
ComponentOne
VSFlexGrid® Pro 8.0
The Agile Grid

Corporate Headquarters
ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com
Web site: http://www.componentone.com

Sales

E-mail: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Technical Support
See *Technical Support* in this manual for information on obtaining technical support.

Trademarks
ComponentOne VSFlexGrid Pro 8.0 and the ComponentOne VSFlexGrid Pro 8.0 logo are trademarks, and ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

Warranty
ComponentOne warrants that the original CD (or diskettes) is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective CD (or disk) to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for a defective CD (or disk) by sending it and a check for $25 (to cover postage and handling) to ComponentOne.
Except for the express warranty of the original CD (or disks) set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. We are not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution
While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only. Please read *End User License Agreement for ComponentOne Software* and *Redistributable Files* sections in this manual before copying and redistributing any ComponentOne VSFlexGrid Pro 8.0 files.

Table of Contents

Overview

Welcome to **ComponentOne VSFlexGrid® Pro 8.0**.

VSFlexGrid 8.0 includes **VSFlexGrid**, a full-featured grid control and **VSFlexString**, a powerful regular expression engine. If you like **VSFlexGrid 8.0**, you can check out our other products by visiting our web site at http://www.componentone.com.

VSFlexGrid 8.0 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.

ComponentOne has a user-friendly distribution policy. We want every programmer to obtain a copy of **VSFlexGrid 8.0** to try for as long as they wish. Those who like the product and find it useful may buy a license for a reasonable price. The only restriction is that unlicensed copies of **VSFlexGrid 8.0** will display a ComponentOne banner every time they are loaded to remind developers to license the product.

We are confident that you will like ComponentOne **VSFlexGrid 8.0**. If you have any suggestions or ideas for new features that you'd like to see included in a future version, or ideas for new controls, please call us or write:

Corporate Headquarters

ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 USA
412.681.4343
412.681.4384 (Fax)

http://www.componentone.com

VSFlexGrid Controls

The **ComponentOne VSFlexGrid 8.0** 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.

What's New in VSFlexGrid 8.0 for Active X

This documentation was last revised on November 14, 2005.

Feature Overview

VSFlexGrid 8.0 now allows resizing of columns that are too wide to fit the control.

Export to Excel now supports the creation of shared string tables, which can save cells with up to 32k of text (the previous limit was 256 characters when saving). SST also reduces the size of the .xls files.

Installing VSFlexGrid 8.0 for Active X

The following sections provide helpful information on installing **VSFlexGrid 8.0**.

SetUp Files

To install **ComponentOne VSFlexGrid 8.0**, 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 ComponentOne products for which you have purchased a registration key at this time as well.

Note: Always use the **SETUP.EXE** utility to install **VSFlexGrid 8.0** 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 ComponentOne banner when you run them.

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

File	Description
VSFLEX8.HLP	This file contains the VSFlexGrid 8.0 online Help topics.
VSFLEX8.CNT	This file contains the VSFlexGrid 8.0 online Help contents.

The following files will be installed into your WINDOWS\SYSTEM directory or, if you use Windows NT, into your WinNT\System and WinNT\System32 directories:

File	Description
VSFLEX8.OCX	This file contains the VSFlexGrid 8.0 control with ADO/OLEDB data-binding.
VSFLEX8D.OCX	This file contains the VSFlexGrid 8.0 control with DAO data-binding.
VSFLEX8L.OCX	This file contains the VSFlexGrid 8.0 control with no data-binding support.
VSFLEX8U.OCX	This file contains a Unicode version of the VSFlexGrid 8.0 control with ADO/OLEDB data-binding.
VSSTR8.OCX	This file contains the VSFlexString 8.0 control (This control used to be part of the VSFLEX6.OCX file).
VSFLEX8N.OCX	This file contains a Unicode version of the VSFlexGrid Light control.
VSPPG8.DLL	This is the property pages file for use with the VSFlexGrid at design time in the Visual Basic IDE.

The following folders will be created by the setup utility:

Folder	Description
ComponentOne Studio	Main ComponentOne folder to store ComponentOne control Information.
ComponentOneStudio\VS FLEXGrid Pro 8	Contains sample Visual Basic projects, utilities, And the README.TXT file which discusses Version specific information.

Installing Demonstration Versions

If you wish to try **VSFlexGrid 8.0** or any of our other products, and do not have a registration key, follow the steps through the installation wizard and use the default serial number.

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 ComponentOne banner will not appear when your users run the applications.

Uninstalling VSFlexGrid 8.0

To uninstall **ComponentOne VSFlexGrid 8.0**, use the **UNSETUP.EXE** utility provided on the installation CD or diskettes. The **UNSETUP.EXE** utility will remove all **VSFLEX8** files from your \Windows\System directory.

Upgrading From Previous Versions

Visual Basic projects that use **VSFlexGrid 7.0** may be upgraded to **VSFlexGrid 8.0** 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 (for example, "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 VSDataObject. 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 **AfterScoll** events.

END-USER LICENSE AGREEMENT FOR COMPONENTONE SOFTWARE

IMPORTANT-READ CAREFULLY: This End User License Agreement (this "**EULA**") contains the terms and conditions that govern your use of the SOFTWARE (as defined below) and imposes material limitations to your rights. You should read this EULA carefully and treat it as valuable property.

I. THIS EULA.

1. **Software Covered by this EULA.** This EULA governs your use of the ComponentOne, LLC ("**C1**") software product(s) enclosed or otherwise accompanied herewith (individually and collectively, the "**SOFTWARE**"). The term "SOFTWARE" includes, to the extent provided by C1: 1) any revisions, updates and/or upgrades thereto; 2) any data, image or executable files, databases, data engines, computer software, or similar items customarily used or distributed with computer software products; 3) anything in any form whatsoever intended to be used with or in conjunction with the SOFTWARE; and 4) any associated media, documentation (including physical, electronic and online) and printed materials (the "**Documentation**").

2. **This EULA is a Legally Binding Agreement Between You and C1.** If you are acting as an agent of a company or another legal person, such as an officer or other employee acting for your employer, then "you" and "your" mean your principal, the entity or other legal person for whom you are acting. However, importantly, even if you are acting as an agent for another, you may still be personally liable for violation of federal and State laws, such as copyright infringement.

   By signifying your acceptance of the terms of this EULA, you intend to be, and hereby are, legally bound to this EULA to the same extent as if C1 and you physically signed this EULA. By installing, copying, or otherwise using the SOFTWARE, you agree to be bound by all the terms and conditions of this EULA. If you do not agree to all of such terms and conditions, **you may not install or use the SOFTWARE**. If you do not agree with any of the terms herewith and, for whatever reason, installation has begun or has been completed, you should cancel installation or un-install the SOFTWARE, as the case may be. Furthermore, you should promptly return the SOFTWARE to the place of business from which you obtained it in accordance with any return policies of such place of business. Return policies may vary among resellers; therefore you must comply with the return policies of your supplier as you agreed at the point of purchase. If the place of business from which you purchased the SOFTWARE does not honor a full refund for a period of thirty (30) days from the date of purchase, you may then return the SOFTWARE directly to C1 for a refund provided that such returns is authorized within the same thirty (30) days time period. To return the product directly to C1, you must first obtain a Return Authorization Number by contacting C1, and you must forward to C1 all items purchased, including the proof of purchase. The return must be postage-prepaid, and post-marked within thirty (30) days from the proof of purchase, time being of the essence. The return option to C1 is only available to the original purchaser of an unopened factory packaged item.

II. YOUR LICENSE TO DEVELOP AND TO DISTRIBUTE.

As provided in more detail below, this EULA grants you two licenses: 1) a license to use the SOFTWARE to develop other software products (the "**Development License**"); and 2) a license to use and/or distribute the Developed Software (the "**Distribution License**"). These licenses (individually and collectively, the "**Licenses**") are explained and defined in more detail below.

1. **Definitions.** The following terms have the respective meanings as used in this EULA:

   "**Network Server**" means a computer with one or more computer central processing units (CPU's) that operates for the purpose of serving other computers logically or physically connected to it, including, but not limited to, other computers connected to it on an internal network, intranet or the Internet.

   "**Web Server**" means a type of Network Server that serves other computers which, are specifically connected to it through either an intranet or the Internet.

   "**Developed Software**" means those computer software products that are developed by or through the use of the SOFTWARE.

   "**Developed Web Server Software**" means those Developed Software products that reside logically or physically on at least one Web Server and are operated (meaning the computer software instruction set is carried out) by the Web Server's central processing unit(s) (CPU).

"**Redistributable Files**" means the SOFTWARE files or other portions of the SOFTWARE that are provided by C1 and are identified as such in the Documentation for distribution by you with the Developed Software.

"**Developer**" means a human being or any other automated device using the SOFTWARE in accordance with the terms and conditions of this EULA.

"**Developer Seat License**" means that each Developer using or otherwise accessing the programmatic interface or the SOFTWARE must obtain the right to do so by purchasing a separate End User License.

"**Source Code**" shall mean computer software code or programs in human readable format, such as a printed listing of such a program written in a high-level computer language. The term "**Source Code**" includes, but is not limited to, documents and materials in support of the development effort of the SOFTWARE, such as flow charts, pseudo code and program notes.

2. **Your Development License.** You are hereby granted a limited, royalty-free, non-exclusive right to use the SOFTWARE to design, develop, and test Developed Software, on the express condition that, and only for so long as, you fully comply with all terms and conditions of this EULA.

   The SOFTWARE is licensed to you on a Developer Seat License basis.

   Developer Seat License basis means that you may perform an installation of the SOFTWARE for use in designing, testing and creating Developed Software by a single Developer on one or more computers, each with a single set of input devices, so long as 1) such computer/computers is/are used only by one single Developer at any given time and not concurrently and, 2) the user is the primary User to whom the license has been granted. Conversely, you may not install or use the SOFTWARE on a computer that is a network server or a computer at which the SOFTWARE is used by more than one Developer. You may not network the SOFTWARE or any component part of it, where it is or may be used by more than one Developer unless you purchase an additional Development License for each Developer. You must purchase another separate license to the SOFTWARE in order to add additional developer seats, whether the additional developers are accessing the SOFTWARE in a stand-alone environment or on a computer network.

   The license rights granted under this Agreement may be limited to a specified number of days after you first install the SOFTWARE unless you supply information required to license or verify your licensed copy, as the case may be, within the time and the manner described during the SOFTWARE setup sequence and/or in the dialog boxes appearing during use of the SOFTWARE. You may need to verify the SOFTWARE through the use of the Internet, email or telephone; toll charges may apply. You may need to re-verify the SOFTWARE if you modify your computer hardware. Product verification is based on the exchange of information between your computer and C1. None of this information contains personally identifiable information nor can they be used to identify any personal information about you or any information you store in your computer. YOU ACKNOWLEDGE AND UNDERSTAND THAT THERE ARE TECHNOLOGICAL MEASURES IN THE SOFTWARE THAT ARE DESIGNED TO PREVENT UNLICENSED OR ILLEGAL USE OF THE SOFTWARE. YOU AGREE THAT C1 MAY USE SUCH MEASURES AND YOU AGREE TO FOLLOW ANY REQUIREMENTS REGARDING SUCH TECHNOLOGICAL MEASURES. YOU ACKNOWLEDGE AND AGREE THAT THE SOFTWARE WILL CEASE TO FUNCTION UNLESS AND UNTIL YOU VERIFY THE APPLICABLE SOFTWARE SERIAL KEY.

   You agree that C1 may audit your use of the SOFTWARE for compliance with these terms at any time, upon reasonable notice. In the event that such audit reveals any use of the SOFTWARE other than in full compliance with the terms of this EULA, you shall reimburse C1 for all reasonable expenses related to such audit in addition to any other liabilities you may incur as a result of such non-compliance.

   In all cases, (a) you may not use C1's name, logo, or trademarks to market your Developed Software without the express written consent of C1; (b) you must include the following C1 copyright notice in

your Developed Software documentation and/or in the "About Box" of your Developed Software, and wherever the copyright/rights notice is located in the Developed Software ("Portions Copyright © ComponentOne, LLC 1991-2005. All Rights Reserved."); (c) you agree to indemnify, hold harmless, and defend C1, its suppliers and resellers, from and against any claims or lawsuits, including attorney's fees that may arise from the use or distribution of your Developed Software; (d) you may use the SOFTWARE only to create Developed Software that is significantly different than the SOFTWARE.

3. **Your Distribution License.**

License to Distribute Developed Software. Subject to the terms and conditions in this EULA, you are granted the license to use and to distribute Developed Software on a royalty-free basis, provided that the Developed Software incorporates the SOFTWARE as an integral part of the Developed Software in machine-language compiled format (customarily an ".exe", or ".dll", etc.). You may not distribute, bundle, wrap or subclass the SOFTWARE as Developed Software which, when used in a "designtime" development environment, exposes the programmatic interface of the SOFTWARE. You may distribute, on a royalty-free basis, Redistributable Files with Developed Software only. You may not add or transfer the SOFTWARE license key to the computer where the Developed Software is installed. Users of the Developed Software may not use the SOFTWARE or the Redistributable Files, directly or indirectly, for development purposes. In particular, if you create a control (or user control) using the SOFTWARE as a constituent control, you are not licensed to distribute the control you created with the SOFTWARE to users for development purposes.

4. **Specific Product Limitations.** Notwithstanding anything in this EULA to the contrary, if the license you have purchased is for any of the following products, then the following additional limitations will apply:

a. ComponentOne Reports for .NET Designer Edition. ComponentOne Reports for .NET Designer Edition includes at least: 1) one dynamic link library file (c1.win.c1reportdesigner.dll) known as C1ReportDesigner Component, 2) one executable file (ReportDesigner.exe) known as C1ReportDesigner Application and, 3) the Source Code of the C1ReportDesigner Application. The C1ReportDesigner Component is subject to the general terms and restrictions set forth in this EULA. The C1ReportDesigner Application is an executable file used to design and prepare reports; the C1ReportDesigner Application may be distributed, free of royalties, only in conjunction with the Developed Software.

C1 hereby also grants you the right to use and to modify the C1ReportDesigner Application Source Code to create derivative works that are based on the licensed Source Code. You may distribute such derivative works, solely in object code format and exclusively in conjunction with and/or as a part of the Developed Software. You **are expressly not granted** the right to distribute, disclose or otherwise make available to any third party the licensed Source Code, any portion, modified version or derivative work thereof, in source code format.

C1 shall retain all right, title and interest in and to the licensed Source Code, and all C1 updates, modifications or enhancements thereof. Nothing herein shall be deemed to transfer any ownership or title rights in and to the licensed Source Code from C1 to you.

SOURCE CODE IS LICENSED TO YOU AS IS. C1 DOES NOT AND SHALL NOT PROVIDE YOU WITH ANY TECHNICAL SUPPORT FOR YOUR SOURCE CODE LICENSE.

b. VSView Reporting Edition (ActiveX). VSView Reporting Edition includes at least one executable file listed as "VSRptX.exe" (where X indicates the version number i.e.7,8, etc.), known as "**Designer**." The file "VSRptX.exe", or any upgrade or future versions of the Designer, are subject to the restrictions set forth in this EULA and may not be distributed with your Developed Software or in any other way.

c. Studio Products. You may not share the component parts of the Studio Products licensed to you with other Developers, nor may you allow the use and/or installation of such components by other Developers.

5. **Updates/Upgrades; Studio Subscription.** Subject to the terms and conditions of this EULA, the Licenses are perpetual. Updates and upgrades to the SOFTWARE may be provided by C1 from time-to-time, and, if so provided by C1, are provided upon the terms and conditions offered at that time by C1 in its sole discretion. C1 may provide updates and upgrades to the SOFTWARE for free or for any charge, at any time or never, and through its chosen manner of access and distribution, all in C1's sole discretion.

   C1 licenses certain of its separately-licensed products bundled together in a product suite, called the C1 "Studio" product line (the "**Studio Products**"). The exact separately-licensed products that are bundled into the Studio Products may change from time-to-time in C1's sole discretion. If the SOFTWARE is identified as a C1 "Studio" product, then the SOFTWARE is one of the Studio Products. The SOFTWARE and the Studio Products are revised from time-to-time (meaning, for example, revised with updates, upgrades and, in the case of Studio products, some times changes to the mix of products included in the bundle). To receive any such revisions to the SOFTWARE or the Studio Products, as the case may be, you must have a valid SOFTWARE license or a valid Studio subscription. Together with the Licenses, the original purchaser is granted a one-year subscription from the date of purchase. Upon expiration, you must renew your license subscription to continue to be entitled to receive SOFTWARE and/or the Studio Products revisions as the case may be.

6. **Serial Number.** With your license, you will be issued a unique serial number (the "**Serial Number**") used for the activation of the SOFTWARE. The Serial Number is subject to the restrictions set forth in this EULA and may not be disclosed or distributed either with your Developed Software or in any other way. The disclosure or distribution of the Serial Number constitutes a breach of this EULA, the effect of which shall be the immediate termination and revocation of all the rights granted herein.

7. **Evaluation Copy.** If you are using an "evaluation copy", specifically designated as such by C1 on its website or elsewhere, then the Licenses are limited as follows: a) you are granted a license to use the SOFTWARE for a period of thirty (30) days counted from the day of installation (the "**Evaluation Period**"); b) upon completion of the Evaluation Period, you shall either i) delete the SOFTWARE from the computer containing the installation, or you may ii) obtain a paid license of the SOFTWARE from C1 or any of its resellers; and c) any Developed Software developed with the Evaluation Copy may not be distributed or used for any commercial purpose.

III. INTELLECTUAL PROPERTY.

1. **Copyright.** You agree that all right, title, and interest in and to the SOFTWARE (including, but not limited to, any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE), and any copies of the SOFTWARE, and any copyrights and other intellectual properties therein or related thereto are owned exclusively by C1, except to the limited extent that C1 may be the rightful license holder of certain third-party technologies incorporated into the SOFTWARE. The SOFTWARE is protected by copyright laws and international treaty provisions. **The SOFTWARE is licensed to you, not sold to you**. C1 reserves all rights not otherwise expressly and specifically granted to you in this EULA.

2. **Backups.** You may make a copy of the SOFTWARE solely for backup or archival purposes. Notwithstanding the foregoing, you may not copy the printed Documentation.

3. **General Limitations.** You may not reverse engineer, decompile, or disassemble the SOFTWARE, except and only to the extent that applicable law expressly permits such activity notwithstanding this limitation.

4. **Software Transfers.** You may not rent or lease the SOFTWARE. You may permanently transfer all of your rights under the EULA, provided that you retain no copies, that you transfer all the SOFTWARE (including all component parts, the media and printed materials, any updates, upgrades, this EULA and, if applicable, the Certificate of Authenticity), and that the transferee agrees to be bound by the terms of this EULA. If the SOFTWARE is an update or upgrade, any transfer must include all prior versions of the SOFTWARE.

5. **Termination.** Without prejudice to any other rights it may have, C1 may terminate this EULA and the Licenses if you fail to comply with the terms and conditions contained herein. In such an event, you must destroy all copies of the SOFTWARE and all of its component parts.

6. **Export Restrictions.** You acknowledge that the SOFTWARE is of U.S. origin. You acknowledge that the license and distribution of the SOFTWARE is subject to the export control laws and regulations of the United States of America, and any amendments thereof, which restrict exports and re-exports of software, technical data, and direct products of technical data, including services and Developed Software. You agree that you will not export or re-export the SOFTWARE or any Developed Software, or any information, documentation and/or printed materials related thereto, directly or indirectly, without first obtaining permission to do so as required from the United States of America Department of Commerce's Bureau of Export Administration ("**BXA**"), or other appropriate governmental agencies, to any countries, end-users, or for any end-uses that are restricted by U.S. export laws and regulations, and any amendments thereof, which include, but are not limited to: Restricted Countries, Restricted End-Users, and Restricted End-Uses.

   These restrictions change from time to time. You represent and warrant that neither the BXA nor any other United States federal agency has suspended, revoked or denied your export privileges. C1 acknowledges that it shall use reasonable efforts to supply you with all reasonably necessary information regarding the SOFTWARE and its business to enable you to fully comply with the provisions of this Section. If you have any questions regarding your obligations under United States of America export regulations, you should contact the Bureau of Export Administration, United States Department of Commerce, Exporter Counseling Division, Washington DC. U.S.A. (202) 482-4811, http://www.bxa.doc.gov.

7. **U.S. Government Restricted Rights.** The SOFTWARE and documentation are provided with RESTRICTED RIGHTS. You will comply with any requirements of the Government to obtain such RESTRICTED RIGHTS protection, including without limitation, the placement of any restrictive legends on the SOFTWARE, and any license agreement used in connection with the distribution of the SOFTWARE. Manufacturer is ComponentOne, LLC, 201 South Highland Avenue , 3rd Floor, Pittsburgh, Pennsylvania 15206 USA. For solicitations issued by the Government on or after December 1, 1995 and the Department of Defense on or after September 29, 1995, the only rights provided in the software and documentation provided herein shall be those contained in this EULA. Under no circumstances shall C1 be obligated to comply with any Governmental requirements regarding the submission of or the request for exemption from submission of cost or pricing data or cost accounting requirements. For any distribution of the SOFTWARE that would require compliance by C1 with the Government's requirements relating to cost or pricing data or cost accounting requirements, you must obtain an appropriate waiver or exemption from such requirements for the benefit of C1 from the appropriate Government authority before the distribution and/or license of the SOFTWARE to the Government.

IV. WARRANTIES AND REMEDIES.

1. **Limited Warranty**. C1 warrants that the original media, if any, are free from defects for ninety (90) days from the date of delivery of the SOFTWARE. C1 also warrants that: (i) it has the full power to enter into this Agreement and grant the license rights set forth herein; (ii) it has not granted and will not grant any rights in the Software to any third party which grant is inconsistent with the rights granted to you in this Agreement; and (iii) the Software does not and will not infringe any trade secret, copyright, trademark or other proprietary right held by any third party and does not infringe any patent held by any third party. **EXCEPT AS OTHERWISE PROVIDED IN THE PRECEDING SENTENCE, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, C1 EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE, DOCUMENTATION AND ANYTHING ELSE PROVIDED BY C1 HEREBY AND C1 PROVIDES THE SAME IN "AS IS" CONDITION WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK ARISING OUT OF USE OR**

PERFORMANCE OF THE SOFTWARE AND DOCUMENTATION REMAINS WITH YOU. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS WHICH VARY FROM STATE TO STATE.

2. **Limited Remedy.** C1 PROVIDES NO REMEDIES OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, FOR ANY SAMPLE APPLICATION CODE, REDISTRIBUTABLE FILES, TRIAL VERSION AND THE NOT FOR RESALE VERSION OF THE SOFTWARE. ANY SAMPLE APPLICATION CODE, TRIAL VERSION AND THE NOT FOR RESALE VERSION OF THE SOFTWARE ARE PROVIDED "AS IS".

   C1's entire liability and your exclusive remedy under this EULA shall be, at C1's sole option, either (a) return of the price paid for the SOFTWARE; (b) repair the SOFTWARE through updates distributed online or otherwise in C1's discretion; or (c) replace the SOFTWARE with SOFTWARE that substantially performs as described in the SOFTWARE documentation, provided that you return the SOFTWARE in the same manner as provided in Section I.2 for return of the SOFTWARE for non-acceptance of this EULA. Any media for any repaired or replacement SOFTWARE will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. THESE REMEDIES ARE NOT AVAILABLE OUTSIDE OF THE UNITED STATES OF AMERICA. **TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL C1 BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFIT, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE, EVEN IF C1 HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES IN CERTAIN CASES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.**

V. MISCELLANEOUS.

1. **This is the Entire Agreement.** This EULA (including any addendum to this EULA included with the SOFTWARE) is the final, complete and exclusive statement of the entire agreement between you and C1 relating to the SOFTWARE. This EULA supersedes any prior and contemporaneous proposals, purchase orders, advertisements, and all other communications in relation to the subject matter of this EULA, whether oral or written. No terms or conditions, other than those contained herein, and no other understanding or agreement which in any way modifies these terms and conditions, shall be binding upon the parties unless entered into in writing executed between the parties, or by other non-oral manner of agreement whereby the parties objectively and definitively act in a manner to be bound (such as by continuing with an installation of the SOFTWARE, etc.). Employees, agents and other representatives of C1 are not permitted to orally modify this EULA.

2. **You Indemnify C1.** You agree to indemnify, hold harmless, and defend C1 and its suppliers and resellers from and against any and all claims or lawsuits, including attorney's fees, which arise out of or result from your distribution of your Developed Software, your Developed Web Server Software or from your breach of any of the terms and conditions of this EULA.

3. **Interpretation of this EULA.** If for any reason a court of competent jurisdiction finds any provision of this EULA, or any portion thereof, to be unenforceable, that provision of this EULA will be enforced to the maximum extent permissible so as to effect the intent of the parties, and the remainder of this EULA will continue in full force and effect. Formatives of defined terms shall have the same meaning of the defined term. Failure by either party to enforce any provision of this EULA will not be deemed a waiver of future enforcement of that or any other provision. Except as otherwise required or superseded by law, this EULA is governed by the laws of the Commonwealth of Pennsylvania, without regard to its conflict of laws principles. The parties consent to the personal jurisdiction and venue of the Commonwealth of Pennsylvania, in the County of Allegheny, and agree that any legal proceedings arising out of this EULA shall be conducted solely in such Commonwealth. If the SOFTWARE was acquired outside the United States, then local law may apply.

