial"
fg.CellFontSize = 8
fg.CellFontBold = True
```

you may write

```vba
fg.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.

For example:

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

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

**Data Type**
Variant

---

**CellAlignment Property**

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

**Syntax**

```vba
[form]!VSFlexGrid.CellAlignment[ = AlignmentSettings ]
```

**Remarks**

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexAlignLeftTop</td>
<td>0</td>
</tr>
<tr>
<td>flexAlignLeftCenter</td>
<td>1</td>
</tr>
<tr>
<td>flexAlignLeftBottom</td>
<td>2</td>
</tr>
<tr>
<td>flexAlignCenterTop</td>
<td>3</td>
</tr>
<tr>
<td>flexAlignCenterCenter</td>
<td>4</td>
</tr>
<tr>
<td>flexAlignCenterBottom</td>
<td>5</td>
</tr>
<tr>
<td>flexAlignRightTop</td>
<td>6</td>
</tr>
<tr>
<td>flexAlignRightCenter</td>
<td>7</td>
</tr>
<tr>
<td>flexAlignRightBottom</td>
<td>8</td>
</tr>
</tbody>
</table>
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.

This sample selects the first seven cells in column 3 and centers the text.

With fg
  'select rows 1 through 7 in Column 3
  .Select 1, 3, 7, 3
  .FillStyle = flexFillRepeat
  .CellAlignment = flexAlignCenterCenter
  'return .FillStyle to its default (if needed)
  .FillStyle = flexFillSingle
End With

**Data Type**
AlignmentSettings (Enumeration)

### CellBackColor Property

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).

The following code only changes the back color of the current cell:

```
FG.CellBackColor = vbRed
```

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.

For example, the following code selects the first seven cells in column 3 and sets the BackColor for those cells:

```
With fg
  .Select 1, 3, 7, 3
  .FillStyle = flexFillRepeat
  .CellBackColor = &HFF       'Red
  'return .FillStyle to its default (if needed)
  .FillStyle = flexFillSingle
  'Make cell 1, 1 the current cell so we can view the change
  .Select 1, 1
End With
```

**Data Type**
Color
CellBorderStyle Method

Draws a border around and within the selected cells.

Syntax

```vbnet
[form!] VSFlexGrid.CellBorderStyle Color As OLE_COLOR, Left As Integer, Top As Integer, Right As Integer, Bottom As Integer, Vertical As Integer, Horizontal As Integer
```

Remarks

The **CellBorderStyle** 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 **CellBorderStyle** method using the following parameters:

- **Color As OLE_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:

```vbnet
Private Sub Form_Load()
  ' draw borders around a table
  fg.Select 1, 1, 4, 4
  fg.CellBorderStyle RGB(0, 0, 125), 2, 3, 2, 2, 1, 1

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

End Sub
```

The result looks like this:
CellButtonClick Event
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.).

By default, cell 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. You may customize their appearance by assigning a picture to the `CellButtonPicture` property.

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 assign the selected color to the cell background:

```vbscript
Private Sub Form_Load()
    fg.Editable = flexEDKbdMouse
    fg.ColComboList(1) = "...
End Sub

Private Sub fg_CellButtonClick(ByVal Row As Long, ByVal Col As Long)
    CommonDialog1.ShowColor
End Sub
```

CellButtonPicture Property
Returns or sets the picture used in cell buttons.

**Syntax**
```
(form!)VSFlexGrid.CellButtonPicture[ = Picture ]
```

**Remarks**
This property allows you to customize the appearance of cell buttons. For details on how to create and handle cell buttons, see the `CellButtonClick` event.

If you want to use a single picture for all cell buttons on the grid, assign the picture to the `CellButtonPicture` property at design time, or at the `Form_Load` event. To change pictures depending on the row, column, or cell being edited, trap the `BeforeEdit` event and set the picture accordingly. For example, the code below uses different pictures depending on the column being edited:

```vbscript
Private Sub fg_BeforeEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
    Select Case Col
        Case 2 ' ellipsis button
            fg.ComboList = "...
            Set fg.CellButtonPicture = Nothing
        Case 3 ' font button
            fg.ComboList = "...
            Set fg.CellButtonPicture = imgFont
    End Select
End Sub
```
Case 4 ' color button
  fg.ComboList = "..."
  Set fg.CellButtonPicture = imgColor
Case Else ' no button
  fg.ComboList = ""
  Set fg.CellButtonPicture = Nothing
End Select
End Sub

The pictures used for cell buttons should fit within the button (larger pictures are truncated). They should also be transparent, so the button face can be seen through the empty parts of the picture. For best results, use small icons (16 x 16 pixels) and draw the picture in the upper left 12 x 12 rectangle within the icon.

Data Type
Picture

CellChanged Event
Fired after a cell's contents change.

Syntax
Private Sub VSFlexGrid_CellChanged( ByVal Row As Long, ByVal Col As Long)

Remarks
This event allows you to perform some processing whenever the contents of a cell change, regardless of how they were changed (e.g. user typed data into the cell, data got loaded from a database, or data was assigned to the grid through code). This is useful to provide conditional formatting and dynamic data summaries (that get updated automatically whenever the data changes).

For example, the following code formats negative values in bold red:

Private Sub Form_Load()
  Dim r&, c&
  fg.Editable = flexEDKbdMouse
  fg.ColFormat(-1) = "#,###.##"
  For r = fg.FixedRows To fg.Rows - 1
    For c = fg.FixedCols To fg.Cols - 1
      fg.TextMatrix(r, c) = Rnd * 1000 - 500
      Next
    Next
  End Sub

Private Sub fg_CellChanged(ByVal Row As Long, ByVal Col As Long)
  If fg.TextMatrix(Row, Col) < 0 Then
    fg.Cell(flexcpForeColor, Row, Col) = RGB(200, 100, 100)
    fg.Cell(flexcpFontBold, Row, Col) = RGB(200, 100, 100)
  Else
    fg.Cell(flexcpCustomFormat, Row, Col) = False
  End If
End Sub
CellChecked Property

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flexNoCheckbox</code></td>
<td>0</td>
<td>The cell has no check box. This is the default setting.</td>
</tr>
<tr>
<td><code>flexChecked</code></td>
<td>1</td>
<td>The cell has a check box that is checked.</td>
</tr>
<tr>
<td><code>flexUnchecked</code></td>
<td>2</td>
<td>The cell has a check box that is not checked.</td>
</tr>
</tbody>
</table>

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.

For example, the code below makes column 1

```vbnet
Private Sub Form_Load()
    Dim r&
    For r = fg.FixedRows To fg.Rows - 1
        fg.Cell(flexcpChecked, r, 1) = flexUnchecked
        fg.Cell(flexcpText, r, 1) = "Row " & r
    Next
    fg.Editable = flexEDKbdMouse
End Sub

Private Sub Command1_Click()
    Dim r&
    For r = fg.FixedRows To fg.Rows - 1
        If fg.Cell(flexcpChecked, r, 1) = flexChecked Then
            Debug.Print fg.TextMatrix(r, 1); " is Checked"
        End If
    Next
End Sub
```

**Data Type**

`CellCheckedSettings` (Enumeration)

---

CellFloodColor Property

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

**Syntax**

```
[form!]VSFlexGrid.CellFloodColor[ = colorref& ]
```

**Remarks**

---

111
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) or **vbBlack**.

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
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.

For example, the following code makes it so the FloodColor will start from the right and fill 25% of the cell.

```
FG.CellFloodPercent = -25
```

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
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
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
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
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**
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**
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**
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**

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) or **vbBlack**.

**Data Type**

Color

---

**CellHeight Property**

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, CellWidth, CellTop, 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**

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**
The `CellHeight`, `CellWidth`, `CellTop`, 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**

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

**Syntax**
```vbscript
[form!]VSFlexGrid.CellPicture[ = Picture ]
```

**Remarks**
The `Picture` object assigned to this property may be retrieved from another control (e.g. the Image control's `Picture` property) or loaded from a disk file using Visual Basic's `LoadPicture` function. Using pictures in Visual C++ is a little more involved. For details on this, see the Using VSFlexGrid in Visual C++ topic.

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. If you want the text to be drawn over the picture, set the `PicturesOver` property to True.

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.

**Data Type**
Picture

**CellPictureAlignment Property**

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

**Syntax**
```vbscript
[form!]VSFlexGrid.CellPictureAlignment[ = PictureAlignmentSettings ]
```

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexPicAlignLeftTop</td>
<td>0</td>
</tr>
<tr>
<td>flexPicAlignLeftCenter</td>
<td>1</td>
</tr>
<tr>
<td>flexPicAlignLeftBottom</td>
<td>2</td>
</tr>
<tr>
<td>flexPicAlignCenterTop</td>
<td>3</td>
</tr>
<tr>
<td>flexPicAlignCenterCenter</td>
<td>4</td>
</tr>
<tr>
<td>flexPicAlignCenterBottom</td>
<td>5</td>
</tr>
</tbody>
</table>
flexPicAlignRightTop 6
flexPicAlignRightCenter 7
flexPicAlignRightBottom 8
flexPicAlignStretch 9
flexPicAlignTile 10

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

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexTextFlat</td>
<td>0</td>
<td>Draw text normally.</td>
</tr>
<tr>
<td>flexTextRaised</td>
<td>1</td>
<td>Draw text with a strong raised 3-D effect.</td>
</tr>
<tr>
<td>flexTextInset</td>
<td>2</td>
<td>Draw text with a strong inset 3-D effect.</td>
</tr>
<tr>
<td>flexTextRaisedLight</td>
<td>3</td>
<td>Draw text with a light raised 3-D effect.</td>
</tr>
<tr>
<td>flexTextInsetLight</td>
<td>4</td>
<td>Draw text with a light inset 3-D effect.</td>
</tr>
</tbody>
</table>

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

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, CellWidth, CellTop, 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
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, CellWidth, CellTop, 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
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 change or a new selection is made from a drop-down list.

You may use this event to provide help while the user browses the contents of a list. The current text being edited can be retrieved using the EditText property. For example:

Private Sub Form_Load()
    fg.Editable = flexEDKbdMouse
    fg.ColComboBox(1) = "Gold|Copper|Silver|Steel|Iron"
End Sub

Private Sub fg_ChangeEdit()
    Debug.Print "The current entry begins with " & Left(fg.EditText, 1)
End Sub

Clear Method (VSFlexGrid Control)
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 As ClearWhereSettings (optional)
Defines what part of the grid will be cleared. Valid settings for this parameter are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexClearEverywhere</td>
<td>0</td>
<td>Clear everywhere (this is the default setting).</td>
</tr>
<tr>
<td>flexClearScrollable</td>
<td>1</td>
<td>Clear scrollable region (excludes fixed rows and columns).</td>
</tr>
<tr>
<td>flexClearSelection</td>
<td>2</td>
<td>Clear current selection.</td>
</tr>
</tbody>
</table>

What As ClearWhatSettings (optional)
Defines what part of the grid’s information will be cleared. Valid settings for this parameter are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexClearEverything</td>
<td>0</td>
<td>Clear cell text, formatting and custom data (this is the default setting).</td>
</tr>
<tr>
<td>flexClearText</td>
<td>1</td>
<td>Clear text only.</td>
</tr>
<tr>
<td>flexClearFormatting</td>
<td>2</td>
<td>Clear custom formatting only (including pictures and cell data).</td>
</tr>
<tr>
<td>flexClearData</td>
<td>3</td>
<td>Clears all custom data (row, column, and cell data).</td>
</tr>
</tbody>
</table>

The Clear method does not affect the number of rows and columns on the grid, and cannot be used to clear data when the grid is bound data source.

You may clear the text or custom formatting in an arbitrary range using the Cell property. For example, the following code clears all text and custom formatting in a range:

```vbnet
fg.Cell(flexcpText, r1, c1, r2, c2) = ""
fg.Cell(flexcpCustomFormat, r1, c1, r2, c2) = False
```

This example clears the text of the selected cell(s) while leaving the picture and cell data intact:

```vbnet
fg.Clear flexClearSelection, flexClearText
```

ClientHeight Property
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.

These properties return values that are slightly smaller than the control's Width and Height properties, because they discount the space taken up by the scrollbars and the control's borders.

**Data Type**

Long
ClientWidth Property

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

Syntax

```vbscript
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.

These properties return values that are slightly smaller than the control's Width and Height properties, because they discount the space taken up by the scrollbars and the control's borders.

For example, the code below scales the columns widths proportionally whenever the grid is resized:

```vbscript
Private Sub Form_Load()
    fg.Cols = 3
    fg.ExtendLastCol = True
    fg.ScrollBars = flexScrollBarVertical
End Sub

Private Sub Form_Resize()
    fg.Move 0, 0, ScaleWidth, ScaleHeight
    fg.ColWidth(0) = fg.ClientWidth * 0.2
    fg.ColWidth(1) = fg.ClientWidth * 0.4
    fg.ColWidth(2) = fg.ClientWidth * 0.4
End Sub
```

Note that the code sets the ExtendLastCol property to True to eliminate any round-off errors.

Data Type

Long

Clip Property

Returns or sets the contents of a range.

Syntax

```vbscript
[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.

The default row and column delimiters may be changed using the ClipSeparators property.

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 ignored. If there are more cells described in the clip string than in the selected region, the extraneous portion of the clip string is ignored. Empty entries in the Clip string will clear existing cell contents.

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 & "2nd" & vbTab & "b"

' paste it over current selection
fg.Clip = s

You may also retrieve or set a clip string for an arbitrary selection by reading the Cell property. For example, the code below copies contents of the first row to the current row:

Private Sub Command1_Click()
    Dim s$
    s = fg.Cell(flexcpText, 1, 0, 1, fg.Cols - 1)
    fg.Cell(flexcpText, fg.Row, 0, fg.Row, fg.Cols - 1) = s
End Sub

Data Type
String

ClipSeparators Property

Returns or sets the characters to be used as column and row separators in Clip strings.

Syntax
[form!]VSFlexGrid.ClipSeparators[ = value As String ]

Remarks
By default, Clip strings are built using tab characters (vbTab or Chr(9)) indicate column breaks, and carriage return characters (vbCr or Chr(13)) indicate row breaks. You may change these characters by assigning a new string to the ClipSeparators property.

The string assigned to the ClipSeparators may be empty, in which case the defaults are used. If it is not empty, it should consist of two distinct characters. The first character will be used as a column separator and the second as a row separator.

The ClipSeparators are used in the following contexts:

1) With the Clip property.
2) With the AddItem method.
3) With the Cell property with the flexcpText setting.
4) With the SaveGrid and LoadGrid methods with the flexFileCustomText setting.

For example, the code below saves the contents of the grid to a text file using pipe character ("\") as column separators. Notice how the code saves and restores the ClipSeparators property to avoid any interference with other parts of the application.

Private Sub Command1_Click()
    Dim cs$
    cs = fg.ClipSeparators
    fg.ClipSeparators = "\" & vbCr
    fg.SaveGrid "c:\pipes.txt", flexFileCustomText
    fg.ClipSeparators = cs
End Sub

Data Type
String
Col Property

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.

The Col property may be set to -1 to hide the selection, to a value between zero and FixedCols - 1 to select a cell in a fixed column, or to a value between FixedCols and Cols - 1 to select a cell in a scrollable column. Setting Col to other values will trigger an Invalid Index error.

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

Setting the Row and Col properties does not ensure that the current cell is visible. To do that, use the ShowCell method.

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

Data Type

Long

ColAlignment Property

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexAlignLeftTop</td>
<td>0</td>
</tr>
<tr>
<td>flexAlignLeftCenter</td>
<td>1</td>
</tr>
<tr>
<td>flexAlignLeftBottom</td>
<td>2</td>
</tr>
<tr>
<td>flexAlignCenterTop</td>
<td>3</td>
</tr>
<tr>
<td>flexAlignCenterCenter</td>
<td>4</td>
</tr>
<tr>
<td>flexAlignCenterBottom</td>
<td>5</td>
</tr>
<tr>
<td>flexAlignRightTop</td>
<td>6</td>
</tr>
<tr>
<td>flexAlignRightCenter</td>
<td>7</td>
</tr>
<tr>
<td>flexAlignRightBottom</td>
<td>8</td>
</tr>
<tr>
<td>flexAlignGeneral</td>
<td>9</td>
</tr>
</tbody>
</table>

The flexAlignGeneral setting aligns text to the left and numbers and dates to the right.

The ColAlignment property affects all cells in the specified column, including those in fixed
rows. You may override this setting for fixed cells using the **FixedAlignment** property. You may override it for individual cells using the **Cell(flexcpAlignment)** property.

This example sets the alignment of the third column to the right and bottom

```vbs
fg.ColAlignment(2) = flexAlignRightBottom
```

You may set the alignment of pictures in cells using the **CellPictureAlignment** or **Cell(flexcpPictureAlignment)** properties.

When setting this property, the **Col** parameter should be set to a value between zero and **Cols** - 1 to set the alignment of a given column, or to -1 to set the alignment of all columns.

**Data Type**

AlignmentSettings (Enumeration)

**ColComboList Property**

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

**Syntax**

```vbs
[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 than 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 can be configured to 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:

```vbs
fg.ColComboList(1) = "#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:

```vbs
```

You may use the **BuildComboList** method to create a **ColComboList** property automatically from a recordset.

For more details on list syntax, including multi-column lists, see the **ComboList** property.
When setting this property, the **Col** parameter should be set to a value between zero and **Cols** - 1 to set the **ColComboList** of a given column, or to -1 to set the **ColComboList** of all columns.

**Note:** The value -1 is reserved and may not be used as an entry ID.

**Data Type**
String

**ColData Property**

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

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.

Because these properties hold Variants, you have extreme flexibility in the types of information you may associate with each row, column or cell. The example below shows some ways in which you can use the **ColData** property:

```
Dim coll As New Collection
coll.Add "Hello"
coll.Add "World"

fg.ColData(1) = 212    ' store a number
fg.ColData(2) = "Hello"    ' store a string
fg.ColData(3) = coll    ' store a pointer to an object
fg.ColData(4) = Me    ' store a pointer to a form

Debug.Print TypeName(fg.ColData(1)), fg.ColData(1)
Debug.Print TypeName(fg.ColData(2)), fg.ColData(2)
Debug.Print TypeName(fg.ColData(3)), fg.ColData(3).Item(2)
Debug.Print TypeName(fg.ColData(4)), fg.ColData(4).Caption
```

This code produces the following output:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>212</td>
</tr>
<tr>
<td>String</td>
<td>Hello</td>
</tr>
<tr>
<td>Collection</td>
<td>World</td>
</tr>
<tr>
<td>Form1</td>
<td>Form1</td>
</tr>
</tbody>
</table>
ColDataType Property

Returns or sets the data type for the column.

**Syntax**
```
[form!]VSFlexGrid.ColDataType(Col As Long) = DataTypesSettings
```

**Remarks**
Valid settings for the ColDataType property are listed below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexDTEmpty</td>
<td>0</td>
</tr>
<tr>
<td>flexDTNull</td>
<td>1</td>
</tr>
<tr>
<td>flexDTShort</td>
<td>2</td>
</tr>
<tr>
<td>flexDTLong</td>
<td>3</td>
</tr>
<tr>
<td>flexDTSingle</td>
<td>4</td>
</tr>
<tr>
<td>flexDTDouble</td>
<td>5</td>
</tr>
<tr>
<td>flexDTCurrency</td>
<td>6</td>
</tr>
<tr>
<td>flexDTDate</td>
<td>7</td>
</tr>
<tr>
<td>flexDTString</td>
<td>8</td>
</tr>
<tr>
<td>flexDTDispatch</td>
<td>9</td>
</tr>
<tr>
<td>flexDTError</td>
<td>10</td>
</tr>
<tr>
<td>flexDTBoolean</td>
<td>11</td>
</tr>
<tr>
<td>flexDTVariant</td>
<td>12</td>
</tr>
<tr>
<td>flexDTUnknown</td>
<td>13</td>
</tr>
<tr>
<td>flexDTDecimal</td>
<td>14</td>
</tr>
<tr>
<td>flexDTLong8</td>
<td>20</td>
</tr>
<tr>
<td>flexDTStringC</td>
<td>30</td>
</tr>
<tr>
<td>flexDTStringW</td>
<td>31</td>
</tr>
</tbody>
</table>

This property is automatically set for each column when the control is bound to a recordset, 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:

- **flexDTDate**
  This setting is taken into account when sorting dates either using the **Sort** property or when the user clicks the **ExplorerBar**. If you don’t set the column type to **flexDTDate**, the dates will be sorted as strings.

