



TRUE DBGRID™ PRO 5.0

About True DBGrid Pro 5.0

{button ,Jl('',`Product_Profile')} Product Profile

{button ,Jl('',`Whats_New')} What's New in Version 5.0?

{button ,Jl('',`License_and_Redistributable_Files')} License and Redistributable Files

{button ,Jl('',`Technical_Support')} Technical Support

Part 1 - True DBGrid 101

{button ,Jl('',`Getting_Started')} Getting Started
Customizing the Grid's Appearance

{button ,Jl('',`The_Basics')} The Basics
Presentation Techniques

{button ,Jl('',`Tutorials')} Tutorials

{button ,Jl('',`Object_Model')} Object Model

{button ,Jl('',`Design_Time_Interaction')} Design Time Interaction
Cell Editing Techniques

{button ,Jl('',`Run_Time_Interaction')} Run Time Interaction

Part 3 - Data Display and Editing

{button ,Jl('',`Customizing_the_Grids_Appearance')}

{button ,Jl('',`Data_Presentation_Techniques')} Data
Presentation Techniques

{button ,Jl('',`How_to_Use_Splits')} How to Use Splits

{button ,Jl('',`How_to_Use_Styles')} How to Use Styles

{button ,Jl('',`Cell_Editing_Techniques')}

Part 2 - Data Access

{button ,Jl('',`Bound_Mode')} Bound Mode
Reference

{button ,Jl('',`Storage_Mode')} Storage Mode

{button ,Jl('',`Application_Mode')} Application Mode

{button ,Jl('',`Unbound_Mode')} Unbound Mode
Reference

{button ,Jl('',`Database_Programming_Techniques')} Database Programming Techniques

Part 4 - Reference

{button ,Jl('',`Property_Reference')} Property

{button ,Jl('',`Method_Reference')} Method Reference

{button ,Jl('',`Event_Reference')} Event Reference

{button ,Jl('',`Constant_Reference')} Constant

{button ,Jl('',`XArray_Reference')} XArray Reference

"Rio Bravo" Version (5.0)

Copyright © 1995-1997 APEX Software Corporation. All rights reserved.

Product Profile

True DBGrid Pro 5.0 is a data-aware ActiveX grid control for Microsoft Visual Basic 4.0 and 5.0 and Visual C++ 4.2 and 5.0. Developed by APEX Software Corporation, True DBGrid Pro 5.0 is the upgrade to the DBGrid control included in these Microsoft products.

True DBGrid Pro 5.0 allows end users to browse, edit, add, and delete data in a tabular format. Using the latest data binding technologies built into Visual Basic, True DBGrid Pro 5.0 completely manages the database interface, allowing developers to concentrate on important application-specific tasks. True DBGrid Pro 5.0 can also be used in unbound mode with a programmer's own data source without binding to the Visual Basic Data control.

True DBGrid Pro 5.0 was designed to be a powerful, versatile, and easy-to-use data presentation tool. Novice programmers can use True DBGrid Pro 5.0 to create a fully functional database browser without writing a single line of code. Professional developers can use the grid control's many properties and events to create sophisticated and user-friendly database front-end applications.

In addition to being the fastest database grid on the market, True DBGrid Pro 5.0 includes dozens of advanced data access, data presentation, and user interface features that enable developers to build intuitive, professional-looking applications:

100% DBGrid compatibility	True DBGrid Pro 5.0 supports all of the features of the Microsoft Data Bound Grid control (DBGrid).
Excel and Word-like styles	Style objects encapsulate font, color, and formatting information, facilitating easy customization of grid components at design time and run time.
Excel-like splits	Developers and end-users can divide the grid into separate vertical panes to provide multiple views of the data. The splits can scroll independently or simultaneously.
Fixed, nonscrolling columns	Splits can also be used to create nonscrolling columns anywhere in the grid (at the left or right edges, or in the middle).
In-cell objects	The grid supports a variety of in-cell objects for data display and editing, including bitmaps, command buttons, check boxes, and radio buttons.
Drop-down objects	The grid supports a variety of drop-down objects for data entry, including a data-aware multicolumn control (TDBDropDown), a combo box, and a multiline text editor. Third-party drop-down controls also supported.
Automatic data translation	Database values can be automatically translated into alternate text or graphics without coding. For example, numeric codes can be rendered as words or even bitmaps.
Data-sensitive display	Powerful regular expression facility can be used to apply different styles to individual cells depending upon their contents. For example, negative numbers can be shown in red, or fields containing a particular substring can be shown in bold.
Drag-and-drop features	Programmers can implement drag-and-drop interfaces that are sensitive to the grid's rows, columns, or individual cells.
Interactive visual editing	Programmers can create columns, retrieve field layouts from a bound data source, resize grid components, and configure all aspects of the grid layout at design time---no coding is required.
Flexible unbound modes	Event-driven unbound modes handle any data source and are ideal for displaying array data, connecting to a proprietary database, or eliminating the overhead associated with data controls.
Unbound columns	The grid supports unbound columns while other columns are bound to a data control.
Excellent documentation	True DBGrid Pro 5.0 includes an extensive manual and on-line help with plenty of tutorials and examples.

Responsive technical support

Free technical support via e-mail, phone, fax, and peer-to-peer newsgroups. Product updates, sample programs, and answers to frequently asked questions are also available from the APEX Web site at www.apexsc.com.

Free run-time distribution

No royalty fees required.

What's New in Version 5.0?