Technical Support

ComponentOne **VSFlexGrid** is developed and supported by ComponentOne LLC, a company formed by the merger of APEX Software Corporation and VideoSoft. You can obtain technical support using any of the following methods:

ComponentOne Web site

The ComponentOne Web site at www.componentone.com provides a wealth of information and software downloads for **VSFlexGrid** users, including:

- Descriptions of the various support options available through the ComponentOne Service Team.
- Answers to frequently asked questions (FAQ's) about our products, organized by functionality. Please consult the FAQ's before contacting us directly, as this can save you time and also introduce you to other useful information pertaining to our products.
- Free product updates, which provide you with bug fixes and new features.

ComponentOne HelpCentral

ComponentOne HelpCentral is the new online resource for Visual Studio developers and Help authors. Visit HelpCentral to get information on ComponentOne products, view online demos, get Tech Tips and answers to frequently asked questions (FAQ's), search the ComponentOne knowledgebase and more!

Internet e-mail

For technical support through the Internet, e-mail us at:

support.vsflex@componentone.com

To help us provide you with the best support, please include the following information when contacting ComponentOne:

- Your ComponentOne product serial number.
- The version and name of your operating system.
- Your development environment and its version.

For more information on technical support, go to:

www.componentone.com/support

Peer-to-Peer newsgroup

ComponentOne also sponsors peer-to-peer newsgroups for VSFlexGrid users. ComponentOne does not offer formal technical support in this newsgroup, but instead sponsors it as a forum for users to post and answer each other's questions regarding VSFlexGrid. However, ComponentOne may monitor the newsgroups to ensure accuracy of information and provide comments when necessary. You can access the newsgroup from the ComponentOne Web site at http://helpcentral.componentone.com/Newsgroups.aspx.

Documentation

ComponentOne documentation is available with each of our products in HTML Help, Microsoft Help 2.0 (.NET, ASP.NET and Mobile Device products only), and PDF format. All of the PDFs are also available on ComponentOne HelpCentral. If you have suggestions on how we can improve our documentation, please email the Documentation team. Please note that e-mail sent to the Documentation team is for documentation feedback only. Technical Support and Sales issues should be sent directly to their respective departments.

Redistributable Files

ComponentOne **VSFlexGrid** is developed and published by ComponentOne LLC. You may use it to develop applications in conjunction with Microsoft Visual Studio or any other programming environment that enables the user to use and integrate the control(s). You may also distribute, free of royalties, the following Redistributable Files with any such application you develop to the extent that they are used separately on a single CPU on the client/workstation side of the network:

- VSFlex8.ocx
- VSFlex8d.ocx
- VSFlex8L.ocx
- VSFlex8U.ocx
- VSSTR8.ocx
- VSFlex8N.ocx

Site licenses are available for groups of multiple developers. Please contact Sales@ComponentOne.com for details.

Note: When shipping your applications, we strongly suggest that you use a professional installation utility such as the Wise Installation System®, InstallShield®, or the setup wizard that ships with Microsoft Visual Basic 5.0 and later.

VSFlexGrid is a dependency-free control and does not require any additional DLLs in order to run.

Adding the VSFlexGrid 8.0 Component to the Toolbox

To use the **VSFlexGrid 8.0** components, they must be added to the Visual Studio Toolbox:

Open the Visual Basic IDE (Microsoft's Integrated Development Environment). Make sure the Toolbox is visible (if necessary, select Toolbox in the View menu).

To set up the **VSFlexGrid 8.0** components to appear on their own tab, right-click anywhere in the Toolbox and select **Add Tab** from the context menu. Enter a tab name, for example, "FlexGrid 8", in the dialog box. Select the new tab. Right-click the gray area under that tab and select **Components** from the context menu. The **Components** dialog box opens.

In the **Components** dialog box, find and select ComponentOne FlexGrid 8.0 Control. Click **OK**.

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 that belongs to a database.

VSFlexGrid 8.0 ships with three versions of the **VSFlexGrid** control. Each version has its own OCX file and supports a different type of data binding:

OCX File	Description
VSFLEX8.OCX	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.
VSFLEX8D.OCX	This version provides support for DAO data binding, the more traditional type of data binding supported by Visual Basic's built-in Data Control.
VSFLEX8L.OCX	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.
VSFLEX8U.OCX	This is a Unicode version that provides support for ADO/OLEDB data binding. It only runs on Windows NT systems.
VSFLEX8N.OCX	This file contains a Unicode version of the VSFlexGrid Light control.

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

Basic Operations (page 14)

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

Editing Cells (page 15)

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

Formatting Cells (page 16)

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 (page 19)

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 (page 17)

Describes how to add subtotals to grids and how to build outline trees.

Data Binding (ADO and DAO) (page 22)

Discusses the basic aspects of ADO/OLEDB and DAO data-binding.

Other types of Data Binding (page 23)

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 (page 21)

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++ (page 24)

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++ (page 30)

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

VSFlexGrid Property Groups (page 32)

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 (for example, "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 (for example, "|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 alone 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:

Product	Region	Associate	Sales
Grand Total			1,736,402
⊟ Total Drums			161,656
⊞ Drums	Total East		18,866
⊞ Drums	Total North		45,342
⊟ Drums	Total South		47,874
Drums	South	John	45,342
		Mike	2,532
⊞ Drums	Total West		49,574
⊟ Total Flutes			262,511
⊟ Flutes	Total East		47,975
Flutes	East	Paul	4,543
		Sylvia	43,432
⊟ Flutes	Total North		75,877
Flutes	North	Mike	75,877

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:

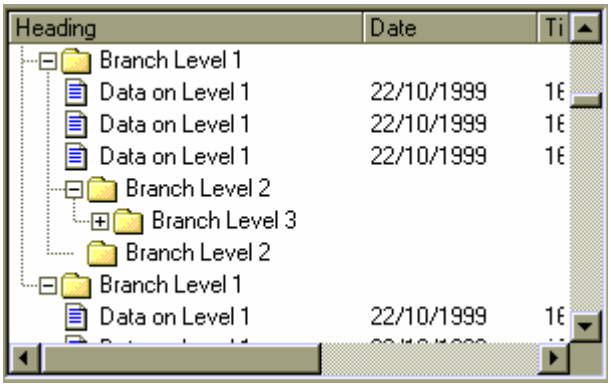
1. Populate the grid.
2. Turn some rows into outline nodes by setting their **IsSubtotal**() property to **True**.
3. Set each node's level in the hierarchy by setting their **RowOutlineLevel**() property. Higher values mean the node is deeper (more indented) into the outline tree.

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

```
' 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

    ' add a regular item
```

```
    Else
      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:



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

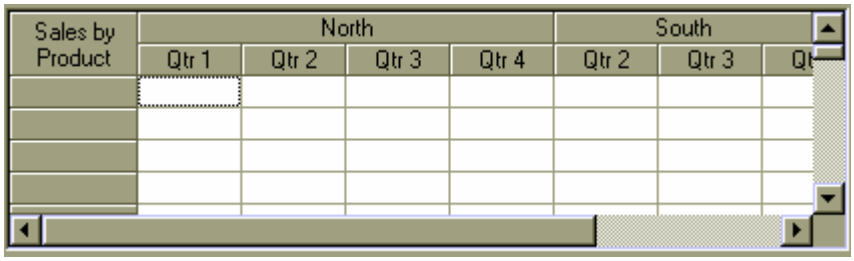
To create merged table headers, you must start by setting the **MergeCells** property to *flexMergeFixedOnly*. Then, designate the rows and columns that you want to merge using the **MergeRow**(), and **MergeCol**() properties. Finally, assign the text to the header cells so that the cells you want to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

The code below shows an example:

```
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
' four cells, will merge
        fg.Cell(flexcpText, 0, 1, 0, 4) = "North"
' four cells, will merge
fg.Cell(flexcpText, 0, 5, 0, 8) = "South"
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
' two cells, will merge
       fg.Cell(flexcpText, 0, 0, 1, 0) = "Sales by " & _ "Product"
        ' align and autosize 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 show 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
```

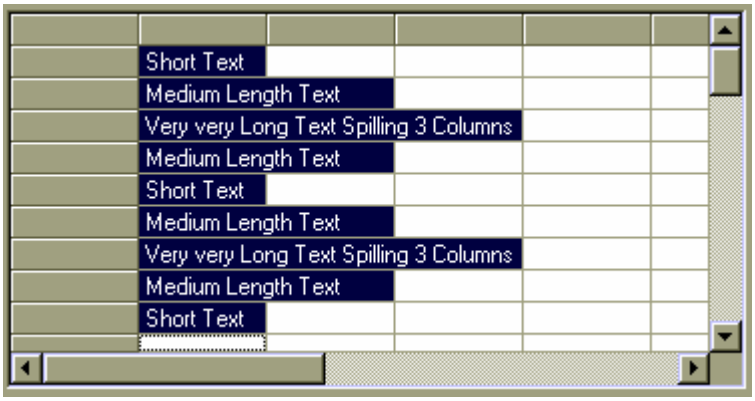
This is the result:

Country	City	CompanyName
Argentina	Buenos Aires	Cactus Comidas para llevar
		Océano Atlántico Ltda.
		Rancho grande
Austria	Graz	Ernst Handel
	Salzburg	Piccolo und mehr
Belgium	Bruxelles	Maison Dewey
	Charleroi	Suprêmes délices
Brazil	Campinas	Gourmet Lanchonetes
	Resende	Wellington Importadora
	Rio de Janeiro	Hanari Carnes
		Que Delícia
		Ricardo Adocicados
	São Paulo	Comércio Mineiro
		Familia Arquibaldo
		Queen Cozinha

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 ComponentOne). 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 (VSFLEX8.OCX) and a DAO version (VSFLEX8D.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 controls data binding: the **DataSource**, **DataMode**, and **VirtualData** properties, and the **AfterDataRefresh** event.

The 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.

The 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.

The 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.

The 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:

Method Name	Description
GetFieldCount()	Returns the number of fields in the data source.
GetRecordCount()	Returns the number of records in the data source.
GetFieldName(Fld&)	Returns the name of field number Fld (ranging from zero to GetFieldCount() - 1).
GetData(Fld&, Rec&)	Returns the data in field Fld , record Rec (ranging from zero to GetRecordCount() - 1).
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:
      // standard constructor
      CMyDlg(CWnd* pParent = NULL);

      // new VSFlexGrid
      CvsFlexGrid m_GridDynamic;
```

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 an "event sink"). 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:

```
#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
  pPic->Release();
}
```

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:

```
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

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

instead of

```
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:

```
// include this statement in the StdAfx.h file
#import "c:\windows\system\vsflex8l.ocx" no_namespace
// note: to use the OLEDB/ADO version of the control,
// you need to #import "msdatsrc.tlb" as well.
//#import "c:\windows\system\msdatsrc.tlb" no_namespace
//#import "c:\windows\system\vsflex8l.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\vsflex8l.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:

```
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
  }
```

7. 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**).

8. 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:

```
  #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);
}
```

That's about it. You could clean up this project by removing the references to the **idl** and **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 **CPictureHolder** class. But you can use the **CPicHolder** class provided in the samples on the distribution CD. **CPicHolder** is actually more powerful than **CPictureHolder** because it provides methods for loading pictures from disk files and from the clipboard. Once you include the **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 **CToolTip**, and a file named **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 **#import** statement in the dialog header file or in the **StdAfx.h** file.

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

```
  CAxWindow m_wndGrid;      // host window
  IVSFlexGridPtr m_spGrid;  // pointer to control
```

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

```
  // 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);
```

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 **IDispEventImpl** declaration to your class so it inherits the ATL event handling mechanisms, and adds an event sink map to your class using **BEGIN_SINK_MAP**, **SINK_ENTRY**, and **END_SINK_MAP** macros. You still need to add the call to **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 **getCtlText**() 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:

```
  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:

```
  // 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:

```
  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:

```
  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:

```
  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:

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

```
      // Set FlexGrid properties.
      fg.setEditable(True);
      fg.setDataMode(vsflex8.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:

```
    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:

```
  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**, **ShowComboButton**, **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**, **GetNodeRow**

7) Merging Cells

MergeCells, **MergeRow**, **MergeCol**, **MergeCompare**, **GetMergedRange**

8) Data Binding

DataSource, **DataMember**, **DataMode**, **VirtualData**, **DataRefresh**, **BeforeDataRefresh**, **AfterDataRefresh**, **AutoResize**, **Error**
BindToArray, **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 Samples

Please be advised that this ComponentOne software title is accompanied by various sample projects and/or demos, which may or not make use of other ComponentOne development tools. While the sample projects and/or demos included with the software are used to demonstrate and highlight the product's features, and how the control may be integrated with the rest of the ComponentOne product line, some of the controls used in the demo/sample project may not be included with the purchase of certain individual products.

The ComponentOne Samples are also available at http://helpcentral.componentone.com/ProductResources.aspx.

Visual Basic Samples

AdjWidth	Resize a form based on the width of the grid. This sample uses the VSFlexGrid control.
AutoComp	Implement Excel-style auto-completion. This sample uses the VSFlexGrid control.
Batch	Browse and edit a disconnected ADO recordset. This sample uses the VSFlexGrid control.
Bind	Bind the FlexGrid to Variant Arrays or to other FlexGrid controls. This sample uses the VSFlexGrid control.
Buttons	Implement custom cell buttons. This sample uses the VSFlexGrid control.
Calendar	Build a calendar control with the VSFlexGrid. This sample uses the VSFlexGrid control.
CellChange	Provide conditional formatting and dynamic data summaries. This sample uses the VSFlexGrid control.
CellNotes	Implement Excel-style cell notes. This sample uses the VSFlexGrid control.
CustData	Implement grid-based views of custom data structures using FlexDataSource. This sample uses the VSFlexGrid control.
CustData2	Implement grid-based views of virtual (calculated) data using FlexDataSource. This sample uses the VSFlexGrid control.
CustDataADO	Bind the grid to ADO data sources using the FlexDataSource property. This sample uses the VSFlexGrid control.
CustDataFMT	Implement data formatting using FlexDataSource. This sample uses the VSFlexGrid control.
CustDataPict	Map data values into pictures using the ColImageList property. This sample uses the VSFlexGrid control.
CustDataSort	Sort data from a custom data source. This sample uses the VSFlexGrid control.
CustomEditor	Edit grid using a custom editor. This sample uses the VSFlexGrid control.

DBase	Browse data from a database with frozen cells, translated text/pictures, and hierarchical recordsets. This sample uses the VSFlexGrid control.
DragBound	Implement data-bound column dragging. This sample uses the VSFlexGrid control.
DragDrop	Implement Explorer-style drag and drop using the VSFlexGrid control. This sample uses the VSFlexGrid control.
DragDrop2	Implement OLE drag/drop of text, pictures, and files. This sample uses the VSFlexGrid control.
DragRows	Use the ExplorerBar property and the DragRow method to drag rows. This sample uses the VSFlexGrid control.
Events	This sample uses the VSFlexGrid control.
DTPick	Use a DateTimePicker control to edit date entries. This sample uses the VSFlexGrid control.
Excel	Save grid contents into an Excel xls file. This sample uses the VSFlexGrid control.
Explorer	Build an Explorer-like interface, populated on demand. This sample uses the VSFlexGrid control.
Fetch	Compare the time it takes to bind to various types of recordset. This sample uses the VSFlexGrid control.
FilterDB	Filter data as it is retrieved or saved to a bound recordset. This sample uses the VSFlexGrid control.
Group	Implement Outlook-style grouping using the FlexGrid. This sample uses the VSFlexGrid and vsfgroup? control.
HyperLnk	Use the FlexGrid control to browse Internet hyperlinks. This sample uses the VSFlexGrid control.
KeepSelection	Set the Row property while keeping the extended row selection. This sample uses the VSFlexGrid control.
KeyHook	Shows how you can subclass the VSFlexGrid editor window and record. This sample uses the VSFlexGrid control.
MaskClean	Extract data from grid entries formatted with an EditMask. This sample uses the VSFlexGrid control.
OwnerDraw	Draw custom borders around selection. This sample uses the VSFlexGrid control.
PageBrk	Print a grid and control where pages break. This sample uses the VSFlexGrid control.
Password	Use a column for entering passwords. This sample uses the VSFlexGrid control.

PIVOT	Build a pivoting data view with the FlexGrid. This sample uses the VSFlexGrid control.
PopEdit	Implement a custom menu for the FlexGrid's editor. This sample uses the VSFlexGrid control.
PrintCustom	This sample uses the VSFlexGrid control.
PropPage	Build a property browser interface using the VSFlexGrid. This sample uses the VSFlexGrid control.
ROLLBACK	Edit disconnected recordsets with the option to commit or rollback the edits. This sample uses the VSFlexGrid control.
ScrlCell	Scroll the contents of individual grid cells. This sample uses the VSFlexGrid control.
Selection	Use code to control extended selections and scrolling. This sample uses the VSFlexGrid control.
SortOutline	Use the Node.Sort method to sort outlines. This sample uses the VSFlexGrid control.
Spell	Use the VSSpell control to spell check grid entries as you type them. This sample uses the VSFlexGrid and VSSpell control.
TreeNode	Manage an outline tree using the VSFlexNode object. This sample uses the VSFlexGrid control.
WallPaper	Implement graphical backgrounds with the WallPaper property. This sample uses the VSFlexGrid control.
XLS	This sample uses the VSFlexGrid control.
XML	Display an XML tree. This sample uses the VSFlexGrid control.
Xml2	Display an XML tree using the MSXML version 2.0 DOM. This sample uses the VSFlexGrid control.
XMLBound	Bind a FlexGrid control to an XML document. This sample uses the VSFlexGrid control.

C++ Samples

AdjWidth	Resize a form based on the width of the grid. This sample uses the VSFlexGrid control.
ADOFilter	Filter records no an ADO recordset. This sample uses the VSFlexGrid control.
Archive	Zip-like utility using Archive and ArchiveInfo methods. This sample uses the VSFlexGrid control.

ATLDDROP	OLE drag and drop in ATL projects. This sample uses the VSFlexGrid control.
AxWin	Create FlexGrid controls as children of the main window. This sample uses the VSFlexGrid control.
BigDemo	View all main features for the VSFlexGrid control. This sample uses the VSFlexGrid control.
CellChange	Monitor changes to the grid to format and total cells. This sample uses the VSFlexGrid control.
Clipboard	Monitor the keyboard and support the clipboard. This sample uses the VSFlexGrid control.
CustDataMFC	Bind the grid to a custom data source. This sample uses the VSFlexGrid control.
DBase	Bind the grid to an ADO data source and customize it. This sample uses the VSFlexGrid control.
DragRows	Drag single rows or groups of rows. This sample uses the VSFlexGrid control.
DTPickATL	Use a DateTimePicker control to edit dates on the grid. This sample uses the VSFlexGrid control.
DTPickMFC	Use a DateTimePicker control to edit dates on the grid. This sample uses the VSFlexGrid control.
EditWindow	Customize the built-in editor. This sample uses the VSFlexGrid control.
EXPLORER	Use the FlexGrid to implement a Windows Explorer clone. This sample uses the VSFlexGrid control.
LoadXL	Load Excel files (xls) into the FlexGrid. This sample uses the VSFlexGrid control.
MergePrt	Print a merged grid and add custom headers/footers. This sample uses the VSFlexGrid control.
MFCData	Bind the grid to an ADO data source. This sample uses the VSFlexGrid control.
MfcDDrop	Implement OLE drag drop between grid and other controls. This sample uses the VSFlexGrid control.
MFCDynamic	Create a grid dynamically using the CreateControl method. This sample uses the VSFlexGrid control.
MFCObject	Store custom objects in the grid. This sample uses the VSFlexGrid control.
MFCPict	Show pictures in grid cells. This sample uses the VSFlexGrid control.
PropWnd	Implement a property grid using the FlexGrid. This sample uses the VSFlexGrid control.

SetupEdit	Use the SetupEditWindow event to customize the combo editor. This sample uses the VSFlexGrid control.
TREENODE	Manage an outline tree using the VSFlexNode object. This sample uses the VSFlexGrid control.
XML2	Display an XML tree using the MSXML version 2.0 DOM. This sample uses the VSFlexGrid control.

HTML Samples

HtmlSamples	Various HTML samples that show licensing, databinding, etc.

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 (page 41)

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 (page 45)

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

Data Analysis Demo (page 48)

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 (page 53)

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 (page 54)

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

Printing Demo (page 55)

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 (page 56)

This tutorial shows how to implement automatic and manual OLE drag and drop.

Visual C++ MFC Demo (page 59)

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.

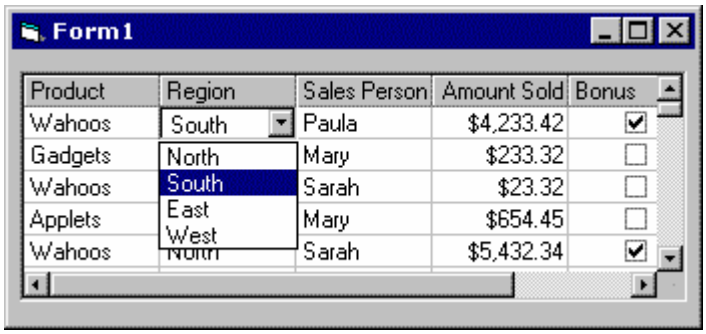
Edit Demo

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:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid 8.0** (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:

```
   Private Sub Form_Load()

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

   End Sub
```

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

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

Step 3: Check Boxes

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

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

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

```
Private Sub Form_Load()

    ' format column 3 (Amount Sold) to display currency
    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 **ColComboList** property. The list consists of a string with choices, separated by pipe characters ("|").

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

```
Private Sub Form_Load()

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

    ' 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:

```
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:

```
    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

    ' paste: ctrl-V, shift-ins
    If KeyCode = vbKeyV And Shift = 2 Then Pst = True
    If KeyCode = vbKeyInsert And Shift = 1 Then Pst = True
    ' 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.

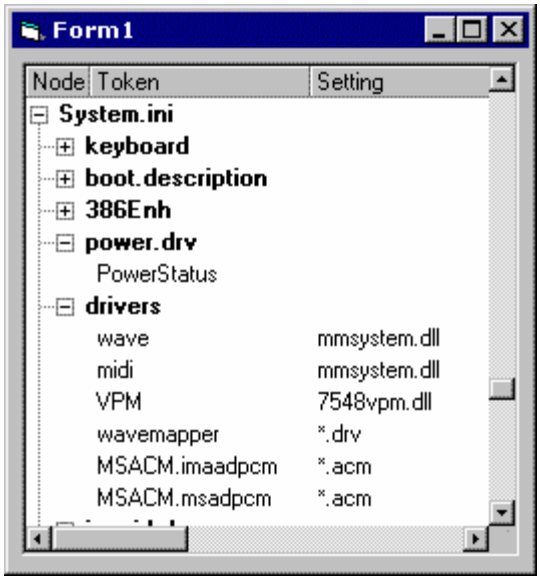
Press F5 to run the project again, then try copying and pasting some data. You will notice that it is possible to paste invalid data, because our paste code does not do any data validation. This is left as an exercise for the reader.

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 contains sections within the corresponding .INI file. Each section node contains branches that show the tokens and settings stored in the corresponding section. Here is how the final project will look:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid 8.0** (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:

```
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:

```
    Sub AddNode(inifile As String)
        Dim ln 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, ln

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

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

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

Next, the routine reads the INI file line by line. Section names are enclosed in square brackets. The code adds them to the control and then marks them as subtotals much the same way it marked the file name. The difference is that here it also sets the **RowOutlineLevel** property to 1, indicating this node is a child of the previous level-0 node (the one that contains the file name).

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

Step 3: Use the Outline

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

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

Step 4: Custom Mouse and Keyboard Handling

The **VSFlexGrid** control 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:

```
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 **DblClick** event handler from the **KeyPress** handler:

```
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.

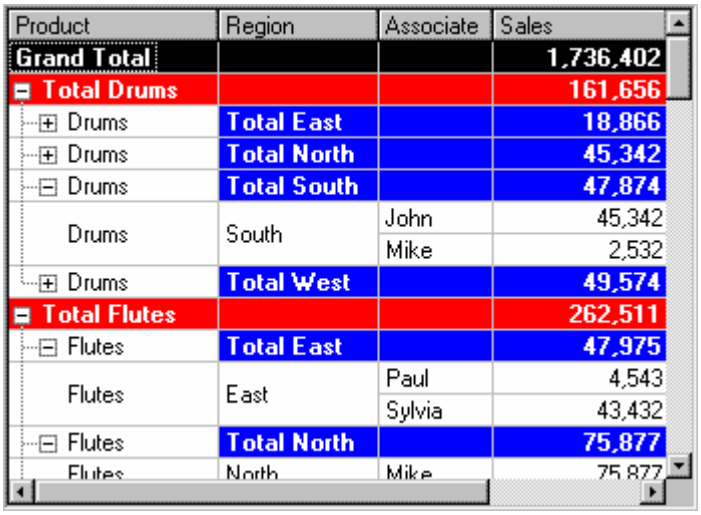
Data Analysis Demo

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:



Step 1: Create the Control

Start a new Visual Basic project including **VSFlexGrid 8.0** (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 Property window to set the control name to **fg**.

Step 2: Initialize and populate the grid

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

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

```
    Private Sub Form_Load()

        ' initialize the control
        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 fa.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 3: Automatic Sorting

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

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

Here is the code:

```
    Private Sub 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** event handler.