- **flexDTBoolean**
  This setting causes the control to 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:
```
fg.ColDataType(1) = flexDTBoolean
fg.TextMatrix(1, 1) = 1    ' checked
fg.TextMatrix(2, 1) = True  ' checked
fg.TextMatrix(3, 1) = "True"  ' checked
fg.TextMatrix(4, 1) = 0     ' not checked
fg.TextMatrix(5, 1) = "False"  ' not checked
fg.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:
fg.ColDataType(2) = flexDTBoolean
fg.ColFormat(2) = "Yes;Not Available" or "True;False", "On;Off", "Yes;No", etc.

When setting this property, the Col parameter should be set to a value between zero and Cols - 1 to set the data type of a given column, or to -1 to set the data type of all columns.

**Data Type**
DataTypeSettings (Enumeration)

---

**ColEditMask Property**

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**

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. The syntax is described below:

**Formatting Numbers:**
The characters used to format numerical values are as follows:

<table>
<thead>
<tr>
<th>Char</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>A locale-dependent currency sign is prepended to the output.</td>
</tr>
<tr>
<td>,</td>
<td>Locale-dependent thousand separators are added to the output.</td>
</tr>
<tr>
<td>(</td>
<td>Negative values are displayed enclosed in parentheses.</td>
</tr>
<tr>
<td>.</td>
<td>The number of decimals is determined by the number of &quot;0&quot; or &quot;&gt;#&quot; characters after the decimal point.</td>
</tr>
<tr>
<td>%</td>
<td>The value is multiplied by 100 and followed by a percent sign.</td>
</tr>
<tr>
<td>,.</td>
<td>The value is divided by 1000 and displayed with thousand separators.</td>
</tr>
</tbody>
</table>

The string "Currency" is also recognized by the control. It formats numbers using the system's currency settings (specified with the Control Panel.)
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:

```vbnet
fg.ColDataType(2) = flexDTBoolean
fg.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. The example below shows several types of format and their effect:

```vbnet
Private Sub Form_Load()
    ' format numbers
    fg.ColFormat(1) = "#,###.##" ' number with thousand separators
    fg.ColFormat(2) = "#.###%" ' percentage
    fg.ColFormat(3) = "#,##" ' thousands
    fg.ColFormat(4) = "Currency" ' thousands
    
    ' format booleans
    fg.ColDataType(5) = flexDTBoolean
    fg.ColFormat(5) = "Probably;Hardly" ' boolean
    
    ' format dates
    fg.ColFormat(6) = "ddd, mmmm d, yyyy"
    fg.ColFormat(7) = "Medium Date"
    fg.ColFormat(8) = "Medium Time"
    
    ' set some cells
    fg.TextMatrix(1, 1) = 1234.56
    fg.TextMatrix(1, 2) = 0.5432
    fg.TextMatrix(1, 3) = 125250
    fg.TextMatrix(1, 4) = -1234.5
    fg.TextMatrix(1, 5) = True
    fg.TextMatrix(1, 6) = #7/4/1969#
    fg.TextMatrix(1, 7) = #7/4/1969#
    fg.TextMatrix(1, 8) = #7/4/1969#
    
    ' display results:
    Dim %
    Debug.Print "Format"; Tab(20); "Content"; Tab(40); "Display"
    Debug.Print "-----------"; Tab(20); "-----------"; Tab(40); "-------------"
    For i = 1 To 8
        Debug.Print fg.ColFormat(i); Tab(20); _
        fg.Cell(flexcpText, 1, i); Tab(40); _
        fg.Cell(flexcpTextDisplay, 1, i)
    Next
End Sub
```

This code produces the following output:
<table>
<thead>
<tr>
<th>Format</th>
<th>Content</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>#,###.##</td>
<td>1234.56</td>
<td>1,234.56</td>
</tr>
<tr>
<td>#.###%</td>
<td>0.5432</td>
<td>54.320%</td>
</tr>
<tr>
<td>#,.##</td>
<td>125250</td>
<td>125.25</td>
</tr>
<tr>
<td>Currency</td>
<td>-1234.5</td>
<td>($1,234.50)</td>
</tr>
<tr>
<td>Probably;Hardly</td>
<td>True</td>
<td>Probably</td>
</tr>
<tr>
<td>ddd, mmm d, yyyy</td>
<td>04/07/1969</td>
<td>Fri, July 4, 1969</td>
</tr>
<tr>
<td>Medium Date</td>
<td>04/07/1969</td>
<td>04-Jul-69</td>
</tr>
<tr>
<td>Medium Time</td>
<td>04/07/1969</td>
<td>12:00 AM</td>
</tr>
</tbody>
</table>

When setting this property, the `Col` parameter should be set to a value between zero and `Cols` - 1 to set the format for a given column, or to -1 to set the format for all columns.

**Data Type**

String

### ColHidden Property

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.

When setting this property, the `Col` parameter should be set to a value between zero and `Cols` - 1 to hide or show a given column, or to -1 to hide or show all columns.

This example hides the second column:

```
fg.ColHidden(1) = True
```

**Data Type**

Boolean

### ColImageList Property

Sets or returns a handle to an ImageList to be used as a source of pictures for a given column.

**Syntax**

```
(form!)VSFlexGrid.ColImageList(Col As Long)[ = value As Long ]
```

**Remarks**

This property is useful if you want to display numeric data contained in a grid column as pictures rather than as numbers.
For example, the "Products" table in the NorthWind database contains a "CategoryID" entry. Instead of displaying the categories as numbers or names, you could create an image list with icons representing each product category, and then bind the "CategoryID" column to the image list. This is easier and more efficient than traversing the recordset and setting the `CellPicture` property for each cell on the column, especially if the number of records is very large.

The mapping between numeric values and image list indices is continuous, starting from zero. Pictures are painted in cells that have numeric values between zero and the number of images on the list minus one. Cells that are empty, contain non-numeric values, or contain values that are outside the image list range, do not get pictures.

When a cell picture comes from an image list, the text contents of the cell are not drawn, and the image is aligned based on the setting of the `ColAlignment` property.

The code below shows how you can use the `ColImageList` property. It assumes you have a form with the following controls on it:

<table>
<thead>
<tr>
<th>Control Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fg</td>
<td>A VSFlexGrid control (OLEDB version).</td>
</tr>
<tr>
<td>imgList</td>
<td>An Image List control with nine images, as shown below.</td>
</tr>
<tr>
<td>dataProducts</td>
<td>An ADO data source control bound to the Products table from the NorthWind database.</td>
</tr>
</tbody>
</table>

The `ImgList` control should have nine images, corresponding to the product categories defined in the NorthWind database. In fact, there are only eight product categories, but they are numbered from one to eight. Thus an extra image with index zero had to be added to the list, even though it won't get used. Here's a picture showing the images defined for the `ImgList` control:

![Image List Control](image)

And here is the code that binds the grid and the image list:

```vbscript
Private Sub Form_Load()
    Set fg.DataSource = dataProducts
End Sub

Private Sub fg_AfterDataRefresh()
    Dim c%
```
c = fg.ColIndex("CategoryID")
fg.ColImageList(c) = imgList.hImageList
fg.ColAlignment(c) = flexAlignCenterCenter
End Sub

The picture below shows the result:

![Image]

Data Type
Long

ColIndex Property
Returns the column index that matches the given key.

Syntax
val& = [form!]VSFlexGrid.ColIndex(Key As String)

Remarks
The ColIndex property is used in conjunction with the ColKey property to identify and refer to columns regardless of their physical position on the grid. These properties are useful when the grid is bound to a recordset or when the user is allowed to move columns around using the ExplorerBar.

To use these properties, assign unique keys to each column using the ColKey property. When you want to refer to a specific column, convert the key into an index using the ColIndex property.

For example, the code below counts how many times the MouseMove event fired and displays the total on a "Counter" column. The user may move the column to a different position with the mouse, and the code will follow the column around:

```vbscript
Private Sub fg_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Dim c%
    c = fg.ColIndex("Counter")
    fg.TextMatrix(1, c) = fg.ValueMatrix(1, c) + 1
End Sub
```

```vbscript
Private Sub Form_Load()
    fg.ExplorerBar = flexExMove
```
When the grid is bound to a recordset, **ColKey** values are automatically set to the field names. This allows you to refer to columns by their field names, as in the following code (which assumes the fg grid is bound to a recordset with a field called "CategoryID"):

```vbnet
fg.ColAlignment(fg.ColIndex("CategoryID")) = flexAlignCenterCenter
```

If the **Key** argument does not correspond to any **ColKey** value, the **ColIndex** property returns -1.

**Data Type**
Long

### ColIsVisible Property

Returns whether a given column is currently within view.

**Syntax**

```vbnet
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.

This example checks to see if the second column is currently within view:

```vbnet
If fg.ColIsVisible(1) Then
    Debug.Print "Column 1 is visible"
End If
```

If a column has zero width or is hidden but is within the scrollable area, **ColIsVisible** returns True.

To ensure a given column is visible, use the **ShowCell** method.

**Data Type**
Boolean

### ColKey Property

Returns or sets a key used to identify the given column.

**Syntax**

```vbnet
[form!]VSFlexGrid.ColKey(Col As Long)[ = value As String ]
```

**Remarks**

The **ColKey** property is used in conjunction with the **ColIndex** property to identify and refer to columns regardless of their physical position on the grid. These properties are useful when the grid is bound to a recordset or when the user is allowed to move columns around using
the ExplorerBar.

To use these properties, assign unique keys to each column using the ColKey property. When you want to refer to a specific column, convert the key into an index using the ColIndex property.

For details an an example, see the ColIndex property.

Data Type
String

ColPos Property
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
Moves a given column to a new position.

Syntax
[f=flexgrid].ColPosition(Col As Long)[ = NewPosition As Long ]

Remarks
The Col and NewPosition parameters must be valid column indices (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.

For example, the following code shows how you can insert a new column at an arbitrary position on the grid:

Sub InsertCol(pos%)
    fg.Cols = fg.Cols + 1
    fg.ColPosition(fg.Cols - 1) = pos
    fg.TextMatrix(0, pos) = "New " & fg.Cols
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
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 run time.

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 cause a run time error. To make sure your code works properly when dealing with large controls, you should add error-handling code to your programs.

If you increase the value of the Cols property, new columns are appended to the right of the grid. To insert columns at specific positions, you need to use the ColPosition property. If you decrease the value of the Cols property, the rightmost columns are removed from the control.

This example sets the number of columns in the grid to four:

\[fg.Cols = 4\]

Data Type
Long

ColSel Property
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 modify a selection or to determine which cells are currently selected. 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, 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.

If the SelectionMode property is set to flexSelectionListBox (3), you should use the IsSelected property to select and deselect rows.

Note that when a range is selected, the value of Row may be greater than or less than RowSel, and Col may be greater than or less than ColSel. This is inconvenient when you need to set up bounds for loops. In these cases, use the GetSelection method to retrieve
selection in an ordered fashion.

**Data Type**
Long

---

**ColSort Property**
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`.

For example, the following code sorts the grid so that the first row is in ascending order, the second is ignored, and the third is in descending order:

```vbs
Private Sub Form_Load()
    fg.Cols = 4
    fg.ColSort(1) = flexSortGenericAscending
    fg.ColSort(2) = flexSortNone
    fg.ColSort(3) = flexSortGenericDescending
End Sub

Private Sub Command1_Click()
    fg.Select 1, 1, 1, 3
    fg.Sort = flexSortUseColSort
End Sub
```

**Data Type**
`SortSettings` (Enumeration)

---

**ColWidth Property**
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. To set column widths automatically, based on the contents of the control, use the **AutoSize** method.
If the `Col` parameter is -1, then the specified width is applied to all columns.

If set `ColWidth` to -1, the column width is reset to its default value, which depends on the size of the control's current font. If set `ColWidth` to zero, 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.

This example sets the width of the third column to 1 inch (1440 twips)

```vbnet
fg.ColWidth(2) = 1440
```

**Data Type**
Long

**ColWidthMax Property**
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. Set it to zero to remove the maximum limit on column widths. Use the `ColWidthMin` property to set a minimum limit to column widths.

Setting limits on column widths may be useful in conjunction with the `AutoSize` method to prevent extremely long entries from making columns too wide or empty columns from becoming too narrow.

This example sets the maximum column width to 2 inches (2880 twips):

```vbnet
fg.ColWidthMax = 2880
```

**Data Type**
Long

**ColWidthMin Property**
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. Set it to zero to remove the minimum limit on column widths. Use the `ColWidthMax` property to set a maximum limit to column widths.

Setting limits on column widths may be useful in conjunction with the `AutoSize` method to prevent extremely long entries from making columns too wide or empty columns from
becoming too narrow.

This example sets the minimum column width to 1/2 inch (720 twips):

```vbscript
fg.ColWidthMin = 720
```

**Data Type**
Long

---

**ComboCount Property**
Returns the number of items in the editor's combo list.

**Syntax**
```vbscript
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`.

```vbscript
Private Sub fg_KeyDownEdit(ByVal Row As Long, ByVal Col As Long, _
KeyCode As Integer, ByVal Shift As Integer)
    Dim i As Long
    If Col = 2 And KeyCode = vbKeyHome Then
        KeyCode = 0 ' eat the key
        For i = 0 To fg.ComboCount - 1 ' select "Cedric"
            If fg.ComboItem(i) = "Cedric" Then
                fg.ComboIndex = i
                Exit For
        Next
    End If
End Sub
```

**Data Type**
Long

---

**ComboData Property**
Returns the long value associated with an item in the editor's combo list.

**Syntax**
```vbscript
val& = [form!]VSFlexGrid.ComboData([ Index As Long ])
```

**Remarks**
You may assign data values to list items when you define the list, using the `ComboList` or `ColComboList` properties. Once the list is created, the data values cannot be changed and become read-only.
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.

**Note:** The value -1 is reserved and may not be used as an entry ID.

**Data Type**
String

**ComboIndex Property**
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**
Returns the string associated with an item in the editor’s combo list.

**Syntax**
```
val$ = form!VSFlexGrid.ComboItem([ Index As Long ])
```

**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**
Returns or sets the list to be used as a drop-down when editing a cell.
Syntax

\[ \text{[form!]} \text{VSFlexGrid.ComboList[ = value As String ]} \]

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 (\texttt{Chr}(9), or \texttt{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.

**Note:** The value -1 is reserved and may not be used as an entry ID.

**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 *ComboList* property. You will need to trap the *BeforeEdit* event and you will have no automatic value translation.

**Data Type**

*String*

**ComboSearch Property**

Returns or sets whether combo lists should support smart searches.

**Syntax**

```plaintext
[form!]V$FlexGrid.ComboSearch[ = ComboSearchSettings ]
```

**Remarks**

When *ComboSearch* is set to a non-zero value, the control searches for entries and highlights them as the user types. This is similar to searching items on the grid using the *AutoSearch* property.

The *ComboSearch* capability makes data-entry substantially easier, especially when working with long lists.

The settings for the *ComboSearch* property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>flexCmbSearchNone</em></td>
<td>0</td>
<td>Don't search as the user types (use only the first letter).</td>
</tr>
<tr>
<td><em>flexCmbSearchLists</em></td>
<td>1</td>
<td>Search drop-down lists, but not combo boxes.</td>
</tr>
<tr>
<td><em>flexCmbSearchCombos</em></td>
<td>2</td>
<td>Search combo boxes, but not drop-down lists.</td>
</tr>
<tr>
<td><em>flexCmbSearchAll</em></td>
<td>3</td>
<td>Search combo boxes and drop-down lists (this is the default setting).</td>
</tr>
</tbody>
</table>
For information on how to build drop-down lists and combo boxes, see the **ComboList** and **ColComboList** properties.

**Data Type**

ComboSearchSettings (Enumeration)

**Default Value**

flexCmbSearchAll (3)

**Compare Event**

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. The result should be:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>If <code>Row1</code> should appear before <code>Row2</code>.</td>
</tr>
<tr>
<td>0</td>
<td>If the rows are equal (as far as sorting goes).</td>
</tr>
<tr>
<td>+1</td>
<td>If <code>Row1</code> should appear after <code>Row2</code>.</td>
</tr>
</tbody>
</table>

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**

Returns or sets the data member.

**Syntax**

`[form!]VSFlexGrid.DataMember[ = value As String ]`

**Remarks**

This property is available only in the ADO (OLEDB) version of the VSFlexGrid control.

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

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**
The settings for the **DataMode** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexDMFree</td>
<td>0</td>
<td>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 <strong>DataRefresh</strong> method. Any direct changes to the database (edits and cursor movements) are ignored by the control. The <strong>flexDMFree</strong> setting is equivalent to the data binding implemented in the <strong>MSFlexGrid</strong> control.</td>
</tr>
<tr>
<td>flexDMBound</td>
<td>1</td>
<td>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 <strong>Row</strong> property changes, the database cursor moves and vice-versa. All edits to the control contents are updated in the database and vice-versa. The <strong>flexDMBound</strong> setting is similar to the data binding implemented in the Microsoft <strong>DBGrid</strong> control.</td>
</tr>
<tr>
<td>flexDMBoundBatch</td>
<td>2</td>
<td>Similar to the <strong>flexDMBound</strong> setting, except the source recordset is not updated automatically after the user edits a cell.</td>
</tr>
<tr>
<td>flexDMBoundImmediate</td>
<td>3</td>
<td>Similar to the <strong>flexDMBound</strong> setting, except the source recordset is updated automatically after the user edits a cell.</td>
</tr>
</tbody>
</table>

The **flexDMBound** mode handles updates to the recordset automatically, based on the setting of the recordset’s **LockType** property. The **flexDMBoundBatch** and **flexDMBoundImmediate** allow you to bypass the recordset’s **LockType** setting. This is usually a bad idea, and these settings should not be used unless you have a good reason to do so.

When the **DataMode** property is set to a value other than **flexDMFree**, some properties and methods are disabled or their behavior is restricted:

<table>
<thead>
<tr>
<th>Property</th>
<th>Limitation when Data-Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddItem</td>
<td>The second parameter of the <strong>AddItem</strong> method, the position where the new row should be inserted, is ignored. New rows are always appended to the database.</td>
</tr>
<tr>
<td>Rows, Cols</td>
<td>These properties become read-only. You may add or remove</td>
</tr>
</tbody>
</table>
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 or using the Sort method on the source recordset.

**IsSubtotal** This property becomes read-only. You may add or clear subtotals using the Subtotal method.

**Data Type**
DataModeSettings (Enumeration)

**DataRefresh Method**
Forces the control to re-fetch all data from its data source.

**Syntax**
\[\text{form}!\].VSFlexGrid.DataRefresh

**DataSource Property**
Returns or sets the data source.

**Syntax**
\[\text{form}!\].VSFlexGrid.DataSource[ = \text{DataSource} ]

**Remarks**
This property behaves differently in the ADO and DAO versions of the VSFlexGrid control.

### OLEDB/ADO version (VSFLEX7.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:

\[\text{ADODC1} = \text{ADODC1} \]

### DAO version (VSFLEX7D.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. You cannot set or retrieve this property at run time.

See also the DataMember and DataMode properties.
Data Type
DataSource

DragRow Method
Starts dragging a row to a new position.

Syntax
[form!]VSFlexGrid.DragRow Row As Long

Remarks
The DragRow method allows you to initiate row dragging programmatically. The user can then move the row to a new position, and you get notified with the BeforeMoveRow and AfterMoveRow events. The method returns the row's new position.

For example, the code below traps the BeforeMouseDown event to initiate row dragging when the user clicks the right mouse button. The code highlights the row being dragged by setting its background color to red, and restores the background after the dragging process ends.

```vbscript
Private Sub fg_BeforeMouseDown (ByVal Button As Integer, ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single, Cancel As Boolean)
    With fg
        If Button = 2 And Shift > 0 Then
            Cancel = True
            Dim r%
            r = .Row
            .Cell (flexcpBackColor, r, l, r, .Cols -1) = vbRed
            r = .Dragrow (r)
            .Cell (flexcpCustmFormat, r, l, r, .Cols -1) = False
            Debug.Print "Dragged to "; r
        End If
    End With
End Sub
```

DrawCell Event
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
This event is fired if the OwnerDraw property is set to a non-zero value, to allow for custom painting 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.

The distribution CD includes sample projects that show how you can use **OwnerDraw** property both in Visual Basic and in Visual C++. Look for the **OwnerDraw** and **PropPage** demos.

---

**Editable Property**

Returns or sets whether the control allows in-cell editing.

**Syntax**

```
[form!]VSFlexGrid.Editable[ = EditableSettings ]
```

**Remarks**

If the **Editable** property is set to a non-zero value, the user may edit the cell contents by typing into the grid.

The settings for the **Editable** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexEDNone</td>
<td>0</td>
<td>The grid contents cannot be edited by the user.</td>
</tr>
<tr>
<td>flexEDKbd</td>
<td>1</td>
<td>The user may initiate edit mode by typing into the current cell.</td>
</tr>
<tr>
<td>flexEDKbdMouse</td>
<td>2</td>
<td>The user may initiate edit mode by typing into the current cell or by double-clicking it with the mouse.</td>
</tr>
<tr>
<td>True</td>
<td>-1</td>
<td>Equivalent to <strong>flexEDKbd</strong>. This setting is used only to keep compatibility with earlier versions of the control.</td>
</tr>
</tbody>
</table>

By default, the control goes into editing mode when the user presses the edit key (F2), the space bar, or any printable character. If the **Editable** property is set to **flexEDKbdMouse** (2) the control will also go into edit mode when the user double-clicks on a cell.

You may force the control into cell-editing mode 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 cancel edit mode using the **Select** statement to select any cell (including the cell being edited).

You may choose to use a regular edit box, drop-down list or drop-down combo, depending on the setting of the **ComboBox** and **ColComboBox** properties. You may also specify an editing mask using the **EditMask** and **ColEditMask** properties. Set these properties in response to the **BeforeEdit** event.

Use the **ValidateEdit** event to perform data validation, and the **AfterEdit** event for post-editing work such as re-sorting the control.

To determine whether the control is in edit mode, use the **EditWindow** property (if it has a non-zero value, the control is in edit mode).

### Data Type

**EditableSettings** (Enumeration)

---

**EditCell Method**

Activates edit mode.

**Syntax**

```
(form!)VSFlexGrid/EditCell
```

**Remarks**

If the **Editable** property is set to a non-zero value, 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.

```vba
Sub fg_MouseDown(Button As Integer, Shift As Integer, X!, Y!)
    If Button = vbRightButton Then
        fg.Select fa.MouseRow, fa.MouseCol
        fg.EditCell
    End If
End Sub
```

**EditMask Property**

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 **ComboBox** 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 even 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.
fg/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:
fg/EditMask = "(###) 000-0000 St\ate\: >LL;q;**"
```

Here are some more commented examples:

<table>
<thead>
<tr>
<th>EditMask String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;St\ate: &gt;LL&quot;</td>
<td>Is a valid format. The 'a' and ';' characters are escaped and thus taken as literals. The '&gt;' is used to ensure that the next two characters will be represented in uppercase.</td>
</tr>
</tbody>
</table>
| "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
"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
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 entered 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.

The following code will allow the user to enter only 4 characters into the cell being edited.

Private Sub fg_BeforeEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
fg.EditMaxLength = 4
End Sub

Data Type
Long

EditSelLength Property
Returns or sets 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.

Notes:
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.
3) Changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
4) Setting **SelText** to a new value replaces the selected text with the new string and sets **SelLength** to 0.

The following code selects characters 6 through 8 whenever a cell is clicked.

```vba
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
End Sub
```

**Data Type**
Long

**EditSelStart Property**
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.

**Notes:**
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.
3) Changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
4) Setting **SelText** to a new value replaces the selected text with the new string and sets **SelLength** to 0.

The following code selects characters 6 through 8 whenever a cell is clicked.

```vba
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
End Sub
```

**Data Type**
Long

**EditSelText Property**
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.

**Notes:**
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.
3) Changing `SelStart` changes the selection to an insertion point and sets `SelLength` to 0.
4) Setting `SelText` to a new value replaces the selected text with the new string and sets `SelLength` to 0.

The following code replaces characters 6 through 8 with the word "COW" whenever a cell is clicked.