True DBGrid Pro 5.0 is fully compatible with its predecessor, True DBGrid 4.0, and includes an add-in migration utility to automate the conversion of existing Visual Basic projects. The following features are new in version 5.0:

Array-based storage mode	The XArray object included with True DBGrid Pro 5.0 works just like a Visual Basic array, but also acts as a data source for the grid. No unbound events to code!
Data-aware drop-down list box	The TDBDropDown control included with True DBGrid Pro 5.0 can be bound to a different data control than the grid or used in unbound mode. It also supports incremental search.
Resuable grid layouts	Grid layouts can be saved to a file, then reused in other projects. Multiple layouts can be stored in a single grid at design time, then loaded as needed in code. End-user layout preferences can also be saved to a file, then recalled the next time the application is run.
Input masking	Input templates similar to Visual Basic format strings can be assigned to columns in order to reduce end-user data entry errors.
Multiline displays	The cells in a single record can now span multiple lines, making all columns visible.
Run-time CellTips	Provides context-sensitive help for end-users.
Alternating row colors	Enhances the readability of the grid's display.
Design Assistant add-in	Automates repetitive tasks, facilitates column and split configuration, and enables per-column color and font customizations that would otherwise require coding.
And much more...	New built-in styles, style properties, and database navigation and manipulation methods.

License and Redistributable Files

True DBGrid Pro 5.0 is developed and published by APEX Software Corporation. You may use it for development with Microsoft Visual Basic 4.0 and 5.0 or any other programming environment. You may also distribute the following control files, royalty free, with any application you develop:

TDBG5.OCX	For Visual Basic 5.0 or other compatible programming environments
TDBG5_32.OCX	For 32-bit Visual Basic 4.0
TDBG5_16.OCX	For 16-bit Visual Basic 4.0
XARRAY32.OCX	For Visual Basic 5.0 or 32-bit Visual Basic 4.0

End-users of your applications are **not** licensed to use True DBGrid for development, and may **not** redistribute any of the above control files.

You are **not** licensed to distribute any True DBGrid file to users for development purposes. You are **not** allowed to add or transfer the True DBGrid license key to the registry of your users' computer(s).

In particular, if you create a control using a True DBGrid component as a constituent control, you are not licensed to distribute the control you created with the True DBGrid component to users for development purposes.

It is your responsibility to make such restrictions clear to your users.

Technical Support

True DBGrid Pro 5.0 is developed and supported by APEX Software Corporation. You can obtain technical support using any of the following methods:

APEX Web site

The APEX Web site at www.apexsc.com provides a wealth of information and software downloads for True DBGrid users:

- Answers to frequently asked questions (FAQs) about True DBGrid, organized by functionality. Please consult the FAQs before contacting us directly, as this can save you time and also introduce you to other useful information pertaining to True DBGrid.
- Free product updates, which provide you with bug fixes and new features.
- Sample programs, which provide detailed illustrations of advanced concepts.
- Carl and Gary's Visual Basic Home Page, a comprehensive resource for Visual Basic developers.

Internet e-mail

The best way to get direct technical support is through Internet e-mail (you can also send and receive Internet e-mail if you have a CompuServe account). We respond to e-mail quickly and efficiently---you will receive a response within one business day:

E-mail support@apexsc.com

Phone and fax

If you don't have an Internet account, please use these telephone numbers:

Voice	(412) 681-4738
Fax	(412) 681-4384

Our office hours are 9 AM to 6 PM (EST).

Peer-to-Peer newsgroup

APEX also sponsors a peer-to-peer newsgroup for True DBGrid users. APEX 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 True DBGrid. However, APEX may monitor the newsgroup to ensure accuracy of information and provide comments when necessary. You can access the newsgroup from the APEX Web site or connect your news reader to vger.apexsc.com.

Getting Started

{button ,JI(``,`Installation`)} Installation

{button ,JI(``,`Adding_True_DBGrid_Pro_5.0_to_a_Visual_Basic_Project`)} Adding True DBGrid Pro 5.0 to a Visual Basic Project

{button ,JI(``,`Migrating_to_True_DBGrid_Pro_5.0`)} Migrating to True DBGrid Pro 5.0

{button ,JI(``,`Syntax_Changes_in_Version_5.0`)} Syntax Changes in Version 5.0

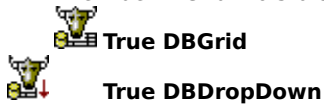
Installation

Insert the True DBGrid Pro 5.0 product CD in your CD-ROM drive. Run SETUP.EXE and follow the instructions.

Adding True DBGrid Pro 5.0 to a Visual Basic Project

After you have opened a new or existing project in Visual Basic, you can add the True DBGrid Pro 5.0 control to the Visual Basic Toolbox by following these instructions:

- If you are using Visual Basic 5.0, click **Project**, then click **Components...** to display the Components dialog.
- If you are using Visual Basic 4.0, click **Tools**, then click **Custom Controls...** to display the Custom Controls dialog.
- The control will be listed as **APEX True DBGrid Pro 5.0** (or **APEX True DBGrid Pro 5.0 for VB4**) in the appropriate dialog. Select the control's check box and then press the **OK** or **Apply** button.
- The True DBGrid Pro 5.0 control icons will be added to the Visual Basic Toolbox.