```
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 4: Cell Merging

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

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

```
Private Sub Form_Load()
    ' initialize the control
    ' define some sample data
    ' populate the control with the data
    ' set up cell merging (all columns)
    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 5: 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 figure column.

Here is the code:

```
    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:

```
    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 6: 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:

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

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

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

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

    ' initialize control
    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

        ' 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 it 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 ComponentOne's **VSPrinter** control, a separate ComponentOne 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.

```
    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:

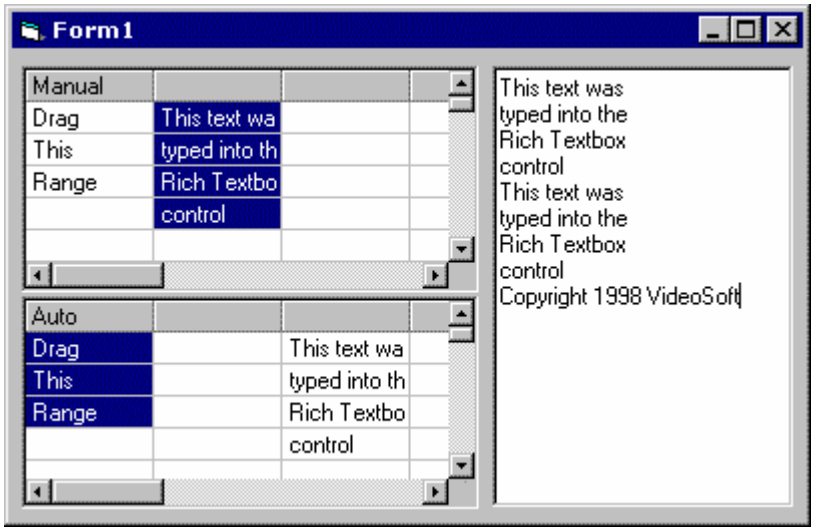
```
    ' BeforePageBreak: controls page breaks
    ' we assume we have subtotals above details,
    ' and prevent subtotal rows from
    ' being the last on a page
    Private Sub fg_BeforePageBreak(ByVal Row As Long, BreakOK As
Boolean)
        ' if this row is a subtotal heading, we can't break here
        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
                Exit Sub
            End If
        Next
    End Sub
```

OLE Drag and Drop Demo

This sample shows how to implement automatic and manual OLE drag and drop using **VSFlexGrid 8.0**.

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

Here is how the final application will look:



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

Step 1: Create the Controls

Start a new Visual Basic project including **VSFlexGrid 8.0** (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:

```
Private Sub Form_Load()

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

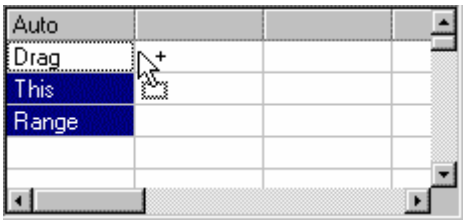
```
    ' initialize auto control
    With fgDDAuto
      .Cell(flexcpText, 0, 0) = "Auto"
      .FixedCols = 0
      .Editable = True
      .OLEDragMode = flexOLEDragAutomatic
      .OLEDropMode = flexOLEDropAutomatic
    End With

End Sub
```

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

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

That's all it takes to implement automatic OLE drag and drop. If you run the project now, you will be able to drag text from the Rich Editbox into the **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:

```
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:

```
    Private Sub fgDDManual_OLEStartDrag(Data As
VSFlex8Ctl.vsDataObject, AllowedEffects As Long)

        ' set contents of data object for manual drag
        Dim s$
        s = fgDDManual.Clip & vbCrLf & "Copyright 2000 ComponentOne"
        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 4: 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:

```
    Private Sub fgDDManual_OLEDragDrop(Data As VSFlex8Ctl.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.

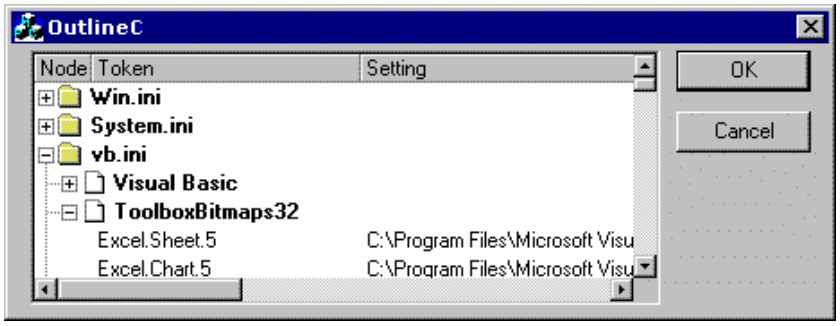
Visual C++ MFC Demo

The **VSFlexGrid 8.0** 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. You will see a dialog with some information about the new project. Click **OK**.

Step 2: Add the VSFlexGrid Control to the Project

Now that the project has been created, 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 ComponentOne **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 4: 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 5: Read the Data and Build the Outline

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

Open the dialog in the Visual C++ resource editor, then type CTRL-W (for Wizard). You will see a dialog that lets you add event handlers to each element on the form.

On the **Object Ids** list, select **IDOK**. On the **Messages** list, select **BN_CLICKED**. Now click the **Add Function** button, and then the **Edit Code** button.

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

```
void COutlineCDlg::OnOK()
{
  // TODO: Add extra validation here
  // comment the following line to avoid closing the
  // dialog when the user clicks OK:
  //CDialog::OnOK();
  // initialize variant to use as optional parameter
  COleVariant varDefault;
  V_VT(&varDefault) = VT_ERROR;
  // suspend repainting to increase speed
  m_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 COutlineCDlg::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_fa.GetRows() - 1;
          m_fg.SetIsSubtotal(row, TRUE);
          m_fg.SetRowOutlineLevel(row, 1);
          m_fg.Select(row, 0, varDefault, varDefault);
          m_fg.SetCellFontBold(TRUE);

      // if this is regular data, add branch
      } else if (strchr(ln, '=')) {
```

```
            char* p = strchr(ln, '=');
            *p = 0;
            CString str = (CString)"\t" + (CString)ln +
                          (CString)"\t" + (CString)(p + 1);
            m_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 6: 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 7: 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 Id**s list, then select each event and click the **Add Function** button. When you are done, click the **Edit Code** button and type the following code:

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

void COutlineCDlg::OnDblClickVsflexgrid1()
{
  // double clicking on a row expands or collapses it
  long r = m_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 8: Cell Pictures

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

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

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

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

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

    // initialize variant to use as optional parameter

    // create file node
    m_fg.SetCellFontBold(TRUE);
    LPDISPATCH pPic = picFile.GetPictureDispatch();
    m_fg.SetCellPicture(pPic);
    pPic->Release();

    // read ini file

        // if this is a section, add section node
            m_fg.SetCellFontBold(TRUE);
            m_fg.SetCellPicture(picSection.GetPictureDispatch());

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

The first line added includes the MFC header file **afxctl.h**. This file defines **CPictureHolder**, a handy class for manipulating OLE pictures. The next three lines added declare a **CPictureHolder** variable for each bitmap, and load the bitmaps using the **CreateFromBitmap** method.

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

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

VSFlexString 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.

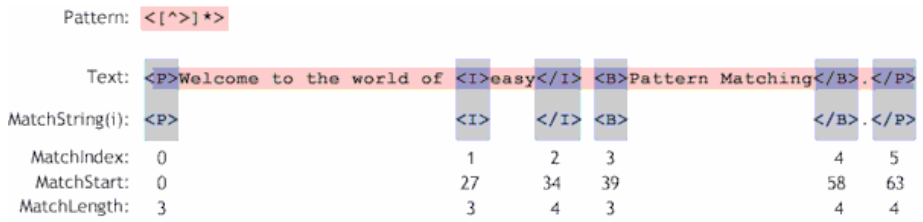
VSFlexString looks for text patterns on its **Text** property, and lets you inspect and change the matches it finds. The text patterns are specified through the **Pattern** property, using a regular expression syntax similar to the ones used in Unix systems.

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:



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.

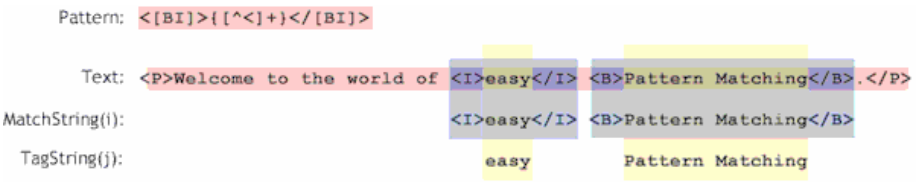
From this point on, you can work on the string either by modifying it directly or by changing the text in the **TagString** property.

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 "**<[BI]>{[^>]*}</[BI]>**", 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.



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:

Char	Description
^	Beginning of a string.
$	End of a string.
.	Any character.
[list]	Any character in list. For example, "[AEIOU]" matches any single uppercase vowel.
[^list]	Any character not in list. For example, "[^]" matches any character except a space.
[A-Z]	Any character between 'A' and 'Z'. For example, "[0-9]" matches any single digit.
?	Repeat previous character zero or one time. For example, "10?" matches "1" and "10".
*	Repeat previous character zero or more times. For example, "10*" matches "1", "10", "1000", etc.
+	Repeat previous character one or more times. For example, "10+" matches "10", "1000", etc.
\	Escape next character. This is required to any of the special characters that are part of the syntax. For example "\.*\+\\" matches ".*+\". It is also required to encode some special non-printable characters (such as tabs) listed below.
{tag}	Tag this part of the match so you can refer to it later using the TagString property.

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

Code	Description
\a	Bell (Chr(7))
\b	Backspace (Chr(8))
\f	Formfeed (Chr(12))
\n	New line (Chr(10), vbLf)
\r	Carriage return (Chr(13), vbCr)
\t	Horizontal tab (Chr(9), vbTab)
\v	Vertical tab (Chr(11))

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
"[A-Z][a-z]*"             ' any uppercase word
"[0-9]+"                  ' any stream of digits
"\."                      ' decimal point (needs escape
                          ' character)

"[1-9]+[1-9]*"            ' any stream of digits not
                          ' starting with 0

"[+\-]?[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,

```
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 **VSFlexString** 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 **VSFlexString** 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:

```
    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:

```
    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.
```

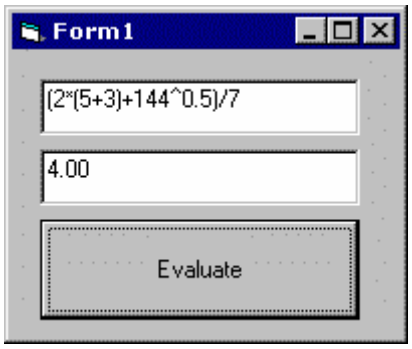
The function works as expected. Note that the last try fails, because the last name is not supposed to contain periods. In this case, the function just returns the original string, which seems like a reasonable thing to do.

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:



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 **VSSTR8.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:

```
' 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:

```
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

Before you can use a **VSFlexGrid** control in your application, you must add the **VSFLEX8.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 **VSFLEX8.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.

VSFlexGrid Properties, Events, and Methods

All of the properties, events, and methods for the **VSFlexGrid** control are listed in the following tables. Properties, events, and methods that apply *only* to this control, or that require special consideration when used with it, are marked with an asterisk (*). These are documented in later sections. For documentation on the remaining properties, see the Visual Basic documentation.

Properties

***AccessibleDescription**	Gets or sets the description of the control used by accessibility client applications.
***AccessibleName**	Gets or sets the name of the control used by accessibility client applications.
***AccessibleRole**	Gets or sets the role of the control used by accessibility client applications.
***AccessibleValue**	Gets or sets the value of the control used by accessibility client applications.
***Aggregate**	Returns an aggregate function (sum, average, etc.) for a given range.
***AllowBigSelection**	Returns or sets whether clicking on the fixed area will select entire columns and rows.
***AllowSelection**	Returns or sets whether the user can select ranges of cells with the mouse and keyboard.
***AllowUserFreezing**	Returns or sets whether the user is allowed to freeze rows and columns with the mouse.
***AllowUserResizing**	Returns or sets whether the user is allowed to resize rows and columns with the mouse.
***Appearance**	Returns or sets the paint style of the control on an MDIForm or Form object.
***ArchiveInfo**	Returns information from a VSFlex archive file.
***AutoResize**	Returns or sets whether column widths will be automatically adjusted when data is loaded.
***AutoSearch**	Returns or sets whether the control will search for entries as they are typed.

***AutoSearchDelay**	Returns or sets the delay, in seconds, before the AutoSearch buffer is reset.
***AutoSizeMode**	Returns or sets whether **AutoSize** will adjust column widths or row heights to fit cell contents.
***AutoSizeMouse**	Returns or sets whether columns should be resized to fit when the user double-clicks on the header row.
***BackColor**	Returns or sets the background color of the non-fixed cells.
***BackColorAlternate**	Returns or sets the background color for alternate rows (set to 0 to disable).
***BackColorBkg**	Returns or sets the background color of the area not covered by any cells.
***BackColorFixed**	Returns or sets the background color of the fixed rows and columns.
***BackColorFrozen**	Returns or sets the background color of the frozen rows and columns.
***BackColorSel**	Returns or sets the background color of the selected cells.
***BottomRow**	Returns the zero-based index of the last row displayed in the control.
***Cell**	Returns or sets cell properties for an arbitrary range.
***CellAlignment**	Returns or sets the alignment of text in the selected cell or range.
***CellBackColor**	Returns or sets the background color of the selected cell or range.
***CellButtonPicture**	Returns or sets the picture used in cell buttons.
***CellChecked**	Returns or sets whether a grid cell has a check mark in it.
***CellFloodColor**	Returns or sets the color to be used for flooding a cell.
***CellFloodPercent**	Returns or sets the percentage of flooding for a cell.
***CellFontBold**	Returns or sets the Bold attribute of the font of the selected cell or range.
***CellFontItalic**	Returns or sets the Italic attribute of the font of the selected cell or range.
***CellFontName**	Returns or sets the name of the font of the selected cell or range.
***CellFontSize**	Returns or sets the size of the font of the selected cell or range.
***CellFontStrikethru**	Returns or sets the Strikethru attribute of the font of the selected cell or range.
***CellFontUnderline**	Returns or sets the Underline attribute of the font of the selected cell or range.

***CellFontWidth**	Returns or sets the width of the font of the selected cell or range.
***CellForeColor**	Returns or sets the foreground color of the selected cell or range.
***CellHeight**	Returns the height of the selected cell, in twips. Also brings the cell into view, scrolling if necessary.
***CellLeft**	Returns the left (x) coordinate of the selected cell relative to the control, in twips. Also brings the cell into view, scrolling if necessary.
***CellPicture**	Returns or sets the picture displayed in a selected cell or range.
***CellPictureAlignment**	Returns or sets the alignment of the pictures in the selected cell or range.
***CellTextStyle**	Returns or sets 3D effects for text in a selected cell or range.
***CellTop**	Returns the top (y) coordinate of the selected cell relative to the control, in twips. Also brings the cell into view, scrolling if necessary.
***CellWidth**	Returns the width of the selected cell, in twips. Also brings the cell into view, scrolling if necessary.
***ClientHeight**	Returns the height of the control's client area, in twips.
***ClientWidth**	Returns the width of the control's client area, in twips.
***Clip**	Returns or sets the contents of a range.
***ClipSeparators**	Returns or sets the characters to be used as column and row separators in **Clip** strings.
***Col**	Returns or sets the zero-based index of the current column.
***ColAlignment**	Returns or sets the alignment of the given column.
***ColComboList**	Returns or sets the list to be used as a drop-down on the specified column.
***ColData**	Returns or sets a user-defined variant associated with the given column.
***ColDataType**	Returns or sets the data type for the column.
***ColEditMask**	Returns or sets the input mask used to edit cells on the specified column.
***ColFormat**	Returns or sets the format used to display numeric values.
***ColHidden**	Returns or sets whether a column is hidden.
***ColImageList**	Sets or returns a handle to an **ImageList** to be used as a source of pictures for a given column.
***ColIndent**	Returns or sets the indentation of the given column, in twips.

***ColIndex**	Returns the column index that matches the given key.
***ColIsVisible**	Returns whether a given column is currently within view.
***ColKey**	Returns or sets a key used to identify the given column.
***ColPos**	Returns the left (x) coordinate of a column relative to the edge of the control, in twips.
***ColPosition**	Moves a given column to a new position.
***Cols**	Returns or sets the total number of columns in the control.
***ColSel**	Returns or sets the extent of a range of columns.
***ColSort**	Returns or sets the sorting order for each column (for use with the **Sort** property).
***ColWidth**	Returns or sets the width of the specified column in twips.
***ColWidthMax**	Returns or sets the maximum column width, in twips.
***ColWidthMin**	Returns or sets the minimum column width, in twips.
***ComboCount**	Returns the number of items in the editor's combo list.
***ComboData**	Returns the long value associated with an item in the editor's combo list.
***ComboIndex**	Returns or sets the zero-based index of the current selection in the editor's combo list.
***ComboItem**	Returns the string associated with an item in the editor's combo list.
***ComboList**	Returns or sets the list to be used as a drop-down when editing a cell.
***ComboSearch**	Returns or sets whether combo lists should support smart searches.
***DataMember**	Returns or sets the data member.
***DataMode**	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).
***DataSource**	Returns or sets the data source.
***DragMode**	Returns/sets a value that determines whether manual or automatic drag mode is used.
***Editable**	Returns or sets whether the control allows in-cell editing.
***EditMask**	Returns or sets the input mask used to edit cells.
***EditMaxLength**	Returns or sets the maximum number of characters that can be entered in the editor.
***EditSelLength**	Returns or the number of characters selected in the editor.
***EditSelStart**	Returns or sets the starting point of text selected in the editor.

***EditSelText**	Returns or sets the string containing the current selection in the editor.
***EditText**	Returns or sets the text in the cell editor.
***EditWindow**	Returns a handle to the grid's editing window, or 0 if the grid is not in edit mode.
***Ellipsis**	Returns or sets whether the control will display ellipsis (...) after long strings.
Enabled	Returns or sets a value that determines whether a form or control can respond to user-generated events. See the Visual Basic documentation for more information.
***ExplorerBar**	Returns or sets whether column headers are used to sort and/or move columns.
***ExtendLastCol**	Returns or sets whether the last column should be adjusted to fit the control's width.
***FillStyle**	Returns or sets whether changes to the **Text** or **Format** properties apply to the current cell or to the entire selection.
***FindRow**	Returns the index of a row that contains a specified string or RowData value.
***FindRowRegex**	Returns the index of the row that contains a match or -1 if no match was found.
***FixedAlignment**	Returns or sets the alignment for the fixed rows in a column.
***FixedCols**	Returns or sets the number of fixed (non-scrollable) columns.
***FixedRows**	Returns or sets the number of fixed (non-scrollable) rows.
***Flags**	Gets or sets flags that affect the behavior of the control.
***FlexDataSource**	Returns or sets a custom data source for the control.
***FloodColor**	Returns or sets the color used to flood cells.
***FocusRect**	Returns or sets the type of focus rectangle to be displayed around the current cell.
Font	Returns a **Font** object. See the Visual Basic documentation for more information.
***FontBold**	Determines whether the font is bold.
***FontItalic**	Determines whether the font is italicized.
***FontName**	Returns or sets the name of the font.
***FontSize**	Determines the size of the font.
***FontStrikethru**	Determines the strikethru of the font.
***FontUnderline**	Determines the font is underlined.
***FontWidth**	Returns or sets the width of the font, in points.

***ForeColor**	Returns or sets the foreground color of the non-fixed cells.
***ForeColorFixed**	Returns or sets the foreground color of the fixed rows and columns.
***ForeColorFrozen**	Returns or sets the foreground color of the frozen rows and columns.
***ForeColorSel**	Returns or sets the foreground color of the selected cells.
***FormatString**	Assigns column widths, alignments, and fixed row and column text.
***FrozenCols**	Returns or sets the number of frozen (editable but non-scrollable) columns.
***FrozenRows**	Returns or sets the number of frozen (editable but non-scrollable) rows.
***GridColor**	Returns or sets the color used to draw the grid lines between the non-fixed cells.
***GridColorFixed**	Returns or sets the color used to draw the grid lines between the fixed cells.
***GridLines**	Returns or sets the type of lines to be drawn between non-fixed cells.
***GridLinesFixed**	Returns or sets the type of lines to be drawn between fixed cells.
***GridLineWidth**	Returns or sets the width of the grid lines, in pixels.
***GroupCompare**	Returns or sets the type of comparison used when grouping cells.
***HighLight**	Returns or sets whether selected cells will be highlighted.
hWnd	Returns a handle to a form or control. See the Visual Basic documentation for more information.
***IsCollapsed**	Returns or sets whether an outline row is collapsed or expanded.
***IsSearching**	Returns a value that indicates whether the grid is in search mode.
***IsSelected**	Returns or sets whether a row is selected (for listbox-type selections).
***IsSubtotal**	Returns or sets whether a row contains subtotals (as opposed to data).
***LeftCol**	Returns or sets the zero-based index of the leftmost non-fixed column displayed in the control.
***MergeCells**	Returns or sets whether cells with the same contents will be merged into a single cell.
***MergeCellsFixed**	Allows users to set different merging criteria for fixed vs. scrollable cells.

***MergeCol**	Returns or sets whether a column will have its cells merged.
***MergeCompare**	Returns or sets the type of comparison used when merging cells.
***MergeRow**	Returns or sets whether a row will have its cells merged.
***MouseCol**	Returns the zero-based index of the column under the mouse pointer.
MouseIcon	Returns or sets a custom mouse icon. See the Visual Basic documentation for more information.
MousePointer	Returns or sets a value indicating the type of mouse pointer displayed when the mouse is over a particular part of an object at run time. See the Visual Basic documentation for more information.
***MouseRow**	Returns the zero-based index of the row under the mouse pointer.
***MultiTotals**	Returns or sets whether subtotals will be displayed in a single row when possible.
***NodeClosedPicture**	Returns or sets the picture to be used for closed outline nodes.
***NodeOpenPicture**	Returns or sets the picture to be used for open outline nodes.
***OLEDragMode**	Returns or sets whether the control can act as an OLE drag source, either automatically or under program control.
***OLEDropMode**	Returns or sets whether the control can act as an OLE drop target, either automatically or under program control.
***OutlineBar**	Returns or sets the type of outline bar that should be displayed.
***OutlineCol**	Returns or sets the column used to display the outline tree.
***OwnerDraw**	Returns or sets whether and when the control will fire the DrawCell event.
***Picture**	Returns a picture of the entire control.
***PicturesOver**	Returns or sets whether text and pictures should be overlaid in cells.
***PictureType**	Returns or sets the type of picture returned by the Picture property.
***Redraw**	Enables or disables redrawing of the **VSFlexGrid** control.
***RightCol**	Returns the zero-based index of the last column displayed in the control.
***RightToLeft**	Returns or sets whether text should be displayed from right to left on Hebrew and Arabic systems.

***Row**	Returns or sets the zero-based index of the current row.
***RowData**	Returns or sets a user-defined variant associated with the given row.
***RowHeight**	Returns or sets the height of the specified row in twips.
***RowHeightMax**	Returns or sets the maximum row height, in twips.
***RowHeightMin**	Returns or sets the minimum row height, in twips.
***RowHidden**	Returns or sets whether a row is hidden.
***RowIsVisible**	Returns whether a given row is currently within view.
***RowOutlineLevel**	Returns or sets the outline level for a subtotal row.
***RowPos**	Returns the top (y) coordinate of a row relative to the edge of the control, in twips.
***RowPosition**	Moves a given row to a new position.
***Rows**	Returns or sets the total number of rows in the control.
***RowSel**	Returns or sets the extent of a range of rows.
***RowStatus**	Returns or sets a value that indicates whether a row has been added, deleted, or modified.
***ScrollBars**	Returns or sets whether the control will display horizontal or vertical scroll bars.
***ScrollTips**	Returns or sets whether tool tips are shown while the user scrolls vertically.
***ScrollTipText**	Returns or sets the tool tip text shown while the user scrolls vertically.
***ScrollTrack**	Returns or sets scrolling should occur while the user moves the scroll thumb.
***SelectedRow**	Returns the position of a selected row when **SelectionMode** is set to flexSelectionListBox.
***SelectedRows**	Returns the number of selected rows when **SelectionMode** is set to flexSelectionListBox.
***SelectionMode**	Returns or sets whether the control will select cells in a free range, by row, by column, or like a listbox.
***SheetBorder**	Returns or sets the color used to draw the border around the sheet.
***ShowComboButton**	Returns or sets whether drop-down buttons are shown when editable cells are selected.
***Sort**	Sets a sorting order for the selected rows using the selected columns as keys.
***SortAscendingPicture**	Gets or sets a custom image to indicate the column sort direction.
***SortDescendingPicture**	Gets or sets a custom image to indicate the column sort direction.

***SubtotalPosition**	Returns or sets whether subtotals should be inserted above or below the totaled data.
***TabBehavior**	Returns or sets whether the tab key will move focus between controls (VB default) or between grid cells.
***Text**	Returns or sets the contents of the selected cell or range.
***TextArray**	Returns or sets the contents of a cell identified by a single index.
***TextMatrix**	Returns or sets the contents of a cell identified by its row and column coordinates.
***TextStyle**	Returns or sets 3D effects for displaying text in non-fixed cells.
***TextStyleFixed**	Returns or sets 3D effects for displaying text in fixed cells.
***TopRow**	Returns or sets the zero-based index of the topmost non-fixed row displayed in the control.
***TreeColor**	Returns or sets the color used to draw the outline tree.
***Value**	Returns the numeric value of the current cell.
***ValueMatrix**	Returns the numeric value of a cell identified by its row and column coordinates.
***Version**	Returns the version of the control currently loaded in memory.
***VirtualData**	Returns or sets whether all data is fetched from the data source at once or as needed.
***WallPaper**	Returns or sets a picture to be used as a background for the control's scrollable area.
***WallPaperAlignment**	Returns or sets the alignment of the WallPaper background picture.
***WordWrap**	Returns or sets whether text wider that its cell will wrap.

Methods

***AddItem**	Adds a row to the control.
***Archive**	Adds, extracts, or deletes files from a vsFlexGrid archive file.
***AutoSize**	Resizes column widths or row heights to fit cell contents.
***BindToArray**	Binds the grid to an array of variants to be used as storage.
***BuildComboList**	Returns a **ColComboList** string from data in a recordset.
***CellBorder**	Draws a border around and within the selected cells.

***CellBorderRange**	Similar to the **CellBorder** method, but allows the user to specify the range instead of using the selection CellBorderRange.
***Clear**	Clears the contents of the control. Optional parameters specify what to clear and where.
***Copy**	Copy selection to the Clipboard.
***Cut**	Cut selection to the Clipboard.
***DataRefresh**	Forces the control to re-fetch all data from its data source.
***Delete**	Deletes the selection.
***DragRow**	Starts dragging a row to a new position.
***EditCell**	Activates edit mode.
***FinishEditing**	Finishes any pending edits and returns the grid to browse mode.
***GetMergedRange**	Returns the range of merged cells that includes a given cell.
***GetNode**	Returns an outline node object for a given subtotal row.
***GetNodeRow**	Returns the number of a row's parent, first, or last child in an outline.
***GetSelection**	Returns the current selection ordered so that Row1 <= Row2 and Col1 <= Col2.
***LoadArray**	Loads the control with data from a Variant array or from another FlexGrid control.
***LoadGrid**	Loads grid contents and format from a file.
***LoadGridURL**	Loads grid contents and format from a URL (created with SaveGrid).
***OLEDrag**	Initiates an OLE drag operation.
***Outline**	Sets an outline level for displaying subtotals.
***Paste**	Pastes the selection from the Clipboard.
***PrintGrid**	Prints the grid on the printer.
Refresh	Forces a complete repaint of a form or control. See the Visual Basic documentation for more information.
***RemoveItem**	Removes a row from the control.
***SaveGrid**	Saves grid contents and format to a file.
***Select**	Selects a range of cells.
***Subtotal**	Inserts rows with summary data.

Events

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

***AfterDataRefresh**	Fired after reading data from the record source.
***AfterEdit**	Fired after the control exits cell edit mode.
***AfterMoveColumn**	Fired after a column is moved by dragging on the ExplorerBar.
***AfterMoveRow**	Fired after a row is moved by dragging on the ExplorerBar or using the **DragRow** method.
***AfterRowColChange**	Fired after the current cell (Row, Col) changes to a different cell.
***AfterScroll**	Fired after the control scrolls.
***AfterSelChange**	Fired after the selected range (**RowSel**, **ColSel**) changes.
***AfterSort**	Fired after a column is sorted by a click on the ExplorerBar.
***AfterUserFreeze**	Fired after the user changes the number of frozen rows or columns.
***AfterUserResize**	Fired after the user resizes a row or a column.
***BeforeCollapse**	Fired before the user expands or collapses one or more rows in an outline.
***BeforeDataRefresh**	Fired before reading data from the record source.
***BeforeEdit**	Fired before the control enters cell edit mode.
***BeforeMouseDown**	Fired before the control processes the **MouseDown** event.
***BeforeMoveColumn**	Fired before a column is moved by dragging on the ExplorerBar.
***BeforeMoveRow**	Fired before a row is moved by dragging on the ExplorerBar or using the **DragRow** method.
***BeforePageBreak**	Fired while printing the control to control page breaks.
***BeforeRowColChange**	Fired before the current cell (Row, Col) changes to a different cell.
***BeforeScroll**	Fired before the control scrolls.
***BeforeScrollTip**	Fired before a scroll tip is shown so you can set the **ScrollTipText** property.
***BeforeSelChange**	Fired before the selected range (**RowSel**, **ColSel**) changes.
***BeforeSort**	Fired before a column is sorted by a click on the **ExplorerBar**.
***BeforeUserResize**	Fired before the user starts resizing a row or column, allows cancel.
***CellButtonClick**	Fired after the user clicks a cell button.
***CellChanged**	Fired after a cell's contents change.
***ChangeEdit**	Fired after the text in the editor has changed.

Click	This event occurs when a user presses and then releases the button of mouse device over an object. See the Visual Basic documentation for more information.
***ComboCloseUp**	Fired before the built-in combobox closes up.
***ComboDropDown**	Fired before the built-in combobox drops down.
***Compare**	Fired when the **Sort** property is set to flexSortCustom, to allow custom comparison of rows.
DblClick	This event occurs when the user double-clicks an object.
***DrawCell**	Fired when the OwnerDraw property is set to allow custom cell drawing.
***EndAutoSearch**	Fired when the grid leaves AutoSearch mode.
***EnterCell**	Fired when a cell becomes active.
***Error**	Fired after a data-access error.
***FilterData**	Fired after a value is read and before a value is written to a recordset to allow custom formatting.
***GetHeaderRow**	Fired while printing the control to set repeating header rows.
KeyDown	This event occurs when a user presses a key while an object has the focus.
***KeyDownEdit**	Fired when the user presses a key in cell-editing mode.
KeyPress	This event occurs when the user presses and releases an ANSI key.
***KeyPressEdit**	Fired when the user presses a key in cell-editing mode.
KeyUp	Occurs when the user releases a key.
***KeyUpEdit**	Fired when the user presses a key in cell-editing mode.
***LeaveCell**	Fired before the current cell changes to a different cell.
MouseDown	Occur when the user presses (MouseDown) a mouse button.
MouseMove	Occurs when the user moves the mouse.
MouseUp	Occur when the user releases (MouseUp) a mouse button.
***OLECompleteDrag**	Fired after a drop to inform the source component that a drag action was either performed or canceled.
***OLEDragDrop**	Fired when a source component is dropped onto a target component.
***OLEDragOver**	Fired when a component is dragged over another.
***OLEGiveFeedback**	Fired after every **OLEDragOver** event to allow the source component to provide visual feedback to the user.

***OLESetCustomDataObject**	Fired after an OLE drag operation is started (manually or automatically), allows you to provide a custom DataObject.
***OLESetData**	Fired on the source component when a target component performs the **GetData** method on the source's DataObject object.
***OLEStartDrag**	Fired after an OLE drag operation is started (manually or automatically).
***RowColChange**	Fired when the current cell (**Row**, **Col**) changes to a different cell.
***SelChange**	Fired after the selected range (**RowSel**, **ColSel**) changes.
***SetupEditStyle**	Fired before the EditWindow is created, used to modify window styles.
***SetupEditWindow**	Fired after the EditWindow has been created and before it is displayed.
***StartAutoSearch**	Fired when the grid enters **AutoSearch** mode.
***StartEdit**	Fired when the control enters cell edit mode (after **BeforeEdit**).
***StartPage**	Fired before each page while the grid is being printed.
***ValidateEdit**	Fired before the control exits cell edit mode.

VSFlexGrid Properties

AccessibleDescription Property

Gets or sets the description of the control used by accessibility client applications.

Syntax

Property **AccessibleDescription** As String

Data Type

String

See Also

VSFlexGrid Control (page 73)

AccessibleName Property

Gets or sets the name of the control used by accessibility client applications.

Syntax

Property **AccessibleName** As String

Data Type

String

See Also

VSFlexGrid Control (page 73)

AccessibleRole Property

Gets or sets the role of the control used by accessibility client applications.

Syntax

Property **AccessibleRole** As Variant

Data Type

Variant

See Also

VSFlexGrid Control (page 73)

AccessibleValue Property

Gets or sets the value of the control used by accessibility client applications.

Syntax

Property **AccessibleValue** As String

Data Type

String

See Also

VSFlexGrid Control (page 73)

Aggregate Property

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

Syntax

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:

Constant	Value	Description
flexSTNone	0	Outline only, no aggregate values
flexSTClear	1	Clear all subtotals
flexSTSum	2	Sum
flexSTPercent	3	Percent of total sum

Constant	Value	Description
flexSTCount	4	Row count
flexSTAverage	5	Average
flexSTMax	6	Maximum
flexSTMin	7	Minimum
flexSTStd	8	Standard deviation
flexSTVar	9	Variance
flexSTStdPop	10	Standard Deviation Population
flexSTVarPop	11	Variance Population

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 continuous).

For example,