```vba
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
    fg.EditSelText = "COW"
End Sub
```

**Data Type**
String

**EditText Property**
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. When `ValidateEdit` event is fired, the cell still contains the original value. The new (edited) value is available only through the `EditText` property.

For example, the code below shows a typical handler for the `ValidateEdit` event. In this case, column 1 only accepts strings, and column 2 only accepts numbers greater than zero:

```vba
Sub fg_ValidateEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
    Dim c$)
    Select Case Col ' different validation rules for each column
        Case 1 ' column 1 only accepts strings
            c = Left$(fg.EditText, 1)
            If UCase$(c) < "A" And UCase$(c) > "Z" Then Beep: Cancel = True
        Case 2 ' column 2 only accepts numbers > 0
            If Val(fg.EditText) <= 0 Then Beep: Cancel = True
    End Select
End Sub
```
Data Type
String

EditWindow Property
Returns a handle to the grid’s editing window, or 0 if the grid is not in edit mode.

Syntax
val& = [form!]VSFlexGrid.EditWindow

Remarks
You can use this property to determine whether the grid is in edit mode. If the grid is in edit mode, EditWindow returns the window handle of the currently active text box or drop-down combo. If the grid is not in edit mode, EditWindow returns zero.

For example, the following code prevents the user from scrolling the grid while a cell is being edited:

Private Sub fg_BeforeScroll(OldTopRow As Long, OldLeftCol As Long, NewTopRow As Long, NewLeftCol As Long, Cancel As Boolean)
    If fg.EditWindow <> 0 And OldTopRow <> NewTopRow Then Cancel = True
End Sub

Data Type
Long

Ellipsis Property
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 settings for the Ellipsis property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexNoEllipsis</td>
<td>0</td>
<td>Long strings are truncated, no ellipsis characters are displayed.</td>
</tr>
<tr>
<td>flexEllipsisEnd</td>
<td>1</td>
<td>Ellipsis characters are displayed at the end of long strings.</td>
</tr>
<tr>
<td>flexEllipsisPath</td>
<td>2</td>
<td>Ellipsis characters are displayed in the middle of long strings.</td>
</tr>
</tbody>
</table>

Data Type
EllipsisSettings (Enumeration)
Default Value
flexNoEllipsis (0)

EnterCell Event
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 Control)
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.

The ErrorCode parameter can have the following values:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>Recordset can't be updated. The record may be locked, or the recordset may be a read-only recordset.</td>
</tr>
<tr>
<td>130</td>
<td>Recordset field can't take this value. The value entered is of the wrong type or would violate database integrity rules.</td>
</tr>
</tbody>
</table>

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, optionally replacing it with a custom one.

ExplorerBar Property
Returns or sets whether column headers are used to sort and/or move columns.

Syntax
pressions]

Remarks
The ExplorerBar property allows users to use column headings to sort and move columns without any code. Valid settings are described below:
### Constant

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexExNone</td>
<td>0</td>
<td>No ExplorerBar. Fixed rows behave as usual.</td>
</tr>
<tr>
<td>flexExSort</td>
<td>1</td>
<td>Users may sort columns by clicking on their headings.</td>
</tr>
<tr>
<td>flexExMove</td>
<td>2</td>
<td>Users may move columns by dragging their headings.</td>
</tr>
<tr>
<td>flexExSortAndMove</td>
<td>3</td>
<td>Users may sort and move columns.</td>
</tr>
<tr>
<td>flexExSortShow</td>
<td>5</td>
<td>Users may sort columns by clicking on their headings. The control will show the current sorting order by drawing an arrow on the heading.</td>
</tr>
<tr>
<td>flexExSortShowAndMove</td>
<td>7</td>
<td>Users may sort and move columns. The control will show the current sorting order by drawing an arrow on the heading.</td>
</tr>
<tr>
<td>flexExMoveRows</td>
<td>8</td>
<td>Allows movement of rows by dragging them by the fixed cells on the row. *</td>
</tr>
</tbody>
</table>

*The setting, flexExMoveRows is actually a flag, and may be combined with other setting using the "Or" operator. For example:

'allow sorting, moving rows, and moving columns
ExplorerBar = flexExMoveRows Or flexExSortShowAndMove

Note that these values are a combination of binary flags and are not sequential.

By default, the **ExplorerBar** works like the one in Microsoft’s Windows File Explorer. 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**.

The new ExplorerBar is easier to use. For example, when sorting, it shows sorting triangle glyphs. Also, when moving columns, it clearly shows the new column position and scrolls the control to allow the column to be moved anywhere.

You must have at least one fixed row to be able to use the **ExplorerBar**.

### Data Type

**ExplorerBarSettings** (Enumeration)

**Default Value**

flexExNone (0)

---

### ExtendLastCol Property

Returns or sets whether the last column should be adjusted to fit the control's width.

**Syntax**

```
[form1]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
Returns or sets whether changes to the Text or format properties apply to the current cell or to the entire selection.

Syntax
\[\text{form!}VSFMLexGrid.FillStyle[ = \text{FillStyleSettings }\]\n
Remarks
The settings for the FillStyle property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexFillSingle</td>
<td>0</td>
<td>Setting the Text property or any of the cell formatting properties affects the current cell only.</td>
</tr>
<tr>
<td>flexFillRepeat</td>
<td>1</td>
<td>Setting the Text property or any of the cell formatting properties affects the entire selected range.</td>
</tr>
</tbody>
</table>

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)

FilterData Event
Fired after a value is read and before a value is written to a recordset to allow custom formatting.

Syntax
Private Sub VSFMLexGrid_FilterData( ByVal Row As Long, ByVal Col As Long, Value As String, ByVal SavingToDB As Boolean, WantThisCol As Boolean)

Remarks
This event is fired whenever data is read from or written to a bound recordset. It allows you to modify the data before it is committed to the grid (when reading) or to the recordset (when writing).

This event is mostly useful when the data is stored in the database using an encoded format that is not ideal for displaying and editing. For example, many databases store dates as a string of digits (usually 6 or 8), without any separators. You could use the FilterData event to insert the separators at the proper places before displaying the data, and to remove them again before writing them back into the database.

The parameters for the FilterData event are described below:

Row As Long, Col As Long
Address of the cell whose value is about to be read from or written to the bound recordset.

Value As String
Value just read from or about to be written to the bound recordset.
SavingToDB As Boolean
If True, the value was read from the grid and is about to be written to the bound recordset.
If False, the value was read from the bound recordset and is about to be written to the grid.

WantThisCol As Boolean
This value is set to False by default. If you set it to True, the control will keep firing the FilterData event for this column. If you don't set it to True, the event will no longer be fired for this column until the grid is bound to a new recordset or the current recordset is refreshed. The WantThisCol parameter is important because the FilterData event is relatively slow. The WantThisCol parameter allows the application to establish which columns need filtering, thus improving performance.

For example, suppose you are dealing with a recordset that contains two date fields called "Birth" and "Hired". These fields contain dates encoded as six-digit strings (e.g. July 4th, 1962 is encoded as "070462") which you would like to display in the "Medium Date" format. The code below shows how you can accomplish this using the FilterData event:

Private Sub fg_FilterData(ByVal Row As Long, ByVal Col As Long, _
    Data As String, ByVal SavingToDB As Boolean, _
    WantThisCol As Boolean)

    ' we are interested only in "Birth" and "Hired" fields
    If fg.ColKey(Col) <> "Birth" And fg.ColKey(Col) <> "Hired" Then Exit Sub

    ' keep getting this column
    WantThisCol = True

    ' format data going out to the recordset (Date -> "mmddyy")
    Dim dt As Date
    If SavingToDB Then
        dt = Data
        Data = Format(dt, "mmddyy")
    Else
        If Len(Data) = 6 Then
            dt = DateSerial(Right(Data, 2), Left(Data, 2), Mid(Data, 3, 2))
            Data = Format(dt, "Medium Date")
        End If
    End If

End Sub

Note: Be careful when writing FilterData code to avoid corrupting your database. The FilterData code should perform symmetrical translations on the data. In other words, if value X gets translated into value Y when it is read from the database, then value Y should be translated back into X when it is written out to the database.

FindRow Property
Returns the index of a row that contains a specified string or RowData value.

Syntax
val& = [form!]VSFlexGrid.FindRow(Item As Variant, [ Row As Long ], [ Col As Long ], [ CaseSensitive As Boolean ], [ FullMatch As Boolean])

Remarks
The **FindRow** method allows you to look up rows based on cell contents or **RowData** values. The search is much faster and easier to implement than a Visual Basic loop.

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

**Item As Variant**  
This parameter contains the data being searched.

**Row As Long** (optional)  
This parameter contains the rows where the search should start. The default value is **FixedRows**.

**Col As Long** (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 contents for the given column.

**CaseSensitive As Boolean** (optional)  
This parameter is True by default, which means the search is case-sensitive. Set it to False if you want a case-insensitive search (e.g. when looking for "FOO" you may find "foo" ). This parameter is only relevant when you are looking for a string.

**FullMatch As Boolean** (optional)  
This parameter is True by default, which means the search is for a full match. Set it to False if you want to allow partial matches (e.g. when looking for "FOO" you may find "FOOBAR"). This parameter is only relevant when you are looking for a string.

**FindRow** returns the index of the row where the data was found, or -1 if the data was not found.

The code below shows how this method is used:

```vba
' assign some data to row 40 and cell (40, 5)  
fg.RowData(30) = "MyRow"  
fg.TextMatrix(40, 5) = "MyCell"

Debug.Print fg.FindRow("MyRow")  ' find a row by its RowData
> 30

Debug.Print fg.FindRow("MyCell")  ' no rows have RowData = "MyCell"
> -1
Debug.Print fg.FindRow("MyCell", , 5)  ' look for text in column 5
> 40

Debug.Print fg.FindRow("MYCELL", , 5)  ' case-sensitive search fails
> -1
Debug.Print fg.FindRow("MYCELL", , 5, False)  ' case-insensitive search succeeds
> 40

Debug.Print fg.FindRow("My", , 5)  ' full-match search fails
> -1
Debug.Print fg.FindRow("My", , 5, False)  ' partial-match search succeeds
> 40
```

**Note:** The **FindRow** method is useful for searching rows through code. To allow users to perform incremental searches by typing into cells, use the **AutoSearch** property instead.
Data Type
Long

FixedAlignment Property
Returns or sets the alignment for the fixed rows in a column.

Syntax
\[\text{form!}!\text{VSFlexGrid.FixedAlignment(Col As Long)[ = AlignmentSettings ]}\]

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)

FixedCols Property
Returns or sets the number of fixed (non-scrollable) columns.

Syntax
\[\text{form!}!\text{VSFlexGrid.FixedCols[ = value As Long ]}\]

Remarks
Fixed columns remain visible when the user scrolls the contents of the grid. They are not selectable or editable by the user (but you can select them with code and even allow the user to edit their contents by selecting them and invoking the EditCell method). You can set FixedCols to any value between zero and the total number of columns.

The following line of code places 3 fixed columns at the left edge of the grid.

\[\text{fg.FixedCols = 3}\]

Fixed columns are typically used in spreadsheet applications to display row numbers or other types of labels.

To format the fixed cells, use the BackColorFixed, ForeColorFixed, GridLinesFixed, and FixedAlignment properties.

You can create non-scrollable columns that can be selected and edited using the FrozenCols property.

If the AllowUserResizing property is set to a non-zero value, the fixed cells allow the user to resize row heights and column widths at run time.
**Data Type**

Long

---

**FixedRows Property**

Returns or sets the number of fixed (non-scrollable) rows.

**Syntax**

\[form\!\]VSFlexGrid.FixedRows[ = value As Long ]

**Remarks**

Fixed rows remain visible when the user scrolls the contents of the grid. They are not selectable or editable by the user (but you can select them with code and even allow the user to edit their contents by selecting them and invoking the EditCell method). You can set FixedRows to any value between zero and the total number of rows.

The following line of code places 3 fixed rows at the top edge of the grid.

fg.FixedRows = 3

Fixed rows are typically used in spreadsheet applications to display column headers, and in database applications to display field names.

To format the fixed cells, use the BackColorFixed, ForeColorFixed, GridLinesFixed, and FixedAlignment properties.

You can create non-scrollable rows that can be selected and edited using the FrozenRows property.

If the AllowUserResizing property is set to a non-zero value, the fixed cells allow the user to resize row heights and column widths at run time. If the ExplorerBar property is set to a non-zero value, the fixed rows allow the user to sort and move columns with the mouse.

---

**FlexDataSource Property**

Returns or sets a custom data source for the control.

**Syntax**

\[form\!\]VSFlexGrid.FlexDataSource[ = IVSFlexDataSource ]

**Remarks**

The VSFlexGrid control can be bound to several types of data source, including ADO or DAO recordsets, Variant arrays, and other VSFlexGrid controls. The FlexDataSource property is yet another option, based on a custom COM interface that is easy to implement and very flexible.

The main advantages of data-binding through the FlexDataSource property are speed and flexibility. The main disadvantage is that you have to write more code than with the other options. You should consider using the FlexDataSource property when you have large amounts of data stored in custom structures or objects (other than database recordsets). By using the FlexDataSource property, you may display and edit the data in-place. There is no
need to copy it to the grid and save it back later. In fact, the data may even be mostly virtual, consisting of dynamically calculated values rather than static information.

When you assign a new data source to the FlexDataSource property, the control will automatically set each column's header text and ColKey property based on the data source's field name. You may change these values after setting the FlexDataSource property, if you wish.

Note: When bound to a FlexDataSource, values on fixed rows are not regarded as bound data. You may set or retrieve them with code without affecting the data source. Fixed columns, on the other hand, are regarded as bound data. Their contents are read from and written to the data source.

To bind the VSFlexGrid control to a FlexDataSource property, you need to implement an object that exposes the IVSFlexDataSource COM interface. The interface is very simple. It consists of only two methods: GetData and SetData. GetData(Row, Col) returns a string to be displayed by the grid at the specified position. It is called by the grid when it needs to display a value. SetData(Row, Col, Data) updates the data source at the specified position with the new value. It is called by the grid when the user edits a cell.

The distribution CD includes sample code that show how to implement and use the FlexDataSource object in a fairly realistic scenario, both in Visual Basic and Visual C++. The following tutorial is a very simple example of how to use the FlexDataSource property.

**Step 1: Create the project**
Start a new Visual Basic project and add a VSFlexGrid control to the form (you may use any version: ADO, DAO, or Light). Change the name of the grid control to fs.

**Step 2: Create the FlexDataSource object**
Add a new class module to your project by selecting the "Project|Add Class Module" command from the Visual Basic menu. On the new class code window, type the following statement:

```
Option Explicit
Implements IVSFlexDataSource
```

This tells the world that the new class implements the IVSFlexDataSource methods and that they are available to anyone who cares to use them. Now select the IVSFlexDataSource item from the "Object Box" (the drop-down list on the top left of the code window). VB will immediately create a "stub" (empty function) for the GetFieldCount method. Now click on the "Procedures/Events Box" (the drop-down list on the top right of the code window) and select each method to create stubs for each one. Here is what the code window should look like by now:

```
Option Explicit
Implements IVSFlexDataSource

Private Function IVSFlexDataSource_GetFieldCount() As Long
End Function

Private Function IVSFlexDataSource_GetFieldName(ByVal Field As Long) As String
End Function

Private Function IVSFlexDataSource_GetRecordCount() As Long
End Function

Private Function IVSFlexDataSource_GetData(ByVal Field As Long, ByVal Record As Long) As String
End Function
```
Private Sub IVSFlexDataSource_SetData(ByVal Field As Long, ByVal Record As Long, ByVal newData As String)
    End Sub

**Step 3: Implement the Data Structure**

In this example, the data will be completely virtual. We will simply display a table of angles in degrees and radians, their sines, and co-sines. The table will have four "fields" and 360 "records". Here is the code needed to implement this structure:

Private Function IVSFlexDataSource_GetFieldCount() As Long
    IVSFlexDataSource_GetFieldCount = 4
End Function

Private Function IVSFlexDataSource_GetRecordCount() As Long
    IVSFlexDataSource_GetRecordCount = 360
End Function

Private Function IVSFlexDataSource_GetFieldName(ByVal Field As Long) As String
    Select Case Field
        Case 0: IVSFlexDataSource_GetFieldName = "Angle (Deg)"
        Case 1: IVSFlexDataSource_GetFieldName = "Angle (Rad)"
        Case 2: IVSFlexDataSource_GetFieldName = "Sine"
        Case 3: IVSFlexDataSource_GetFieldName = "Co-Sine"
    End Select
End Function

Now that we have defined the data structure, we need to implement the functions that will supply the actual data.

**Step 3: Implement the GetData and SetData methods**

The **GetData** method is responsible for providing data to the consumer. In this example, there is no static data, only calculated fields:

Private Function IVSFlexDataSource_GetData(ByVal Field As Long, ByVal Record As Long) As String
    Select Case Field
        Case 0: IVSFlexDataSource_GetData = Record
        Case 1: IVSFlexDataSource_GetData = Record / 180# * 3.1416
        Case 2: IVSFlexDataSource_GetData = Sin(Record / 180# * 3.1416)
        Case 3: IVSFlexDataSource_GetData = Cos(Record / 180# * 3.1416)
    End Select
End Function

The **SetData** method is responsible for updating the data new information supplied by the user (e.g. by editing a grid cell). In this case, the data cannot be changed, so any attempt to do it will simply raise an error:

Private Sub IVSFlexDataSource_SetData(ByVal Field As Long, ByVal Record As Long, ByVal newData As String)
    Err.Raise 666, "IVSFlexDataSource", "This data is read-only."
End Sub

The **SetData** method is trivial in this case, but in a more realistic application it could be used to perform data-validation and to allow editing of certain columns only.

**Step 5: Hook up the VSFlexGrid and the Data Provider**

Now that the **FlexDataSource** object is ready, all we need to do is hook it up to the grid.
Double click on the main form (the one with the grid on it), and add the following code to the `Form_Load` event:

```vbnet
Private Sub Form_Load()
    Dim fds As New Class1 ' create the data source object
    fg.FlexDataSource = fds ' assign it to the grid
    fg.ColFormat(-1) = "#.##" ' format grid columns
    fg.ColFormat(0) = ""
End Sub
```

Run the project and you will see that it loads very quickly, and displays the information as expected. Just for fun, try changing the value returned by the `GetRecordCount` method to a really large value (say 500,000 or so) and run the project again. You will notice there's little or no speed degradation.

This is the end of the tutorial. For more details, refer to the samples on the distribution CD.

**Data Type**
IVSFlexDataSource

---

### FloodColor Property

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.

Any of the following FloodColor lines will paint the left half of the current cell blue.

- `fg.CellFloodPercent = 50`
- `fg.FloodColor = &HFF0000 ' using a hex value`
- `fg.FloodColor = vbBlue ' using a VB constant`
- `fg.FloodColor = RGB(0, 0, 255) ' using the RGB function`

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

Returns or sets the type of focus rectangle to be displayed around the current cell.

**Syntax**

```
(form!)VSFlexGrid.FocusRect[ = FocusRectSettings ]
```
Remarks
The settings for the FocusRect property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexFocusNone</td>
<td>0</td>
<td>Do not show a focus rectangle.</td>
</tr>
<tr>
<td>flexFocusLight</td>
<td>1</td>
<td>Show a one-pixel wide focus rectangle.</td>
</tr>
<tr>
<td>flexFocusHeavy</td>
<td>2</td>
<td>Show a two-pixel wide focus rectangle.</td>
</tr>
<tr>
<td>flexFocusSolid</td>
<td>3</td>
<td>Show focus rectangle as a flat frame (the color is determined by the BackColorSel property).</td>
</tr>
<tr>
<td>flexFocusRaised</td>
<td>4</td>
<td>Show focus rectangle as a raised frame.</td>
</tr>
<tr>
<td>flexFocusInset</td>
<td>5</td>
<td>Show focus rectangle as an inset frame.</td>
</tr>
</tbody>
</table>

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 selected background color (BackColorSel).

Data Type
FocusRectSettings (Enumeration)

Default Value
flexFocusLight (1)

ForeColorFixed Property
Returns or sets the foreground color of the fixed rows and columns.

Syntax
\[[form!]VSFlexGrid.GForeColorFixed\[ = \text{colorref} & \]

Remarks
This property works in conjunction with the ForeColor, ForeColorSel, and ForeColorFrozen properties to specify the color used to draw text.

ForeColor is the color used to draw text in the scrollable area of the control.
ForeColorFixed is the color used used to draw text in the fixed rows and columns (see FixedRows and FixedCols).
ForeColorSel is the color used to draw text in selected cells (see Highlight).
ForeColorFrozen is the color used to draw frozen cells (see FrozenRows and FrozenCols).

You may set the text color of individual cells using the Cell(flexcpForeColor) property.

Data Type
Color

ForeColorFrozen Property
Returns or sets the foreground color of the frozen rows and columns.

Syntax
\[[form!]VSFlexGrid.GForeColorFrozen\[ = \text{colorref} & \]

Remarks
This property works in conjunction with the ForeColor, ForeColorSel, and ForeColorFixed
properties to specify the color used to draw text.

ForeColor
is the color used to draw text in the scrollable area of the control.
ForeColorFixed
is the color used used to draw text in the fixed rows and columns (see FixedRows and FixedCols).
ForeColorSel
is the color used to draw text in selected cells (see Highlight).
ForeColorFrozen
is the color used to draw frozen cells (see FrozenRows and FrozenCols).

You may set the text color of individual cells using the Cell(flexcpForeColor) property.

Data Type
Color

ForeColorSel Property

Returns or sets the foreground color of the selected cells.

Syntax

[form!]VSFlexGrid.ForeColorSel[ = colorref& ]

Remarks

This property works in conjunction with the ForeColor, ForeColorFrozen, and ForeColorFixed properties to specify the color used to draw text.

ForeColor          is the color used to draw text in the scrollable area of the control.
ForeColorFixed    is the color used to draw text in the fixed rows and columns (see FixedRows and FixedCols).
ForeColorSel       is the color used to draw text in selected cells (see Highlight).
ForeColorFrozen   is the color used to draw frozen cells (see FrozenRows and FrozenCols).

You may set the text color of individual cells using the Cell(flexcpForeColor) property.

Data Type
Color

FormatString Property

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.

The `FormatString` property is obsolete. Use the `Columns` property page to set up your columns instead.

Data Type
String

FrozenCols Property
Returns or sets the number of frozen (editable but non-scrollable) columns.

Syntax
```vbscript
[form!]VF FlexGrid.FrozenCols[ = value As Long ]
```

Remarks
Cells in frozen columns can be selected and edited, but they remain visible when the user scrolls the contents of the control horizontally.

Freezing columns is useful when the grid is used as a data browser. It allows users to user to scroll the contents of the control while keeping the `FrozenCols` leftmost columns visible. If you set the `AllowUserFreezing` property to a non-zero value, the may freeze or thaw columns at run time by dragging the solid line between the frozen and scrollable areas of the grid.