Migrating to True DBGrid Pro 5.0

If you have projects which use versions of True DBGrid other than True DBGrid Pro 5.0, you can easily convert them to use True DBGrid Pro 5.0 by running the appropriate add-in migration utility. The migration utilities handle the following conversions:

Platform	From	To
16-bit VB4	DBGrid16.OCX for VB4	TDBG5_16.OCX
	TDBGS16.OCX	TDBG5_16.OCX
	TDBG16.OCX	TDBG5_16.OCX
	EDBG5_16.OCX	TDBG5_16.OCX
32-bit VB4	DBGrid32.OCX for VB4	TDBG5_32.OCX
	TDBGS32.OCX	TDBG5_32.OCX
	TDBG32.OCX	TDBG5_32.OCX
	EDBG5_32.OCX	TDBG5_32.OCX
VB5	DBGrid32.OCX for VB5	TDBG5.OCX
	TDBGS32.OCX	TDBG5.OCX
	TDBG32.OCX	TDBG5.OCX
	TDBGS5.OCX	TDBG5.OCX
	EDBG5.OCX	TDBG5.OCX
	TDBG5_32.OCX	TDBG5.OCX

Follow these instructions to add the True DBGrid Pro 5.0 Migration Utility to the Visual Basic **Add-Ins** menu:

- If you are using Visual Basic 5.0, run Visual Basic and select **Add-In Manager...** from the **Add-Ins** menu. The Add-In Manager dialog will appear. Check the box labeled **True DBGrid Pro 5.0 Migration Utility**, then press the **OK** button. The True DBGrid Pro 5.0 Migration Utility icon will be placed on the toolbar.



- If you are using Visual Basic 4.0, run Visual Basic and select **Add-In Manager...** from the **Add-Ins** menu. The Add-In Manager dialog will appear. Check the box labeled **True DBGrid Pro 5.0 Migration Utility (16-bit)** or **True DBGrid Pro 5.0 Migration Utility (32-bit)**. The **Add-Ins** menu now contains a menu item labeled **True DBGrid Pro 5.0 Migration Utility**.

After you have added the True DBGrid Pro 5.0 Migration Utility to the list of available add-ins, you can use it by following these steps:

- Make backup copies of any projects that you plan to convert.
- Open a project that contains one of the controls listed in the **From** column in the migration chart.
- If you are using Visual Basic 4.0, select **True DBGrid Pro 5.0 Migration Utility (16-bit)** or **True DBGrid Pro 5.0 Migration Utility (32-bit)** from the **Add-Ins** menu.
- If you are using Visual Basic 5.0, click the **True DBGrid Pro 5.0 Migration Utility** icon on the toolbar.



- Choose the type of migration that applies to your project (for example, from DBGrid 1.0 to True DBGrid Pro

5.0, or from True DBGrid 4.0 to True DBGrid Pro 5.0), and then click **OK**.

- If the conversion succeeds, a message box will appear to inform you that "The current project was converted successfully."
- To avoid conflict, the migration utilities also remove the **From** control from the Visual Basic Toolbox.

Please note that the migration utilities may also need to modify your source code. Whenever source code is modified, the original code will be commented out and the modification will be tagged with the following comment:

```
*** APEX Migration Utility Code Change ***
```

Syntax Changes in Version 5.0

If you are using DBGrid 1.0, True DBGrid Standard Edition 1.0, or True DBGrid 4.0, you should be aware of the following changes in syntax that may require modifications to your existing code.

FetchCellStyle argument is now qualified

In True DBGrid 4.0, the last argument of the **FetchCellStyle** event is declared as follows:

```
CellStyle As Object
```

In True DBGrid 5.0, this was changed to accommodate the Automatic Code Completion feature:

```
CellStyle As TrueDBGrid50.StyleDisp
```

In Visual Basic 4.0, object qualifiers are not automatically supplied for event arguments, so the last argument appears in the code window as follows:

```
CellStyle As StyleDisp
```

The migration utilities will change the **FetchCellStyle** event handlers accordingly.

FirstRowChange and LeftColChange events pass a split index

In True DBGrid 4.0, the **FirstRowChange** and **LeftColChange** events did not pass any arguments and were only fired for the current split:

```
Private Sub TDBGrid1_FirstRowChange()  
Private Sub TDBGrid1_LeftColChange()
```

In True DBGrid Pro 5.0, the **FirstRowChange** and **LeftColChange** events pass the index of the split in which the change occurred:

```
Private Sub TDBGrid1_FirstRowChange(ByVal SplitIndex As Integer)  
Private Sub TDBGrid1_LeftColChange(ByVal SplitIndex As Integer)
```

The migration utilities will change these event handlers accordingly.

Add method returns the newly added object

In True DBGrid 4.0, you can invoke the **Add** method of the **Columns** and **Splits** collections as follows, ignoring the return value:

```
TDBGrid1.Columns.Add 0  
TDBGrid1.Splits.Add 0
```

This is also true of the **Columns** collection in DBGrid 1.0 and True DBGrid Standard Edition 1.0. Although this syntax is convenient at times, it is incompatible with the Automatic Code Completion feature of Visual Basic 5.0, which requires that the **Add** method return the object that was just added to the collection. For this reason, the old syntax is no longer supported in True DBGrid Pro 5.0, and the standard collection syntax must be used instead:

```
Dim C As TrueDBGrid50.Column  
Set C = TDBGrid1.Columns.Add(0)  
  
Dim S As TrueDBGrid50.Split  
Set S = TDBGrid1.Splits.Add(0)
```

The migration utilities will **not** make any of these changes.