```
    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

See Also

VSFlexGrid Control (page 73)

AllowBigSelection Property

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

Syntax

[form!]VSFlexGrid.**AllowBigSelection**[= {**True** | **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

See Also

VSFlexGrid Control (page 73)

AllowSelection Property

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

Syntax

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

Remarks

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

See Also

VSFlexGrid Control (page 73)

AllowUserFreezing Property

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

Syntax

[form!]VSFlexGrid.**AllowUserFreezing**[= 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:

Constant	Value	Description
flexFreezeNone	0	The user cannot change the number of frozen rows or columns.
flexFreezeColumns	1	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

AllowUserFreezeSettings (Enumeration)

Default Value

flexFreezeNone (0)

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexResizeNone	0	The user may not resize rows or columns.
flexResizeColumns	1	The user may resize column widths.
flexResizeRows	2	The user may resize row heights.
flexResizeBoth	3	The user may resize column widths and row heights.
flexResizeBothUniform	4	The user may resize column widths and row heights. When a row height is resized, the new height is applied to all rows.

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:

```
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)

See Also

VSFlexGrid Control (page 73)

Appearance Property

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

Syntax

Property **Appearance** As AppearanceSettings

Remarks

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

Constant	Value	Description
flexFlat	0	Flat appearance
flex3D	1	3D appearance
flex3Dlight	2	3D Light appearance
flexXPThemes	3	If the application is theme-enabled, the control paints fixed cells using themes.

Notes on XP Themes

A visual style is included in the Windows XP release. In addition, other themes or visual styles are available in the Windows XP Plus Pack. You can use helper libraries and application programming interfaces (APIs) to incorporate a Windows XP visual style into an application with few code changes.

Windows XP applies a visual style to the non-client (frame and caption) area by default. To apply a visual style to common controls in the client area, you must use version 6 or later of the ComCtl32.dll file. ComCtl32.dll version 6 is not a redistributable system component. ComCtl32.dll version 6 contains both the user controls and the common controls. By default, applications use the controls that are defined in the User32.dll file. In addition, applications use the common controls that are defined in ComCtl32.dll version 5 by default.

To use the Windows XP visual styles from an application, you must add an application manifest file. This application manifest file should specify that ComCtl32.dll version 6 be used if it is available. One of the features that is included with this component is support for changing the appearance of controls in a window. Therefore, you must follow two steps to enable the Windows XP theme or visual style in Visual Basic 6.0:

1. Call the InitCommonControls function.
2. Add an application manifest file.

Example:

1. Call the InitCommonControls function:

You must call the InitCommonControls function in the Form_Initialize event:

```
Private Declare Sub InitCommonControls Lib "comctl32.dll" ()
Private Sub Form_Initialize()
    InitCommonControls
End Sub
```

Note: Do not call InitCommonControls in the **Form_Load** event. When you call InitCommonControls from the **Form_Load** event, the form cannot load.

2. Add a manifest file to your application:

You must add a file named YourApp.exe.manifest to the same folder as your executable file. For example, if your application is named Generic.exe, include a manifest file that is named Generic.exe.manifest. The application manifest file has Extensible Markup Language (XML) format similar to the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
 <assemblyIdentity
     version="1.0.0.0"
     processorArchitecture="X86"
     name="CompanyName.ProductName.YourApp"
     type="win32"
 />
<description>Your application description here.</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

After you place the application manifest file in the same folder as the executable file, you can run the compiled executable file to display the Windows XP visual style in the application.

Note: You cannot view visual styles when you run the compiled executable from the Visual Basic 6.0 Integrated Development Environment (IDE).

Although you can enable a Windows XP theme or visual style in Visual Basic 6.0 by calling InitCommonControls and by using an application manifest file, Microsoft does not officially support this feature.

If you enable a Windows XP theme in Visual Basic 6.0, you may encounter unexpected behavior. For example, if you place option buttons on top of a Frame control and then enable a Windows XP theme or visual style, the option buttons on the Frame control appear as black blocks when you run the executable file.

You can also embed the manifest into the executable file. In this case, you won't need the separate manifest file.

The manifest file should be embedded into the executable using a resource editor. The manifest should be embedded as a resource of type RT_RESOURCE and ID 1.

For details on this procedure, please refer to MSDN (Using Themes with Windows XP).

See Also

VSFlexGrid Control (page 73)

ArchiveInfo Property

Returns information from a VSFlex archive file.

Syntax

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

Remarks

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

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

ArcFileName as String

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

InfoType As ArchiveInfoSettings

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

Constant	Value	Description
ArcFileCount	0	Returns the number of files in the archive.
arcFileName	1	Returns the name of a file in the archive.
arcFileSize	2	Returns the original size of a file in the archive.
arcFileCompressedSize	3	Returns the compressed size of a file in the archive.
arcFileDate	4	The date of the last modification made to a file in the archive.

Index As Integer (optional)

This parameter specifies which file in the archive should be processed. It ranges from zero to the number of files in the archive minus 1. 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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexSearchNone	0	No auto search.
FlexSearchFromTop	1	When the user types, start searching from The first row.
FlexSearchFromCursor	2	When the user types, start searching from The current row.

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 **AutoSearchDelay** property.

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

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 **ComboSearch** property.

Data Type

AutoSearchSettings (Enumeration)

Default Value

flexSearchNone (0)

See Also

VSFlexGrid Control (page 73)

AutoSearchDelay Property

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

Syntax

[form!]VSFlexGrid.**AutoSearchDelay**[= value As Single]

Remarks

This property is only used when the **AutoSearch** property is set to a non-zero value.

Data Type

Single

Default Value

2

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexAutoSizeColWidth	0	Adjust column widths to accommodate the widest entry in each column.
flexAutoSizeRowHeight	1	Adjust row heights to accommodate the longest entry in each row.

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

BackColor Property

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

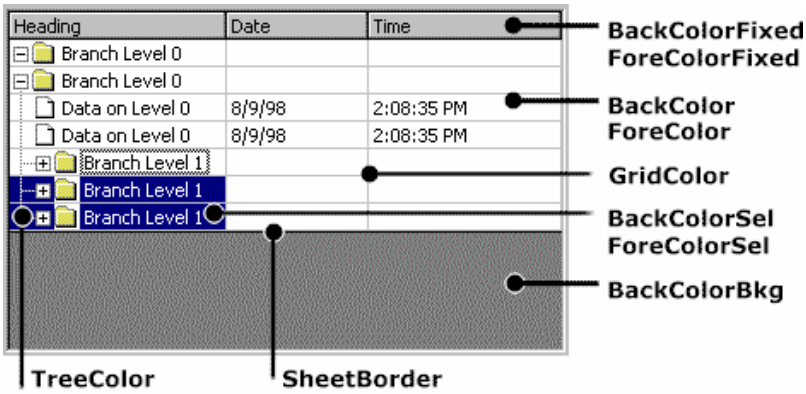
Syntax

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

Remarks

The **VSFlexGrid** control has several properties that allow you to customize its colors.

The picture below shows these properties and to which part of the control each one refers:



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

Data Type

Color

See Also

VSFlexGrid Control (page 73)

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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 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:

Constant	Value	Description
FlexcpText	0	Returns or sets the cell's text (or clip string for selections).
FlexcpTextStyle	1	Returns or sets the cell's text style (see the **CellTextStyle** property).
FlexcpAlignment	2	Returns or sets the cell's text alignment (see the **CellAlignment** property).
FlexcpPicture	3	Returns or sets the cell's picture (see the **CellPicture** property).
FlexcpPictureAlignment	4	Returns or sets the cell's picture alignment (see the **CellPictureAlignment** property).
FlexcpChecked	5	Returns or sets the state of the cell's check box (see the **CellChecked** property).
FlexcpBackColor	6	Returns or sets the cell's back color (see the **CellBackColor** property).
FlexcpForeColor	7	Returns or sets the cell's fore color (see the **CellForeColor** property).
flexcpFloodPercent	8	Returns or sets the cell's flood percent (see **CellFloodPercent** property).
flexcpFloodColor	9	Returns or sets the cell's flood color (see the **CellFloodColor** property).
flexcpFont	10	Returns or sets the cell's font.
flexcpFont*	11-17	Returns or sets properties of the cell's font (see the **CellFontName** property etc.).
flexcpValue	18	Returns the value of the cell's text (read-only).
flexcpTextDisplay	19	Returns the cell's formatted text (read only).
flexcpData	20	Returns or sets a Variant attached to the cell.
flexcpCustomFormat	21	Returns True if the cell has custom formatting (set to False to remove all custom formatting).
FlexcpLeft	22	Returns a cell's left coordinate, in twips, taking merging into account (read-only).
FlexcpTop	23	Returns a cell's top coordinate, in twips, taking merging into account (read-only).
FlexcpWidth	24	Returns a cell's width, in twips, taking merging into account (read-only).
FlexcpHeight	25	Returns a cell's height, in twips, taking merging into account (read-only).

Constant	Value	Description
flexcpVariantValue	26	Returns a double if the cell contains a numeric value or a string otherwise (read-only).
flexcpRefresh	27	Set to True to force a cell or range to be repainted.
FlexcpSort	28	Allows you to sort a range without changing the selection.

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:

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

you may write

```
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:

```
' 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

See Also

VSFlexGrid Control (page 73)

CellAlignment Property

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

Syntax

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

Remarks

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

Constant	Value
FlexAlignLeftTop	0
FlexAlignLeftCenter	1
FlexAlignLeftBottom	2
FlexAlignCenterTop	3
FlexAlignCenterCenter	4
FlexAlignCenterBottom	5
FlexAlignRightTop	6
FlexAlignRightCenter	7
FlexAlignRightBottom	8
FlexAlignGeneral	9

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

```
    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
            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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexNoCheckbox	0	The cell has no check box. This is the default setting.
FlexChecked	1	The cell has a check box that is checked.
FlexUnchecked	2	The cell has a check box that is not checked.
FlexTSChecked	3	Tri-state Checked (when clicked, changes state to FlexTSGrayed)
FlexTSGrayed	4	Tri-state Grayed (when clicked, changes state to FlexTSUnchecked)
FlexTSUnchecked	5	Tri-state Unchecked (when clicked, changes state to FlexTSChecked)

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 **FlexTSChecked** and **FlexTSUnchecked** settings cause the grid to display checked and unchecked boxes that are identical to the **FlexChecked** and **FlexUnchecked**. The difference is that the former are tri-state settings. They cause the check box to cycle through checked, grayed, and unchecked states instead of simply toggling between checked and unchecked.

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

```
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)

See Also

VSFlexGrid Control (page 73)

CellFloodColor Property

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

Syntax

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

Remarks

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

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

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

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

See Also

VSFlexGrid Control (page 73)

CellPicture Property

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

Syntax

[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

See Also

VSFlexGrid Control (page 73)

CellPictureAlignment Property

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

Syntax

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

Remarks

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

Constant	Value
flexPicAlignLeftTop	0
flexPicAlignLeftCenter	1
flexPicAlignLeftBottom	2
flexPicAlignCenterTop	3
flexPicAlignCenterCenter	4
flexPicAlignCenterBottom	5
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)

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexTextFlat	0	Draw text normally.
flexTextRaised	1	Draw text with a strong raised 3-D effect.
flexTextInset	2	Draw text with a strong inset 3-D effect.
flexTextRaisedLight	3	Draw text with a light raised 3-D effect.
flexTextInsetLight	4	Draw text with a light inset 3-D effect.

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

ClientWidth Property

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

Syntax

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

Remarks

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

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:

```
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

See Also

VSFlexGrid Control (page 73)

Clip Property

Returns or sets the contents of a range.

Syntax

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

Remarks

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

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value
FlexAlignLeftTop	0
FlexAlignLeftCenter	1
FlexAlignLeftBottom	2
FlexAlignCenterTop	3
FlexAlignCenterCenter	4
FlexAlignCenterBottom	5
FlexAlignRightTop	6
FlexAlignRightCenter	7

Constant	Value
FlexAlignRightBottom	8
FlexAlignGeneral	9

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

```
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)

See Also

VSFlexGrid Control (page 73)

ColComboList Property

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

Syntax

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

Remarks

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

Another difference is that the **ColComboList** property 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:

```
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:

```
    Debug.Print fg.Cell(flexcpText, fg.Row, 1),
fg.Cell(flexcpTextDisplay, fg.Row, 1)
        23          Part time
```

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

See Also

VSFlexGrid Control (page 73)

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:

```
Integer        212
String        Hello
Collection    World
Form1         Form1
```

Data Type

Variant

See Also

VSFlexGrid Control (page 73)

ColDataType Property

Returns or sets the data type for the column.

Syntax

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

Remarks

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

Constant	Value
FlexDTEmpty	0
FlexDTNull	1
FlexDTShort	2
FlexDTLong	3
FlexDTSingle	4
FlexDTDouble	5
FlexDTCurrency	6
FlexDTDate	7
FlexDTString	8
FlexDTDispatch	9
FlexDTError	10
FlexDTBoolean	11
FlexDTVariant	12
FlexDTUnknown	13
FlexDTDecimal	14
flexDTLong8	20
FlexDTStringC	30
FlexDTStringW	31

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

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

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:

```
    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:

```
    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 i%
        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:

```
Format              Content             Display
-----------         -----------         -------------
#,###.##            1234.56             1,234.56
#.###%              0.5432              54.320%
#,.##               125250              125.25
Currency            -1234.5             ($1,234.50)
Probably;Hardly     True                Probably
ddd, mmmm d, yyyy   04/07/1969          Fri, July 4, 1969
Medium Date         04/07/1969          04-Jul-69
Medium Time         04/07/1969          12:00 AM
```

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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.

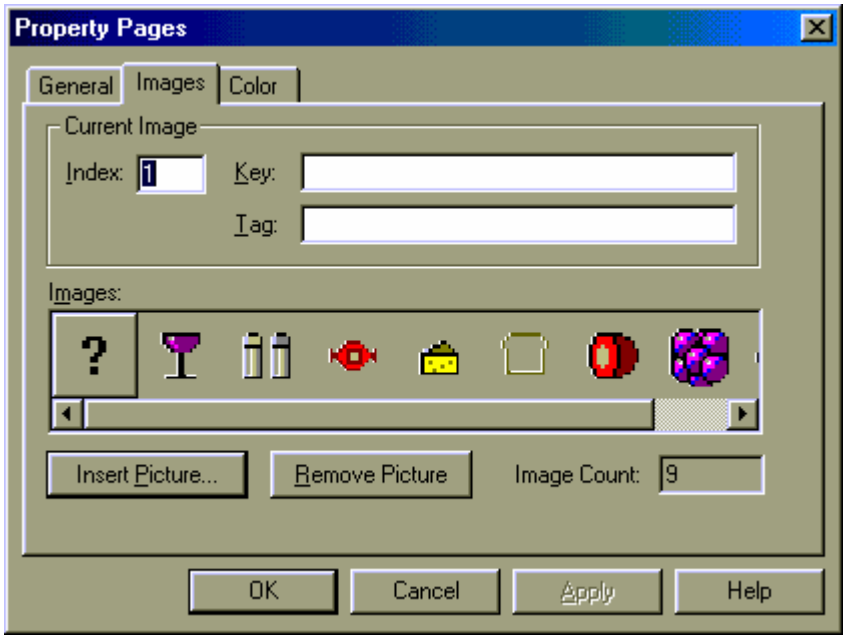
The mapping between numeric values and image list indices is continuous, starting from zero. Pictures are painted in cells that have numeric values between zero and the number of images on the list minus one. Cells that are empty, contain non-numeric values, or contain values that are outside the image 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:

Control Name	Description
Fg	A VSFlexGrid control (OLEDB version).
ImgList	An Image List control with nine images, as shown below.
DataProducts	An ADO data source control bound to the Products table from the NorthWind database.