You may customize the appearance of the frozen areas of the grid using the `BackColorFrozen` and `ForeColorFrozen` properties. The solid line between the frozen and scrollable areas of the grid is drawn using the color specified by the `SheetBorder` property.

Data Type
Long

Default Value
0

FrozenRows Property
Returns or sets the number of frozen (editable but non-scrollable) rows.

Syntax
```vbscript
[form!]VF FlexGrid.FrozenRows[ = value As Long ]
```

Remarks
Cells in frozen rows can be selected and edited, but they remain visible when the user scrolls the contents of the control vertically.

Freezing rows is useful when the top rows are used to display information that should be kept visible, such as subtotals or a “query-by-example” search row. If you set the `AllowUserFreezing` property to a non-zero value, the may freeze or thaw rows at run time by dragging the solid line between the frozen and scrollable areas of the grid.
You may customize the appearance of the frozen areas of the grid using the `BackColorFrozen` and `ForeColorFrozen` properties. The solid line between the frozen and scrollable areas of the grid is drawn using the `SheetBorder` property.

**Data Type**

Long

**Default Value**

0

---

**GetHeaderRow Event**

Fired while printing the control to set repeating header rows.

**Syntax**

```vbscript
Private Sub VSFlexGrid_GetHeaderRow( ByVal Row As Long, HeaderRow As Long)
```

**Remarks**

This event is fired while the control is being printed, to create a 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.

The control may be printed with the `PrintGrid` method or with a `VSPrinter` control. The advantage of using the `VSPrinter` control is that it provides print previewing, the ability to integrate many grids and other graphical elements on a single document, and complete control over the printer. The `VSPrinter` control is available from VideoSoft as a separate product.

---

**GetMergedRange Method**

Returns the range of merged cells that includes a given cell.

**Syntax**

```vbscript
(form!)VSFlexGrid.GetMergedRange Row As Long, Col As Long, R1 As Long, C1 As Long, R2 As Long, C2 As Long
```

**Remarks**

The `VSFlexGrid` control can merge cells based on their contents. This method allows you to determine whether a cell is merged with its neighboring cells.

For example, the following code changes the contents of a merged cell preserving the
merged range:

' create a merged range
fg.MergeCells = flexMergeFree
fg.MergeRow(1) = True
fg.Cell(flexcpText, 1, 1, 1, 4) = "Merged Range"

' this changes only cell 1, 1
fg.Cell(flexcpText, 1, 1) = "Merged Range Has Changed"

' this changes the whole merged range
Dim r1&, c1&, r2&, c2&
fg.GetMergedRange 1, 2, r1, r2, c1, c2
fg.Cell(flexcpText, 1, 1, 1, 4) = "Merged Range Has Changed"

For more details on cell merging, see the **MergeCells** property.

---

**GetNode Method (VSFlexGrid Control)**

Returns an outline node object for a given subtotal row.

**Syntax**

```
[form!]VSFlexGrid.GetNode [ Row As Variant ]
```

**Remarks**

GetNode method returns a VSFlexNode object that corresponds to an outline row.

The VSFlexNode object has methods for adding, removing, locating, moving, sorting, and changing outline nodes.

This method takes a Variant parameter named **Row** that may be one of the following:

1. Omitted:  Defaults to grid's current row
2. A row number:  If a row is not a node, the row's parent node is used
3. A string containing a key:  The key corresponds to a RowData string used to identify the row

For example, you could use this code to "promote" an outline node (move it deeper into the outline):

```
Dim nd As VsFlexNode
Set nd = fg.GetNode ' defaults to current row
nd. Move flexNMIn
```

**GetNodeRow Method**

Returns the number of a row's parent, first, or last child in an outline.

**Syntax**

```
[form!]VSFlexGrid.GetNodeRow Row As Long, Which As NodeTypeSettings
```

**Remarks**

When the grid is used in outline mode, this method allows you to find nodes that are related to a given node.

The parameters for the **GetNodeRow** property are described below:
**Row As Long**
The row number of the node whose related nodes you want to find.

**Which As NodeTypeSettings**
Which node to return. Valid settings for this parameter are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexNTRoot</td>
<td>0</td>
<td>Returns the index of the node's top level ancestor.</td>
</tr>
<tr>
<td>flexNTParent</td>
<td>1</td>
<td>Returns the index of the node's immediate parent.</td>
</tr>
<tr>
<td>flexNTFirstChild</td>
<td>2</td>
<td>Returns the index of the node's first child node.</td>
</tr>
<tr>
<td>flexNTLastChild</td>
<td>3</td>
<td>Returns the index of the node's last child node.</td>
</tr>
<tr>
<td>flexNTFirstChild</td>
<td>4</td>
<td>Returns the index of the node's first sibling node (may be same row)</td>
</tr>
<tr>
<td>flexNTLastSibling</td>
<td>5</td>
<td>Returns the index of the node's last sibling node (may be same row)</td>
</tr>
<tr>
<td>flexNTPreviousSibling</td>
<td>6</td>
<td>Returns the index of the node's previous sibling node (-1 if this is the first sibling)</td>
</tr>
<tr>
<td>flexNTNextSibling</td>
<td>7</td>
<td>Returns the index of the node's next sibling node (-1 if this is the last sibling)</td>
</tr>
</tbody>
</table>

If the node requested cannot be found, **GetNodeRow** returns -1. For example when requesting the parent of the root node, or the child of an empty node.

The code below shows two typical uses for the **GetNodeRow** property:

```vba
' traverse an outline and return the full path to a given node
Private Function GetFullPath(r As Long)
    Dim s$ As String
    While r > -1
        s = fg.TextMatrix(r, 0) & "\" & s
        r = fg.GetNodeRow(r, flexNTParent)
    Wend
    GetFullPath = s
End Function

' delete an outline node and all its children
Private Sub DeleteNode(r As Long)
    Dim nRows&
    nRows = fg.GetNodeRow(r, flexNTLastChild) - r + 1
    While nRows > 0
        fg.RemoveItem r
        nRows = nRows - 1
    Wend
End Sub
```

**GetSelection Method**
Returns the current selection ordered so that Row1 <= Row2 and Col1 <= Col2.

**Syntax**

```vba
(form!)VSFlexGrid.GetSelection Row1 As Long, Col1 As Long, Row2 As Long, Col2 As Long
```

**Remarks**
When programming the VSFlexGrid control, a common task is looping through the currently selected range to perform some action on the selected cells, defined by the values of the
Row, RowSel, Col, and ColSel properties. When setting up such loops, you should take into account the fact that Row may be greater than or smaller than RowSel, and Col may be greater than or smaller than ColSel. Instead of comparing these values to set up the loop bounds, use the GetSelection to obtain the loop bounds in the proper order.

For example, the code below prints the contents of the selected range:

```vbs
Dim r&, c&, r1&, c1, r2&, c2&
fg.GetSelection r1, c1, r2, c2
For r = r1 To r2
    For c = c1 To c2
        Debug.Print fg.TextMatrix
    Next
Next
```

**GridColumn Property**

Returns or sets the color used to draw the grid lines between the non-fixed cells.

**Syntax**

```
[form!]VSFlexGrid.GridColumn[ = colorref& ]
```

**Remarks**

The GridColor and GridLines properties determine the appearance of the grid lines displayed in the scrollable area of the grid. GridColorFixed and GridLinesFixed determine the appearance of the grid lines displayed in the fixed area of the grid.

The GridColor property is ignored when GridLines 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

**GridColumnFixed Property**

Returns or sets the color used to draw the grid lines between the fixed cells.

**Syntax**

```
[form!]VSFlexGrid.GridColumnFixed[ = colorref& ]
```

**Remarks**

The GridColorFixed and GridLinesFixed properties determine the appearance of the grid lines displayed in the fixed area of the grid. GridColor and GridLines determine the appearance of the grid lines displayed in the scrollable area of the grid.

The GridColorFixed property is ignored when GridLinesFixed 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

Returns or sets the type of lines to be drawn between non-fixed cells.

**Syntax**

```plaintext
[form!] VSFlexGrid.GridLines[ = GridStyleSettings ]
```

**Remarks**

The `GridLines` and `GridColor` properties determine the appearance of the grid lines displayed in the scrollable area of the grid. `GridLinesFixed` and `GridColorFixed` determine the appearance of the grid lines displayed in the fixed area of the grid.

The settings for the `GridLines` property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexGridNone</td>
<td>0</td>
<td>Do not draw grid lines between cells.</td>
</tr>
<tr>
<td>flexGridFlat</td>
<td>1</td>
<td>Draw flat lines with color and width determined by the <code>GridColor</code> and <code>GridLineWidth</code> properties.</td>
</tr>
<tr>
<td>flexGridInset</td>
<td>2</td>
<td>Draw inset lines between cells.</td>
</tr>
<tr>
<td>flexGridRaised</td>
<td>3</td>
<td>Draw raised lines between cells.</td>
</tr>
<tr>
<td>flexGridFlatHorz</td>
<td>4</td>
<td>Draw flat lines between rows, no lines between columns.</td>
</tr>
<tr>
<td>flexGridInsetHorz</td>
<td>5</td>
<td>Draw inset lines between rows, no lines between columns.</td>
</tr>
<tr>
<td>flexGridRaisedHorz</td>
<td>6</td>
<td>Draw raised lines between rows, no lines between columns.</td>
</tr>
<tr>
<td>flexGridSkipHorz</td>
<td>7</td>
<td>Draw an inset effect around every other row.</td>
</tr>
<tr>
<td>flexGridFlatVert</td>
<td>8</td>
<td>Draw flat lines between columns, no lines between rows.</td>
</tr>
<tr>
<td>flexGridInsetVert</td>
<td>9</td>
<td>Draw inset lines between columns, no lines between rows.</td>
</tr>
<tr>
<td>flexGridRaisedVert</td>
<td>10</td>
<td>Draw raised lines between columns, no lines between rows.</td>
</tr>
<tr>
<td>flexGridSkipVert</td>
<td>11</td>
<td>Draw an inset effect around every other column.</td>
</tr>
<tr>
<td>flexGridExplorer</td>
<td>12</td>
<td>Draw button-like frames around each cell.</td>
</tr>
<tr>
<td>flexGridExcel</td>
<td>13</td>
<td>Draw button-like frames around each cell, highlighting the headings for the current selection. This setting should only be applied to the <code>GridLinesFixed</code> property.</td>
</tr>
</tbody>
</table>

The `GridColor` property is ignored when `GridLines` 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**

`GridStyleSettings` (Enumeration)

**Default Value**

`flexGridFlat` (1)

GridLinesFixed Property

Returns or sets the type of lines to be drawn between fixed cells.

**Syntax**

```plaintext
[form!] VSFlexGrid.GridLinesFixed[ = GridStyleSettings ]
```

**Remarks**

The `GridLinesFixed` and `GridColorFixed` properties determine the appearance of the grid
GridLines and GridColor determine the appearance of the grid lines displayed in the scrollable area of the grid.

The settings for the GridLinesFixed property are the same as those used for the GridLines property. The flexGridExcel (13) setting should only be used with the GridLinesFixed property. This setting causes the fixed rows and columns to show a highlighted area corresponding to the current selection. This makes it easy for users to identify which rows and columns are selected on large grids.

The GridColorFixed property is ignored when GridLinesFixed 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**
GridStyleSettings (Enumeration)

**Default Value**
flexGridInset (2)

---

### GridLineWidth Property

Returns or sets the width of the grid lines, in pixels.

**Syntax**

\[
i(form!)VSFlexGrid.GridLineWidth[ = value As Integer ]
\]

**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 have fixed width and cannot be changed.

**Data Type**
Integer

**Default Value**
1

---

### HighLight Property

Returns or sets whether selected cells will be highlighted.

**Syntax**

\[
i(form!)VSFlexGrid.HighLight[ = ShowSelSettings ]
\]

**Remarks**

The settings for the HighLight property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexHighlightNever</td>
<td>0</td>
<td>Never highlight the selected range. Selected ranges will not be visible to the user.</td>
</tr>
<tr>
<td>flexHighlightAlways</td>
<td>1</td>
<td>Always highlight the selected range.</td>
</tr>
<tr>
<td>flexHighlightWithFocus</td>
<td>2</td>
<td>Highlight the selected range only when the control has the focus.</td>
</tr>
</tbody>
</table>
Highlighting ranges that contain merged cells may lead to non-rectangular shapes being highlighted. If this is undesirable, you may disable the highlighting by setting `HighLight` to `flexHighlightNever`.

You may also prevent extended selections of any kind by setting the `AllowSelection` property to `False`.

**Data Type**
ShowSelSettings (Enumeration)

**Default Value**
`flexHighlightAlways (1)`

### IsCollapsed Property

Returns or sets whether an outline row is collapsed or expanded.

**Syntax**

```vba
form!VSFlexGrid.IsCollapsed(Row As Long) [ = CollapsedSettings ]
```

**Remarks**

This property is used when the grid is in outline mode. It allows you to expand or collapse outline nodes through code. If the `OutlineBar` property is set to a non-zero value, the user can also collapse and expand nodes with the mouse.

The settings for the `IsCollapsed` property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flexOutlineExpanded</code></td>
<td>0</td>
<td>Show all child nodes.</td>
</tr>
<tr>
<td><code>flexOutlineSubtotals</code></td>
<td>1</td>
<td>Show child nodes but collapse them.</td>
</tr>
<tr>
<td><code>flexOutlineCollapsed</code></td>
<td>2</td>
<td>Hide all child nodes.</td>
</tr>
</tbody>
</table>

When a node is collapsed or expanded, the control fires the `BeforeCollapse` and `AfterCollapse` events. You may trap these events to prevent certain nodes from being collapsed or expanded, or to populate the outline asynchronously. See the `BeforeCollapse` event for an example.

Setting the `IsCollapsed` property for a row that is not an outline node applies the setting to the row's parent node. If a row has no parent node, an `Invalid Index` error will occur.

For more details on creating and using outlines, see the [Outline Demo](#).

**Data Type**
CollapsedSettings (Enumeration)

### IsSelected Property

Returns or sets whether a row is selected (for listbox-type selections).

**Syntax**

```vba
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).

You may enumerate the selected rows using the SelectedRows and SelectedRow properties.

Data Type
Boolean

IsSubtotal Property
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
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 **ComboBox** and **ColComboBox** 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 **ComboBox**, **ComboBoxIndex**, **ComboBoxCount**, and **ComboBoxData** 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 **ComboBox** and **ColComboBox** 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 **ComboBox**, **ComboBoxIndex**, **ComboBoxCount**, and **ComboBoxData** 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 fg_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 < vbKey0 Or KeyAscii > vbKey9 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
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
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
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).

To ensure that a given cell is visible, use the ShowCell method instead.

**Data Type**
Long

---

**LoadArray Method**
Loads the control with data from a Variant array or from another FlexGrid control.

**Syntax**

```vbscript
[form!]VSFlexGrid.LoadArray [ VariantArray As Variant ], [ RowDim As Long ], [ ColDim As Long ], [ PageDim As Long ], [ CurrentPage As Long ]
```

**Remarks**
The `LoadArray` method loads the grid with data from a Variant array or from another grid. It is identical to the `BindToArray` method, except `BindToArray` keeps the grid connected to the source array or control. `LoadArray` simply loads the data and does not keep a connection between the control and the data source.

For a description of the parameters and some examples, see the `BindToArray` method.

---

**LoadGrid Method**
 Loads grid contents and format from a file.

**Syntax**

```vbscript
[form!]VSFlexGrid.LoadGrid FileName As String, LoadWhat As SaveLoadSettings, [ FixedCells As Boolean ]
```

**Remarks**
The 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**
The name of the file to load, including the path. This file must have been created by the `SaveGrid` method, or an **Invalid File Format** error will occur (error #321).

* **LoadWhat As SaveLoadSettings**
This parameter specifies what should be loaded. Valid options are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexFileAll</td>
<td>0</td>
<td>Load all data and formatting information available in the file.</td>
</tr>
<tr>
<td>flexFileData</td>
<td>1</td>
<td>Load only the data, ignoring formatting information.</td>
</tr>
<tr>
<td>flexFileFormat</td>
<td>2</td>
<td>Load only the formatting, ignoring the data.</td>
</tr>
<tr>
<td>flexFileCommaText</td>
<td>3</td>
<td>Load data from a comma-delimited text file.</td>
</tr>
<tr>
<td>flexFileTabText</td>
<td>4</td>
<td>Load data from a tab-delimited text file.</td>
</tr>
</tbody>
</table>

---

174
Load data from a text file using the delimiters specified by the **ClipSeparators** property.

**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**
Returns or sets whether cells with the same contents will be merged into a single cell.

**Syntax**

```vba
[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 (except 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.

The algorithm used to compare cell contents and determine whether they should be merged is set through the **MergeCompare** property.

The settings for the **MergeCells** property are described below:

<table>
<thead>
<tr>
<th><strong>Constant</strong></th>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>flexMergeNever</em></td>
<td>0</td>
<td>Do not merge cells.</td>
</tr>
<tr>
<td><em>flexMergeFree</em></td>
<td>1</td>
<td>Merge any adjacent cells with same contents (if they are on a row with <strong>RowMerge</strong> set to True or a column with <strong>MergeCol</strong> set to True).</td>
</tr>
<tr>
<td><em>flexMergeRestrictRows</em></td>
<td>2</td>
<td>Merge rows only if cells above are also merged.</td>
</tr>
<tr>
<td><em>flexMergeRestrictColumns</em></td>
<td>3</td>
<td>Merge columns only if cells to the left are also merged.</td>
</tr>
<tr>
<td><em>flexMergeFixedOnly</em></td>
<td>4</td>
<td>Merge only fixed cells. This setting is useful for setting up complex headers for the data and preventing the data itself from being merged.</td>
</tr>
<tr>
<td><em>flexMergeSpill</em></td>
<td>6</td>
<td>Allow long entries to spill into empty adjacent cells.</td>
</tr>
<tr>
<td><em>flexMergeOutline</em></td>
<td>7</td>
<td>Makes entries in subtotal rows spill to fill adjacent empty cells. This setting is useful when you want to display only a node name on the outline nodes and data on the regular (non-node) rows.</td>
</tr>
</tbody>
</table>

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 into 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 into the cells to the right. The picture below shows an example using the \textit{flexMergeSpill} setting. Notice how some cells with long entries spill into adjacent empty cells or get truncated if the adjacent cell is not empty:

<table>
<thead>
<tr>
<th>Product</th>
<th>Associate</th>
<th>Region</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor Oil SAE37-666</td>
<td>Lower East Side</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor Oil SAE37-666</td>
<td>Upper West Side</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor Oil SAE37-666</td>
<td>Lower East Side</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor Oil SAE37-666</td>
<td>Upper West Side</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Motor Oil SAE37-666</td>
<td>Upper West Side</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drums</td>
<td>Sylvia</td>
<td>Lower East</td>
<td>45,342</td>
</tr>
<tr>
<td>Flutes</td>
<td>Donna</td>
<td>South</td>
<td>45,342</td>
</tr>
<tr>
<td>Flutes</td>
<td>John</td>
<td>East</td>
<td>43,432</td>
</tr>
<tr>
<td>Flutes</td>
<td>Mike</td>
<td>West</td>
<td>4,543</td>
</tr>
<tr>
<td>Flutes</td>
<td>Paul</td>
<td>North</td>
<td>4,543</td>
</tr>
</tbody>
</table>

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

\begin{verbatim}
' regular spreadsheet view
fg.MergeCells  = flexMergeNever
fg.MergeCol(0) = True: fg.MergeCol(1) = True: fg.MergeCol(2) = True
fg.MergeCol(3) = False
\end{verbatim}

<table>
<thead>
<tr>
<th>Product</th>
<th>Associate</th>
<th>Region</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drums</td>
<td>Donna</td>
<td>East</td>
<td>2,522</td>
</tr>
<tr>
<td>Drums</td>
<td>John</td>
<td>East</td>
<td>14,323</td>
</tr>
<tr>
<td>Drums</td>
<td>John</td>
<td>North</td>
<td>4,543</td>
</tr>
<tr>
<td>Drums</td>
<td>Paul</td>
<td>North</td>
<td>4,232</td>
</tr>
<tr>
<td>Drums</td>
<td>Paula</td>
<td>East</td>
<td>45,342</td>
</tr>
<tr>
<td>Flutes</td>
<td>Donna</td>
<td>South</td>
<td>45,342</td>
</tr>
<tr>
<td>Flutes</td>
<td>John</td>
<td>East</td>
<td>43,432</td>
</tr>
<tr>
<td>Flutes</td>
<td>Mike</td>
<td>West</td>
<td>4,543</td>
</tr>
<tr>
<td>Flutes</td>
<td>Paul</td>
<td>North</td>
<td>4,543</td>
</tr>
</tbody>
</table>

' free merging: notice how the first region cell (East) merges across employees (Donna and John) to its left.
fg.MergeCells  = flexMergeFree
fg.MergeCol(0) = True: fg.MergeCol(1) = True: fg.MergeCol(2) = True
fg.MergeCol(3) = False
Data Type
MergeSettings (Enumeration)

Default Value
flexMergeNever (0)

MergeCol Property

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.
The **Col** parameter should be set to a value between zero and **Cols** - 1 to set **MergeCol** for a single column, or -1 to set all columns.

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

**Data Type**

Boolean

---

**MergeCompare Property**

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 should be compared when determining whether to merge them or not.

The settings for the **MergeCompare** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexMCExact</td>
<td>0</td>
<td>Cells are merged only if their contents match exactly.</td>
</tr>
<tr>
<td>flexMCNoCase</td>
<td>1</td>
<td>Cells are merged if their contents match in a case-insensitive comparison.</td>
</tr>
<tr>
<td>flexMCTrimNoCase</td>
<td>2</td>
<td>Cells are merged if their contents match after trimming blanks in a case-insensitive comparison.</td>
</tr>
</tbody>
</table>

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

**Data Type**

MergeCompareSettings (Enumeration)

**Default Value**

flexMCExact (0)

---

**MergeRow Property**

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.
The `Row` parameter should be set to a value between zero and `Rows - 1` to set `MergeRow` for a single row, or -1 to set all rows.

You don't need to set `MergeRow` to True when `MergeCells` is set to `flexMergeSpill` (6).

For more details and examples, see the `MergeCells` property.

**Data Type**

Boolean

---

### MouseCol Property

Returns the zero-based index of the column under the mouse pointer.

**Syntax**

```vb
val& = [form!]VSFlexGrid.MouseCol
```

**Remarks**

The `MouseRow` and `MouseCol` properties are 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 while the left button is not pressed. Finally, they are also good for detecting clicks on the fixed areas of the grid.