Note the use of the object qualifier `TrueDBGrid50` in the preceding code samples. Although not required, supplying a qualifier eliminates any chance that an object name such as `Column` will conflict with an object of the same name in another control. Object qualifiers are not automatically provided for event handlers in Visual Basic 4.0; however, you can still qualify True DBGrid Pro 5.0 for VB4 objects with the `TrueDBGrid45`

object qualifier.

The Basics

This chapter explains the three fundamental concepts that you need to master in order to use True DBGrid effectively:

1. Data sources
2. Column layouts
3. Bookmarks

{button ,JI('',`Specifying_a_Data_Source')} Specifying a Data Source

{button ,JI('',`Choosing_a_Column_Layout')} Choosing a Column Layout

{button ,JI('',`Configuring_Columns_at_Design_Time')} Configuring Columns at Design Time

{button ,JI('',`Configuring_Columns_at_Run_Time')} Configuring Columns at Run Time

{button ,JI('',`Understanding_Bookmarks')} Understanding Bookmarks

Specifying a Data Source

True DBGrid offers unprecedented flexibility in choosing a data source. You can bind directly to Visual Basic's intrinsic Data control, an external ActiveX control such as Microsoft's Remote Data Control (RDC) or ActiveX Data Connector (ADC), or a third-party data control such as APEX's MyData Control. Using True DBGrid in bound mode greatly simplifies database development by enabling you to focus on your application's interface instead of data access details.

However, there are times when binding to a data control is neither practical nor desirable, and an unbound mode of operation is necessary. By their nature, data controls add layers of overhead, which may result in performance degradation for large datasets. Bound mode is not an option if you are using a proprietary database format or one that is not supported by the standard data controls. You may even need to display a simple two-dimensional array within a grid--why bother with a database?

True DBGrid provides a sensible strategy for handling all of these situations. Regardless of which data access strategy you choose, you won't be penalized for switching to another one at a later date, as True DBGrid provides a clean separation between the data source and the grid's programmatic interface. In other words, if you write data validation or record manipulation code that works in bound mode, it will continue to work if you switch to unbound mode.

True DBGrid supports the following data access modes:

Bound	The grid receives data and notifications from an intrinsic or external data control according to the Microsoft data binding specifications.
Storage	The grid receives data from an APEX XArray object, which can be redimensioned and populated in code much like a standard Visual Basic array.
Application	The grid fires data retrieval and update events for individual grid cells.
Unbound	The grid fires data retrieval and update events for a small set of records all at once.

```
{button ,JI(`,`What_is_bound_mode?')} What is bound mode?
```

```
{button ,JI(`,`What_is_storage_mode?')} What is storage mode?
```

```
{button ,JI(`,`What_is_application_mode?')} What is application mode?
```

```
{button ,JI(`,`What_is_unbound_mode?')} What is unbound mode?
```

What is bound mode?

When the **DataMode** property of a **TDBGrid** control is set to the default value of 0 - Bound, the grid communicates directly with an intrinsic or external data control to retrieve and update data. If you are using a data source that is supported by the Visual Basic built-in Data control or the Microsoft Remote Data Control (RDC), bound mode is your best option. Simply configure the data control as you normally would, then attach it to the **DataSource** property of a **TDBGrid** control at design time.

For more information on using True DBGrid in bound mode, see [Bound Mode](#).

What is storage mode?

The easiest way to display two-dimensional array data in a grid is with storage mode. At design time, set the **DataMode** property of a **TDBGrid** control to 4 - Storage. At run time, create an instance of the APEX **XArray** object included with True DBGrid, populate it as you would a standard Visual Basic array, then attach it to the **Array** property of a **TDBGrid** control in code.

For more information on using True DBGrid in storage mode, see [Storage Mode](#).

NOTE: APEX does not provide a 16-bit XArray object, so storage mode is only available on 32-bit development platforms.

What is application mode?

If you prefer to work with standard Visual Basic arrays, or cannot use storage mode because you need to deliver 16-bit versions of your programs, application mode is recommended, as it is well-suited to array manipulation.

To use application mode, set the **DataMode** property of a **TDBGrid** control to 3 - Application at design time. At a minimum, you will need to write code to handle two events: **ClassicRead** and **UnboundGetRelativeBookmark**. The former is fired whenever the grid requests a value to be displayed in a particular cell; the latter is fired whenever the grid needs to determine the bookmark used to identify a particular row.

For more information on using True DBGrid in application mode, see [Application Mode](#).

What is unbound mode?

If you are working with a database API that supports multiple-row fetches (such as ODBC), or are converting applications that use unbound **DBGrid** controls, then the row-based unbound mode should be used. Although unbound mode is the most difficult data access mode to implement, it tends to be more efficient than application mode, since fewer events need to be fired.

DataMode setting 2 - Unbound Extended is the preferred method. Setting 1 - Unbound is a remnant of the original **DBGrid** control and is included for backward compatibility.

For more information on using True DBGrid in unbound mode, see [Unbound Mode](#).

Choosing a Column Layout

In a True DBGrid display, each column represents a single field of data. For each column, the grid needs to know the *field name* associated with the data, and optionally a heading to be displayed above the data column.

True DBGrid gets information about field names and headings in one of three ways:

1. Automatic layouts, derived at run time from the Data control's **Recordset**.
2. Customized layouts, derived at design time from the Data control's **Recordset**, and optionally tailored using the control's property pages.
3. Run-time layouts, created or modified in code by manipulating the **Columns** collection and its **Column** object members.