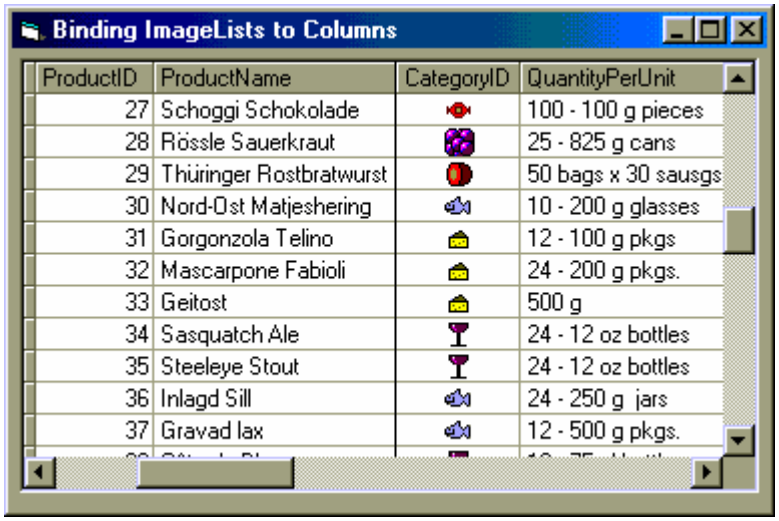
The **imgList** control should have nine images, corresponding to the product categories defined in the NorthWind database. In fact, there are only eight product categories, but they are numbered from one to eight. Thus an extra image with index zero had to be added to the list, even though it won't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



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

```
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:



Data Type

Long

See Also

VSFlexGrid Control (page 73)

ColIndent Property

Returns or sets the indentation of the given column, in twips.

Syntax

[form!]VSFlexGrid.**ColIndent**(Col As Long)[= value As Long]

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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:

```
    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
    Private Sub Form_Load()
        fg.ExplorerBar = flexExMove
        fg.TextMatrix(0, 1) = "Counter"
        fg.ColKey(1) = "Counter"
    End Sub
```

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*):

```
    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

See Also

VSFlexGrid Control (page 73)

ColIsVisible Property

Returns whether a given column is currently within view.

Syntax

val% = [form!]VSFlexGrid.**ColIsVisible**(Col As Long)

Remarks

The **ColIsVisible** and **RowIsVisible** properties are used to determine whether the specified column or row is within the visible area of the control or whether it has been scrolled off the visible part of the control.

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

```
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

See Also

VSFlexGrid Control (page 73)

ColKey Property

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

Syntax

[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 example, see the **ColIndex** property.

Data Type

String

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

ColPosition Property

Moves a given column to a new position.

Syntax

[form!]vsFlexGrid.**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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

```
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)

See Also

VSFlexGrid Control (page 73)

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)

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

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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):

```
fg.ColWidthMax = 2880
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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):

```
fg.ColWidthMin = 720
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

ComboCount Property

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

Syntax

val& = [form!]VSFlexGrid.**ComboCount**

Remarks

The **ComboCount** property allows you to customize editing when using drop-down or combo lists. It is valid only while the user is editing a value using a list.

For example, the code below traps the HOME key and selects a specific name instead of moving the cursor to the first item on the list. The example also illustrates the use of other related properties, **ComboItem** and **ComboIndex**.

```
Private Sub 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
                End If
        Next
        End If
    End Sub
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

ComboData Property

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

Syntax

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

Long

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

ComboList Property

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

Syntax

[form!]VSFlexGrid.**ComboList**[= value As String]

Remarks

The **ComboList** property controls the type of editor to be used when editing a cell. You may use a text box, drop-down list, drop-down combo, or an edit button to pop up custom editor forms.

To use the **ComboList** property, set the **Editable** property to **True**, and respond to the **BeforeEdit** event by setting the **ComboList** property to a string containing the proper options, described below.

Editing Options

To edit the cell using a regular text box, set the **ComboList** property to an empty string (""). You may also define an edit mask using the **EditMask** property.

To edit the cell using a drop-down list, set the **ComboList** property to a string containing the available options, separated by pipe characters ("|"). For example:

```
ComboList = "ListItem 1|ListItem 2".
```

To edit the cell using a drop-down combo, set the **ComboList** property to a string containing the available options, separated by pipe characters ("|") and starting with a pipe character. For example:

```
ComboList = "|ComboItem 1|ComboItem 2".
```

You can also use edit masks with drop-down combos using the **EditMask** property.

To display an edit button, set the **ComboList** property to a string containing an ellipsis (...). Edit buttons look like regular push buttons, aligned to the right of the cell, with an ellipsis as a caption. When the user clicks on the edit button, the control fires the **CellButtonClick** event. For example:

```
ComboList = "...".
```

List Syntax

In addition to the basic list syntax described above, you may create lists that define multi-column drop-downs and translated lists (lists where each item has an associated numerical value).

To define multi-column lists, separate columns with tab characters (Chr(9), or vbTab). When you define a multi-column combo, only one column is displayed in the cell (the others are visible only on the drop-down list). By default, the first column is the one that is displayed in the cell. To display a different column instead, add a string with the format "*nnn;" to the first item, where nnn is the zero-based index of the column to be displayed.

To create a translated list, attach a numerical value to each list item by adding a string with format "#xxx;" to the beginning of the row, where xxx is the numerical value. This value may be read while editing the cell using the **ComboData** property.

For example:

```
  s = "|#10*1;Getz"   & vbTab & "Stan"  & vbTab & "1 Sansome" & vbTab &
"972-4323" & _
      "|#20;Mindelis" & vbTab & "Nuno"  & vbTab & "2 5th"     & vbTab &
"972-2321" & _
      "|#30;Davis"    & vbTab & "Miles" & vbTab & "1 High"    & vbTab &
"345-2342" & _
      "|#40;Johnson"  & vbTab & "Bob"   & vbTab & "5 Hemlock" & vbTab &
"342-2321"
  fa.ComboList = s
```

The code above will display a drop-down combo with four columns. The items will have associated data values 10, 20, 30, and 40. The value displayed in the cells will be the one in column 1 (first name). Because the first character is a pipe, the box will be a drop-down combo, as opposed to a drop-down list box.

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

See Also

VSFlexGrid Control (page 73)

ComboSearch Property

Returns or sets whether combo lists should support smart searches.

Syntax

[form!]VSFlexGrid.**ComboSearch**[= ComboSearchSettings]

Remarks

When **ComboSearch** is set to a non-zero value, the control search for entries and highlight 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:

Constant	Value	Description
FlexCmbSearchNone	0	Don't search as the user types (use only the first letter).
FlexCmbSearchLists	1	Search drop-down lists, but not combo boxes.
FlexCmbSearchCombos	2	Search combo boxes, but not drop-down lists.
FlexCmbSearchAll	3	Search combo boxes and drop-down lists (this is the default setting).

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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexDMFree	0	This setting causes the data to be read from the database when the program starts, when the data source is refreshed, and when the user calls the DataRefresh method. Any direct changes to the database (edits and cursor movements) are ignored by the control. The flexDMFree setting is equivalent to the data binding implemented in the MSFlexGrid control.

Constant	Value	Description
flexDMBound	1	This setting causes the data in the database to be permanently synchronized with the control. The current row is linked to the database cursor, so when the Row property changes, the database cursor moves and vice-versa. All edits to the control contents are updated in the database and vice-versa. The flexDMBound setting is similar to the data binding implemented in the Microsoft DBGrid control.
flexDMBoundBatch	2	Similar to the flexDMBound setting, except the source recordset is not updated automatically after the user edits a cell.
flexDMBoundImmediate	3	Similar to the flexDMBound setting, except the source recordset is updated automatically after the user edits a cell.
flexDMBoundNoRowCount	4	Please provide description.

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:

Property	Limitation when Data-Bound
AddItem	The second parameter of the AddItem method, the position where the new row should be inserted, is ignored. New rows are always appended to the database.
Rows, Cols	These properties become read-only. You may add or remove records from the database one at a time using the AddItem and RemoveItem methods.
FixedRows, FixedCols	These properties become read-only at runtime. You need to decide how many fixed rows and columns you want at design time.
Sort, RowPosition	These properties are disabled. You may sort the database records by modifying the SQL statement in the data source 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)

See Also

VSFlexGrid Control (page 73)

DataSource Property

Returns or sets the data source.

Syntax

[form!]VSFlexGrid.**DataSource**[= DataSource]

Remarks

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

OLEDB/ADO version (VSFLEX8.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 Property window. When you select the **DataSource** property, you will get a drop-down list enumerating the sources available. These include sources defined with Visual Basic's Data Environment as well as any controls defined as data sources, such as the Microsoft ADO data control.

You may also set the **DataSource** property at runtime using the Visual Basic Set statement, as shown below:

```
' ADODC1 is a Microsoft ADO Data control
Set fg.DataSource = ADODC1
```

DAO version (VSFLEX8D.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

See Also

VSFlexGrid Control (page 73)

DragMode Property

Returns/sets a value that determines whether manual or automatic drag mode is used.

Syntax

[[form!]VSFlexGrid.**DragMode**[= value As Integer]

See Also

VSFlexGrid Control (page 73)

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:

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

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 **ComboList** and **ColComboList** properties. You may also specify an editing mask using the **EditMask** and **ColEditMask** properties. Set these properties in response to the **BeforeEdit** event.

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)

See Also

VSFlexGrid Control (page 73)

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 **ComboList** property.

If the same mask is used to edit all values in a column, use the **ColEditMask** property instead. This tends to simplify the code because you don't need to trap the **BeforeEdit** event.

When the user is done editing, the **ValidateEdit** event will be fired as usual. The *Cancel* parameter will be set to **True** if the mask was not filled out properly, so in most cases you don't event need to implement the handler. The default behavior ensures that only valid data will be entered.

The **EditMask** must be a string composed of the following symbols:

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

Localized characters

. localized decimal separator

, localized thousand separator

: localized time separator

/ localized date separator

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:

If a lowercase 'q' is present, the control edits in 'quiet' mode (no beeps on invalid characters),

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:

EditMask String	Description
"St\ate\; >LL"	Is a valid format. The 'a' and ';' characters are escaped and thus taken as literals. The '>' is used to ensure that the next two characters will be represented in uppercase.
"St\ate\; >LL;q;*"	Is a valid format. It is similar to the previous example, but the 'q' after the delimiter puts the control in quiet mode. An asterisk '*' is used as placeholder instead of the underscore, because that is the last character after the delimiter.
"St; >LL"	This is an invalid format. The mask itself is just "St" (the part to the left of the ';' delimiter. There are no wildcards, so the user can't type anything. If he could, the placeholder character would be "L" (last character after the ';' delimiter).
"; >LL"	This is an invalid format. The first character is a delimiter, so there is no real mask at all.

Data Type

String

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

EditSelLength Property

Returns or the number of characters selected in the editor.

Syntax

[form!]VSFlexGrid.**EditSelLength**[= value As Long]

Remarks

This property works in conjunction with the **EditSelStart** and **EditSelText** properties, while the control is in cell-editing mode.

Use these properties for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in the editor, or clearing text. Used in conjunction with the Visual Basic Clipboard object, these properties are useful for copy, cut, and paste operations.

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.

```
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
End Sub
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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.

```
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
End Sub
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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 sub-strings 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.

```
Private Sub fg_Click()
    fg.EditCell
    fg.EditSelStart = 5
    fg.EditSelLength = 3
    fg.EditSelText = "COW"
End Sub
```

Data Type

String

See Also

VSFlexGrid Control (page 73)

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:

```
Sub fg_ValidateEdit(ByVal Row As Long, ByVal Col As Long, Cancel As
Boolean)
        Dim c$
        ' different validation rules for each column
        Select Case Col

          ' column 1 only accepts strings
          Case 1
              c = Left$(fg.EditText, 1)
              If UCaseS(c) < "A" And UCase$(c) > "Z" Then
                  Beep: Cancel = True
                            End If

          ' column 2 only accepts numbers > 0
          Case 2
              If Val(fg.EditText) <= 0 Then
                  Beep: Cancel = True
                            End If
        End Select
    End Sub
```

Data Type

String

See Also

VSFlexGrid Control (page 73)

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

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexNoEllipsis	0	Long strings are truncated, no ellipsis characters are displayed.
FlexEllipsisEnd	1	Ellipsis characters are displayed at the end of long strings.
FlexEllipsisPath	2	Ellipsis characters are displayed in the middle of long strings.

Data Type

EllipsisSettings (Enumeration)

Default Value

flexNoEllipsis (0)

See Also

VSFlexGrid Control (page 73)

ExplorerBar Property

Returns or sets whether column headers are used to sort and/or move columns.

Syntax

[form!]VSFlexGrid.**ExplorerBar**[= ExplorerBarSettings]

Remarks

The **ExplorerBar** property allows users to use column headings to sort and move columns without any code. Valid settings are described below:

Constant	Value	Description
FlexExNone	0	No ExplorerBar. Fixed rows behave as usual.
FlexExSort	1	Users may sort columns by clicking on their headings.
FlexExMove	2	Users may move columns by dragging their headings.
FlexExSortAndMove	3	Users may sort and move columns.
FlexExSortShow	5	Users may sort columns by clicking on their headings. The control will show the current sorting order by drawing an arrow on the heading.
FlexExSortShowAndMove	7	Users may sort and move columns. The control will show the current sorting order by drawing an arrow on the heading.
FlexExMoveRows	8	Users may move rows by dragging their headings (fixed cells on the left of each row).

Note that these values are a combination of binary flags and are not sequential. You may combine settings using the Or operator. For example:

```
' allow sorting, moving rows, and moving columns
fg.ExplorerBar = flexExMoveRows Or flexExSortShowAndMove.
```

By default, the **ExplorerBar** works like the one in Microsoft's Internet 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**.

You must have at least one fixed row to be able to use the **ExplorerBar's** column moving and sorting capabilities, and at least one fixed column to use the row moving capability. To move rows by dragging non-fixed cells, see the **DragRow** method.

Data Type

ExplorerBarSettings (Enumeration)

Default Value

flexExNone (0)

See Also

VSFlexGrid Control (page 73)

ExtendLastCol Property

Returns or sets whether the last column should be adjusted to fit the control's width.

Syntax

[form!]VSFlexGrid.**ExtendLastCol**[= {**True** | **False**}]

Remarks

This property only affects painting. It does not modify the **ColWidth** property of the last column.

Data Type

Boolean

Default Value

False

See Also

VSFlexGrid Control (page 73)

FillStyle Property

Returns or sets whether changes to the **Text** or **Format** properties apply to the current cell or to the entire selection.

Syntax

[form!]VSFlexGrid.**FillStyle**[= FillStyleSettings]

Remarks

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

Constant	Value	Description
flexFillSingle	0	Setting the Text property or any of the cell formatting properties affects the current cell only.
flexFillRepeat	1	Setting the Text property or any of the cell formatting properties affects the entire selected range.

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

Note

The **FillStyle** property does not work over discontinuous selections if the **SelectionMode** property is set to *flexSelectionListBox*. For example, if you select rows 1, 4, and 10, only the selection that contains the current cell will be modified.

Data Type

FillStyleSettings (Enumeration)

Default Value

flexFillSingle (0)

See Also

VSFlexGrid Control (page 73)

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:

```
    ' 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

See Also

VSFlexGrid Control (page 73)

FindRowRegex Property

Returns the index of the row that contains a match or -1 if no match was found.

Syntax

Property **FindRowRegex**(*Pattern* As String, *Row* As Long, *Col* As Long) As Long

Remarks

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

Pattern As String

Pattern containing the regular expression to look for (see the **Pattern** property in the VBScript Regex object for regular expression syntax).

Row As Long

The row where the search should start (use -1 to start at the first scrollable row).

Col As Long

The column to search.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

FixedAlignment Property

Returns or sets the alignment for the fixed rows in a column.

Syntax

[form!]VSFlexGrid.**FixedAlignment**(*Col* As Long)[= AlignmentSettings]

Remarks

The **FixedAlignment** property behaves like the **ColAlignment** property except that it only affects the alignment of fixed cells. You can use this property to align headings differently than the rest of the cells.

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

For a list of valid settings, see the **ColAlignment** property.

Data Type

AlignmentSettings (Enumeration)

See Also

VSFlexGrid Control (page 73)

FixedCols Property

Returns or sets the number of fixed (non-scrollable) columns.

Syntax

[form!]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.

```
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

See Also

VSFlexGrid Control (page 73)

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.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

Flags Property

Gets or sets flags that affect the behavior of the control.

Syntax

[form!]VSFlexGrid.**Flags**[= ControlFlagsSettings]

Remarks

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

Constant	Value	Description
flexCFNone	0	No flags
flexCFV7SelectionEvents	1	Selection events (**BeforeSelChange**, **AfterSelChange**) are fired only when the Row/Col parameters refer to valid coordinates.
flexCFAutoClipboard	2	Causes the control to handle clipboard keys automatically Copy: Ctrl+C, Ctrl+Ins Cut: Ctrl+X, Shift+Del (if editable) Paste: Ctrl+V, Shift+Ins (if editable) Delete: Del (if editable)
flexCFNoEditIndent	4	Causes the edit control to use the old behavior: align to the left, no indent.
flexCFBindToBinaryFields	8	Causes the grid to show binary fields when it is bound to a data source.

By default, **VSFlexGrid** fires selection change events whenever the selection changes, even when the cell coordinates are invalid (e.g., the selection was removed by setting Row = -1, or the old row was removed with Rows = 1).

If you set fg.Flags = flexCFV7SelectionEvents, then the control will fire selection events (**BeforeSelChange**, **AfterSelChange**) only when the Row/Col parameters refer to valid coordinates. (This is the same behavior as in VSFlexGrid version 7.)

See Also

VSFlexGrid Control (page 73)

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). Chang 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:

```
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:

```
    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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexFocusNone	0	Do not show a focus rectangle.
flexFocusLight	1	Show a one-pixel wide focus rectangle.
flexFocusHeavy	2	Show a two-pixel wide focus rectangle.
flexFocusSolid	3	Show focus rectangle as a flat frame (the color is determined by the BackColorSel property).
flexFocusRaised	4	Show focus rectangle as a raised frame.
flexFocusInset	5	Show focus rectangle as an inset frame.

If a focus rectangle is drawn, then the current cell is painted using the regular background color, as in most spreadsheets and grids. Otherwise, the current cell is painted using the selected background color (**BackColorSel**).

Data Type

FocusRectSettings (Enumeration)

Default Value

flexFocusLight (1)

See Also

VSFlexGrid Control (page 73)

FontBold Property

Determines whether the font is bold.

Syntax

[form!]VSFlexGrid.**FontBold** [= {**True** | **False**}]

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

FontItalic Property

Determines whether the font is italicized.

Syntax

[form!]VSFlexGrid.**FontItalic** [= {**True** | **False**}]

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

FontName Property

Returns or sets the name of the font.

Syntax

[form!]VSFlexGrid.**FontName** [= value As String]

Data Type

String

See Also

VSFlexGrid Control (page 73)

FontSize Property

Determines the size of the font.

Syntax

[form!]VSFlexGrid.**FontSize** [= value As Single]

Data Type

Single

See Also

VSFlexGrid Control (page 73)

FontStrikethru Property

Determines the strikethru of the font.

Syntax

[form!]VSFlexGrid.**FontStrikethru** [= {**True** | **False**}]

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

FontUnderline Property

Determines the font is underlined.

Syntax

[form!]VSFlexGrid.**FontUnderline** [= {**True** | **False**}]

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

FontWidth Property

Returns or sets the width of the font, in points.

Syntax

[form!]VSFlexGrid.**FontWidth** [= value As Single]

Data Type

Single

See Also

VSFlexGrid Control (page 73)

ForeColor Property

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

Syntax

[form!]VSFlexGrid.**ForeColor**[= colorref&]

Remarks

This property works in conjunction with the **ForeColorFixed**, **ForeColorSel**, and **ForeColorFrozen** properties to specify the color used to draw text.

ForeColor	The color used to draw text in the scrollable area of the control.
ForeColorFixed	The color used to draw text in the fixed rows and columns (see the **FixedRows** and **FixedCols** properties).
ForeColorSel	The color used to draw text in selected cells (see the **HighLight** property).
ForeColorFrozen	The color used to draw frozen cells (see the **FrozenRows** and **FrozenCols** properties).

You may set the text color of individual cells using the **Cell**(*flexcpForeColor*) property.

Data Type

Color

See Also

VSFlexGrid Control (page 73)

ForeColorFixed Property

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

Syntax

[form!]VSFlexGrid.**ForeColorFixed**[= colorref&]

Remarks

This property works in conjunction with the **ForeColor**, **ForeColorSel**, and **ForeColorFrozen** properties to specify the color used to draw text.

ForeColor	The color used to draw text in the scrollable area of the control.
ForeColorFixed	The color used to draw text in the fixed rows and columns (see the **FixedRows** and **FixedCols** properties).
ForeColorSel	The color used to draw text in selected cells (see the **HighLight** property).
ForeColorFrozen	The color used to draw frozen cells (see the **FrozenRows** and **FrozenCols** properties).

You may set the text color of individual cells using the **Cell**(*flexcpForeColor*) property.

Data Type

Color

See Also

VSFlexGrid Control (page 73)

ForeColorFrozen Property

Returns or sets the foreground color of the frozen rows and columns.

Syntax

[form!]VSFlexGrid.**ForeColorFrozen**[= colorref&]

Remarks

This property works in conjunction with the **ForeColor**, **ForeColorSel**, and **ForeColorFixed** properties to specify the color used to draw text.

ForeColor	The color used to draw text in the scrollable area of the control.
ForeColorFixed	The color used to draw text in the fixed rows and columns (see the **FixedRows** and **FixedCols** properties).
ForeColorSel	The color used to draw text in selected cells (see the **HighLight** property).

ForeColorFrozen	The color used to draw frozen cells (see the **FrozenRows** and **FrozenCols** properties).

You may set the text color of individual cells using the **Cell**(*flexcpForeColor*) property.

Data Type

Color

See Also

VSFlexGrid Control (page 73)

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	The color used to draw text in the scrollable area of the control.
ForeColorFixed	The color used to draw text in the fixed rows and columns (see the **FixedRows** and **FixedCols** properties).
ForeColorSel	The color used to draw text in selected cells (see the **HighLight** property).
ForeColorFrozen	The color used to draw frozen cells (see the **FrozenRows** and **FrozenCols** properties).

You may set the text color of individual cells using the **Cell**(*flexcpForeColor*) property.

Data Type

Color

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

FrozenCols Property

Returns or sets the number of frozen (editable but non-scrollable) columns.

Syntax

[form!]VSFlexGrid.**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

See Also

VSFlexGrid Control (page 73)

FrozenRows Property

Returns or sets the number of frozen (editable but non-scrollable) rows.

Syntax

[form!]VSFlexGrid.**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 color specified by the **SheetBorder** property.

Data Type

Long

Default Value

0

See Also

VSFlexGrid Control (page 73)

GridColor Property

Returns or sets the color used to draw the grid lines between the non-fixed cells.

Syntax

[form!]VSFlexGrid.**GridColor**[= 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

See Also

VSFlexGrid Control (page 73)

GridColorFixed Property

Returns or sets the color used to draw the grid lines between the fixed cells.

Syntax

[form!]VSFlexGrid.**GridColorFixed**[= 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

See Also

VSFlexGrid Control (page 73)

GridLines Property

Returns or sets the type of lines to be drawn between non-fixed cells.

Syntax

[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:

Constant	Value	Description
flexGridNone	0	Do not draw grid lines between cells.
flexGridFlat	1	Draw flat lines with color and width determined by the GridColor and GridLineWidth properties.
flexGridInset	2	Draw inset lines between cells.
flexGridRaised	3	Draw raised lines between cells.
flexGridFlatHorz	4	Draw flat lines between rows, no lines between columns.
flexGridInsetHorz	5	Draw inset lines between rows, no lines between columns.
flexGridRaisedHorz	6	Draw raised lines between rows, no lines between columns.
flexGridSkipHorz	7	Draw an inset effect around every other row.
flexGridFlatVert	8	Draw flat lines between columns, no lines between rows.
flexGridInsetVert	9	Draw inset lines between columns, no lines between rows.
flexGridRaisedVert	10	Draw raised lines between columns, no lines between rows.
flexGridSkipVert	11	Draw an inset effect around every other column.
flexGridExplorer	12	Draw button-like frames around each cell.
flexGridExcel	13	Draw button-like frames around each cell, highlighting the headings for the current selection. This setting should only be applied to the GridLinesFixed property.
flexGridDataGrid	14	Please provide description.

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)

See Also

VSFlexGrid Control (page 73)

GridLinesFixed Property

Returns or sets the type of lines to be drawn between fixed cells.

Syntax

[form!]VSFlexGrid.**GridLinesFixed**[= GridStyleSettings]

Remarks

The **GridLinesFixed** and **GridColorFixed** properties determine the appearance of the grid lines displayed in the fixed area 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)

See Also

VSFlexGrid Control (page 73)

GridLineWidth Property

Returns or sets the width of the grid lines, in pixels.

Syntax

[form!]VSFlexGrid.**GridLineWidth**[= value As Integer]

Remarks

The **GridLineWidth** property determines the thickness, in pixels, of the grid lines when the **GridLineWidth** property or **GridLinesFixed** property is set to one of the flat styles (*flexGridFlat*, *flexGridFlatHorz*, *flexGridFlatVert*). Raised and inset grid lines have fixed width and cannot be changed.

Data Type

Integer

Default Value

1

See Also

VSFlexGrid Control (page 73)

GroupCompare Property

Returns or sets the type of comparison used when grouping cells.

Syntax

Property **GroupCompare** As MergeCompareSettings

Remarks

By default, the **Subtotal** method will group cells when there is an exact match between adjacent cells. This property allows you to control the comparison parameters (case-insensitive and trimming, like **MergeCompare**).

Data Type

MergeCompareSettings

See Also

VSFlexGrid Control (page 73)

HighLight Property

Returns or sets whether selected cells will be highlighted.

Syntax

[form!]VSFlexGrid.**HighLight**[= ShowSelSettings]

Remarks

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

Constant	Value	Description
FlexHighlightNever	0	Never highlight the selected range. Selected Ranges will not be visible to the user.
FlexHighlightAlways	1	Always highlight the selected range.
FlexHighlightWithFocus	2	Highlight the selected range only when the Control has the focus.

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)

See Also

VSFlexGrid Control (page 73)

IsCollapsed Property

Returns or sets whether an outline row is collapsed or expanded.

Syntax

[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:

Constant	Value	Description
FlexOutlineExpanded	0	Show all child nodes.
FlexOutlineSubtotals	1	Show child nodes but collapse them.
FlexOutlineCollapsed	2	Hide all child nodes.

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)

See Also

VSFlexGrid Control (page 73)

IsSearching Property

Returns a value that indicates whether the grid is in search mode.

Syntax

Property **IsSearching** As Boolean

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

IsSelected Property

Returns or sets whether a row is selected (for listbox-type selections).

Syntax

[form!]VSFlexGrid.**IsSelected**(*Row* As Long)[= {**True** | **False**}]

Remarks

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

To implement this type of row selection, you will typically set the **SelectionMode** property to *flexSelectionListBox*, which allows the user to select individual rows using the mouse or the keyboard, and to toggle the selection for a row by CTRL-clicking on it. If you set **SelectionMode** property to something other than *flexSelectionListBox*, you may still select and de-select rows using the **IsSelected** property, but the user will not be able to alter the selection with the mouse or keyboard (unless you write the code to do it).

You may enumerate the selected rows using the **SelectedRows** and **SelectedRow** properties.

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

MergeCells Property

Returns or sets whether cells with the same contents will be merged into a single cell.

Syntax

[form!]VSFlexGrid.**MergeCells**[= MergeSettings]

Remarks

The **MergeCells** property is used in conjunction with the **MergeRow**, **MergeCol**, and **MergeCompare** properties to control whether and how cells are merged for display. Merging cells allows you to display data in a clear, appealing way because it highlights groups of identical information. It also gives you flexibility to build tables similar to the ones you can create in HTML or using Microsoft Word, both of which support merged cells.

To create tables with merged cells, you must set the **MergeCells** property to a value other than *flexMergeNever*, and then set the **MergeRow** and **MergeCol** properties to **True** for the rows and columns you wish to merge (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:

Constant	Value	Description
flexMergeNever	0	Do not merge cells.
flexMergeFree	1	Merge any adjacent cells with same Contents (if they are on a row with **RowMerge** set to True or a column with **MergeCol** set to True).
flexMergeRestrictRows	2	Merge rows only if cells above are also Merged.
flexMergeRestrictColumns	3	Merge columns only if cells to the left are also merged.
flexMergeRestrictAll	4	Merge cells only if cells above or to the left are also merged.
flexMergeFixedOnly	5	Merge only fixed cells. This setting is Useful for setting up complex headers for the data and preventing the data itself from being merged.
flexMergeSpill	6	Allow long entries to spill into empty Adjacent cells.
flexMergeOutline	7	Allow entries in subtotal rows (outline nodes) to spill into empty adjacent cells.

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 *flexMergeSpill* setting. Notice how some cells with long entries spill into adjacent empty cells or get truncated if the adjacent cell is not empty:

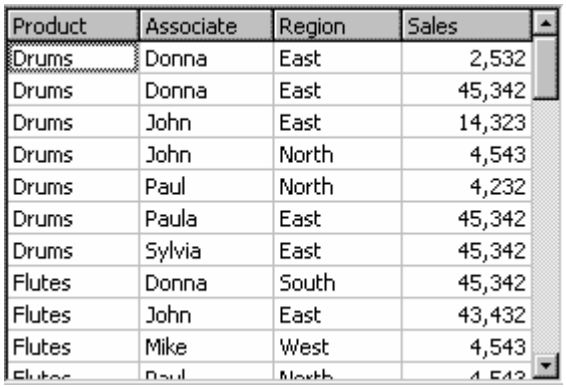
Product	Associate	Region	Sales
Motor Oil SAE37-666		Lower East Side	
Motor Oil SAE37-666		Upper West Side	
Motor Oil SAE37-666		Lower East Side	
Motor Oil SAE37-666		Upper West Side	
Motor Oil SAI	Paul	Upper West	4,232
Motor Oil SAI	Paula	Upper West	45,342
Drums	Sylvia	Lower East S	45,342
Flutes	Donna	South	45,342
Flutes	John	East	43,432
Flutes	Mike	West	4,543
Flutes	Paul	North	4,543

The *flexMergeOutline* setting is similar to *flexMergeSpill*, except it merges cells in subtotal rows with empty adjacent cells. This is good when you want to display only a node name on the subtotal rows (nodes) and data on the regular (non-node) rows.

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

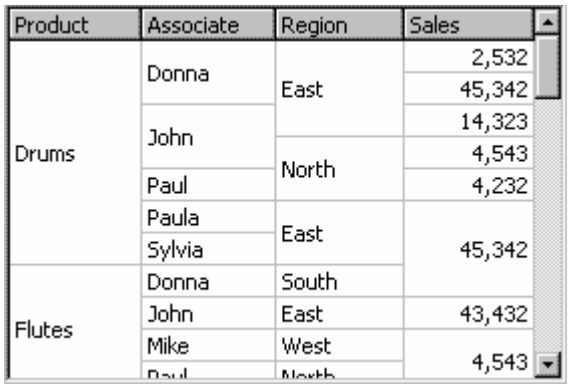
The examples below illustrate the difference.