The code below will highlight the current column when the mouse is moved over a non-fixed cell.

```vb
Private Sub fg_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If fg.MouseRow >= fg.FixedRows And fg.MouseCol >= fg.FixedCols Then
    End If
End Sub
```

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.

**Note:** `MouseRow` and `MouseCol` return -1 if they are invoked while the mouse is not over the control or is over the empty area of the control.

**Data Type**

Long

---

### MouseRow Property

Returns the zero-based index of the row under the mouse pointer.

**Syntax**

```vb
val& = [form!]VSFlexGrid.MouseRow
```
The **MouseRow** and **MouseCol** properties are 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 while the left button is not pressed. Finally, they are also good for detecting clicks on the fixed areas of the grid.

The code below will highlight the current column when the mouse is moved over a non-fixed cell.

```vbscript
Private Sub fg_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If fg.MouseRow >= fg.FixedRows And fg.MouseCol >= fg.FixedCols Then
    End If
End Sub
```

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.

**Note:** **MouseRow** and **MouseCol** return -1 if they are invoked while the mouse is not over the control or is over the empty area of the control.

**Data Type**

Long

**MultiTotals Property**

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 aggregate value.

The examples below show the difference:

```
fg.MultiTotals = True
fg.Subtotal flexSTClear
fg.Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
fg.Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True
```
fg.MultiTotals = False
fg.Subtotal flexSTClear
fg.Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
fg.Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True

Data Type
Boolean

Default Value
True

NodeClosedPicture Property
Returns or sets the picture to be used for closed outline nodes.

Syntax
[form!]VSFlexGrid.NodeClosedPicture[ = Picture ]

Remarks
If a custom picture is not provided, closed outline nodes are represented by a plus sign in a rectangle.

To customize the picture used to represent closed outline nodes, use the following line:

    fg.NodeClosedPicture = LoadPicture(App.Path & "close.ico")
Data Type
Picture

NodeOpenPicture Property
Returns or sets the picture to be used for open outline nodes.

Syntax
[form!]VSFlexGrid.NodeOpenPicture[ = Picture ]

Remarks
If a custom picture is not provided, closed outline nodes are represented by a minus sign in a rectangle.

To customize the picture used to represent open outline nodes, use the following line:

   fg.NodeOpenPicture = LoadPicture(App.Path & "open.ico")

Data Type
Picture

OLECompleteDrag Event
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 target 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDropEffectNone</td>
<td>0</td>
<td>Drop operation was cancelled.</td>
</tr>
<tr>
<td>vbDropEffectCopy</td>
<td>1</td>
<td>Drop results in a copy from the source to the target. The original data remains.</td>
</tr>
<tr>
<td>vbDropEffectMove</td>
<td>2</td>
<td>Drop moves the data from the source to the target. The original data should be deleted.</td>
</tr>
</tbody>
</table>

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.
OLEDrag Method
Initiates an OLE drag operation.

Syntax
[form!]VSFlexGrid.OLEDrag

Remarks
When the OLEDrag method is called, the control’s OLEStartDrag event occurs, allowing it to supply data to a target component.

OLEDragDrop Event
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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDropEffectNone</td>
<td>0</td>
<td>Drop operation was cancelled.</td>
</tr>
<tr>
<td>vbDropEffectCopy</td>
<td>1</td>
<td>Drop results in a copy from the source to the target. The original data remains.</td>
</tr>
<tr>
<td>vbDropEffectMove</td>
<td>2</td>
<td>Drop moves the data from the source to the target. The original data should be deleted.</td>
</tr>
</tbody>
</table>

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 (vb Left Button), the right button is bit 1 (vb Right Button), and the middle button is bit 2 (vb Middle Button). 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 (vb Shift Mask), 2 (vb Ctrl Mask), and 4 (vb Alt Mask), 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.

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the
OLE Drag and Drop Demo.

OLEDragMode Property

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

The settings for the OLEDragMode property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexOLEDragManual</td>
<td>0</td>
<td>When OLEDragMode is set to flexOleDragManual, you must call the OleDrag method to start dragging, which then triggers the OLEStartDrag event. A good place to call the OLEDrag method is in response to the BeforeMouseDown event.</td>
</tr>
<tr>
<td>flexOLEDragAutomatic</td>
<td>1</td>
<td>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.</td>
</tr>
</tbody>
</table>

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.

**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

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDropEffectNone</td>
<td>0</td>
<td>Drop operation was cancelled.</td>
</tr>
<tr>
<td>vbDropEffectCopy</td>
<td>1</td>
<td>Drop results in a copy from the source to the target. The original data remains.</td>
</tr>
<tr>
<td>vbDropEffectMove</td>
<td>2</td>
<td>Drop moves the data from the source to the target. The original data should be deleted.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbEnter</td>
<td>0</td>
<td>Source component is being dragged within the range of a target.</td>
</tr>
<tr>
<td>vbLeave</td>
<td>1</td>
<td>Source component is being dragged out of the range of a target.</td>
</tr>
<tr>
<td>vbOver</td>
<td>2</td>
<td>Source component has moved from one position in the target to another.</td>
</tr>
</tbody>
</table>

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.

OLEDropMode Property
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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexOleDropNone</td>
<td>0</td>
<td>The control does not accept OLE drops and displays the No Drop cursor.</td>
</tr>
<tr>
<td>flexOleDropManual</td>
<td>1</td>
<td>The target component triggers the OLE drop events, allowing the programmer to handle the OLE drop operation in code.</td>
</tr>
<tr>
<td>flexOleDropAutomatic</td>
<td>2</td>
<td>The control automatically accepts OLE drops if the DataObject object contains data in string or file formats.</td>
</tr>
</tbody>
</table>

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.

Data Type
OLEDropModeSettings (Enumeration)

Default Value
flexOLEDropNone (0)

OLEGiveFeedback Event
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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDropEffectNone</td>
<td>0</td>
<td>Drop operation was cancelled.</td>
</tr>
<tr>
<td>vbDropEffectCopy</td>
<td>1</td>
<td>Drop results in a copy from the source to the target. The original data remains.</td>
</tr>
<tr>
<td>vbDropEffectMove</td>
<td>2</td>
<td>Drop moves the data from the source to the target. The original data should be deleted.</td>
</tr>
</tbody>
</table>

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.

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.

OLESetData Event
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.

For an example of implementing OLE drag and drop with the VSFlexGrid control, see the OLE Drag and Drop Demo.

**OLEStartDrag Event**

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbDropEffectNone</td>
<td>0</td>
<td>Drop operation was cancelled.</td>
</tr>
<tr>
<td>vbDropEffectCopy</td>
<td>1</td>
<td>Drop results in a copy from the source to the target. The original data remains.</td>
</tr>
<tr>
<td>vbDropEffectMove</td>
<td>2</td>
<td>Drop moves the data from the source to the target.</td>
</tr>
</tbody>
</table>
target. The original data should be deleted.

For an example of implementing OLE drag and drop with the \texttt{VSFlexGrid} control, see the OLE Drag and Drop Demo.

Outline Method
Sets an outline level for displaying subtotals.

\textbf{Syntax}
\[
\texttt{[form!]VSFlexGrid.Outline Level As Integer}
\]

\textbf{Remarks}
The \texttt{Outline} method collapses or expands an outline to the level specified, collapsing or expanding multiple nodes simultaneously.

The method shows all nodes that have \texttt{RowOutlineLevel} smaller than or equal to the \texttt{Level} parameter specified. Thus, small \texttt{Level} values collapse the outline more, and large values expand it more. If \texttt{Level} is set to zero, only the root node is visible. If \texttt{Level} is set to a very large value (say 100 or so), the outline is totally expanded.

Setting \texttt{Level} to -1 causes the outline to be totally expanded.

When the nodes are collapsed or expanded, the control fires the \texttt{BeforeCollapse} and \texttt{AfterCollapse} events. You may trap these events to cancel the action. See the \texttt{BeforeCollapse} event for an example.

To set up an outline structure using automatic subtotals, see the \texttt{Subtotal} method. To set up a custom outline structure, see the \texttt{IsSubtotal} property. For more details on creating and using outlines, see the \texttt{Outline Demo}.

OutlineBar Property
Returns or sets the type of outline bar that should be displayed.

\textbf{Syntax}
\[
\texttt{[form!]VSFlexGrid.OutlineBar[ = OutlineBarSettings ]}
\]

\textbf{Remarks}
This property determines whether the control should display an outline bar when it is used in outline mode. 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.

The settings for the \texttt{OutlineBar} property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{flexOutlineBarNone}</td>
<td>0</td>
<td>No outline bar.</td>
</tr>
<tr>
<td>\texttt{flexOutlineBarComplete}</td>
<td>1</td>
<td>Complete outline tree plus button row on top. (Buttons are only displayed if the \texttt{OutlineBar} is on a fixed column).</td>
</tr>
<tr>
<td>\texttt{flexOutlineBarSimple}</td>
<td>2</td>
<td>Complete outline tree, no buttons across the top.</td>
</tr>
<tr>
<td>\texttt{flexOutlineBarSymbols}</td>
<td>3</td>
<td>Outline symbols but no connecting lines.</td>
</tr>
<tr>
<td>\texttt{flexOutlineBarCompleteLeaf}</td>
<td>4</td>
<td>Similar to \texttt{flexOutlineBarComplete}, but empty nodes are displayed without symbols.</td>
</tr>
<tr>
<td>\texttt{flexOutlineBarSimpleLeaf}</td>
<td>5</td>
<td>Similar to \texttt{flexOutlineBarSimple}, but empty nodes are...</td>
</tr>
</tbody>
</table>
The following properties affect how each row is displayed on the **OutlineBar**:

<table>
<thead>
<tr>
<th>Property</th>
<th>Affect</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsSubtotal</td>
<td>If True and the row has children, the row is displayed as a node, with a collapse/expand symbol.</td>
</tr>
<tr>
<td>IsCollapsed</td>
<td>If set to flexOutlineCollapsed, the row is displayed with a plus sign that the user can click to expand the node. Otherwise, the row is displayed with a minus sign that the user can click to collapse the node.</td>
</tr>
<tr>
<td>RowOutlineLevel</td>
<td>Controls the indentation of the node. Higher values cause the node to be more indented.</td>
</tr>
</tbody>
</table>

The **OutlineBar** recognizes the following mouse actions: Clicking on a collapsed node (with a plus sign) expands it. Clicking on an expanded node (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. The **OutlineBar** can have a row of buttons across the top that allow the user to collapse the entire outline to a certain level.

By default, the outline bar is drawn on the first column of the control. You may 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.

When a node is collapsed or expanded, the control fires the **BeforeCollapse** and **AfterCollapse** events. You may trap these events to prevent certain nodes from being collapsed or expanded, or to populate the outline asynchronously. See the **BeforeCollapse** event for an example.

For more details on creating and using outlines, see the **Outline Demo**.

**Data Type**
OutlineBarSettings (Enumeration)

**Default Value**
flexOutlineBarNone (0)

**OutlineCol Property**

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.

You should normally use the **AutoSize** method after setting this property, to ensure that the tree and data on the **OutlineCol** column are fully visible.
OwnerDraw Property

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 settings for the OwnerDraw property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexODNone</td>
<td>0</td>
<td>The control performs all drawing itself. The DrawCell event does not get fired at all. (This is the default setting.)</td>
</tr>
<tr>
<td>flexODOver</td>
<td>1</td>
<td>The control draws the cell, then fires the DrawCell event so the application can add text or graphics over the default cell contents.</td>
</tr>
<tr>
<td>flexODContent</td>
<td>2</td>
<td>The control draws the cell background, including any pictures, but no text. It fires the DrawCell event so the application can draw the text.</td>
</tr>
<tr>
<td>flexODComplete</td>
<td>3</td>
<td>The control draws nothing at all in the cell. It fires the DrawCell event and the application is responsible for drawing the entire cell.</td>
</tr>
<tr>
<td>flexODOverFixed</td>
<td>4</td>
<td>Similar to flexODOver, except only fixed cells are owner-drawn.</td>
</tr>
<tr>
<td>flexODContentFixed</td>
<td>5</td>
<td>Similar to flexODContent, except only fixed cells are owner-drawn.</td>
</tr>
<tr>
<td>flexODCompleteFixed</td>
<td>6</td>
<td>Similar to flexODComplete, except only fixed cells are owner-drawn.</td>
</tr>
</tbody>
</table>

For more details, see the DrawCell event.

Data Type
OwnerDrawSettings (Enumeration)

Default Value
flexODNone (0)

PicturesOver Property

Returns or sets whether text and pictures should be overlaid in cells.

Syntax
[form!] VSFlexGrid.PicturesOver[ = {True | False} ]
Remarks
If **PicturesOver** 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 **PicturesOver** 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
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:

The settings for the **PictureType** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexPictureColor</td>
<td>0</td>
<td>Causes the <strong>Picture</strong> property to generate a color bitmap of the control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This mode creates high quality pictures, but they may be quite large and slow to manipulate.</td>
</tr>
<tr>
<td>flexPictureMonochrome</td>
<td>1</td>
<td>Causes the <strong>Picture</strong> property to generate a monochrome bitmap of the control. The mode creates lower quality pictures which consume less memory and are faster to manipulate.</td>
</tr>
</tbody>
</table>

For more details and sample code, see the **Picture** property.

Data Type
PictureTypeSettings (Enumeration)

PrintGrid Method
Prints the grid on the printer.

Syntax

```
[form!]VSFlexGrid.PrintGrid [ DocName As String ], [ ShowDialog As Boolean ],
[ Orientation As Integer ], [ MarginLR As Long ], [ MarginTB As Long ]
```

Remarks
The parameters for the **PrintGrid** method are described below:
DocName As String  (optional)
Contains the name of the document being printed. This string appears in the printer
window's job list and is also used as a footer.

ShowDialog As Variant  (optional, default value = False)
If set to True, a printer selection/setup dialog is displayed before the document start printing.
The user can then select which printer to use, page orientation etc.

Orientation As Variant  (optional, default value = printer default)
Set this parameter to 1 to print the grid in Portrait mode, or set it to 2 to print the grid in
Landscape mode. The default setting, zero, uses the default printer orientation.

MarginLR As Variant  (optional, default value = 1440)
Left and right margins, in twips. The margins must be equal. The default value, 1440,
corresponds to a one-inch margin.

MarginTB As Variant  (optional)
Top and bottom margins, in twips. The margins must be equal. The default value, 1440,
corresponds to a one-inch margin.

The grid is printed using the same fonts used to display it on the screen, so to achieve best
results, make sure the grid's Font property is set to a TrueType font (such as Arial, Times
New Roman, Tahoma, or Verdana).

While printing the grid, the control fires the BeforePageBreak and GetHeaderRow events.
These events allow you to control page breaks and setup repeating headers.

The PrintGrid method prints the entire grid, possibly spilling across and down to new pages.
To print only a part of the grid, hide to rows and columns you don't want to print, call the
PrintGrid method, and restore the hidden rows and columns when you are done.

The code below shows how you can do this:

Private Sub PrintSelection(fg As VSFlexGrid, Row1&, Col1&, Row2&, Col2&)
  ' save current settings
  Dim hl%, tr&, lc&, rd%
  fg.HighLight = 0
  fg.Redraw = flexRDNone

  ' hide non-selected rows and columns
  Dim i&
  For i = fg.FixedRows To fg.Rows - 1
    If i < Row1 Or i > Row2 Then fg.RowHidden(i) = True
  Next
  For i = fg.FixedCols To fg.Cols - 1
    If i < Col1 Or i > Col2 Then fg.ColHidden(i) = True
  Next

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

  ' print visible area
  fg.PrintGrid

  ' restore control
  fg.RowHidden(-1) = False
  fg.ColHidden(-1) = False
End Sub
Redraw Property

Enables or disables redrawing of the VSFlexGrid control.

Syntax

\[
\text{[form!]VSFlexGrid.Redraw} = \text{RedrawSettings}
\]

Remarks

The settings for the Redraw property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexRDNone</td>
<td>0</td>
<td>The grid does not repaint itself.</td>
</tr>
<tr>
<td>flexRDDirect</td>
<td>1</td>
<td>The grid paints its contents directly on the screen. This is the fastest</td>
</tr>
<tr>
<td></td>
<td></td>
<td>repaint mode, but there occasionally it may cause a little flicker.</td>
</tr>
<tr>
<td>flexRDBuffered</td>
<td>2</td>
<td>The grid paints its contents on an off-screen buffer, then transfers the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complete image to the screen. This mode is slightly slower than flexRDDirect,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>but it eliminates flicker.</td>
</tr>
<tr>
<td>True</td>
<td>-1</td>
<td>Equivalent to flexRDDirect. This setting is allowed for compatibility with</td>
</tr>
<tr>
<td></td>
<td></td>
<td>previous versions of the control.</td>
</tr>
</tbody>
</table>

The Redraw property is used in one of the main optimizations available to VSFlexGrid users. Before making extensive changes to the grid, set the Redraw property to flexRDNone to suspend repainting until you are done with the changes. Then restore Redraw to its previous setting. Doing this will reduce flicker and increase speed. This optimization is especially effective when adding large numbers of rows to the grid, because the control needs to recalculate the scroll ranges each time a row is added.

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.

```vbscript
Sub UpdateGrid()
    fa.Redraw = flexRDNone ' suspend redrawing to avoid flicker
    fg.Rows = 1
    Dim i As Long
    For i = 1 To 10000
        fg.AddItem "Row " & i
    Next
    fg.Redraw = True       ' resume redrawing
End Sub
```

Data Type

RedrawSettings (Enumeration)

Default Value

flexRDDirect (1)

RemoveItem Method

Removes a row from the control.
Syntax

[form!]VSFlexGrid.RemoveItem [ Row As Long ]

Remarks
The Row parameter determines which row should be removed from the control. If provided, it must be in the range between 0 and Rows-1, or an Invalid Index error will occur. If omitted, the last row is deleted.

RightCol Property

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 the contents of the control set the TopRow and LeftCol properties instead. To ensure that a cell is visible, use the ShowCell method.

Data Type
Long

Row Property (VSFlexGrid Control)

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.

The Row property may be set to -1 to hide the selection, to a value between zero and FixedRows - 1 to select a cell in a fixed row, or to a value between FixedRows and Rows - 1 to select a cell in a scrollable row. Setting Row to other values will trigger an Invalid Index error.

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

Setting the Row and Col properties does not ensure that the current cell is visible. To do that, use the ShowCell method.

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

Data Type
Long
RowColChange Event
Fired when the current cell (Row, Col) changes to a different cell.

Syntax
Private Sub VSFlexGrid_RowColChange()

Remarks
RowColChange is fired when the Row or Col properties change, either as a result of user actions (mouse or keyboard) or through code. It 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
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. You may also associate values to individual cells using the Cell(flexcpData) property.

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.

Because these properties hold Variants, you have extreme flexibility in the types of information you may associate with each row, column or cell. The example below shows some ways in which you can use the RowData property:

```vbnet
Dim coll As New Collection
    coll.Add "Hello"
    coll.Add "World"
    fg.RowData(1) = 212 ' store a number
    fg.RowData(2) = "Hello" ' store a string
    fg.RowData(3) = coll ' store a pointer to an object
    fg.RowData(4) = Me ' store a pointer to a form
```

Debug.Print TypeName(fg.RowData(1)), fg.RowData(1)
Debug.Print TypeName(fg.RowData(2)), fg.RowData(2)
Debug.Print TypeName(fg.RowData(3)), fg.RowData(3).Item(2)
Debug.Print TypeName(fg.RowData(4)), fg.RowData(4).Caption

This code produces the following output:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>212</td>
</tr>
<tr>
<td>String</td>
<td>Hello</td>
</tr>
<tr>
<td>Collection</td>
<td>World</td>
</tr>
<tr>
<td>Form1</td>
<td>Form1</td>
</tr>
</tbody>
</table>
You can search for rows that contain specific RowData values using the FindRow property.

**Data Type**
Variant

**RowHeight Property**
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 the Row parameter is -1, then the specified height is applied to all rows.

If you set RowHeight to -1, the row height is reset to its default value, which depends on the size of the control's current font. If you set RowHeight to zero, 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.

To set row heights automatically, based on the contents of the control, set the AutoSizeMode property to flexAutoSizeRowHeight (1) and call theAutoSize method.

**Data Type**
Long

**RowHeightMax Property**
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 theAutoSize method to automatically set row heights, to prevent some rows from becoming too large.

See also theColWidthMin, ColWidthMax, and RowHeightMin properties.

**Data Type**
Long

**RowHeightMin Property**
Returns or sets the minimum row height, in twips.
Syntax

\[ \text{[form!]} \text{VSFlexGrid.RowHeightMin[ = value As Long ]} \]

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

RowHidden Property
Returns or sets whether a row is hidden.

Syntax

\[ \text{[form!]} \text{VSFlexGrid.RowHidden(Row As Long)[ = \{True | False\}]} \]

Remarks
Use the RowHidden property to hide and display rows. This is a better approach than setting the row's RowHeight property to zero, it allows you to display the row later with its original height.

When the control collapses or expands an outline branch, either as a result of user mouse action or programmatically (see the Outline method and IsCollapsed property), it sets the RowHidden property accordingly.

Hidden rows are ignored by the AutoSize method.

When setting this property, the Row parameter should be set to a value between zero and Rows - 1 to hide or show a given row, or to -1 to hide or show all rows.

Data Type
Boolean

RowIsVisible Property
Returns whether a given row is currently within view.

Syntax

\[ \text{val\%} = \text{[form!]} \text{VSFlexGrid.RowIsVisible(Row As Long)} \]

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.
To ensure a given row is visible, use the ShowCell method.

**Data Type**
Boolean

### RowOutlineLevel Property
Returns or sets the outline level for a subtotal row.

**Syntax**

```
(form!)VSFlexGrid.RowOutlineLevel(Row As Long)[ = value As Integer ]
```

**Remarks**
The RowOutlineLevel property is used for two closely related purposes.

1) When using the grid in outline mode, RowOutlineLevel is used to set the hierarchical level of a node. Nodes with high outline level are children of rows with low outline level. The root node has the lowest outline level. You may change the relationship between nodes by modifying the value of the RowOutlineLevel property. For more details on creating and using outlines, see the Outline Demo.

2) When using the Subtotal method to create subtotals automatically, RowOutlineLevel stores the number of the column being used for grouping the data. For more details on creating and using automatic subtotals, see the Subtotal method.

The RowOutlineLevel is only used by the control if the IsSubtotal property is set to True.

**Data Type**
Integer

### RowPos Property
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
Moves a given row to a new position.
**Syntax**

```
[form!]vsFlexGrid.RowPosition(Row As Long)[ = NewPosition As Long ]
```

**Remarks**

The Row and NewPosition parameters must be valid row indices (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**

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 cause a run time error. To make sure your code works properly when dealing with large controls, you should add error-handling code to your programs.