{button ,JI(`,`Automatic_layouts')} Automatic layouts

{button ,JI(`,`Customized_layouts')} Customized layouts

{button ,JI(`,`Run-time_layouts')} Run-time layouts

{button ,JI(`,`Switching_between_layout_types')} Switching between layout types

Automatic layouts

If you do not define a column layout at design time, True DBGrid will automatically create one based upon the database used when you run your program. All fields from the Data control's **Recordset** will be displayed, using the field names for column captions. At run time, you can perform database actions that may alter the layout needed to display the data. For example, you may change the **DatabaseName**, **RecordSource**, or **Recordset** properties of the Data control, resulting in a different **Recordset**. When the new **Recordset** is created the grid will automatically sense the new column layout and reconfigure. This mode is the most automatic and is quite useful for most applications. You can cancel the grid's automatic layout behavior by invoking the grid's **HoldFields** method in code.

Customized layouts

At design time, you can cause True DBGrid to configure to the Data control's **Recordset** by selecting **Retrieve Fields** from the grid's context menu. The grid will create a column for each field in the **Recordset**, using the corresponding field name for each column's caption. You can customize each column using the Columns and Layout property pages. The design-time custom layout can be canceled using the **Clear Fields** option of the grid's context menu, or by invoking the grid's **ClearFields** method in code.

Run-time layouts

True DBGrid gives you complete control over the grid layout at run time via **Column** object properties and **Columns** collection methods. You can always modify the grid layout at run time using code, regardless of whether you use the grid's automatic layout feature or define your own.

Switching between layout types

If you define a design-time column layout, the grid will **not** automatically change the layout at run time, as it assumes that you want total control of the display. The grid considers you to have defined a design-time column layout if you chose the **Retrieve Fields** option from the grid's context menu or modified any properties in either the Columns or Layout property pages.

You can clear the design-time layout by choosing the **Clear Fields** option of the grid's context menu, or by invoking the grid's **ClearFields** method in code:

```
TDBGrid1.ClearFields ' Clear column layout
```

After this statement is executed, the grid will again respond automatically to layout changes at run time.

Conversely, you can cancel the grid's automatic layout behavior by invoking the grid's **HoldFields** method in code:

```
TDBGrid1.HoldFields ' Cancel automatic layout
```

After this statement is executed, the grid will stop automatically changing the layout at run time, and uses the current column layout for all subsequent **Recordset** display. This is especially useful if you need to Refresh the data control the grid is bound to while maintaining the current grid layout.

By using the **ClearFields** and **HoldFields** methods, you can alternate the grid's display between automatic layout and customized layout.

Configuring Columns at Design Time

True DBGrid provides unique visual editing capabilities that streamline design-time column configuration. Instead of adding and removing columns with command buttons on a property page, you manipulate the grid directly on the form with the mouse. You can even copy columns to the Clipboard and paste them into another grid on a different form!

Once you have created and resized columns to your liking, you can use the Columns and Layout property pages to further refine their appearance and behavior.

{button ,JI(`,`Visual_editing')} Visual editing

{button ,JI(`,`Specifying_database_fields')} Specifying database fields

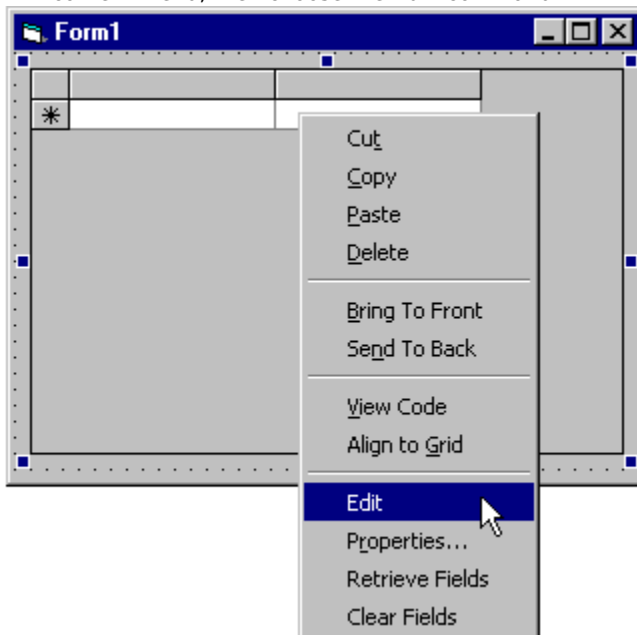
{button ,JI(`,`Specifying_other_column_properties')} Specifying other column properties

Visual editing

At design time, you can use True DBGrid's visual editing mode to perform the following tasks:

- Add and remove columns.
- Copy columns to and from the Clipboard.
- Move and resize columns.
- Adjust the grid's row height.
- Retrieve field layouts from a bound data source.
- Split the grid into separate vertical scrolling regions.
- Save the current grid layout to a file.
- Load an existing grid layout from a file.
- Access the grid's property pages.

To enter visual editing mode, click anywhere on the grid with the right mouse button to display the grid's context menu, then choose the **Edit** command.



The grid control is now activated in-place, which means that you can work with its columns directly on the form. For example, if you point to a dividing line between two columns, the mouse pointer changes to the following symbol.



This indicates that the column you are pointing to is ready to be resized. If you drag the dividing line to a different position, the column will change its width accordingly, and the grid will reposition any adjacent columns. Similarly, if you point to a column header, the mouse pointer changes again.