```
' regular spreadsheet view
fg.MergeCells  = flexMergeNever
fg.MergeCol(0) = True: fg.MergeCol(1) = True: fg.MergeCol(2) = True
fg.MergeCol(3) = False
```



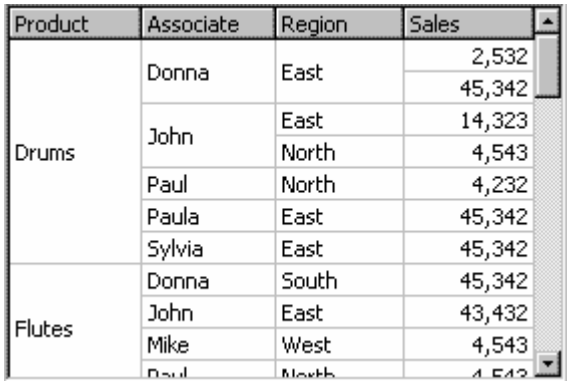
Product	Associate	Region	Sales
Drums	Donna	East	2,532
Drums	Donna	East	45,342
Drums	John	East	14,323
Drums	John	North	4,543
Drums	Paul	North	4,232
Drums	Paula	East	45,342
Drums	Sylvia	East	45,342
Flutes	Donna	South	45,342
Flutes	John	East	43,432
Flutes	Mike	West	4,543

```
' free merging: notice how the first region cell (East) merges
' across employees (Donna and John) to its left.
fg.MergeCells  = flexMergeFree
fg.MergeCol(0) = True: fg.MergeCol(1) = True: fg.MergeCol(2) = True
fg.MergeCol(3) = False
```



Product	Associate	Region	Sales
Drums	Donna	East	2,532
			45,342
	John		14,323
		North	4,543
	Paul		4,232
	Paula	East	45,342
	Sylvia		
Flutes	Donna	South	
	John	East	43,432
	Mike	West	4,543

```
' restricted merging: notice how the first region cell (East)
' no longer merges across employees to its left.
fg.MergeCells  = flexMergeRestrictAll
fg.MergeCol(0) = True: fg.MergeCol(1) = True: fg.MergeCol(2) = True
fg.MergeCol(3) = False
```



Product	Associate	Region	Sales
Drums	Donna	East	2,532
			45,342
	John	East	14,323
		North	4,543
	Paul	North	4,232
	Paula	East	45,342
	Sylvia	East	45,342
Flutes	Donna	South	45,342
	John	East	43,432
	Mike	West	4,543

Data Type

MergeSettings (Enumeration)

Default Value

flexMergeNever (0)

See Also

VSFlexGrid Control (page 73)

MergeCellsFixed Property

Allows users to set different merging criteria for fixed vs. scrollable cells.

Syntax

Property **MergeCellsFixed** As MergeSettings

Remarks

Setting **MergeCells** automatically sets **MergeCellsFixed** to the same value (for compatibility with existing code).

If the **MergeCells** and **MergeCellFixed** settings are different, files saved with **SaveGrid**(All) method will not be read by older versions of the control.

Data Type

MergeSettings

See Also

VSFlexGrid Control (page 73)

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 the **MergeCells** property 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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexMCExact	0	Cells are merged only if their contents match exactly.
FlexMCNoCase	1	Cells are merged if their contents match in a case-insensitive comparison.
FlexMCTrimNoCase	2	Cells are merged if their contents match after trimming blanks in a case-insensitive comparison.
FlexMCIncludeNulls	3	Empty cells are merged.

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

Data Type

MergeCompareSettings (Enumeration)

Default Value

flexMCExact (0)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

MouseCol Property

Returns the zero-based index of the column under the mouse pointer.

Syntax

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.

```
Private Sub fg_MouseMove(Button As Integer, Shift As Integer, X As
Single, Y As Single)

      fg.Cell(flexcpBackColor, fg.FixedRows, fg.FixedCols, fg.Rows - 1,
fg.Cols - 1) = vbDefault
      If fg.MouseRow >= fg.FixedRows And fg.MouseCol >= fg.FixedCols
Then
         fg.Cell(flexcpBackColor, fg.FixedRows, fg.MouseCol, fg.Rows -
1, fg.MouseCol) = vbYellow
      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

See Also

VSFlexGrid Control (page 73)

MouseRow Property

Returns the zero-based index of the row under the mouse pointer.

Syntax

val& = [form!]VSFlexGrid.**MouseRow**

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.

```
Private Sub fg_MouseMove(Button As Integer, Shift As Integer, X As
Single, Y As Single)

       fg.Cell(flexcpBackColor, fg.FixedRows, fg.FixedCols, fg.Rows -
1, fg.Cols - 1) =        vbDefault
       If fg.MouseRow >= fg.FixedRows And fg.MouseCol >= fg.FixedCols
Then
        fg.Cell(flexcpBackColor, fg.FixedRows, fg.MouseCol, fg.Rows -
1, fg.MouseCol) = vbYellow
       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

See Also

VSFlexGrid Control (page 73)

MultiTotals Property

Returns or sets whether subtotals will be displayed in a single row when possible.

Syntax

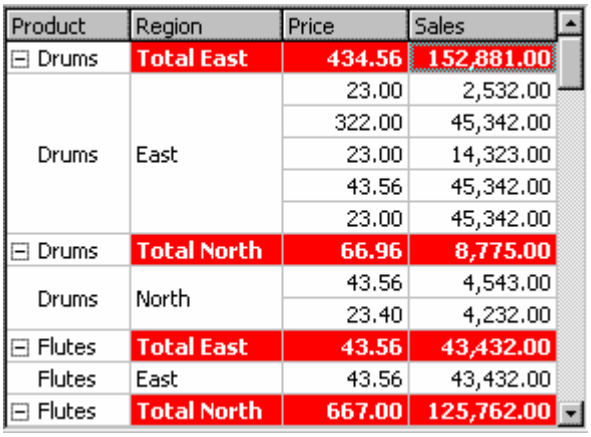
[form!]VSFlexGrid.**MultiTotals**[= {**True** | **False**}]

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.

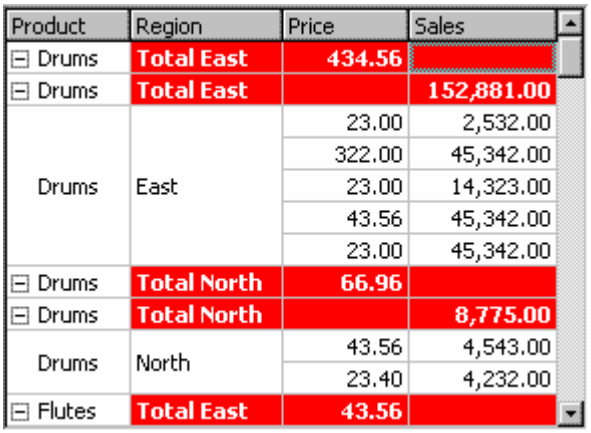
The examples below show the difference:

```
fg.MultiTotals = True
fg.Subtotal flexSTClear
fg.Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
fg.Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True
```



Product	Region	Price	Sales
⊟ Drums	**Total East**	**434.56**	**152,881.00**
Drums	East	23.00	2,532.00
		322.00	45,342.00
		23.00	14,323.00
		43.56	45,342.00
		23.00	45,342.00
⊟ Drums	**Total North**	**66.96**	**8,775.00**
Drums	North	43.56	4,543.00
		23.40	4,232.00
⊟ Flutes	**Total East**	**43.56**	**43,432.00**
Flutes	East	43.56	43,432.00
⊟ Flutes	**Total North**	**667.00**	**125,762.00**

```
fg.MultiTotals = False
fg.Subtotal flexSTClear
fg.Subtotal flexSTSum, 1, 2, , vbRed, vbWhite, True
fg.Subtotal flexSTSum, 1, 3, , vbRed, vbWhite, True
```



Product	Region	Price	Sales
⊟ Drums	**Total East**	**434.56**	
⊟ Drums	**Total East**		**152,881.00**
Drums	East	23.00	2,532.00
		322.00	45,342.00
		23.00	14,323.00
		43.56	45,342.00
		23.00	45,342.00
⊟ Drums	**Total North**	**66.96**	
⊟ Drums	**Total North**		**8,775.00**
Drums	North	43.56	4,543.00
		23.40	4,232.00
⊟ Flutes	**Total East**	**43.56**	

Data Type

Boolean

Default Value

True

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexOLEDragManual	0	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.

Constant	Value	Description
FlexOLEDragAutomatic	1	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.

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)

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexOleDropNone	0	The control does not accept OLE drops and displays the No Drop cursor.
flexOleDropManual	1	The target component triggers the OLE drop events, allowing the programmer to handle the OLE drop operation in code.
flexOleDropAutomatic	2	The control automatically accepts OLE drops if the DataObject object contains data in string or file formats.

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)

See Also

VSFlexGrid Control (page 73)

OutlineBar Property

Returns or sets the type of outline bar that should be displayed.

Syntax

[form!]VSFlexGrid.**OutlineBar**[= OutlineBarSettings]

Remarks

This property determines whether the control should display an outline bar when it is used 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 **OutlineBar** property are described below:

Constant	Value	Description
FlexOutlineBarNone	0	No outline bar.
FlexOutlineBarComplete	1	Complete outline tree plus button row on top. (Buttons are only displayed if the OutlineBar is on a fixed column).
FlexOutlineBarSimple	2	Complete outline tree, no buttons across the top.
FlexOutlineBarSymbols	3	Outline symbols but no connecting lines.
FlexOutlineBarCompleteLeaf	4	Similar to flexOutlineBarComplete, but empty nodes are displayed without symbols.
FlexOutlineBarSimpleLeaf	5	Similar to flexOutlineBarSimple, but empty nodes are displayed without symbols.
FlexOutlineBarSymbolsLeaf	6	Similar to flexOutlineBarSymbols, but empty nodes are displayed without symbols.

The following properties affect how each row is displayed on the **OutlineBar**:

Property Effect

If **IsSubtotal** is set to **True** and the row has children, the row is displayed as a node, with a collapse/expand symbol.

If **IsCollapsed** is 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.

The **RowOutlineLevel** property controls the indentation of the node. Higher values cause the node to be more indented.

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)

See Also

VSFlexGrid Control (page 73)

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.

Data Type

Long

Default Value

0

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexODNone	0	The control performs all drawing itself. The DrawCell event does not get fired at all. (This is the default setting.)
FlexODOver	1	The control draws the cell, and then fires the DrawCell event so the application can add text or graphics over the default cell contents.
FlexODContent	2	The control draws the cell background, Including any pictures, but no text. It fires the DrawCell event so the application can draw the text.
FlexODComplete	3	The control draws nothing at all in the cell. It fires the DrawCell event and the application is responsible for drawing the entire cell.
FlexODOverFixed	4	Similar to flexODOver, except only fixed cells are owner-drawn.
FlexODContentFixed	5	Similar to flexODContent, except only fixed cells are owner-drawn.
FlexODCompleteFixed	6	Similar to flexODComplete, except only fixed cells are owner-drawn.

For more details, see the **DrawCell** event.

Data Type

OwnerDrawSettings (Enumeration)

Default Value

flexODNone (0)

See Also

VSFlexGrid Control (page 73)

Picture Property

Returns a picture of the entire control.

Syntax

val% = [form!]VSFlexGrid.**Picture**

Remarks

This property returns a picture (bitmap) representation of the entire control, including rows and columns that are not visible on the screen. If you have a control with 1000 rows, for example, the bitmap will include all of them, and the picture will be huge. To create a picture of a part of the control, write a routine to hide all the elements you don't want to show, get the picture, and then restore the control.

To reduce memory requirements for the bitmap and increase speed, you may consider setting the **PictureType** property to *flexPictureMonochrome*. The picture will not look as nice, but it will require less memory.

The example below shows a routine that creates a picture of the current selection. It traps out-of-memory errors and automatically switches to monochrome mode if required.

```
Private Sub CopySelectionAsBitmap(fg As VSFlexGrid)

    ' save current settings
    Dim hl%, tr&, lc&, rd%
    hl = fg.HighLight
    tr = fg.TopRow
    lc = fg.LeftCol
    rd = fg.Redraw
    fg.HighLight = 0
    fg.Redraw = flexRDNone

    ' hide non-selected rows and columns
    Dim i&, r1&, c1&, r2&, c2&
    fg.GetSelection r1, c1, r2, c2
    For i = fg.FixedRows To fg.Rows - 1
       If i < r1 Or i > r2 Then fg.RowHidden(i) = True
    Next
    For i = fg.FixedCols To fg.Cols - 1
       If i < c1 Or i > c2 Then fg.ColHidden(i) = True
    Next

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

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

    ' restore control
    fg.RowHidden(-1) = False
    fg.ColHidden(-1) = False
    fg.TopRow = tr
    fg.LeftCol = lc
    fg.HighLight = hl
    fg.Redraw = rd
End Sub
```

Data Type

Picture

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexPictureColor	0	Causes the Picture property to generate a color bitmap of the control. This mode creates high quality pictures, but they may be quite large and slow to manipulate.
FlexPictureMonochrome	1	Causes the Picture property to generate a monochrome bitmap of the control. This mode creates lower quality pictures which consume less memory and are faster to manipulate.
flexPictureEnhMetafile	2	Please provide description.

For more details and sample code, see the **Picture** property.

Data Type

PictureTypeSettings (Enumeration)

See Also

VSFlexGrid Control (page 73)

Redraw Property

Enables or disables redrawing of the **VSFlexGrid** control.

Syntax

[form!]VSFlexGrid.**Redraw**[= RedrawSettings]

Remarks

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

Constant	Value	Description
FlexRDNone	0	The grid does not repaint itself.
FlexRDDirect	1	The grid paints its contents directly on the screen. This is the fastest repaint mode, but there occasionally it may cause a little flicker.
FlexRDBuffered	2	The grid paints its contents on an off-screen buffer, then transfers the complete image to the screen. This mode is slightly slower than flexRDDirect, but it eliminates flicker.
True	-1	Equivalent to flexRDDirect. This setting is allowed for compatibility with previous versions of the control.

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.

```
Sub UpdateGrid()
  fa.Redraw = flexRDNone ' suspend redrawing/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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

RightToLeft Property

Returns or sets whether text should be displayed from right to left on Hebrew and Arabic systems.

Syntax

[form!]VSFlexGrid.**RightToLeft** [= {**True** | **False**}]

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

Row Property

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

See Also

VSFlexGrid Control (page 73)

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:

```
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:

```
Integer        212
String        Hello
Collection    World
Form1         Form1
```

You can search for rows that contain specific **RowData** values using the **FindRow** property.

Data Type

Variant

See Also

VSFlexGrid Control (page 73)

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 the **AutoSize** method.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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 the **AutoSize** method to automatically set row heights, to prevent some rows from becoming too large.

See also the **ColWidthMin**, **ColWidthMax**, and **RowHeightMin** properties.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

RowHeightMin Property

Returns or sets the minimum row height, in twips.

Syntax

[form!]VSFlexGrid.**RowHeightMin**[= value As Long]

Remarks

Set this property to a non-zero value to set a minimum limit to row heights. This is often useful when you use the **AutoSize** method to automatically set row heights, to prevent some rows from becoming too short. This may also be useful when you want to use small fonts, but don't want the rows to become short.

See also the **ColWidthMin**, **ColWidthMax**, and **RowHeightMax** properties.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

RowHidden Property

Returns or sets whether a row is hidden.

Syntax

[form!]VSFlexGrid.**RowHidden**(*Row* As Long)[= {**True** | **False**}]

Remarks

Use the **RowHidden** property to hide and display rows. This is a better approach than setting the row's **RowHeight** property to zero, 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

See Also

VSFlexGrid Control (page 73)

RowIsVisible Property

Returns whether a given row is currently within view.

Syntax

val% = [form!]VSFlexGrid.**RowIsVisible**(*Row* As Long)

Remarks

The **ColIsVisible** and **RowIsVisible** properties are used to determine whether the specified column or row is within the visible area of the control or whether it has been scrolled off the visible part of the control.

If a row has zero height or is hidden but is within the scrollable area, **RowIsVisible** will return **True**.

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

Data Type

Boolean

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexrsNew	0	When the row is created.
flexrsUpdated	1	When the program modifies a row by writing to it.
flexrsModified	2	When the user modifies a row by editing it.
flexrsDeleted	3	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)).

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)

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexScrollBarNone	0	Do not display any scrollbars.
FlexScrollBarHorizontal	1	Display a horizontal scrollbar.
FlexScrollBarVertical	2	Display a vertical scrollbar.
FlexScrollBarBoth	3	Display horizontal and vertical scrollbars.

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.

Data Type

ScrollBarsSettings (Enumeration)

Default Value

flexScrollBarBoth (3)

See Also

VSFlexGrid Control (page 73)

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:

```
    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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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 **SelectedRows** property.

Data Type

Long

See Also

VSFlexGrid Control (page 73)

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	Value	Description
flexSelectionFree	0	Allows selections to be made as usual, spreadsheet-style.
flexSelectionByRow	1	Forces selections to span entire rows. Useful for implementing record-based displays.
flexSelectionByColumn	2	Forces selections to span entire columns. Useful for selecting ranges for a chart or fields for sorting.
flexSelectionListBox	3	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.

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.