If you increase the value of the Rows property, new rows are appended to the bottom of the grid. To insert rows at specific positions, use the AddItem method. If you decrease the value of the Rows property, the bottom rows are removed from the control. To remove rows at specific positions, use the RemoveItem method.

**Data Type**

Long

---

**RowSel Property**

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 modify a selection or to determine which cells are currently selected. Rows and columns 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, 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.

If the SelectionMode property is set to flexSelectionListBox (3), you should use the IsSelected property to select and deselect rows.

Note that when a range is selected, the value of Row may be greater than or less than RowSel, and Col may be greater than or less than ColSel. This is inconvenient when you need to set up bounds for loops. In these cases, use the GetSelection method to retrieve selection in an ordered fashion.

Data Type
Long

RowStatus Property
Returns or sets 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexrsNew</td>
<td>0</td>
<td>When the row is created.</td>
</tr>
<tr>
<td>flexrsUpdated</td>
<td>1</td>
<td>When the program modifies a row by writing to it.</td>
</tr>
<tr>
<td>flexrsModified</td>
<td>2</td>
<td>When the user modifies a row by editing it.</td>
</tr>
<tr>
<td>flexrsDeleted</td>
<td>3</td>
<td>Not assigned by the control, but used as a return value if you request the status of a row that does not exist (e.g. RowStatus(-1)).</td>
</tr>
</tbody>
</table>

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 set to flexrsNew (0). If you then assign values to one of the rows, its status will become flexrsUpdated (1). If the user then edits one or more values on this row, the status becomes flexrsModified (2).

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
Saves grid contents and format to a file.

Syntax
[form!]VSFlexGrid.SaveGrid FileName As String, SaveWhat As SaveLoadSettings, [ FixedCells As Boolean ]

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
The name of the file to create, including the path. If a file with the same name already exists, it is overwritten.

SaveWhat As SaveLoadSettings
This parameter specifies what should be saved. Valid options are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexFileAll</td>
<td>0</td>
<td>Save all data and formatting information.</td>
</tr>
<tr>
<td>flexFileData</td>
<td>1</td>
<td>Save only the data, ignoring formatting information.</td>
</tr>
<tr>
<td>flexFileFormat</td>
<td>2</td>
<td>Save only the global formatting, ignoring the data.</td>
</tr>
<tr>
<td>flexFileCommaText</td>
<td>3</td>
<td>Save data to a comma-delimited text file.</td>
</tr>
<tr>
<td>flexFileTabText</td>
<td>4</td>
<td>Save data to a tab-delimited text file.</td>
</tr>
<tr>
<td>flexFileCustomText</td>
<td>5</td>
<td>Save data to a text file using the delimiters specified by the ClipSeparators property.</td>
</tr>
</tbody>
</table>

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.

If your application requires you to save several grids, you should consider using the Archive method to compress and combine them all into a single archive file. You can later use the ArchiveInfo method to retrieve information from the archive file.

ScrollBars Property
Returns or sets whether the control will display horizontal or vertical scroll bars.

Syntax
[form!]VSFlexGrid.ScrollBars[ = ScrollBarsSettings ]

Remarks
The settings for the ScrollBars property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>

201
Scroll bars are displayed only if the contents of the control extend beyond its borders. For example, if `ScrollBars` is set to `flexScrollBarHorizontal`, a horizontal scroll bar is displayed only if the control is not wide enough to display all columns at once.

If the control has no scroll bars in a 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.

**Default Value**
`flexScrollBarBoth (3)`

### ScrollTips Property

Returns or sets whether tool tips are shown while the user scrolls vertically.

**Syntax**

```
[form]!VSFlexGrid.ScrollTips = {True | False}
```

**Remarks**

Use the `ScrollTips` 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:

```vbnet
Private Sub Form_Load()
    fg.ScrollTrack = False
    fg.ScrollTips = True
End Sub

Private Sub fg_BeforeScrollTip(ByVal Row As Long)
    ' the ScrollTip will show a string such as
    ' "Row 5: Accounts Receivable"
    fg.ScrollTipText = "Row " & Row & ": " & fg.TextMatrix(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
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
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.

The example below scrolls the grid without waiting for the thumb to be released:

   fg.ScrollTrack = True

Data Type
Boolean

SelChange Event
Fired after the selected range (RowSel, ColSel) changes.

Syntax
Private Sub VSFlexGrid_SelChange()

Remarks
SelChange is fired after 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.

The example below prints the coordinates of the current selection to the debug window when the selection changes:
Private Sub fg_SelChange()


End Sub

Select Method (VSFlexGrid Control)
Selects a range of cells.

Syntax
[form!]VSFlexGrid.Select Row As Long, Col As Long, [ RowSel As Long ], [ ColSel As Long ]

Remarks
The Select method allows you to select ranges or cells (by omitting the last two parameters) with a single command.

The following code selects the entire scrollable (data) part of the control:


This method is more efficient than setting the Row, Col, RowSel, and ColSel properties separately and makes the code more readable.

You may use the GetSelection method to retrieve the current selection.

SelectedRow Property
Returns the position of a selected row when SelectionMode is set to flexSelectionListBox.

Syntax
[form!]VSFlexGrid.SelectedRow(Index As Long)[ = value 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 (3), which allows the user to select multiple, non-adjacent rows.

Using the SelectedRows and SelectedRow properties to enumerate all selected rows is equivalent to, but faster than scanning the entire control for selected rows by reading the IsSelected property.

For example,

        For i = 0 to fg.SelectedRows - 1
            Debug.Print "Row "; fg.SelectedRow(i); " is selected"
        Next

is faster than

        For i = 0 to fg.Rows
            If fg.IsSelected(i) Then Debug.Print "Row "; i; " is selected"
        Next
SelectedRows Property

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 (3), which allows the user to select multiple, non-adjacent rows.

The code below prints the number of selected rows to the debug window when the selection changes:

Private Sub Form_Load()
    fg.SelectionMode = flexSelectionListBox
End Sub

Private Sub fg_SelChange()
    Debug.Print fg.SelectedRows
End Sub

Using the SelectedRows and SelectedRow properties to enumerate all selected rows is equivalent to, but faster than scanning the entire control for selected rows by reading the IsSelected property.

For an example, see the SelectedRow property.

Data Type

Long

SelectionMode Property

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:
### Constant

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Allows selections to be made as usual, spreadsheet-style.</td>
</tr>
<tr>
<td>1</td>
<td>Forces selections to span entire rows. Useful for implementing record-based displays.</td>
</tr>
<tr>
<td>2</td>
<td>Forces selections to span entire columns. Useful for selecting ranges for a chart or fields for sorting.</td>
</tr>
<tr>
<td>3</td>
<td>Similar to flexSelectionByRow, but allows non-continuous selections.CTRL-clicking with the mouse toggles the selection for an individual row. Dragging the mouse over a group of rows toggles their selected state.</td>
</tr>
</tbody>
</table>

When **SelectionMode** is set to **flexSelectionListBox**, you may use the **IsSelected** property allows you to read and set the selected status of individual rows, and the **SelectedRows** property to enumerate selected rows.

Private Sub fg_KeyDown()
    If fg.MouseRow < fg.FixedRows And fg.MouseCol > -fg.FixedCols Then
        fg.SelectionMode = flexSelectionByColumn
    End If
    If fg.MouseCol < fg.FixedRows And fg.MouseRow >= fg.FixedRows Then
        fg.SelectionMode = flexSelectionByRow
    End If
End Sub

To prevent range selection altogether, set the **AllowSelection** property to False.

**Data Type**
SelModeSettings (Enumeration)

**Default Value**
flexSelectionFree (0)

### SheetBorder Property

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
Returns or sets whether drop-down buttons are shown when editable cells are selected.

Syntax
\[
\text{[form!] VSFlexGrid.ShowComboButton[ = \{ True | False \}]}
\]

Remarks
If \text{ShowComboButton} is set to True, the control displays 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 \text{ShowComboButton} 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 \text{EditCell} method is used.)

The following code prepares the grid to show a button for the combo list as soon as a cell is selected:

\begin{verbatim}
With fg
  .Editable = flexEDKbdMouse
  .ComboList = "Item 1|Item 2|Item 3"
  .ShowComboButton = True
End With
\end{verbatim}

To determine whether or not a given cell has an associated \text{ComboList}, the control fires the \text{BeforeEdit} event every time it needs to paint a cell with the focus. If this behavior is undesirable, set \text{ShowComboButton} to False and the event will no longer be fired. Also set \text{ShowComboButton} 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 (VSFlexGrid Control)
Sets a sorting order for the selected rows using the selected columns as keys.

Syntax
\[
\text{[form!] VSFlexGrid.Sort = SortSettings}
\]

Remarks
The \text{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 \text{Sort} property also honors outline structures. It will only sort data rows, and will not scramble nodes.

The range of rows to be sorted is specified by setting the \text{Row} and \text{RowSel} properties. If \text{Row} and \text{RowSel} are the same, the control sorts all non-fixed rows.

They keys used for sorting are determined by the \text{Col} and \text{ColSel} properties, always from the left to the right. For example, if \text{Col} = 3 and \text{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 described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexSortNone</td>
<td>0</td>
<td>Ignore this column when sorting. This setting is useful when you assign it to a column’s Cols property, then set Sort to flexSortUseColSort.</td>
</tr>
<tr>
<td>flexSortGenericAscending</td>
<td>1</td>
<td>Sort strings and numbers in ascending order.</td>
</tr>
<tr>
<td>flexSortGenericDescending</td>
<td>2</td>
<td>Sort strings and numbers in descending order.</td>
</tr>
<tr>
<td>flexSortNumericAscending</td>
<td>3</td>
<td>Sort numbers in ascending order.</td>
</tr>
<tr>
<td>flexSortNumericDescending</td>
<td>4</td>
<td>Sort numbers in descending order.</td>
</tr>
<tr>
<td>flexSortStringNoCaseAscending</td>
<td>5</td>
<td>Sort strings in ascending order, ignoring capitalization.</td>
</tr>
<tr>
<td>flexSortStringNoCaseDescending</td>
<td>6</td>
<td>Sort strings in descending order, ignoring capitalization.</td>
</tr>
<tr>
<td>flexSortStringAscending</td>
<td>7</td>
<td>Sort strings in ascending order.</td>
</tr>
<tr>
<td>flexSortStringDescending</td>
<td>8</td>
<td>Sort strings in descending order.</td>
</tr>
<tr>
<td>flexSortCustom</td>
<td>9</td>
<td>Fire a Compare event and use the return value to sort the columns.</td>
</tr>
<tr>
<td>flexSortUseColSort</td>
<td>10</td>
<td>This setting 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.</td>
</tr>
</tbody>
</table>

The flexSortCustom is the most flexible setting. It fires a Compare event that allows you to compare rows in any way you wish, using any columns in any order. However, it is also much slower than the others, so it should be used only when really necessary or when the grid has only a few rows. If you want to sort based on arbitrary criteria (e.g. "Urgent", "High", "Medium", "Low"), consider using a hidden column with numerical values that correspond to the criteria you are using.

To sort dates, make sure the column containing the dates has its ColDataType property set to flexDTDate (7). This will allow the control to sort them properly. For example:

```plaintext
fg.ColDataType(i) = flexDTDate
fg.Col = i
fg.Sort = flexSortGenericAscending
```

The example below shows how the Sort property is used:

```plaintext
' fill control with random data
fg.Cols = 2
fg.FixedCols = 0
FillColumn fg, 0, "Name|Andrew|John|Paul|Mary|Tom|Dick|Harry"
FillColumn fg, 1, "Number|12|32|45|2|65|8|87|34"
```
'sort by name
fg.Select 1, 0
fg.Sort = flexSortGenericAscending

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>32</td>
</tr>
<tr>
<td>Dick</td>
<td>34</td>
</tr>
<tr>
<td>John</td>
<td>8</td>
</tr>
<tr>
<td>Tom</td>
<td>32</td>
</tr>
<tr>
<td>John</td>
<td>87</td>
</tr>
<tr>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>Paul</td>
<td>12</td>
</tr>
<tr>
<td>Andrew</td>
<td>34</td>
</tr>
<tr>
<td>John</td>
<td>2</td>
</tr>
</tbody>
</table>

'sort by name and number
fg.Select 1, 0, 1, 1
fg.Sort = flexSortGenericAscending

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>Andrew</td>
<td>34</td>
</tr>
<tr>
<td>Andrew</td>
<td>2</td>
</tr>
<tr>
<td>Dick</td>
<td>34</td>
</tr>
<tr>
<td>Dick</td>
<td>45</td>
</tr>
<tr>
<td>John</td>
<td>8</td>
</tr>
<tr>
<td>John</td>
<td>87</td>
</tr>
<tr>
<td>John</td>
<td>2</td>
</tr>
<tr>
<td>John</td>
<td>8</td>
</tr>
</tbody>
</table>

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:

fg.ColSort(0)=flexSortGenericAscending
fg.ColSort(1) = flexSortGenericDescending
fg.Select 1, 0, 1, 1
fg.Sort = flexSortUseColSort
The cell property has a new sort setting that allows you to sort a range without changing the selection.

**Data Type**
SortSettings (Enumeration)

---

**StartEdit Event**
Fired when the control enters cell edit mode (after BeforeEdit).

**Syntax**
Private Sub VSFlexGrid_StartEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)

**Remarks**
This event is fired before the control enters edit mode. It allows you to prevent editing by setting the Cancel parameter to True, to supply a list of choices for a combo list with the ComboList property, or to specify an edit mask with the EditMask property.

If the choices or the mask are the same for a whole column, you may set them with the ColComboList and ColEditMask properties, and you don't need to handle the StartEdit event.

The parameters for the StartEdit event are described below:

- **Row As Long, Col As Long**
  Indicate which cell is about to be edited or repainted.

- **Cancel As Boolean**
  Allows you to cancel the editing operation.

The difference between the BeforeEdit and StartEdit events is that the former gets fired every time the current cell is repainted, and does not guarantee that the control is really about to enter edit mode. The StartEdit event, on the other hand, is only fired when the cell is about to enter edit mode.

The following code sets the grid to show a different background color for the cell that is currently being edited:

```vbnet
Private Sub fg_StartEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
    fg.CellBackColor = vbYellow
End Sub

Private Sub fg_AfterEdit(ByVal Row As Long, ByVal Col As Long)
```
fg.CellBackColor = vbDefault

End Sub

If the user starts editing by pressing a key, several events get fired in the following sequence:

KeyDown 65        ' user pressed the 'a' key (65 = 'A')
BeforeEdit 1 1   ' allows you to cancel the editing
**StartEdit** 1 1 ' not canceled, edit is about to start
KeyPressEdit 97   ' 'a' key passed to editor (97 = 'a')
KeyUpEdit 65      ' user released the key
KeyDownEdit 13    ' user pressed Enter
ValidateEdit      ' allows you to validate the edit
AfterEdit 1 1     ' editing done
BeforeEdit 1 1 ' repainting cell
KeyUp 13          ' user released Enter key

**StartPage Event**

Fired before each page while the grid is being printed.

**Syntax**

```vbscript
Private Sub VSFlexGrid_**StartPage**( ByVal hDC As Long, ByVal Page As Long, Cancel As Boolean)
```

**Remarks**

This event gets fired once for each page while the grid is being printed with the **PrintGrid** method.

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

- **hDC As Long**
  This parameter contains a handle to the printer's device context. The **hDC** parameter is required by all Windows GDI calls, and you may use it to add graphical elements or text to the page.

- **Page As Long**
  The number of the page being printed.

- **Cancel As Boolean**
  Set this parameter to True to cancel the print job.

The following code outputs the current status of the print job to the debug window:

```vbscript
Private Sub fg_**StartPage**(ByVal hDC As Long, ByVal Page As Long, Cancel As Boolean)
    Debug.Print "Printing page " & Page & "..."
End Sub
```
Subtotal Method

Inserts rows with summary data.

**Syntax**

```vbscript
[form1]!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 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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexSTNone</td>
<td>0</td>
<td>Outline only, no aggregate values</td>
</tr>
<tr>
<td>flexSTClear</td>
<td>1</td>
<td>Clear all subtotals</td>
</tr>
<tr>
<td>flexSTSum</td>
<td>2</td>
<td>Sum</td>
</tr>
<tr>
<td>flexSTPercent</td>
<td>3</td>
<td>Percent of total sum</td>
</tr>
<tr>
<td>flexSTCount</td>
<td>4</td>
<td>Row count</td>
</tr>
<tr>
<td>flexSTAverage</td>
<td>5</td>
<td>Average</td>
</tr>
<tr>
<td>flexSTMax</td>
<td>6</td>
<td>Maximum</td>
</tr>
<tr>
<td>flexSTMin</td>
<td>7</td>
<td>Minimum</td>
</tr>
<tr>
<td>flexSTStd</td>
<td>8</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>flexSTVar</td>
<td>9</td>
<td>Variance</td>
</tr>
</tbody>
</table>

**GroupOn** 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
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. If this parameter is omitted, the column's default format (defined by the ColFormat property) is used.

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
fg.ColFormat(3) = "$(#,###.00)"  ' set format for calculated totals
fg.Subtotal flexSTClear        ' remove old values

' calculate subtotals (the order doesn't matter)
' (sales values to be added are in column 3)
' col 0: product
fg/Subtotal flexSTSum, 0, 3, , vbRed
' col 1: employee
fg/Subtotal flexSTSum, 1, 3, , vbGreen
' col 2: region
fg/Subtotal flexSTSum, 2, 3, , vbBlue

' total on a negative column to get a grand total
fg/Subtotal flexSTSum, -1, 3, , vbblue, vbwhite, true
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:

```vbscript
fg.ColFormat(3) = "$(#,###.00)" ' set format for calculated totals
fg.Subtotal flexSTClear         ' remove old values

' calculate subtotals (the order doesn't matter)
' (sales values to be added are in column 3)
' col 0: product
fg.Subtotal flexSTSum, 0, 3, , vbRed ,,," TotPrd %s",,True
' col 1: employee
fg.Subtotal flexSTSum, 1, 3, , vbGreen,,"  TotEmp %s",,True
' col 2: region
fg.Subtotal flexSTSum, 2, 3, , vbBlue ,,,"   TotRgn %s",,True

' total on a negative column to get a grand total
fg.Subtotal flexSTSum, -1, 3, , vbblue, vbwhite, true
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Associate</th>
<th>Region</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand Total</td>
<td></td>
<td></td>
<td>$1736.10</td>
</tr>
<tr>
<td>Total Drums</td>
<td></td>
<td></td>
<td>$161.66</td>
</tr>
<tr>
<td>Drums</td>
<td>Total Donna</td>
<td></td>
<td>$47.67</td>
</tr>
<tr>
<td>Drums</td>
<td>Donna</td>
<td>Total East</td>
<td>$37.67</td>
</tr>
<tr>
<td>Drums</td>
<td>Donn</td>
<td>East</td>
<td>$2,532.00</td>
</tr>
<tr>
<td>Drums</td>
<td>Donn</td>
<td>East</td>
<td>$45,342.00</td>
</tr>
<tr>
<td>Drums</td>
<td>Total John</td>
<td></td>
<td>$18.67</td>
</tr>
<tr>
<td>Drums</td>
<td>John</td>
<td>Total East</td>
<td>$14.32</td>
</tr>
</tbody>
</table>

SubtotalPosition Property

Returns or sets whether subtotals should be inserted above or below the totaled data.

**Syntax**

`[form!]VSFlexGrid.SubtotalPosition[ = SubtotalPositionSettings ]`

**Remarks**

The settings for the SubtotalPosition property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexSTBelow</td>
<td>0</td>
<td>Subtotals are inserted below the data. This setting creates grids that look like reports.</td>
</tr>
<tr>
<td>flexSTAbove</td>
<td>1</td>
<td>Subtotals are inserted above the data. This setting creates grids that look like outline trees (this is the...</td>
</tr>
</tbody>
</table>
The **SubtotalPosition** property affects the placement of subtotals added using the **Subtotal** method and also the appearance and behavior of the **OutlineBar**. When **SubtotalPosition** is set to `flexSTAbove`, parent nodes appear at the top of the grid, like a standard tree control. When **SubtotalPosition** is set to `flexSTBelow`, parent nodes appear at the bottom of the grid, giving the tree an upside-down appearance.

The example below shows how to insert a subtotal positioned below the calculated values:

```vba
Private Sub Form_Load()
    fg.Rows = 3
    fg.TextMatrix(1, 1) = "Rent": fg.TextMatrix(1, 2) = "234"
    fg.TextMatrix(2, 1) = "Rent": fg.TextMatrix(2, 2) = "335"
    fg.SubtotalPosition = flexSTBelow
    fg.Subtotal flexSTSum, 1, 2
End Sub
```

**Data Type**
SubtotalPositionSettings (Enumeration)

**Default Value**
flexSTAbove (1)

---

**TabBehavior Property**

Returns or sets whether the tab key will move focus between controls (VB default) or between grid cells.

**Syntax**

```vba
[form!]VSFlexGrid.TabBehavior[ = TabBehaviorSettings ]
```

**Remarks**
The settings for the **TabBehavior** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flexTabControls</code></td>
<td>0</td>
<td>Tab key is used to move to the next or previous control on the form.</td>
</tr>
<tr>
<td><code>flexTabCells</code></td>
<td>1</td>
<td>Tab key is used to move to the next or previous cell on the control.</td>
</tr>
</tbody>
</table>

The example below sets the tab key to move to the next control when in the last cell of the grid:

```vba
Private Sub fg_GotFocus()
    fg.Select fg.FixedRows, fg.FixedCols
End Sub

Private Sub fg_EnterCell()
    If fg.Col = fg.Cols - 1 And fg.Row = fg.Rows - 1 Then
        fg.TabBehavior = flexTabControls
    Else
        fg.TabBehavior = flexTabCells
    End If
End Sub
```
End Sub

**Data Type**
TabBehaviorSettings (Enumeration)

**Default Value**
flexTabControls (0)

### Text Property (VSFlexGrid Control)

Returns or sets the contents of the selected cell or range.

**Syntax**

\[
\text{[form!]}\text{VSFlexGrid}\text{.Text}[ = \text{value As String }]
\]

**Remarks**

The `Text` property retrieves the contents of the current cell, defined by the `Row` and `Col` properties. When a string is assigned to the `Text` property, it is applied either to the current cell or copied over the current selection, depending on the settings of the `FillStyle` property.

The code below sets the contents of the fourth column of the fourth row to "Apple"