This symbol indicates that the column is ready to be selected. If you click its header, the entire column is highlighted. You can also drag the mouse pointer within the column header area to extend the selection to other adjacent columns. To cancel the selection and return the columns to their normal, unhighlighted state, click any cell within the grid's data area.

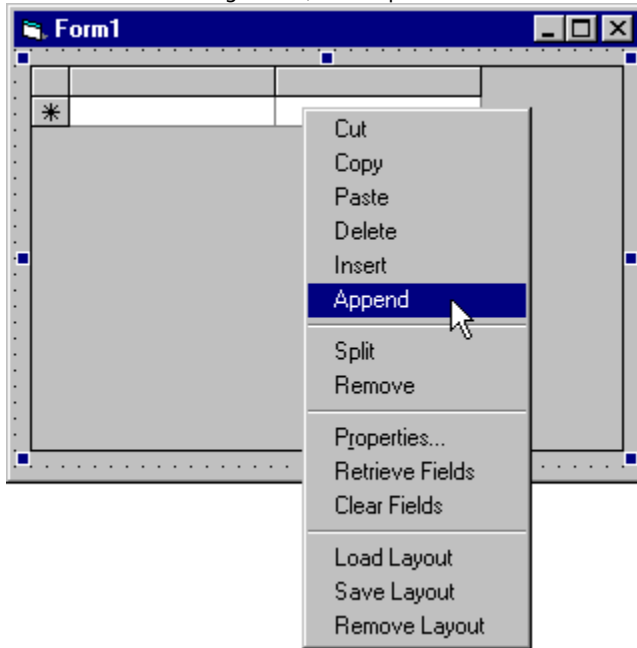
Column selection serves two purposes in visual editing mode:

1. Selected columns can be moved to a different position within the grid by dragging within the column

header (provided that **AllowColMove** is True for the current split).

2. Selected columns act as arguments for some visual editing menu commands.

If the grid is already in visual editing mode, right-clicking it again displays a different context menu. This is the visual editing menu, which provides commands for manipulating columns, splits, and layouts.

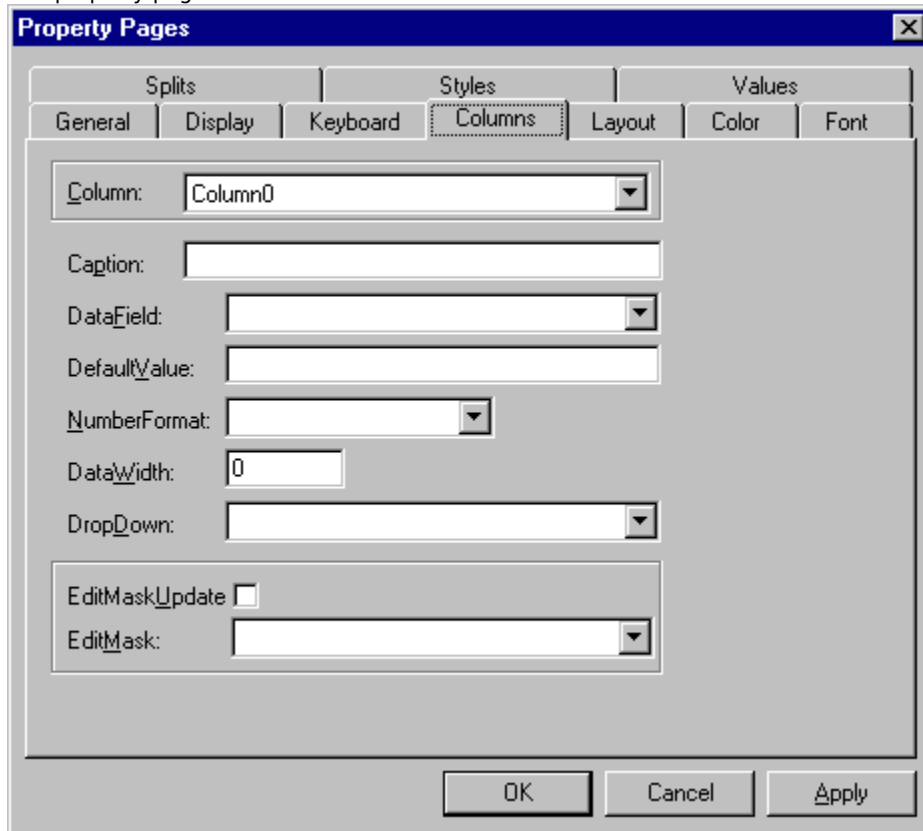


For more information on visual editing, as well as an explanation of the visual editing menu commands, see [Design Time Interaction](#).

Specifying database fields

At design time, the easiest way to bind database fields to grid columns is with the **Retrieve Fields** command. However, if the grid is not bound at design time, this command has no effect. Fortunately, you can still set column properties manually using the grid's property pages, but you must first create blank columns using the **Insert** or **Append** commands of visual editing mode.

To associate a database field with a grid column, choose **Properties...** from the visual editing menu (or context menu) to display the Property Pages dialog, then click the Columns tab to display the Columns property page.