For example:

```
Private Sub fg_Click()

  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)

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

ShowComboButton Property

Returns or sets whether drop-down buttons are shown when editable cells are selected.

Syntax

[form!]VSFlexGrid.**ShowComboButton**[= **ShowButtonSettings**]

Remarks

The **ShowComboButton** property allows you to set whether drop-down buttons are shown when editable cells are selected. To use the **ShowComboButton** property the cell must have values listed in an associated **ComboList**.

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

Constant	Value	Description
flexSBEditing	0	The user may initiate the drop-down button by typing into the current cell or by double-clicking it with the mouse
flexSBFocus	1	The user may initiate the drop-down button by clicking the current cell with the mouse.
flexSBAlways	2	The control displays drop-down buttons automatically.

Note that the user may edit the cells directly, by clicking the button with the mouse when the **Editable** property is set to *flexEDKbdMouse*. If the **Editable** property is set to *flexEDNone*, then the cell contents cannot be edited by the user.

The following code prepares the grid to show a button for the combo list as soon as a cell is selected:

```
    With fg
      .Editable = flexEDKbdMouse
      .ComboList = "Item 1|Item 2|Item 3"
      .ShowComboButton = flexSBFocus
  End With
```

Data Type

ShowButtonSettings (Enumeration)

Default Value

flexSBFocus

See Also

VSFlexGrid Control (page 73)

Sort Property

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

Syntax

[form!]VSFlexGrid.**Sort** = SortSettings

Remarks

The **Sort** property allows you to sort a range or rows in ascending or descending order based on the values in one or more columns.

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

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

The sorting algorithm used by the **VSFlexGrid** control is "stable": this means that the sorting keeps the relative order of records when the sorting key is the same. For example, if you sort a list of files by name, then by extension, file names will still be sorted within each extension group.

Valid settings for the **Sort** property are described below:

Constant	Value	Description
flexSortNone	0	Ignore this column when sorting. This setting is useful when you assign it to a column's Cols property, then set Sort to flexSortUseColSort.
flexSortGenericAscending	1	Sort strings and numbers in ascending order.
flexSortGenericDescending	2	Sort strings and numbers in descending order.
flexSortNumericAscending	3	Sort numbers in ascending order.
flexSortNumericDescending	4	Sort numbers in descending order.
flexSortStringNoCaseAscending	5	Sort strings in ascending order, ignoring capitalization.
flexSortStringNoCaseDescending	6	Sort strings in descending order, ignoring capitalization.
flexSortStringAscending	7	Sort strings in ascending order.
FlexSortStringDescending	8	Sort strings in descending order.
flexSortCustom	9	Fire a Compare event and use the return value to sort the columns.
flexSortUseColSort	10	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.

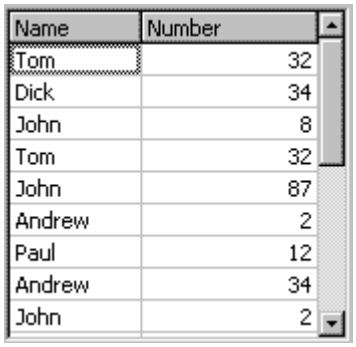
The *flexSortCustom* is the most flexible setting. It fires a **Compare** event that allows you to compare rows in any way you wish, using any columns in any order. However, it is also much slower than the others, so it should be used only when really necessary or when the grid has only a few rows. If you want to sort based on arbitrary criteria (for example, "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:

```
fg.ColDataType(i) = flexDTDate
fg.Col = i
fg.Sort = flexSortGenericAscending
```

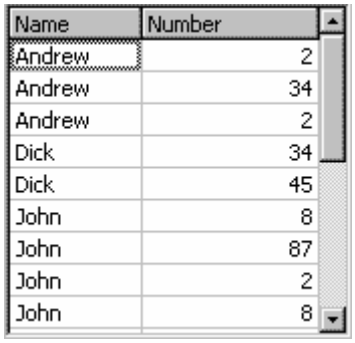
The example below shows how the **Sort** property is used:

```
' 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"
```



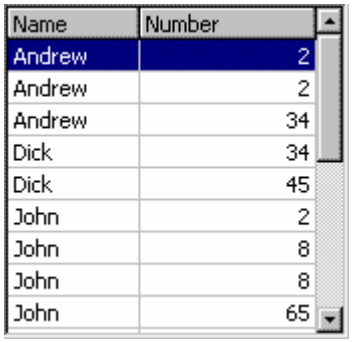
Name	Number
Tom	32
Dick	34
John	8
Tom	32
John	87
Andrew	2
Paul	12
Andrew	34
John	2

```
' sort by name
fg.Select 1, 0
fg.Sort = flexSortGenericAscending
```



Name	Number
Andrew	2
Andrew	34
Andrew	2
Dick	34
Dick	45
John	8
John	87
John	2
John	8

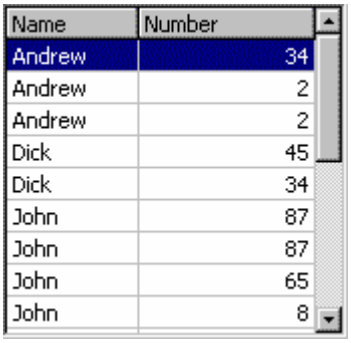
```
' sort by name and number
fg.Select 1, 0, 1, 1
fg.Sort = flexSortGenericAscending
```



Name	Number
Andrew	2
Andrew	2
Andrew	34
Dick	34
Dick	45
John	2
John	8
John	8
John	65

If you want to select different sorting orders for each column, either sort them one by one or use the **ColSort** property and the *flexSortUseColSort* setting. Here is an example that sorts the names in ascending order and the numbers in descending order:

```
fg.ColSort(0)=flexSortGenericAscending
fg.ColSort(1) = flexSortGenericDescending
fg.Select 1, 0, 1, 1
fg.Sort = flexSortUseColSort
```



Name	Number
Andrew	34
Andrew	2
Andrew	2
Dick	45
Dick	34
John	87
John	87
John	65
John	8

Data Type

SortSettings (Enumeration)

Note

The **Sort** property honors outline structures. It will only sort data rows, and will not scramble nodes.

See Also

VSFlexGrid Control (page 73)

SortAscendingPicture Property

Gets or sets a custom image to indicate the column sort direction.

Syntax

Property **SortAscendingPicture** As IPictureDisp

Data Type

IPictureDisp

See Also

VSFlexGrid Control (page 73)

SortDescendingPicture Property

Gets or sets a custom image to indicate the column sort direction.

Syntax

Property **SortDescendingPicture** As IPictureDisp

Data Type

IPictureDisp

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexSTBelow	0	Subtotals are inserted below the data. This setting creates grids that look like reports.
flexSTAbove	1	Subtotals are inserted above the data. This setting creates grids that look like outline trees (this is the default value).

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:

```
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)

See Also

VSFlexGrid Control (page 73)

TabBehavior Property

Returns or sets whether the TAB key will move focus between controls (VB default) or between grid cells.

Syntax

[form!]VSFlexGrid.**TabBehavior**[= TabBehaviorSettings]

Remarks

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

Constant	Value	Description
flexTabControls	0	TAB key is used to move to the next or previous control on the form.
flexTabCells	1	TAB key is used to move to the next or previous cell on the control.

The example below sets the TAB key to move to the next control when in the last cell of the grid:

```
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
```

Data Type

TabBehaviorSettings (Enumeration)

Default Value

flexTabControls (0)

See Also

VSFlexGrid Control (page 73)

Text Property (VSFlexGrid)

Returns or sets the contents of the selected cell or range.

Syntax

[form!]VSFlexGrid.**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, 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**:

```
    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

See Also

VSFlexGrid Control (page 73)

TextArray Property

Returns or sets the contents of a cell identified by a single index.

Syntax

[form!]VSFlexGrid.**TextArray**(*Index* As Long)[= value As String]

Remarks

This property is provided for backward compatibility with earlier versions of this control. New applications should use the **Cell**(*flexcpText*) or **TextMatrix** properties.

The following code places the text "Apple" in the second row and second column of a five column grid:

```
fg.TextArray(6) = "Apple"
```

Data Type

String

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexTextFlat	0	Draw text normally.
flexTextRaised	1	Draw text with a strong raised 3-D effect.
flexTextInset	2	Draw text with a strong inset 3-D effect.

Constant	Value	Description
flexTextRaisedLight	3	Draw text with a light raised 3-D effect.
flexTextInsetLight	4	Draw text with a light inset 3-D effect.

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)

See Also

VSFlexGrid Control (page 73)

TextStyleFixed Property

Returns or sets 3D effects for displaying text in fixed cells.

Syntax

[form!]VSFlexGrid.**TextStyleFixed**[= TextStyleSettings]

Remarks

Valid settings for this property are the same as those for the **TextStyle** property.

The example below prints text with a raised 3-D effect in fixed cells:

```
fg.TextStyleFixed = flexTextRaised
```

Data Type

TextStyleSettings (Enumeration)

Default Value

flexTextFlat (0)

See Also

VSFlexGrid Control (page 73)

TopRow Property

Returns or sets the zero-based index of the topmost non-fixed row displayed in the control.

Syntax

[form!]VSFlexGrid.**TopRow**[= value As Long]

Remarks

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

See Also

VSFlexGrid Control (page 73)

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

See Also

VSFlexGrid Control (page 73)

Value Property

Returns the numeric value of the current cell.

Syntax

val# = [form!]VSFlexGrid.**Value**

Remarks

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

For example, if the current cell contains the string "$(1,234.56)", the **Value** property will return the value -1234.56. 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 **Cell**(*flexcpValue*) or the **ValueMatrix** properties.

Note

This property is not an expression evaluator. If the current cell contains the string "2+2", for example, the **Value** property will return 2 instead of 4. The Visual Basic statement Val("2+2") also returns 2.

Data Type

Double

See Also

VSFlexGrid Control (page 73)

ValueMatrix Property

Returns the numeric value of a cell identified by its row and column coordinates.

Syntax

val# = [form!]VSFlexGrid.**ValueMatrix**(*Row* As Long, *Col* As Long)

Remarks

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

The following code outputs the value of an arbitrary cell to the debug window:

```
Debug.Print fg.ValueMatrix(1, 1)
```

Data Type

Double

See Also

VSFlexGrid Control (page 73)

Version Property (VSFlexGrid)

Returns the version of the control currently loaded in memory.

Syntax

val% = [form!]VSFlexGrid.**Version**

Remarks

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

The version number is a three-digit integer where the first digit represents the major version number and the last two represent the minor version number. For example, version 7.00 returns 700.

The example below displays the version number of the control:

```
Private Sub Command1_Click()
     MsgBox "VSFlexGrid Version " & fg.Version, vbOKOnly

End Sub
```

Note

If you want your application to run correctly with older builds of VSFlex8, check the **Version** property before using these new features. For example:

```
if fg.Version >= 801 then fg.Copy
```

Data Type

Integer

See Also

VSFlexGrid Control (page 73)

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:

```
    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

See Also

VSFlexGrid Control (page 73)

WallPaper Property

Returns or sets a picture to be used as a background for the control's scrollable area.

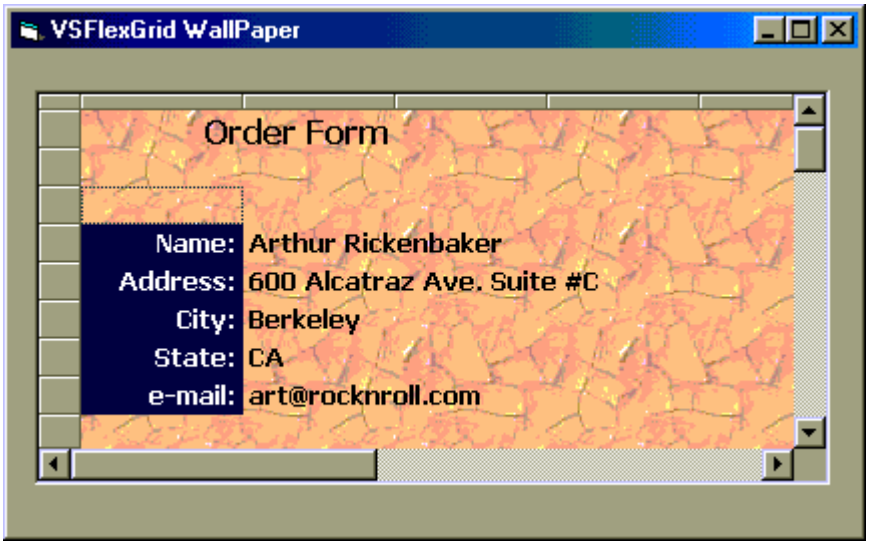
Syntax

[form!]VSFlexGrid.**WallPaper**[= Picture]

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 picture below shows a grid with wallpaper:



When creating or selecting pictures for use as wallpaper, consider the following points:

The **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 **ForeColor** property to a light value so the grid contents will be clearly visible.

When using wallpaper, remember to set the **BackColor** property to a value that matches the predominant color on the wallpaper picture. Even with wallpaper, the grid's **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:

```
Private Sub Form_Load()
    fg.WallPaper = LoadPicture("c:\temp\shark.gif")
    fg.WallPaperAlignment = flexPicAlignCenterCenter

End Sub
```

Data Type

Picture

See Also

VSFlexGrid Control (page 73)

WallPaperAlignment Property

Returns or sets the alignment of the WallPaper background picture.

Syntax

[form!]VSFlexGrid.**WallPaperAlignment**[= 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)

See Also

VSFlexGrid Control (page 73)

WordWrap Property

Returns or sets whether text wider that its cell will wrap.

Syntax

[form!]VSFlexGrid.**WordWrap** [= {**True** | **False**}]

Default Value

False

See Also

VSFlexGrid Control (page 73)

VSFlexGrid Methods

AddItem Method

Adds a row to the control.

Syntax

[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:

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

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.

See Also

VSFlexGrid Control (page 73)

Archive Method

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

Syntax

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

Remarks

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

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

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

ArcFileName As String

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

FileName As String

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

Action As ArchiveSettings

This parameter specifies the action to perform on the archive. Valid settings are:

Constant	Value	Description
arcAdd	0	Adds the file FileName to the archive ArcFileName, compressing it. If the archive file does not exist, it is created. If the file is already present in the archive, it is overwritten with the new contents.
arcStore	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
```

See Also

VSFlexGrid Control (page 73)

AutoSize Method

Resizes column widths or row heights to fit cell contents.

Syntax

[form!]VSFlexGrid.**AutoSize** *Col1* As Long, [*Col2* As Long], [*Equal* As Boolean], [*ExtraSpace* As Single]

Remarks

The parameters for the AutoSize method are described below:

Col1 As Long, *Col2* As Long

Specify the first and last columns to be resized so their widths fit the widest entry in each column. The valid range for these parameters is between 0 and Cols -1. Col2 is optional. If it is omitted, only Col1 is resized.

Equal As Boolean (optional)

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

ExtraSpace As 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.

See Also

VSFlexGrid Control (page 73)

BindToArray Method

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

Syntax

[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.

```
' ** 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.

See Also

VSFlexGrid Control (page 73)

BuildComboList Method

Returns a **ColComboList** string from data in a recordset.

Syntax

[form!]VSFlexGrid.**BuildComboList** *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 **ColComboList**.

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

The parameters for the **BuildComboList** 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 **BuildComboList** 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
```

See Also

VSFlexGrid Control (page 73)

CellBorder Method

Draws a border around and within the selected cells.

Syntax

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

Remarks

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

Color As 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.

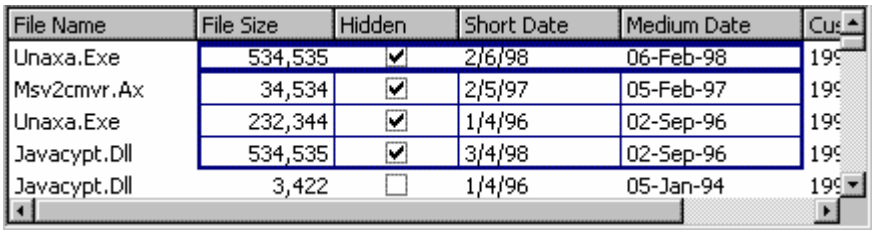
Vertical, *Horizontal* As Integer

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 width of the existing border.

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

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

End Sub
```

The result looks like this:



See Also

VSFlexGrid Control (page 73)

CellBorderRange Method

Similar to the **CellBorder** method, but allows the user to specify the range instead of using the selection CellBorderRange.

Syntax

Sub **CellBorderRange**(Row1 As Long, Col1 As Long, Row2 As Long, Col2 As Long, Color As OLE_COLOR, Left As Integer, Top As Integer, Right As Integer, Bottom As Integer, Vertical As Integer, Horizontal As Integer)

Remarks

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

Row1, Col1, Row2, Col2 As Long

These parameters specify the range where the border will be applied.

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.

See Also

VSFlexGrid Control (page 73)

Clear Method

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:

Constant	Value	Description
flexClearEverywhere	0	Clear everywhere (this is the default setting).
flexClearScrollable	1	Clear scrollable region (excludes fixed rows and columns).
flexClearSelection	2	Clear current selection.

What As ClearWhatSettings (optional)

Defines what part of the grid's information will be cleared. Valid settings for this parameter are:

Constant	Value	Description
FlexClearEverything	0	Clear cell text, formatting and custom data (this is the default setting).
FlexClearText	1	Clear text only.
FlexClearFormatting	2	Clear custom formatting only (including pictures and cell data).
FlexClearData	3	Clears all custom data (row, column, and cell data).

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:

```
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:

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

See Also

VSFlexGrid Control (page 73)

Copy Method

Copy selection to the Clipboard.

Syntax

Sub **Copy()**

See Also

VSFlexGrid Control (page 73)

Cut Method

Cut selection to the Clipboard.

Syntax

Sub **Cut()**

See Also

VSFlexGrid Control (page 73)

DataRefresh Method

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

Syntax

[form!]VSFlexGrid.**DataRefresh**

See Also

VSFlexGrid Control (page 73)

Delete Method

Deletes the selection.

Syntax

Sub **Delete()**

See Also

VSFlexGrid Control (page 73)

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.

```
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 Then
              Cancel = True
              Dim r%
              r = .Row
              .Cell(flexcpBackColor, r, 1, r, .Cols - 1) = vbRed
              r = .DragRow(r)
                 .Cell(flexcpCustomFormat, r, 1, r, .Cols - 1) = False
              Debug.Print "Dragged to "; r
            End If
          End With
End Sub
```

See Also

VSFlexGrid Control (page 73)

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.

```
    Sub fg_MouseDown(Button As Integer, Shift As Integer, X!, Y!)
        If Button = vbRightButton Then
            fg.Select fg.MouseRow, fg.MouseCol
            fg.EditCell
        End If
    End Sub
```

See Also

VSFlexGrid Control (page 73)

FinishEditing Method

Finishes any pending edits and returns the grid to browse mode.

Syntax

[form!]VSFlexGrid.**FinishEditing** *Cancel* As Boolean

See Also

VSFlexGrid Control (page 73)

GetMergedRange Method

Returns the range of merged cells that includes a given cell.

Syntax

[form!]VSFlexGrid.**GetMergedRange** *Row* As Long, *Col* As Long, *R1* As Long, *C1* As Long, *R2* As Long, *C2* As Long

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.

See Also

VSFlexGrid Control (page 73)

GetNode Method

Returns an outline node object for a given subtotal row.

Syntax

[form!]VSFlexGrid.**GetNode** *Row* As VSFlexNod

See Also

VSFlexGrid Control (page 73)

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)[= *value* As Long]

Remarks

When the grid is used in outline mode, this method allows you to determine a node's parent, first, or last child nodes.

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

Row As Long

The row number of the node whose parent or child nodes you want to determine.

Which As NodeTypeSettings

Which node to return. Valid settings for this parameter are:

Constant	Value	Description
FlexNTRoot	0	Returns the index of the node's top level ancestor.
FlexNTParent	1	Returns the index of the node's immediate parent.
FlexNTFirstChild	2	Returns the index of the node's first child node.
FlexNTLastChild	3	Returns the index of the node's last child node.
FlexNTFirstSibling	4	Returns the index of the node's first sibling node (may be same row)
FlexNTLastSibling	5	Returns the index of the node's last sibling node (may be same row)
FlexNTPreviousSibling	6	Returns the index of the node's previous sibling node (-1 if this is the first sibling)
FlexNTNextSibling	7	Returns the index of the node's next sibling node (-1 if this is the last sibling)

If the node requested cannot be found, **GetNodeRow** returns -1. For example, the root node has no parent, and empty nodes have no children.

The code below shows two typical uses for the **GetNodeRow** property:

```
    ' traverse an outline and return the full path to a given node
    Private Function GetFullPath(r As Long)
        Dim s$
        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.GetOutlineNode(r, flexNTLastChild) - r + 1
    While nRows > 0
        fg.RemoveItem r
        nRows = nRows - 1
    Wend
End Sub
```

Data Type

Long

See Also

VSFlexGrid Control (page 73)

GetSelection Method

Returns the current selection ordered so that Row1 <= Row2 and Col1 <= Col2.

Syntax

[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:

```
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
```

See Also

VSFlexGrid Control (page 73)

LoadArray Method

Loads the control with data from a Variant array or from another FlexGrid control.

Syntax

[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.

See Also

VSFlexGrid Control (page 73)

LoadGrid Method

Loads grid contents and format from a file.

Syntax

[form!]VSFlexGrid.**LoadGrid** *FileName* As String, *LoadWhat* As SaveLoadSettings, [*Options* As Variant]

Remarks

This method loads grid from a file previously saved with the **SaveGrid** method, comma-delimited text file (CSV format) such as an Excel text file, or a tab-delimited text file.

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

FileName As String

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:

Constant	Value	Description
flexFileAll	0	Load all data and formatting information available in the file.
flexFileData	1	Load only the data, ignoring formatting information.
flexFileFormat	2	Load only the formatting, ignoring the data.
flexFileCommaText	3	Load data from a comma-delimited text file.
flexFileTabText	4	Load data from a tab-delimited text file.
flexFileCustomText	5	Load data from a text file using the delimiters specified by the ClipSeparators property.
flexFileExcel	6	Load a sheet from an Excel97 file (you can specify which sheet to load using the Options parameter). This filter does not support frozen color rows or columns.

Options As Variant (optional)

When saving and loading text files, this parameter allows you to specify whether fixed cells are saved and restored. The default is False, which means fixed cells are not saved or restored.

When saving and loading Excel files, this parameter allows you to specify the name or index of the sheet to be loaded, or the name of the sheet to be saved. If omitted, the first sheet is loaded.

Notes

The *flexFileExcel* option is new in Version 8. It does not require Excel to be present on the machine. You can load and save Excel97 sheets (BIFF9 format), one sheet per workbook only (when loading, you can specify which sheet using the *Options* parameter).

The Excel filter supports cell values (including formula values), fonts, formats, colors, row and column dimensions. It does not support features that don't translate into the grid, such as macros, charts, rotated text, cell borders, and other advanced formatting.

Starting in build 200, the control also recognizes a string parameter when saving/loading text files. If the string contains an 'f', fixed cells will be included when saving/loading. If the string contains a 'v', only visible cells will be saved to the text files.

For example:

```
fg.Save("c:\export\)
```

See Also

VSFlexGrid Control (page 73)

LoadGridURL Method

Loads grid contents and format from a URL (created with SaveGrid).

Syntax

[form!]VSFlexGrid.**LoadGridURL** *URL* As String, *LoadWhat* As SaveLoadSettings, [*Options* As Variant]

Remarks

URL As String

The URL to load, including the path.

LoadWhat As SaveLoadSettings

This parameter specifies what should be loaded. Valid options are:

Constant	Value	Description
flexFileAll	0	Load all data and formatting information available in the file.
flexFileData	1	Load only the data, ignoring formatting information.
flexFileFormat	2	Load only the formatting, ignoring the data.
flexFileCommaText	3	Load data from a comma-delimited text file.
flexFileTabText	4	Load data from a tab-delimited text file.
flexFileCustomText	5	Load data from a text file using the delimiters specified by the ClipSeparators property.
flexFileExcel	6	Load a sheet from an Excel97 file (you can specify which sheet to load using the Options parameter). This filter does not support frozen color rows or columns.

Options As Variant (optional)

When saving and loading text files, this parameter allows you to specify whether fixed cells are saved and restored. The default is False, which means fixed cells are not saved or restored.

When saving and loading Excel files, this parameter allows you to specify the name or index of the sheet to be loaded, or the name of the sheet to be saved. If omitted, the first sheet is loaded.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

Outline Method

Sets an outline level for displaying subtotals.

Syntax

[form!]VSFlexGrid.**Outline** *Level* As Integer

Remarks

The **Outline** method collapses or expands an outline to the level specified, collapsing or expanding multiple nodes simultaneously.

The method shows all nodes that have *RowOutlineLevel* smaller than or equal to the *Level* parameter specified. Thus, small *Level* values collapse the outline more, and large values expand it more. If Level is set to zero, only the root node is visible. If *Level* is set to a very large value (say 100 or so), the outline is totally expanded.

Setting *Level* to -1 causes the outline to be totally expanded.

When the nodes are collapsed or expanded, the control fires the **BeforeCollapse** and **AfterCollapse** events. You may trap these events to cancel the action. See the **BeforeCollapse** event for an example.

To set up an outline structure using automatic subtotals, see the **Subtotal** method. To set up a custom outline structure, see the **IsSubtotal** property. For more details on creating and using outlines, see the Outline Demo.

See Also

VSFlexGrid Control (page 73)

Paste Method

Pastes the selection from the Clipboard.

Syntax

Sub **Paste()**

See Also

VSFlexGrid Control (page 73)

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%
        hl = fg.HighLight: tr = fg.TopRow: lc = fg.LeftCol: rd =
fg.Redraw
        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
    fg.TopRow = tr: fg.LeftCol = lc: fg.HighLight = hl
    fg.Redraw = rd
  End Sub
```

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
flexFileAll	0	Save all data and formatting information.

Constant	Value	Description
flexFileData	1	Save only the data, ignoring formatting information.
flexFileFormat	2	Save only the global formatting, ignoring the data.
flexFileCommaText	3	Save data to a comma-delimited text file.
flexFileTabText	4	Save data to a tab-delimited text file.
flexFileCustomText	5	Save data to a text file using the delimiters specified by the ClipSeparators property.
flexFileExcel	6	Save all data and formatting information to an Excel97 file. This filter does not support frozen color rows or columns.

Options As Variant (optional)

When saving and loading text files, this parameter allows you to specify whether fixed cells are saved and restored. The default is False, which means fixed cells are not saved or restored.

When saving and loading Excel files, this parameter allows you to specify the name or index of the sheet to be loaded, or the name of the sheet to be saved. If omitted, the first sheet is loaded.

The options for saving fixed rows, columns, and translated combo values include:

Constant	Value	Description
flexXLSaveFixedCells	3	Saves fixed cells.
flexXLSaveFixedRows	2	Saves fixed rows.
flexXLSaveFixedCols	1	Saves fixed columns.
flexXLSaveRaw	4	Saves raw (untranslated) data.

For example, the options can be written as:

```
fg.SaveGrid "book1.xls", flexFileExcel
fg.SaveGrid "book1.xls", flexFileExcel, "sheetName"
fg.SaveGrid "book1.xls", flexFileExcel, flexXLSaveFixedCells
fg.SaveGrid "book1.xls", flexFileExcel, flexXLSaveFixedRows
fg.SaveGrid "book1.xls", flexFileExcel, flexXLSaveFixedCols
fg.SaveGrid "book1.xls", flexFileExcel, flexXLSaveRaw
fg.SaveGrid "book1.xls", flexFileExcel, _
   flexXLSaveFixedCells Or flexXLSaveRaw
```

Notes

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.