```vbnet
fg.Select 3, 3
fg.Text = "Apple"
```

To read or set the contents of an arbitrary cell, use the `Cell(flexcpText)` or `TextMatrix` properties. To read the formatted contents of a cell using the `Cell(flexcpTextDisplay)` property. To read the numeric value of a cell, use the `Cell(flexcpValue)`, `Value`, or `ValueMatrix` properties.

**Data Type**

String

### TextArray Property

Returns or sets the contents of a cell identified by a single index.

**Syntax**

\[
\text{[form!]}\text{VSFlexGrid}\text{.TextArray(Index As Long)}[ = \text{value As String }]
\]

**Remarks**

This property is provided for backward compatibility with earlier versions of this control. New applications should use the `Cell(flexcpText)` or `TextMatrix` properties.

The following code places the text "Apple" in the second row and second column of a five column grid:

```vbnet
fg.TextArray(6) = "Apple"
```

**Data Type**

String
TextMatrix Property
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.

The following code places text into an arbitrary cell:

fg.TextMatrix(1, 1) = "Apple"

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
Returns or sets 3D effects for displaying text in non-fixed cells.

Syntax
[form!]VSFlexGrid TextStyle[ = TextStyleSettings ]

Remarks
The settings for the TextStyle property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexTextFlat</td>
<td>0</td>
<td>Draw text normally.</td>
</tr>
<tr>
<td>flexTextRaised</td>
<td>1</td>
<td>Draw text with a strong raised 3-D effect.</td>
</tr>
<tr>
<td>flexTextInset</td>
<td>2</td>
<td>Draw text with a strong inset 3-D effect.</td>
</tr>
<tr>
<td>flexTextRaisedLight</td>
<td>3</td>
<td>Draw text with a light raised 3-D effect.</td>
</tr>
<tr>
<td>flexTextInsetLight</td>
<td>4</td>
<td>Draw text with a light inset 3-D effect.</td>
</tr>
</tbody>
</table>

Constants flexTextRaised and flexTextInset work best for large and bold fonts. Constants flexTextRaisedLight and flexTextInsetLight work best for small regular fonts.

The example below prints text with an inset 3-D effect in non-fixed cells:

fg.TextStyle = flexTextInset

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
Returns or sets 3D effects for displaying text in fixed cells.

Syntax
\[
\text{form!} VSFlexGrid.TextStyleFixed = \text{TextStyleSettings }
\]

Remarks
Valid settings for this property are the same as those for the TextStyle property.

The example below prints text with a raised 3-D effect in fixed cells:

\[\text{fg TextStyleFixed} = \text{flexTextRaised}\]

Data Type
TextStyleSettings (Enumeration)

Default Value
flexTextFlat (0)

TopRow Property
Returns or sets the zero-based index of the topmost non-fixed row displayed in the control.

Syntax
\[
\text{form!} VSFlexGrid.TopRow = \text{value As Long }
\]

Remarks
Setting the TopRow property causes the control to scroll through its contents vertically so that the given row becomes the top visible row. 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.

The following code selects an arbitrary row and makes it the topmost row displayed in the grid:

Private Sub Command1_Click()
  fg.SelectionMode = flexSelectionListBox
  fg.IsSelected(5) = True
  fg.TopRow = 5
End Sub

To scroll horizontally, use the LeftCol property.

When setting this property, the largest possible row number is the total number of rows minus the number of rows that will fit the display. Attempting to set TopRow to a greater value will cause the control to set it to the largest possible value (no error will occur).

To ensure that a given cell is visible, use the ShowCell method instead.

Data Type
Long
TreeColor Property
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
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
This 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 fails validation, 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.

For example, the code below shows a typical handler for the ValidateEdit event. In this case, column 1 only accepts strings, and column 2 only accepts numbers greater than zero:

Sub fg_ValidateEdit(ByVal Row As Long, ByVal Col As Long, Cancel As Boolean)
Dim c$
Select Case Col ' different validation rules for each column
Case 1 ' column 1 only accepts strings
  c = Left$(fg.EditText, 1)
  If UCaseS(c) < "A" And UCase$(c) > "Z" Then Beep: Cancel = True
Case 2 ' column 2 only accepts numbers > 0
  If Val(fg.EditText) <= 0 Then Beep: Cancel = True
End Select
End Sub

Value Property
Returns the numeric value of the current cell.
**Syntax**

\[\text{val} = \text{[form!]VSFlexGrid.Value}\]

**Remarks**

This property is similar to Visual Basic's \textbf{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 \textbf{Value} property will return the value -1234.56.

The following code outputs the value of the current cell to the debug window:

```
Private Sub Command1_Click()
    Debug.Print fg.Value
End Sub
```

To retrieve the value of an arbitrary cell without selecting it first, use the \textbf{Cell(flexcpValue)} or the \textbf{ValueMatrix} properties.

**Note**

This property is not an expression evaluator. If the current cell contains the string "2+2", for example, the \textbf{Value} property will return 2 instead of 4. The Visual Basic statement \textbf{Val("2+2")} also returns 2.

**Data Type**

Double

---

**ValueMatrix Property**

Returns the numeric value of a cell identified by its row and column coordinates.

**Syntax**

\[\text{val} = \text{[form!]VSFlexGrid.ValueMatrix(Row As Long, Col As Long)}\]

**Remarks**

This property is similar to the \textbf{Value} property, except it allows you to specify the cell whose value is to be retrieved.

The following code outputs the value of an arbitrary cell to the debug window:

```
Debug.Print fg.ValueMatrix(1, 1)
```

**Data Type**

Double

---

**Version Property (VSFlexGrid Control)**

Returns the version of the control currently loaded in memory.

**Syntax**

\[\text{val\%} = \text{[form!]VSFlexGrid.Version}\]
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 7.00 returns 700.

The example below displays the version number of the control:

```vbnet
Private Sub Command1_Click()
    MsgBox "VSFlexGrid Version " & fg.Version, vbOKOnly
End Sub
```

**Data Type**
Integer

---

VirtualData Property

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 is relevant only when the grid is bound to a recordset.

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 because the data is retrieved from the recordset in small chunks, so the control is never tied up reading large amounts of data that are not immediately needed.

The example below configures binding so that data is retrieved from the data source only when it is needed:

```vbnet
Private Sub Form_Load()
    fg.VirtualData = True
    fg.DataMode = flexDMBound
    fg.DataSource = Data1.Recordset

End Sub
```

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
WallPaper Property

Returns or sets a picture to be used as a background for the control's scrollable area.

Syntax

\[
[\text{form}!]\text{VSFlexGrid.WallPaper} = \text{Picture}
\]

Remarks

The \text{WallPaper} and \text{WallPaperAlignment} properties are used to provide the \text{VSFlexGrid} control with a graphical background such as a texture or a logo.

The picture below shows a grid with wallpaper:

![Grid with wallpaper](image-url)

When creating or selecting pictures for use as wallpaper, consider the following points:

The \text{WallPaper} is a static backdrop. It does not scroll along with the grid contents. Thus, you cannot use it to add graphical elements that are related to grid contents such as boxes around specific cells or range highlights. The wallpaper is applied only to the scrollable areas of the grid. Fixed cells are not affected.

The WallPaper picture is displayed "behind" the grid contents. If you use a picture that has strong colors, the grid contents may be obscured. You may want to use an image editor such as PaintBrush or the Microsoft Image Composer to create "faded" versions of pictures for use as wallpaper. If you want to use dark pictures, set the \text{ForeColor} property to a light value so the grid contents will be clearly visible.

When using wallpaper, remember to set the \text{BackColor} property to a value that matches the predominant color on the wallpaper picture. Even with wallpaper, the grid's \text{BackColor} is still used to paint the small areas beyond the grid, next to the fixed cells, the background of edit controls, and the background of the control itself, if the wallpaper picture is transparent.

You can use transparent pictures such as icons, metafiles, or GIFs as wallpaper. However, rendering transparent GIFs is generally much slower than rendering solid pictures. We recommend using solid, compressed JPG pictures as wallpaper. This yields good results both in terms of rendering speed and disk space.

The code below loads a picture center-aligned into the background of the grid:

```vbnet
Private Sub Form_Load()
```


WallPaperAlignment Property

Returns or sets the alignment of the WallPaper background picture.

Syntax

\[ \text{[form!]}\text{VSFlexGrid.WallPaperAlignment} = \text{PictureAlignmentSettings} \]

Remarks

The WallPaper and WallPaperAlignment properties are used to provide the VSFlexGrid control with a graphical background such as a texture or a logo.

The settings for the WallPaperAlignment property are the same as those used for the CellPicture property. The default setting, flexPicAlignStretch (9), has the advantage of ensuring that the entire picture will be visible, and that no areas of the grid will be left unpainted. However, this setting also causes the slowest repaint. If the WallPaper picture is large, consider using the flexPicAlignLeftTop (0) setting instead. If the WallPaper picture is small, consider using the flexPicAlignTile (10) setting instead.

For details on using grid wallpaper, see the WallPaper property.

Data Type

PictureAlignmentSettings (Enumeration)

Default Value

flexPicAlignStretch (9)
VSDataObject Control

Control Name: VSDataObject
Description: VSFLEX7.OCX
File Name: VSFLEX7.OCX
Help File: VSFLEX7.HLP
GUID: {d76d712a-4a96-11d3-bd95-d296dc2dd072}
Properties: 4
Events: 0
Methods: 2

Before you can use a VSDataObject control in your application, you must add the VSFLEX7.OCX file to your project. If you use the control in most of your VB projects, you may want to add it to VB's Autoload file.

To distribute applications you create with the VSDataObject control, 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.

Clear Method (VSDataObject Control)

Syntax
[form!]VSDataObject.Clear

FileCount Property

Syntax
[form!]VSDataObject.FileCount[ = value As Integer ]

Data Type
Integer

Files Property

Syntax
[form!]VSDataObject.Files(Index As Integer)[ = value As String ]

Data Type
String

GetData Property

Syntax
[form!]VSDataObject.GetData(Format As Integer)[ = value As Variant ]

Data Type
Variant
GetFormat Property

Syntax

\[ \text{[form!]} \text{VSDObject.GetFormat(Format As Integer)}[ = \{ \text{True} \mid \text{False} \} ] \]

Data Type

Boolean

SetData Method

Syntax

\[ \text{[form!]} \text{VSDObject.SetData Value As Variant, [ Format As Variant ]} \]
VSFlexNode Control

Control Name: VSFlexNode
Description: File Name: VSFLEX7.OCX
Help File: VSFLEX7.HLP
GUID: {82f2e221-92e8-11d3-9a1d-f2a67fd05a28}
Properties: 7
Events: 0
Methods: 7

Before you can use a VSFlexNode control in your application, you must add the VSFLEX7.OCX file to your project. If you use the control in most of your VB projects, you may want to add it to VB's Autoload file.

To distribute applications you create with the VSFlexNode control, 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.

AddNode Method

Syntax

\[form!\]VSFlexNode.AddNode AddWhere As NodeTypeSettings, Item As String, \[ Key As Variant \], \[ Image As Variant \]

Remarks
This Method adds a node at a specified position based on the current node (e.g. FirstChild, LastChild).

Children Property

Syntax

\[form!\]VSFlexNode.Children[ = value As Long ]

Remarks
This property returns the number of child nodes.

Data Type
Long

EnsureVisible Method

Syntax

\[form!\]VSFlexNode.EnsureVisible

Remarks
This brings the node into view, scrolling the grid if necessary.
Expanded Property

**Syntax**

\[ \text{[form!]}\text{VSFlexNode.} \text{Expanded} = \{ \text{True} | \text{False} \} \]

**Remarks**
This property returns or sets whether the node is expanded or collapsed.

**Data Type**
Boolean

GetNode Method (VSFlexNode Control)

**Syntax**

\[ \text{[form!]}\text{VSFlexNode.} \text{GetNode Which As NodeTypeSettings} \]

**Remarks**
This method returns a node at a specified position based on the current node (e.g. FirstChild, LastChild).

Image Property

**Syntax**

\[ \text{[form!]}\text{VSFlexNode.} \text{Image} = \text{Picture} \]

**Remarks**
This property returns or sets the node picture (same as CellPicture (r, OutlineCol)).

**Data Type**
Picture

Key Property

**Syntax**

\[ \text{[form!]}\text{VSFlexNode.} \text{Key} = \text{value As String} \]

**Remarks**
This property returns or sets the node key (same as RowData(r), use with grid’s GetNode method).

**Data Type**
String

Level Property

**Syntax**

\[ \text{[form!]}\text{VSFlexNode.} \text{Level} = \text{value As Integer} \]

**Remarks**
This property returns the node’s outline level (higher values are more indented).
Data Type
Integer

Move Method
Syntax
[form!] VSFlexNode.Move MoveTo As NodeMoveSettings
Remarks
This method moves the node up, down, in or out of the tree.
It returns a value of True if successful.

RemoveNode Method
Syntax
[form!] VSFlexNode.RemoveNode
Remarks
This method removes a node and all its child nodes from the grid.

Row Property (VSFlexNode Control)
Syntax
[form!] VSFlexNode.Row[ = value As Long ]
Remarks
This property returns or sets the node's row number in the grid.
Data Type
Long

Select Method (VSFlexNode Control)
Syntax
[form!] VSFlexNode.Select
Remarks
This method selects the node.

Sort Method (VSFlexNode Control)
Syntax
[form!] VSFlexNode.Sort How As SortSettings
Remarks
This method sorts the node's child nodes (does not affect regular rows).
Text Property (VSFlexNode Control)

Syntax

\[ form! ] VSFlexNode.Text[ = value As String ]

Remarks
This property returns or sets the node text (same as TextMatrix(r, OutlineCol)).

Data Type
String
VSFlexString Control

Control Name: VSFlexString
File Name: VSSTR7.OCX
Properties: 17
Events: 0
Methods: 0

Before you can use a VSFlexString control in your application, you must add the VSSTR7.OCX file to your project. In VB, right-click the toolbox and select the VSFlexString Control from the list. In VC++, right-click on the dialog box and select the VSFlexString Control from the list, or use the #import statement to import the VSSTR7.OCX file into the project.

To distribute applications you create with the VSFlexString control, 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.

The VSFlexString control allows you to incorporate regular-expression text matching into your applications. This allows you to parse complex text 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 a regular expression syntax similar to the ones used in Unix systems.

CaseSensitive Property

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.

For example:

```vbnet
Dim fs As New VSFlexString
fs.Text = "This text contains the 'this' word."
fs.Pattern = "This"
fs.CaseSensitive = True
Debug.Print "Case Sensitive: "; fs.MatchCount; " Match(es)"
For i = 0 To fs.MatchCount - 1
    Debug.Print " "; fs.MatchString(i)
Next
fs.CaseSensitive = False
Debug.Print "Case Insensitive: "; fs.MatchCount; " Match(es)"
For i = 0 To fs.MatchCount - 1
    Debug.Print " "; fs.MatchString(i)
Next
```
This code would produce the following output:

Case Sensitive: 1 Match(es)
This
Case Insensitive: 2 Match(es)
This
this

**Data Type**
Boolean

**Default Value**
True

---

**Error Property (VSFlexString Control)**

Returns status information after each pattern-matching operation.

**Syntax**

`val% = [form!].VSFlexString.Error`

**Remarks**

Setting or getting most properties in the VSFlexString control cause it to perform a pattern-matching operation. If an error occurs during this operation, an error code is returned in the **Error** property.

Possible values for the **Error** property are described below:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flexErrNone</code></td>
<td>0</td>
<td>No Error.</td>
</tr>
<tr>
<td><code>flexErrOutOfMemory</code></td>
<td>1</td>
<td>Out of Memory.</td>
</tr>
<tr>
<td><code>flexErrSquareB</code></td>
<td>2</td>
<td>The pattern has unmatched square brackets (e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;[oops]&quot;).</td>
</tr>
<tr>
<td><code>flexErrCurlyB</code></td>
<td>3</td>
<td>The pattern has unmatched square brackets (e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;{oops}&quot;).</td>
</tr>
<tr>
<td><code>flexErrBadPattern</code></td>
<td>4</td>
<td>Invalid pattern for replace operation (e.g. Pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= &quot;: Replace = &quot;Hello&quot;)</td>
</tr>
<tr>
<td><code>flexErrBadTagName</code></td>
<td>5</td>
<td>Tag index out of range (e.g. Pattern =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;([a-z]){[a-z]}&quot;: TagString(2))</td>
</tr>
<tr>
<td><code>flexErrNoMatch</code></td>
<td>6</td>
<td>No match was found (e.g. Text = &quot;Paul&quot;: Pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= &quot;John&quot;)</td>
</tr>
<tr>
<td><code>flexErrInvalidMatchIndex</code></td>
<td>7</td>
<td>Match index out of range (e.g. Text = &quot;Paul&quot;: Pattern</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= &quot;Paul&quot;: ? MatchString(2))</td>
</tr>
</tbody>
</table>

**Data Type**
StringErrorSettings (Enumeration)

---

**MatchCount Property**

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

**Syntax**

`val& = [form!].VSFlexString.MatchCount`
Remarks
Looking for a pattern in a string may result in several matches. The **MatchCount** value is normally used as an upper bound in loops that enumerate the matches. Information about each specific match can be retrieved using the **MatchString**, **MatchStart**, and **MatchLength** properties.

For example:

```vbs
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = "\[qbf\][a-z]*"
Debug.Print "Matches found: "; fs.MatchCount
For i = 0 To fs.MatchCount - 1
    Debug.Print " "; fs.MatchString(i)
Next
```

This code produces the following output:

```
Matches found:  3
  quick
  brown
  fox
```

**Data Type**
Long

---

**MatchIndex Property**

Returns or sets the zero-based index of the current match when there are multiple matches.

**Syntax**

```vbs
[form!]VSFlexString.MatchIndex[ = value As Long ]
```

**Remarks**
Looking for a pattern in a string may result in several matches. Setting **MatchIndex** to a value between zero and **MatchCount** - 1 defines the current match to be used by the **MatchString**, **MatchStart**, and **MatchLength** properties.

For example:

```vbs
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = "\[qbf\][a-z]*"
Debug.Print "Matches found: "; fs.MatchCount
For i = 0 To fs.MatchCount - 1
    fs.MatchIndex = i
    Debug.Print " "; fs.MatchString; fs.MatchLength
Next
```

This code produces the following output:

```
Matches found:  3
  quick 5
  brown 5
  fox 3
```

Note that you can also specify the index as an optional parameter when referring to these properties. Doing so will automatically set the **MatchIndex** property to the new index. For example:
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = "[qbf][a-z]*"
Debug.Print "Matches found: "; fs.MatchCount
For i = 0 To fs.MatchCount - 1
  Debug.Print " "; fs.MatchString(i); fs.MatchIndex
Next

This code produces the following output:

Matches found: 3
quick 0
brown 1
fox 2

**Data Type**
Long

**MatchLength Property**
Returns the length of the current match, in characters.

**Syntax**
`val& = [form!]VSFlexString.MatchLength([ MatchIndex As Long ])`

**Remarks**
Looking for a pattern in a string may result in several matches. You can retrieve information about each match using the MatchLength, MatchStart, and MatchString properties.

The optional parameter MatchIndex should be a number between zero and MatchCount - 1. The default value is the current value of the MatchIndex property.

The MatchStart and MatchLength properties are useful when you need to work on the original string, stored in the Text property. For example, the code below searches for a pattern in a RichEdit control and then underlines each match:

```
fs = rtfEdit.Text
fs.Pattern = txtPattern
For i = 0 To fs.MatchCount - 1
  rtfEdit.SelStart = fs.MatchStart(i)
  rtfEdit.SelLength = fs.MatchLength(i)
  rtfEdit.SelUnderline = True
Next
```

**Data Type**
Long

**MatchStart Property**
Returns the zero-based position of the current match within the Text string.

**Syntax**
`val& = [form!]VSFlexString.MatchStart([ MatchIndex As Long ])`
Remarks
Looking for a pattern in a string may result in several matches. You can retrieve information about each match using the **MatchLength**, **MatchStart**, and **MatchString** properties.

The optional parameter **MatchIndex** should be a number between zero and **MatchCount** - 1. The default value is the current value of the **MatchIndex** property.

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

**Data Type**
Long

**MatchString Property**
Returns or sets the string corresponding to the current match.

**Syntax**
```
[form!]VSFlexString.MatchString([ MatchIndex As Long ])[ = value As String ]
```

**Remarks**
Looking for a pattern in a string may result in several matches. You can retrieve information about each match using the **MatchLength**, **MatchStart**, and **MatchString** properties.

The optional parameter **MatchIndex** should be a number between zero and **MatchCount** - 1. The default value is the current value of the **MatchIndex** property.

If you assign a new value to **MatchString**, the original text -- stored in the **Text** property -- is modified and a new match is attempted automatically.

For example:
```
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = "^[A-Z]*[a-z]+"
For i = 0 To fs.MatchCount - 1
    s = fs.MatchString(i)
    fs.MatchString(i) = UCase(Left(s, 1)) & Mid(s, 2)
Next
Debug.Print fs.Text
```

This code capitalizes the first letter of each word, producing the following output:

The Quick Brown Fox Jumped Over The Lazy Dog.

**Data Type**
String

**Pattern Property**
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:

<table>
<thead>
<tr>
<th>Char</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Beginning of a string.</td>
</tr>
<tr>
<td>$</td>
<td>End of a string.</td>
</tr>
<tr>
<td>.</td>
<td>Any character.</td>
</tr>
<tr>
<td>[list]</td>
<td>Any character in list. For example, &quot;[AEIOU]&quot; matches any single uppercase vowel.</td>
</tr>
<tr>
<td>[^list]</td>
<td>Any character not in list. For example, &quot;[^ ]&quot; matches any character except a space.</td>
</tr>
<tr>
<td>[A-Z]</td>
<td>Any character between 'A' and 'Z'. For example, &quot;[0-9]&quot; matches any single digit.</td>
</tr>
<tr>
<td>?</td>
<td>Repeat previous character zero or one time. For example, &quot;10?&quot; matches &quot;1&quot; and &quot;10&quot;.</td>
</tr>
<tr>
<td>*</td>
<td>Repeat previous character zero or more times. For example, &quot;10*&quot; matches &quot;1&quot;, &quot;10&quot;, &quot;1000&quot;, etc.</td>
</tr>
<tr>
<td>+</td>
<td>Repeat previous character one or more times. For example, &quot;10+&quot; matches &quot;10&quot;, &quot;1000&quot;, etc.</td>
</tr>
<tr>
<td>\</td>
<td>Escape next character. This is required to any of the special characters that are part of the syntax. For example &quot;.<em>+&quot; matches &quot;.</em>+&quot;. It is also required to encode some special non-printable characters (such as tabs) listed below.</td>
</tr>
<tr>
<td>{tag}</td>
<td>Tag this part of the match so you can refer to it later using the TagString property.</td>
</tr>
</tbody>
</table>

In addition to the characters listed above, there are seven special characters encoded using the backslash. These are listed below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>ASCII Code</th>
<th>VB Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Bell (alert)</td>
<td>7</td>
<td>N/A</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>8</td>
<td>N/A</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed</td>
<td>12</td>
<td>N/A</td>
</tr>
<tr>
<td>\n</td>
<td>New line</td>
<td>10</td>
<td>vbLf</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
<td>13</td>
<td>vbCr</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
<td>9</td>
<td>vbTab</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab</td>
<td>11</td>
<td>N/A</td>
</tr>
</tbody>
</table>

For some examples and more details, see the Regular Expressions topic.

Data Type
String

Replace Property
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 string you want to modify.

For example:
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = " [fd][a-z]*"
fs.Replace = " animal"
Debug.Print fs.Text

This code produces the following output:

The quick brown animal jumped over the lazy animal.

The **Replace** property is particularly useful for changing marked up text documents such as HTML or XML. For example, the code below converts bold HTML text into bold underlined text:

```vbs
fs = "<P>This is some <B>HTML</B> text.</P>"
fs.CaseSensitive = False
fs.Pattern = "<B>"
fs.Replace = "<B><U>"
fs.Pattern = "</B>"
fs.Replace = "</U></B>"
Debug.Print fs
```

This code produces the following output:

<P>This is some <B><U>HTML</U></B> text.</P>

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:

```vbs
' 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
```

This code produces the following output:

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

**Data Type**

String

**Soundex Property**

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**
- Rumpelstiltskin, Runpilztiskin, Rumpel: **R514**

Some advantages of this system are the code is short, it will rarely miss a match, and 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
Returns the number of tags found after setting the Pattern, Text, or MatchIndex properties.

Syntax
val$ = [form!]VSFlexString.TagCount

Remarks
Tags are parts of the search pattern delimited with curly braces ("{}"). Using tags allow you to refer to parts of each match. The TagCount value is normally used as an upper bound in loops that enumerate the tags for a specific match. Information about each specific tag can be retrieved using the TagString, TagStart, and TagLength properties.

For example:

```basic
fs.Text = "Mary had a little lamb"
fs.Pattern = "Mary had {.*}" Debug.Print fs.TagCount; "tag: [" & fs.TagString(0) & "]"
```

This code produces the following output:

1 tag: [a little lamb]

For a more detailed example, see the TagString property.

Data Type
Long
TagIndex Property
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
Tags are parts of the search pattern delimited with curly braces ("{}"). Using tags allow you to refer to parts of each match. Setting TagIndex to a value between zero and TagCount - 1 defines the current tag to be used by the TagString, TagStart, and TagLength properties.

Alternatively, you may specify the TagIndex as an index when you read the TagString property.

For an example, see the TagString property.

Data Type
Long

TagLength Property
Returns the length of the current tag, in characters.

Syntax
val& = [form!]VSFlexString.TagLength([ TagIndex As Long ])

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

The optional parameter MatchIndex should be a number between zero and TagCount - 1. The default value is the current value of the TagIndex property.

Data Type
Long

TagStart Property
Returns the position of the current tag within the Text string, starting from zero.

Syntax
val& = [form!]VSFlexString.TagStart([ TagIndex As Long ])

Remarks
You can retrieve information about the current tag by reading the TagLength, TagStart, and TagString properties.
The optional parameter *MatchIndex* should be a number between zero and *TagCount* - 1. The default value is the current value of the *TagIndex* property.

**Data Type**
Long

### TagString Property

Returns or sets the string corresponding to the current tag.

**Syntax**
```
[form!]VSFlexString.TagString([ TagIndex As Long ])[ = value As String ]
```

**Remarks**
Tags are parts of the search pattern delimited with curly braces ("{}"). Using tags allow you to refer to parts of each match. You can retrieve information about each tag by reading the *TagLength*, *TagStart*, and *TagString* properties.

The optional parameter *MatchIndex* should be a number between zero and *TagCount* - 1. The default value is the current value of the *TagIndex* property.

The following example shows how you can use tags to extract additional information from each match:

```vbscript
fs.Text = "Mary had a little lamb, Joe has a Porsche, " & vbCrLf & _
  "Matt has problems..., Mike will have many options, " & vbCrLf & _
  "Bill had better stop smoking, And I have a headache!"
fs.Pattern = "
  ([A-Z][a-z]*) ha[ds]{0,1}ve? [ a]*{[A-Za-z ]+} "' who had/has/have stuff?
For i = 0 To fs.MatchCount - 1
  fs.MatchIndex = i
  Debug.Print fs.TagString(1) & " BELONGS TO " & fs.TagString(0)
Next
```

This code produces the following output:

little lamb BELONGS TO Mary
Porsche BELONGS TO Joe
problems BELONGS TO Matt
better stop smoking BELONGS TO Bill
headache BELONGS TO I

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. For example:

```vbscript
fs.Text = "The quick brown fox jumped over the lazy dog."
fs.Pattern = "([A-Za-z])\{a-z\}*"
For i = 0 To fs.MatchCount - 1
  fs.MatchIndex = i
  fs.TagString(0) = UCase(fs.TagString(0))
Next
Debug.Print fs
```

This code produces the following output:
The Quick Brown Fox Jumped Over The Lazy Dog.

**Data Type**
String

**Text Property (VSFlexString Control)**
Returns or sets the text to be scanned searching for the Pattern string.

**Syntax**

```
[form!] VSFlexString.Text[ = value As String ]
```

**Remarks**

This is the **VSFlexString**'s default property.

Whenever a new string is assigned to the **Text** or **Pattern** properties, **VSFlexString** will scan the text looking for parts that fit the pattern.

To find out how many matches were found, read the **MatchCount** property. To retrieve information about each match, read the **MatchLength**, **MatchStart**, and **MatchString** properties.

If you assign new values to the **MatchString** property, the changes will be reflected in the contents of the **Text** property.

For details on how to build patterns, see the **Pattern** property.

**Data Type**
String

**Version Property (VSFlexString Control)**
Returns the version of the control 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 7.00 returns 700.

**Data Type**
Integer

**Default Value**
700
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 VSFLEX6 to VSFlexGrid Pro 7.0?

Use CONVERT, the conversion utility provided with VSFlexGrid Pro 7.0. The CONVERT utility also allows you to convert between the ADO/RDO and OLEDB/ADO versions of the grid.

The CONVERT utility is written in Visual Basic and is supplied in source code format, so you may modify it if you need to.

What is difference between VSFLEX7.OCX, VSFLEX7D.OCX, and VSFLEX7L.OCX?

The VSFlexGrid Pro 7.0 package includes three versions of the grid:

VSFLEX7.OCX This version supports OLEDB/ADO data-binding. You may bind the control to any ADO data source, including the ADO data control that ships with VB6.

VSFLEX7D.OCX This version supports DAO/RDO data-binding. You may bind the control to the traditional data sources (built-in DAO data control, RDO data control).

VSFLEX7L.OCX This version has no data-binding support in the traditional sense (you can still bind the control to arrays or use the FlexDataSource property).

The controls included in each file are functionally identical, but have different identifiers (GUIDs and class names). 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 7.0, you must decide which version to use. The following information will help you make the decision:

1) If you are planning to use the grid to display and edit information coming from OLEDB data sources, use VSFLEX7.OCX. This version requires ADO to be installed on the computer.
2) If you are planning to use the grid to display and edit information coming from DAO data sources (such as the built-in data control), use VSFLEX7D.OCX.
3) If you are using the VSFlexGrid in unbound mode (i.e., not bound to any databases), use VSFLEX7L.OCX.

Whichever version you decide to use, you may easily switch later using the CONVERT utility supplied with VSFlexGrid Pro 7.0.

Does VSFlexGrid 7.0 work with VB4-16 or any other 16-bit environments?

Sorry, it does not. VSFlexGrid Pro 7.0 is a 32-bit-only product. Please contact VideoSoft’s customer service department if you require a 16-bit version of VSFLEX.

When adding VSFLEX7.OCX to my VB4 or VB5 project, I get the following error message: "Error loading DLL". What is wrong?

VSFLEX7.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 **VSFLEX7.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 **VSFLEX7L.OCX** version of the control, which is not subject to this limitation.

**Does VSFlexGrid 7.0 work with VB4, VB5 and VB6?**

**V SFlexGrid Pro 7.0** 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:

```vb
[v =] fg.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.

**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** property, and the slowest is using the **AddItem** method.

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 restore its value 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.

```vb
Private Sub Command1_Click(Index As Integer)
    With fg
        ' 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?

Use the PrintGrid method.

To add advanced features such as print preview, or include one or more grids into a single document, consider 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. To omit optional parameters, use Variants of type VT_ERROR. For example:

```cpp
VARIANT v;
V_VT(&v) = VT_ERROR;
fg.AddItem("hello\my friend", v);
```

Note that using ActiveX controls in Visual C++ is a little different, depending on whether you are using MFC or not. The wrapper classes generated by the MFC Class Wizard require you to pass optional parameters as illustrated above. The wrapper classes generated by the #import statement supply default values for optional parameters, so you may simply omit them.

For more details and tips on using the VSFlexGrid control in Visual C++, see the Using VSFlexGrid in Visual C++ topic in the documentation.

How do I handle Pictures in VSFlexGrid when using C++?

If you are using MFC, the best way is to use MFC's CPictureHolder class. Here's an example that shows how you can set the VSFlexGrid's CellPicture property (or any other ActiveX
1) Using the AppWizard, generate a new project with as a dialog-based app with the OLE controls option set to True.

2) Add a VSFlexGrid control to the form and connect it to the m_flex member variable.

3) Add a bitmap resource and set its ID to IDB_ARROWPIC.

4) 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());
   }

If you are not using MFC, refer to the Using VSFlexGrid in Visual C++ topic in the documentation. It shows how you can use pictures without MFC support and much more.
# INDEX

<table>
<thead>
<tr>
<th>A</th>
<th>BindTo陣列 Method</th>
<th>BookMark Property</th>
<th>BottomRow Property</th>
<th>BuildComboList Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddItem Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddNode Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterCollapse Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterDataRefresh Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterEdit Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterMoveColumn Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterMoveRow Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterScroll Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterSelChange Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterSort Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterUserFreeze Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterUserResize Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AfterScroll Event</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregate Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllowBigSelection Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllowSelection Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllowUserFreezing</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllowUserResizing Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appearance* Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Archive Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ArchiveInfo Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoResize Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSearch Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSearchDelay Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSize Method</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSizeMode Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSizeMode Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoSizeModeMouse Property</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| B                  |                  |                  |                     |                       |
| BackColorAlternate Property |             |                   |                     |                       |
| BackColorBkg Property |                   |                   |                     |                       |
| BackColorFixed Property |                   |                   |                     |                       |
| BackColorFrozen Property |                 |                   |                     |                       |
| BackColorSel Property |                   |                   |                     |                       |
| Basic Operations     |                   |                   |                     |                       |
| BeforeCollapse Event |                   |                   |                     |                       |
| BeforeDataRefresh Event |               |                   |                     |                       |
| BeforeEdit Event     |                   |                   |                     |                       |
| BeforeMouseDown Event |                   |                   |                     |                       |
| BeforeMoveColumn Event|                   |                   |                     |                       |
| BeforeMoveRow Event  |                   |                   |                     |                       |
| BeforePageBreak Event |                   |                   |                     |                       |
| BeforeRowColChange Event |             |                   |                     |                       |
| BeforeScroll Event   |                   |                   |                     |                       |
| BeforeScrollTip Event |                   |                   |                     |                       |
| BeforeSelChange Event |                   |                   |                     |                       |
| BeforeSort Event     |                   |                   |                     |                       |
| BeforeUserResize Event |                |                   |                     |                       |

| C                  |                  |                  |                     |                       |
| Calculator Demo     |                   |                   |                     |                       |
| CaseSensitive Property |                 |                   |                     |                       |
| Cell Flooding Demo  |                   |                   |                     |                       |
| Cell Property       |                   |                   |                     |                       |
| CellAlignment Property |               |                   |                     |                       |
| CellBackColor Property |                 |                   |                     |                       |
| CellBorder Method   |                   |                   |                     |                       |
| CellButtonClick Event |                 |                   |                     |                       |
| CellButtonPicture Property |           |                   |                     |                       |
| CellChanged Event   |                   |                   |                     |                       |
| CellChecked Property |                   |                   |                     |                       |
| CellFontColor Property |               |                   |                     |                       |
| CellFontPercent Property |             |                   |                     |                       |
| CellFontBold Property |                 |                   |                     |                       |
| CellFontItalic Property |              |                   |                     |                       |
| CellFontName Property |                |                   |                     |                       |
| CellFontSize Property |                 |                   |                     |                       |
| CellFontStrikethru Property |         |                   |                     |                       |
| ChangeEdit Event    |                   |                   |                     |                       |
| Changes Property     |                   |                   |                     |                       |
| Children Property   |                   |                   |                     |                       |
| Clear Method (VSDataObject Control) |         |                   |                     |                       |
| Clear Method (VSFlexGrid Control) |       |                   |                     |                       |
| ClientHeight Property |               |                   |                     |                       |
| ClientWidth Property |                   |                   |                     |                       |
| Clip Property       |                   |                   |                     |                       |
| ClipSeparators Property |             |                   |                     |                       |
| Col Property        |                   |                   |                     |                       |
| ColAlignment Property |                 |                   |                     |                       |
| ColComboList Property |                |                   |                     |                       |
| ColData Property     |                   |                   |                     |                       |
| ColDataType Property |                   |                   |                     |                       |
| ColEditMask Property |                   |                   |                     |                       |
| ColFormat Property   |                   |                   |                     |                       |
| ColHidden Property   |                   |                   |                     |                       |

245
<p>| K | Key Property .................................. 227 |
|   | KeyDownEdit Event ................................ 171 |
|   | KeyPressEdit Event ................................ 172 |
|   | KeyUpEdit Event ................................ 173 |
| L | LeaveCell Event .................................. 173 |
|   | LeftCol Property .................................. 173 |
|   | Level Property ..................................... 227 |
|   | LoadArray Method .................................. 174 |
|   | LoadGrid Method ................................... 174 |
| M | MatchCount Property .............................. 231 |
|   | MatchIndex Property ................................ 232 |
|   | Matching Demo ...................................... 72 |
|   | MatchLength Property .............................. 233 |
|   | MatchStart Property ................................ 233 |
|   | MatchString Property ............................... 234 |
|   | MergeCells Property ................................ 175 |
|   | MergeCol Property .................................. 177 |
|   | MergeCompare Property ............................. 178 |
|   | MergeRow Property .................................. 178 |
|   | Merging Cells ...................................... 18 |
|   | MouseCol Property .................................. 179 |
|   | MouseRow Property .................................. 179 |
|   | Move Method ....................................... 228 |
|   | MultiTotals Property ................................ 180 |
| N | New Features in VSFlexGrid Pro 7.0.5 ............ 58 |
|   | NodeClosedPicture Property .................... 181 |
|   | NodeOpenPicture Property ....................... 182 |
| O | OLE Drag and Drop Demo ............................ 58 |
|   | OLECompleteDrag Event ............................ 182 |
|   | OLEDrag Method .................................... 183 |
|   | OLEDragDrop Event .................................. 183 |
|   | OLEDragMode Property ................................ 184 |
|   | OLEDragOver Event .................................. 184 |
|   | OLEDropMode Property .............................. 185 |
|   | OLEGiveFeedback Event ............................ 186 |
|   | OLESetData Event ................................... 186 |
|   | OLEStartDrag Event ................................ 187 |
|   | Other types of Data Binding ...................... 24 |
|   | Outline Demo ....................................... 43 |
|   | Outline Method ..................................... 188 |
|   | OutlineBar Property ................................ 188 |
|   | OutlineCol Property ............................... 189 |
|   | Outlining and Summarizing ....................... 15 |
|   | Overview .......................................... 2 |
|   | OwnerDraw Property ................................ 190 |
|   | Pattern Property ................................... 234 |
|   | PicturesOver Property ............................. 190 |
|   | PictureType Property ................................ 191 |
|   | PrintGrid Method ................................... 191 |
|   | Printing Demo ...................................... 56 |
|   | Product Support .................................... 8 |
| P | Redraw Property .................................... 193 |
|   | Regular Expressions ............................... 70 |
|   | RemoveItem Method ................................... 193 |
|   | RemoveNode Method ................................... 228 |
|   | Replace Property ................................... 235 |
|   | Replacing Demo ..................................... 73 |
|   | RightCol Property .................................. 194 |
|   | Row Property (V SFlexGrid Control) ............ 194 |
|   | Row Property (V SFlexNode Control) .......... 228 |
|   | RowColChange Event ................................ 195 |
|   | RowData Property ................................... 195 |
|   | RowHeight Property ................................ 196 |
|   | RowHeightMax Property ............................ 196 |
|   | RowHeightMin Property ............................. 196 |
|   | RowHidden Property ................................ 197 |
|   | RowIsVisible Property ............................. 197 |
|   | RowOutlineLevel Property ....................... 198 |
|   | RowPos Property .................................... 198 |
|   | RowPosition Property .............................. 198 |
|   | Rows Property ...................................... 199 |
|   | RowSel Property .................................... 199 |
|   | RowStatus Property ................................ 200 |
|   | SaveGrid Method .................................... 200 |
|   | Saving, Loading, and Printing ................... 21 |
|   | ScrollBars Property ................................ 201 |
|   | ScrollTips Property ................................ 202 |
|   | ScrollTipText Property ............................ 203 |
|   | ScrollTrack Property .............................. 203 |
|   | Select Change Event .............................. 203 |
|   | Select Method (VS FlexGrid Control) .......... 204 |
|   | Select Method (VS FlexNode Control) .......... 228 |
| S | Outline Property ................................... 234 |
|   | OutlineCol Property ............................... 190 |
|   | Outlining and Summarizing ....................... 15 |
|   | Overview .......................................... 2 |
|   | OwnerDraw Property ................................ 190 |
|   | Pattern Property ................................... 234 |
|   | PicturesOver Property ............................. 190 |
|   | PictureType Property ................................ 191 |
|   | PrintGrid Method ................................... 191 |
|   | Printing Demo ...................................... 56 |
|   | Product Support .................................... 8 |</p>
<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectedRow Property</td>
<td>204</td>
</tr>
<tr>
<td>SelectedRows Property</td>
<td>205</td>
</tr>
<tr>
<td>SelectionMode Property</td>
<td>205</td>
</tr>
<tr>
<td>SetData Method</td>
<td>225</td>
</tr>
<tr>
<td>SheetBorder Property</td>
<td>206</td>
</tr>
<tr>
<td>ShowComboBoxButton Property</td>
<td>207</td>
</tr>
<tr>
<td>Sort Method (VSFlexNode Control)</td>
<td>228</td>
</tr>
<tr>
<td>Sort Property (VSFlexGrid Control)</td>
<td>207</td>
</tr>
<tr>
<td>Soundex Property</td>
<td>236</td>
</tr>
<tr>
<td>StartEdit Event</td>
<td>210</td>
</tr>
<tr>
<td>StartPage Event</td>
<td>211</td>
</tr>
<tr>
<td>Subtotal Method</td>
<td>212</td>
</tr>
<tr>
<td>SubtotalPosition Property</td>
<td>214</td>
</tr>
<tr>
<td>TabBehavior Property</td>
<td>215</td>
</tr>
<tr>
<td>TagCount Property</td>
<td>237</td>
</tr>
<tr>
<td>TagIndex Property</td>
<td>238</td>
</tr>
<tr>
<td>TagLength Property</td>
<td>238</td>
</tr>
<tr>
<td>TagStart Property</td>
<td>238</td>
</tr>
<tr>
<td>TagString Property</td>
<td>239</td>
</tr>
<tr>
<td>Text Property (VSFlexGrid Control)</td>
<td>216</td>
</tr>
<tr>
<td>Text Property (VSFlexNode Control)</td>
<td>229</td>
</tr>
<tr>
<td>Text Property (VSFlexString Control)</td>
<td>240</td>
</tr>
<tr>
<td>TextArray Property</td>
<td>216</td>
</tr>
<tr>
<td>TextMatrix Property</td>
<td>217</td>
</tr>
<tr>
<td>TextStyle Property</td>
<td>217</td>
</tr>
<tr>
<td>TextStyleFixed Property</td>
<td>218</td>
</tr>
<tr>
<td>ToolTip Demo</td>
<td>55</td>
</tr>
<tr>
<td>TopRow Property</td>
<td>218</td>
</tr>
<tr>
<td>TreeColor Property</td>
<td>219</td>
</tr>
<tr>
<td>Upgrading From Previous Versions</td>
<td>4</td>
</tr>
<tr>
<td>Using VSFlexGrid in Visual C++</td>
<td>26</td>
</tr>
<tr>
<td>Using VSFlexGrid in Visual J++</td>
<td>32</td>
</tr>
<tr>
<td>ValidateEdit Event</td>
<td>219</td>
</tr>
<tr>
<td>Value Property</td>
<td>219</td>
</tr>
<tr>
<td>ValueMatrix Property</td>
<td>220</td>
</tr>
<tr>
<td>Version Property (VSFlexGrid Control)</td>
<td>220</td>
</tr>
<tr>
<td>Version Property (VSFlexString Control)</td>
<td>240</td>
</tr>
<tr>
<td>VirtualData Property</td>
<td>221</td>
</tr>
<tr>
<td>Visual C++ MFC Demo</td>
<td>62</td>
</tr>
<tr>
<td>VSDataObject Control</td>
<td>224</td>
</tr>
<tr>
<td>VSFlexGrid Control</td>
<td>79</td>
</tr>
<tr>
<td>VSFlexGrid Introduction</td>
<td>9</td>
</tr>
<tr>
<td>VSFlexGrid Property Groups</td>
<td>35</td>
</tr>
<tr>
<td>VSFlexGrid Property</td>
<td>37</td>
</tr>
<tr>
<td>VSFlexNode Control</td>
<td>226</td>
</tr>
<tr>
<td>VSFlexString Control</td>
<td>230</td>
</tr>
<tr>
<td>VSFlexString Introduction</td>
<td>68</td>
</tr>
<tr>
<td>Wallpaper Property</td>
<td>222</td>
</tr>
<tr>
<td>WallpaperAlignment Property</td>
<td>223</td>
</tr>
</tbody>
</table>