The image shows a screenshot of the "Property Pages" dialog box, specifically the "Columns" tab. The dialog has a title bar with a close button (X) and a tabbed interface. The tabs are "Splits", "Styles", and "Values". Under "Styles", there are sub-tabs: "General", "Display", "Keyboard", "Columns" (which is selected and highlighted with a dotted border), "Layout", "Color", and "Font". The "Columns" tab contains the following controls:

- Column:** A dropdown menu showing "Column0".
- Caption:** An empty text input field.
- DataField:** A dropdown menu.
- DefaultValue:** An empty text input field.
- NumberFormat:** A dropdown menu.
- DataWidth:** A text input field containing the value "0".
- DropDown:** A dropdown menu.
- EditMaskUpdate:** A checkbox that is currently unchecked.
- EditMask:** A dropdown menu.

At the bottom of the dialog, there are three buttons: "OK", "Cancel", and "Apply".

Select a column from the combo box at the top of the page, then choose or type a **DataField** value. You can also enter a value for the **Caption** property, which specifies the text to be displayed in the column header.

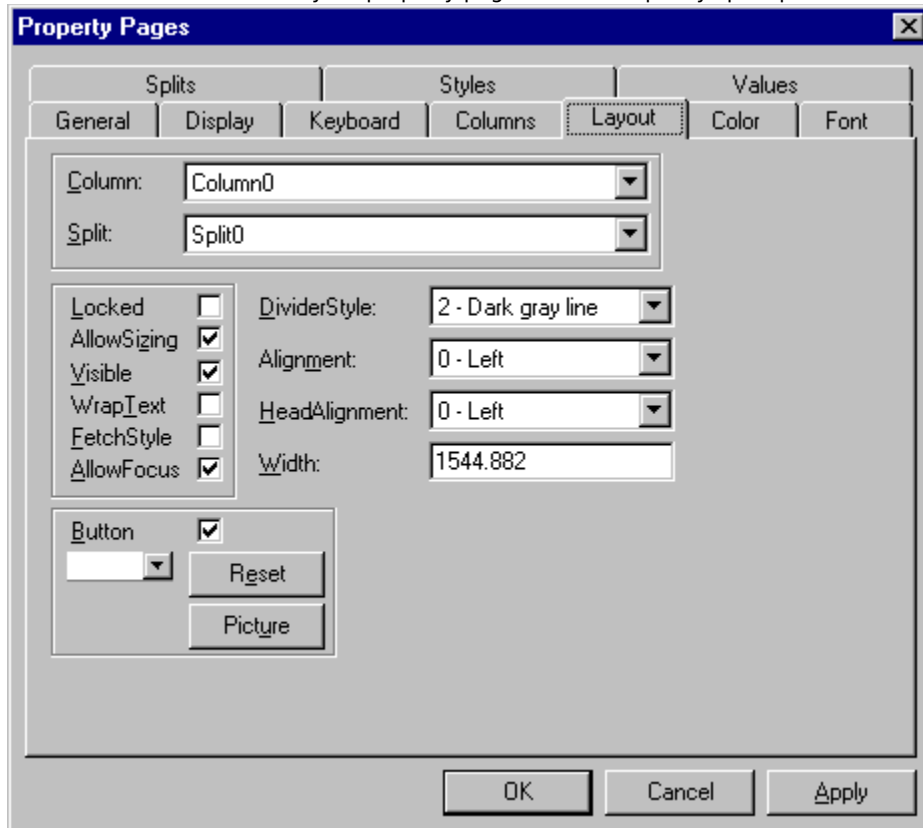
When you are done specifying column properties, click the **OK** button.

Specifying other column properties

Not all column properties can be set from the Columns property page. This is because some properties, such as **Width**, may differ from split to split. In True DBGrid, a *split* is similar to the split window features of products such as Microsoft Excel and Word, and is often used to present data in multiple vertical panes. Two common applications of splits in True DBGrid are:

1. Independent vertical scrolling panes
2. Fixed nonscrolling columns

If you are just getting started with True DBGrid, you don't need to learn about splits right away, but you should know that the Layout property page is used to specify split-specific column properties.



Configuring Columns at Run Time

True DBGrid provides complete control over column layouts at run time. Regardless of which data access mode you are using, you can always add, remove, and manipulate columns in code.

The techniques used to configure columns in code follow the conventions for collection objects in Visual Basic.

{button ,JI(`,`Adding_and_removing_columns')} Adding and removing columns

{button ,JI(`,`Referencing_column_objects')} Referencing column objects

{button ,JI(`,`Adjusting_column_properties_in_code')} Adjusting column properties in code

Adding and removing columns

By manipulating the **Columns** collection, you can add or remove columns from the grid at run time. You can even perform complete grid configurations in code, rather than using the visual editing features.

Here is an example of how a column can be added to the grid using the **Columns** collection:

```
' Create a new Column 0
Dim C As TrueDBGrid50.Column
Set C = TDBGrid1.Columns.Add(0)

' Initialize the new Column 0
With C
    .Visible = True           ' Make it visible
    .DataField = "LAST"      ' Set the column's database field
    .Caption = "Last Name"   ' Set the column's caption
End With

' Make Column 0 as wide as Column 1
C.Width = TDBGrid1.Columns(1).Width
```

Several key points should be noted in this example:

- The **Columns** collection is referenced as `TDBGrid1.Columns`, while an individual **Column** object is referenced with a numeric index, `TDBGrid1.Columns(1)`. All indexes for a collection are zero-based, so index position 1 refers to the second column. This is the general syntax for referencing a collection and its individual elements.
- You can add a new **Column** object to the **Columns** collection using the collection's **Add** method, which accepts a numeric index and returns the newly created object. This is the general technique used to add an item to a collection.
- The Visual Basic `Set` statement is needed to store the new column object in the variable `C`. Without it, the run-time error "Object variable or With block variable not set" will occur.
- The newly added column is Column 0 of the grid. The previous Column 0 becomes Column 1, the previous Column 1 becomes Column 2, and so on.