The *flexFileExcel* option is new in Version 8. It does not require Excel to be present on the machine. You can load and save Excel97 sheets (BIFF9 format), one sheet per workbook only (when loading, you can specify which sheet using the *Options* parameter).

The Excel filter supports cell values (including formula values), fonts, formats, colors, row and column dimensions. Several improvements have been made so that it now also supports more than 30k rows, word-wrapping and all ColorAlternate and ColorFrozen properties. It does not support features that don't translate into the grid, such as macros, charts, rotated text, cell borders, and other advanced formatting.

Starting in build 200, the control also recognizes a string parameter when saving/loading text files. If the string contains an 'f', fixed cells will be included when saving/loading. If the string contains a 'v', only visible cells will be saved to the text files. For example:

```
fg.SaveGrid "book1.csv", flexFileCommaText, "fv"
```

See Also

VSFlexGrid Control (page 73)

Select Method

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:

```
 fg.Select _
fg.FixedRows, fg.FixedCols, fg.Rows - 1, fg.Cols - 1
```

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.

See Also

VSFlexGrid Control (page 73)

ShowCell Method

Brings a given cell into view, scrolling the contents if necessary.

Syntax

[form!]VSFlexGrid.**ShowCell** [Row As Long], [Col As Long]

See Also

VSFlexGrid Control (page 73)

Subtotal Method

Inserts rows with summary data.

Syntax

[form!]vsFlexGrid.**Subtotal** *Function* As SubtotalSettings, [*GroupOn* As Long], [*TotalOn* As Long], [*Format* As String], [*BackColor* As Color], [*ForeColor* As Color], [*FontBold* As Boolean], [*Caption* As String], [*MatchFrom* As Long], [*TotalOnly* As Boolean]

Remarks

The **Subtotal** method adds subtotal rows which summarize the data in the control.

Subtotal rows are used for summarizing data and for displaying outlines. You may use the **Subtotal** method to create subtotal rows automatically, or the **IsSubtotal** property to create them manually.

Each subtotal row has a level that is used to indicate which column is being grouped. The subtotal level is also used for outlining. When you created subtotals using the **Subtotal** method, the level is set automatically based on the *GroupOn* parameter. When you create an outline manually, use the **RowOutlineLevel** property to set the outline level for each subtotal row.

Subtotal rows may be added at the top or at the bottom of the values being summarized. This is determined by the **SubtotalPosition** property. When creating outlines, you will typically use the **SubtotalPosition** property is used to place the subtotals above the data. When creating reports, you will typically use the **SubtotalPosition** property to place the subtotals below the data.

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

Function As SubtotalSettings

This parameter specifies the type of aggregate function to be used for the subtotals. Valid settings are:

Constant	Value	Description
flexSTNone	0	Outline only, no aggregate values
flexSTClear	1	Clear all subtotals
flexSTSum	2	Sum
flexSTPercent	3	Percent of total sum
flexSTCount	4	Row count
flexSTAverage	5	Average
flexSTMax	6	Maximum
flexSTMin	7	Minimum
flexSTStd	8	Standard deviation
flexSTVar	9	Variance
flexSTStdPop	10	Standard Deviation Population
flexSTVarPop	11	Variance Population

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

```
fg.Subtotal flexSTSum, 2, 3, , , , , 2
```

TotalOn As Long (optional)

This parameter specifies the column that contains the values to use when calculating the total.

Format As String (optional)

This parameter specifies the format to be used for displaying the results. The syntax for the format string is similar but not identical to the syntax used with Visual Basic's Format command. For a detailed description of the syntax used to specify formats, see the **ColFormat** property. 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 (for instance, "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.

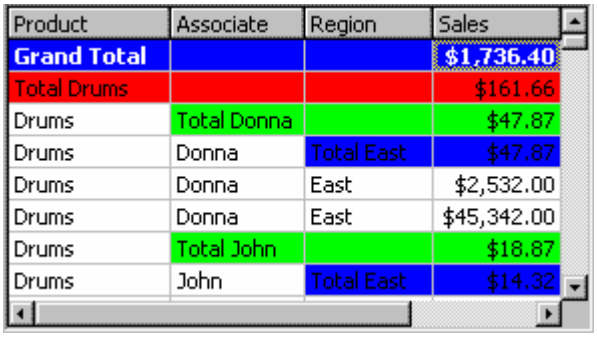
TotalOnly As Boolean (optional)

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
```

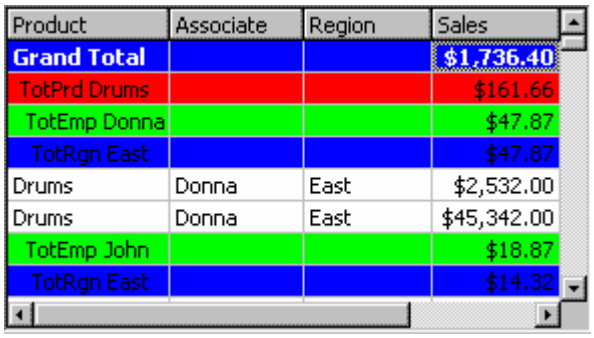
```
    ' total on a negative column to get a grand total
    fg.Subtotal flexSTSum, -1, 3, , vbblue, vbwhite, True
```



Product	Associate	Region	Sales
Grand Total			$1,736.40
Total Drums			$161.66
Drums	Total Donna		$47.87
Drums	Donna	Total East	$47.87
Drums	Donna	East	$2,532.00
Drums	Donna	East	$45,342.00
Drums	Total John		$18.87
Drums	John	Total East	$14.32

The parameters in the **Subtotal** method allow a great deal of customization. The example below shows how the *Caption* and *TotalOnly* parameters can be used to generate report-type subtotals:

```
    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
```



Product	Associate	Region	Sales
Grand Total			$1,736.40
TotPrd Drums			$161.66
TotEmp Donna			$47.87
TotRgn East			$47.87
Drums	Donna	East	$2,532.00
Drums	Donna	East	$45,342.00
TotEmp John			$18.87
TotRgn East			$14.32

See Also

VSFlexGrid Control (page 73)

VSFlexGrid Events

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 node are being collapsed or expanded, this parameter is set to the new outline level being displayed.

Possible node states are:

Constant	Value	Description
flexOutlineExpanded	0	The node is expanded: all subordinate nodes are visible.
flexOutlineSubtotals	1	The node is partially expanded: subordinates nodes are visible but collapsed.
flexOutlineCollapsed	2	The node is collapsed: all subordinate nodes are hidden.

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.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

AfterEdit Event

Fired after the control exits cell edit mode.

Syntax

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

Remarks

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

See Also

VSFlexGrid Control (page 73)

AfterMoveColumn Event

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

Syntax

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

Remarks

This event is only fired if the column was moved by dragging it 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.

See Also

VSFlexGrid Control (page 73)

AfterMoveRow Event

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

Syntax

Private Sub VSFlexGrid_**AfterMoveRow**(ByVal *Row* As Long, *Position* As Long)

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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 *NewCol* 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 **BeforeRowColChange** and **BeforeSelChange** events instead.

See Also

VSFlexGrid Control (page 73)

AfterSort Event

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

Syntax

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

Remarks

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

This event is useful if you want to update user interface elements to reflect the new sorting. Note that the **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 **BeforeSort** event instead.

See Also

VSFlexGrid Control (page 73)

AfterUserFreeze Event

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

Syntax

Private Sub VSFlexGrid_**AfterUserFreeze**()

Remarks

This event is fired whenever the **FrozenRows** or **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 **AllowUserFreezing** property.

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

See Also

VSFlexGrid Control (page 73)

AfterUserResize Event

Fired after the user resizes a row or a column.

Syntax

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

Remarks

The user may resize rows and columns by dragging the edges of fixed rows and columns, depending on the setting of the **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 **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.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
FlexOutlineExpanded	0	The node is expanded: all subordinate nodes are visible.

Constant	Value	Description
FlexOutlineSubtotals	1	The node is partially expanded: subordinates nodes are visible but collapsed.
FlexOutlineCollapsed	2	The node is collapsed: all subordinate nodes are hidden.

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.

```
    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
    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.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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** event.

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:

```
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
```

See Also

VSFlexGrid Control (page 73)

BeforeMouseDown Event

Fired before the control processes the **MouseDown** event.

Syntax

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

Remarks

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

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

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

```
    Private Sub fa_BeforeMouseDown(ByVal Button As Integer, ByVal Shift
As Integer, _
ByVal X As Single, ByVal Y As Single, Cancel As Boolean)
                    ' use shift to drag (ctrl selects)
       If Shift <> 1 Then Exit Sub
                    ' cancel remaining mouse events
                    Cancel = True
                    ' build a list of what we'll be dragging
                    Dim i As Long
                    fa.Tag = ""
        For i = 0 To fa.SelectedRows - 1
            fa.Tag = fa.Tag & vbCrLf & vbTab &    fa.Cell(flexcpText,
fa.SelectedRow(i), 0)
        Next
        ' start dragging
        fa.Drag
    End Sub
```

See Also

VSFlexGrid Control (page 73)

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
```

See Also

VSFlexGrid Control (page 73)

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)

See Also

VSFlexGrid Control (page 73)

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 ComponentOne as a separate product.

See Also

VSFlexGrid Control (page 73)

BeforeRowColChange Event

Fired before the current cell (Row, Col) changes to a different cell.

Syntax

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
```

See Also

VSFlexGrid Control (page 73)

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)

Remarks

This event allows you to prevent the user from scrolling the grid while an operation is being performed on the current selection.

For example, the code below prevents the user from scrolling the grid vertically while a cell is being edited (it allows horizontal scrolling):

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

See Also

VSFlexGrid Control (page 73)

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
```

See Also

VSFlexGrid Control (page 73)

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 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
    ' 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
```

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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:

```
    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
```

See Also

VSFlexGrid Control (page 73)

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:

```
    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
        fg.Cell(flexcpBackColor, Row, Col) = CommonDialog1.Color
    End Sub
```

See Also

VSFlexGrid Control (page 73)

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
```

See Also

VSFlexGrid Control (page 73)

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.ColComboList(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
```

See Also

VSFlexGrid Control (page 73)

ComboCloseUp Event

Fired before the built-in combobox closes up.

Syntax

Private Sub VSFlexGrid_**ComboCloseUp** (*Row* As Long, *Col* As Long, *FinishEdit* As Boolean)

Remarks

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

Row As Long

Col As Long

FinishEdit As Boolean

See Also

VSFlexGrid Control (page 73)

ComboDropDown Event

Fired before the built-in combobox drops down.

Syntax

Private Sub VSFlexGrid_**ComboDropDown** (*Row* As Long, *Col* As Long)

Remarks

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

Row As Long

Col As Long

See Also

VSFlexGrid Control (page 73)

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:

Value	Description
-1	If Row1 should appear before Row2.
0	If the rows are equal (as far as sorting goes).
+1	If Row1 should appear after Row2.

Note that custom sorts are orders of magnitude slower than the built-in sorts, so you should avoid using them unless your data sets are small. Usually, there are good alternatives to a custom sort:

If you are sorting dates, set the **ColDataType** property to *flexDTDate* and the generic sorting settings will sort the dates correctly.

If you are sorting international strings, the generic and string settings will sort the value correctly.

If you want to sort based on arbitrary criteria (for example, "Urgent", "High", "Medium", "Low"), use a hidden column with numerical values that correspond to the criteria you are using.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

EndAutoSearch Event

Fired when the grid leaves AutoSearch mode.

Syntax

Event **EndAutoSearch**()

See Also

VSFlexGrid Control (page 73)

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 programmatically.

See Also

VSFlexGrid Control (page 73)

Error Event

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:

Error Code	Description
129	Recordset can't be updated. The record may be locked, or the recordset may be a read-only recordset.
130	Recordset field can't take this value. The value entered is of the wrong type or would violate database integrity rules.

If you do not handle this event, the control will display a message box informing the user that an error occurred. Execution will continue normally and the control will display the value as retrieved from the database.

You may trap this event to suppress the dialog box, optionally replacing it with a custom one.

See Also

VSFlexGrid Control (page 73)

FilterData Event

Fired after a value is read and before a value is written to a recordset to allow custom formatting.

Syntax

Private Sub VSFlexGrid_**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")

        ' format data coming in from the recordset ("mmddyy" -> Date)
        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.

See Also

VSFlexGrid Control (page 73)

GetHeaderRow Event

Fired while printing the control to set repeating header rows.

Syntax

Private Sub VSFlexGrid_**GetHeaderRow**(ByVal *Row* As Long, HeaderRow As Long)

Remarks

This event is fired while the control is being printed, to create a headers 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 ComponentOne as a separate product.

See Also

VSFlexGrid Control (page 73)

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 **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.

See Also

VSFlexGrid Control (page 73)

KeyPressEdit Event

Fired when the user presses a key in cell-editing mode.

Syntax

Private Sub VSFlexGrid_**KeyPressEdit**(ByVal *Row* As Long, ByVal *Col* As Long, *KeyAscii* As Integer)

Remarks

This event is similar to the standard **KeyPress** event, except it is fired while the grid is in edit mode.

The editor has three modes: text, drop-down combo, or drop-down list. The mode used is determined by the **ComboList** and **ColComboList** properties.

While editing with the text editor or with a drop-down combo, you may set or retrieve the contents of the editor using the **EditText** property. You may manipulate the contents of the editor using the **EditSelStart**, **EditSelLength**, and **EditSelText** properties.

While editing with drop-down lists or drop-down combos, you may set or retrieve the contents of the editor using the **ComboItem**, **ComboIndex**, **ComboCount**, and **ComboData** properties.

The main use for this event is to filter keys as they are typed while the control is in cell-editing mode. For example, the code below shows how you can convert input to upper-case or restrict data entry to numeric values only.

```
Sub 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.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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 programmatically.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
vbDropEffectNone	0	Drop operation was cancelled.
vbDropEffectCopy	1	Drop results in a copy from the source to the Target. The original data remains.
vbDropEffectMove	2	Drop moves the data from the source to the target. The original data should be deleted.

For an example of implementing OLE drag and drop with the **VSFlexGrid** control, see the OLE Drag and Drop Demo.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
vbDropEffectNone	0	Drop operation was cancelled.
vbDropEffectCopy	1	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	2	Drop moves the data from the source to the target. The original data should be deleted.

Button As Integer

An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0 (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.

See Also

VSFlexGrid Control (page 73)

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 ordering together all active effects. The target component should check these effects and other parameters to determine which actions are appropriate for it, and then set this parameter to one of the allowable effects (as specified by the source) to specify which actions will be performed if the user drops the selection on the component. The possible values are:

Constant	Value	Description
VbDropEffectNone	0	Drop operation was cancelled.
VbDropEffectCopy	1	Drop results in a copy from the source to the target. The original data remains.
VbDropEffectMove	2	Drop moves the data from the source to the target. The original data should be deleted.

Button As Integer

An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0, the right button is bit 1, and the middle button is bit 2. These bits correspond to the values 1, 2, and 4, respectively. It indicates the state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are depressed.

Shift As Integer

An integer which acts as a bit field corresponding to the state of the SHIFT, CTRL, and ALT keys when they are depressed. The SHIFT key is bit 0, the CTRL key is bit 1, and the ALT key is bit 2. These bits correspond to the values 1, 2, and 4, respectively. The shift parameter indicates the state of these keys; some, all, or none of the bits can be set, indicating that some, all, or none of the keys are depressed. For example, if both the CTRL and ALT keys are depressed, the value of shift would be 6.

X, Y As Single

These parameters specify the current location of the mouse pointer, in twips.

State As Integer

An integer that corresponds to the transition state of the control being dragged in relation to a target form or control. The possible values are:

Constant	Value	Description
vbEnter	0	Source component is being dragged within the range of a target.
vbLeave	1	Source component is being dragged out of the range of a target.
vbOver	2	Source component has moved from one position in the target to another.

For an example of implementing OLE drag and drop with the **VSFlexGrid** control, see the OLE Drag and Drop Demo.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
vbDropEffectNone	0	Drop operation was cancelled.
vbDropEffectCopy	1	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	2	Drop moves the data from the source to the target. The original data should be deleted.

DefaultCursors As Boolean

A boolean value which determines whether Visual Basic uses the default or a user-defined mouse cursor. If you set this parameter to **False**, the mouse cursor must be set with the **MousePointer** property of the Screen object.

For an example of implementing OLE drag and drop with the **VSFlexGrid** control, see the OLE Drag and Drop Demo.

See Also

VSFlexGrid Control (page 73)

OLESetCustomDataObject Event

Fired after an OLE drag operation is started (manually or automatically), allows you to provide a custom DataObject.

Syntax

Private Sub VSFlexGrid_**OLESetCustomDataObject** (CustomDataObject)

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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:

Constant	Value	Description
vbDropEffectNone	0	Drop operation was cancelled.
vbDropEffectCopy	1	Drop results in a copy from the source to the target. The original data remains.
vbDropEffectMove	2	Drop moves the data from the source to the target. The original data should be deleted.

For an example of implementing OLE drag and drop with the **VSFlexGrid** control, see the OLE Drag and Drop Demo.

See Also

VSFlexGrid Control (page 73)

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.

See Also

VSFlexGrid Control (page 73)

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()

     Debug.Print fg.Row & ", " & fg.Col & ", " & fg.RowSel & ", " &
fg.ColSel

  End Sub
```

See Also

VSFlexGrid Control (page 73)

SetupEditStyle Event

Fired before the EditWindow is created, used to modify window styles.

Syntax

Private Sub VSFlexGrid_**SetupEditStyle** (*Row* As Long, *Col* As Long, *IsCombo* As Boolean, *Style* As Long, *StyleEx* As Long)

Remarks

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

Row As Long

Col As Long

IsCombo As Boolean

Style As Long

StyleEx As Long

See Also

VSFlexGrid Control (page 73)

SetupEditWindow Event

Fired after the EditWindow has been created and before it is displayed.

Syntax

Private Sub VSFlexGrid_**SetupEditWindow** (*Row* As Long, *Col* As Long, *EditWindow* As Long, *IsCombo* As Boolean)

Remarks

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

Row As Long

Col As Long

EditWindow As Long

IsCombo As Boolean

See Also

VSFlexGrid Control (page 73)

StartAutoSearch Event

Fired when the grid enters **AutoSearch** mode.

Syntax

Event **StartAutoSearch**()

See Also

VSFlexGrid Control (page 73)

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:

```
    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
```

See Also

VSFlexGrid Control (page 73)

StartPage Event

Fired before each page while the grid is being printed.

Syntax

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:

```
Private Sub fg_StartPage(ByVal hDC As Long, ByVal Page As Long, Cancel
As Boolean)

        Debug.Print "Printing page " & Page & "..."

End Sub
```

See Also

VSFlexGrid Control (page 73)

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
```

See Also

VSFlexGrid Control (page 73)

VSFlexString Control

Before you can use a **VSFlexString** control in your application, you must add the VSSTR8.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 VSSTR8.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.

VSFlexString Properties, Events, and Methods

All of the properties, events, and methods for the **VSFlexString** control are listed in the following tables. Properties, events, and methods that apply only to this control, or that require special consideration when used with it, are marked with an asterisk (*). These are documented in later sections. For documentation on the remaining properties, see the Visual Basic documentation.

Properties

***CaseSensitive**	Returns or sets whether matching is case-sensitive.
***Error**	Returns status information after each pattern-matching operation.
***MatchCount**	Returns the number of matches found after setting the **Pattern** or **Text** properties.
***MatchIndex**	Returns or sets the zero-based index of the current match when there are multiple matches.
***MatchLength**	Returns the length of the current match, in characters.
***MatchStart**	Returns the zero-based position of the current match within the **Text** string.
***MatchString**	Returns or sets the string corresponding to the current match.
***Pattern**	Returns or sets the regular expression used for matching against the Text string.
***Replace**	Sets a string to replace all matches.
***Soundex**	Returns a phonetic code representing the current **Text** string.
***TagCount**	Returns the number of tags found after setting the **Pattern**, **Text**, or **MatchIndex** properties.
***TagIndex**	Returns or sets the index of the current tag when there are multiple tags in the Pattern string.
***TagLength**	Returns the length of the current tag, in characters.

***TagStart**	Returns the position of the current tag within the **Text** string, starting from zero.
***TagString**	Returns or sets the string corresponding to the current tag.
***Text**	Returns or sets the text to be scanned searching for the **Pattern** string.
***Version**	Returns the version of the control currently loaded in memory.

VSFlexString Properties

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:

```
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

See Also

VSFlexString Control (page 257)

Error Property

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:

Constant	Value	Description
flexErrNone	0	No Error.
flexErrOutOfMemory	1	Out of Memory.
flexErrSquareB	2	The pattern has unmatched square brackets (e.g. "[oops").
flexErrCurlyB	3	The pattern has unmatched square brackets (e.g. "{oops").
flexErrBadPattern	4	Invalid pattern for replace operation (e.g. Pattern = "": Replace = "Hello")
flexErrBadTagIndex	5	Tag index out of range (e.g. Pattern = "{[a-z]}{[a-z]}": ? TagString(2))
flexErrNoMatch	6	No match was found (e.g. Text = "Paul": Pattern = "John")
flexErrInvalidMatchIndex	7	Match index out of range (e.g. Text = "Paul": Pattern = "Paul": ? MatchString(2))

Data Type

StringErrorSettings (Enumeration)

See Also

VSFlexString Control (page 257)

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:

```
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

See Also

VSFlexString Control (page 257)

MatchIndex Property

Returns or sets the zero-based index of the current match when there are multiple matches.

Syntax

[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:

```
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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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:

Char	Description
^	Beginning of a string.
$	End of a string.
.	Any character.
[list]	Any character in list. For example, "[AEIOU]" matches any single uppercase vowel.
[^list]	Any character not in list. For example, "[^]" matches any character except a space.
[A-Z]	Any character between 'A' and 'Z'. For example, "[0-9]" matches any single digit.
?	Repeat previous character zero or one time. For example, "10?" matches "1" and "10".
*	Repeat previous character zero or more times. For example, "10*" matches "1", "10", "1000", etc.
+	Repeat previous character one or more times. For example, "10+" matches "10", "1000", etc.
\	Escape next character. This is required to any of the special characters that are part of the syntax. For example "\.*\+\\" matches ".*+\". It is also required to encode some special non-printable characters (such as tabs) listed below.
{tag}	Tag this part of the match so you can refer to it later using the TagString property.

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

Code	Description	ASCII Code	VB Symbol
\a	Bell (alert)	7	N/A
\b	Backspace	8	N/A
\f	Formfeed	12	N/A
\n	New line	10	VbLf
\r	Carriage return	13	VbCr
\t	Horizontal tab	9	VbTab
\v	Vertical tab	11	N/A

For some examples and more details, see the Regular Expressions topic.

Data Type

String

See Also

VSFlexString Control (page 257)

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:

```
    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:

```
' 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

See Also

VSFlexString Control (page 257)

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
Johnston: J523
Rumpelstiltskin, Runpilztiskin, Rumpel: R514
```

The advantages of this system are that 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

See Also

VSFlexString Control (page 257)

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:

```
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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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

See Also

VSFlexString Control (page 257)

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:

```
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[dsv]e? [ 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:

```
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

See Also

VSFlexString Control (page 257)

Text Property (VSFlexString)

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

See Also

VSFlexString Control (page 257)

Version Property (VSFlexString)

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

See Also

VSFlexString Control (page 257)

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 VSFLEX7 to VSFlexGrid 8.0?

Use **CONVERT**, the conversion utility provided with **VSFlexGrid 8.0**. 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 VSFLEX8.OCX, VSFLEX8D.OCX, and VSFLEX8L.OCX?

The **VSFlexGrid 8.0** package includes three versions of the grid:

VSFLEX8.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.
VSFLEX8D.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).
VSFLEX8L.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 8.0**, 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 **VSFLEX8.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 **VSFLEX8D.OCX**.
3. If you are using the VSFlexGrid in unbound mode (i.e., not bound to any databases), use **VSFLEX8L.OCX**.

Whichever version you decide to use, you may easily switch later using the **CONVERT** utility supplied with **VSFlexGrid 8.0**.

Does VSFlexGrid 7.0 work with VB4-16 or any other 16-bit environments?

It does not. **VSFlexGrid 8.0** is a 32-bit-only product. Please contact ComponentOne's customer service department if you require a 16-bit version of VSFLEX.

When adding VSFLEX8.OCX to my VB4 or VB5 project, I get the following error message: "Error loading DLL". What is wrong?

VSFLEX8.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 **VSFLEX8.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 **VSFLEX8L.OCX** version of the control, which is not subject to this limitation.

Does VSFlexGrid 7.0 work with VB4, VB5 and VB6?

VSFlexGrid 8.0 works with any 32-bit version of Visual Basic. Ideally, however, you should use it with VB5 or later.

When used with VB4, the optional parameters in some properties are not interpreted as optional by VB. The most important property affected by this is the **Cell** property, which has the following syntax:

```
[v =] 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.

```
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 **ComponentOne'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:

```
VARIANT v;
V_VT(&v) = VT_ERROR;
fg.AddItem("hello\tmy 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 Picture property) from C++:

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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W