You can insert a new column at any position. For example:

```
Set C = TDBGrid1.Columns.Add(3)
```

After this statement executes, the new column will be Column 3. The previous Column 3 becomes Column 4, the previous Column 4 becomes Column 5, and so on.

After a new column is added, the **Count** property of the **Columns** collection will be automatically incremented by one. You cannot create a column with an index larger than the current value of the **Count** property. The **Count** property is read-only, so you cannot append columns by setting it to a larger value.

To delete a member of the **Columns** collection and remove it from the grid's display, use the **Remove** method. This is the general technique to remove an item from a collection:

```
TDBGrid1.Columns.Remove 1
```

Or, to remove all columns from a grid:

```
While TDBGrid1.Columns.Count <> 0
    TDBGrid1.Columns.Remove 0
Wend
```

At run time, a newly created column is made invisible to avoid unnecessary flicker when multiple columns are created. Therefore, you must explicitly set its **Visible** property to `True`. Also, you must set the column's

DataField and **Caption** properties, otherwise the grid will display a blank column with no heading.

Note that when you set the **DataField** property of a column in code, you must **ReBind** the grid to the data source in order for the new column binding to take effect.

Referencing column objects

When a column is added to or removed from a grid, the associated **Column** object is added to or removed from the grid's **Columns** collection. This may cause a change in the index numbers of the existing columns, making it very inconvenient to reference columns numerically. For this reason, True DBGrid also allows you to reference columns using either the **DataField** or **Caption** strings. Thus, the following references are identical:

```
TDBGrid1.Columns(n)           ' Reference by the Column index
TDBGrid1.Columns("LAST")     ' Reference by the DataField name
TDBGrid1.Columns("Last Name") ' Reference by the Caption string
```

Referencing column objects by **DataField** or **Caption** is not case-sensitive. `TDBGrid1.Columns("LAST")` refers to the same column as `TDBGrid1.Columns("last")`.

When you reference a **Column** object and its properties at run time, Visual Basic creates an instance of the object. For example, if you duplicate certain properties of a column:

```
TDBGrid1.Columns("First").Width = _
    TDBGrid1.Columns("Last").Width
TDBGrid1.Columns("First").Alignment = _
    TDBGrid1.Columns("Last").Alignment
TDBGrid1.Columns("First").AllowSizing = _
    TDBGrid1.Columns("Last").AllowSizing
```

The `Columns("First")` and `Columns("Last")` objects will each be created and discarded three times in the preceding example. The same results are achieved more efficiently by creating object variables that refer to these columns:

```
' Declare Column objects
Dim FirstCol As TrueDBGrid50.Column
Dim LastCol As TrueDBGrid50.Column

' Reference First and Last Column objects
Set FirstCol = TDBGrid1.Columns("First")
Set LastCol = TDBGrid1.Columns("Last")

' Copy properties from Last to First
FirstCol.Width = LastCol.Width
FirstCol.Alignment = LastCol.Alignment
FirstCol.AllowSizing = LastCol.AllowSizing
```

The same technique can be applied to other objects in Visual Basic. For more details, see [Object Model](#).

Adjusting column properties in code

Properties of the **Column** object can be changed at run time using Visual Basic code. For example, changing the **DataField** property can be done as follows:

```
With TDBGrid1.Columns(0)
    .DataField = "New DataField"
TDBGrid1.ReBind
    .Caption = "New Caption"
End With
```

Note that after changing the **DataField** property, you must **ReBind** the grid columns so that the new data will appear in the column. You should also change the caption to describe the new field.

Other **Column** object properties can be changed in a similar fashion. Please refer to [Column Object Properties](#) for a complete listing.

Understanding Bookmarks

Both True DBGrid and the Microsoft Data Access Objects (DAO) library use *bookmarks* to identify records and navigate through the database. A bookmark is a variant that uniquely identifies a particular row in a database. As such, it is a generalization of the concept of *row numbers*.

Programmers who are accustomed to using row numbers to reference a record (as with dBASE databases) may need to adjust conceptually. In a relational database, the ordinal position of a record (that is, its row number) is irrelevant, since the total number of rows in the database or in a query result set is generally not available. After performing certain operations such as **FindFirst** or **FindNext**, the current record moves an unspecified number of rows forward and there is no efficient way to determine how many. To avoid time-consuming counting operations, most relational database systems have abandoned the practice of using row numbers and have adopted the bookmark approach.

Bookmarks are actually quite simple to use. The following are the basic rules to remember when using bookmarks in True DBGrid and in Visual Basic:

- Each record, or row, has a unique bookmark.
- You can move to a specific record by setting the **Bookmark** property of either the grid or the Data control:

```
TDBGrid1.Bookmark = SomeBookmark  
Data1.Recordset.Bookmark = SomeBookmark
```

SomeBookmark is usually a bookmark you have obtained from the Data control, a clone, or a collection of bookmarks, such as True DBGrid's **SelBookmarks** collection. The **Bookmark** property of the grid and the Data control will always contain the bookmark of the current record.

- You navigate through the database by moving to the first or last record, or by moving relative (next or previous) to the current bookmark:

```
Data1.Recordset.MoveFirst  
Data1.Recordset.MoveLast  
Data1.Recordset.MoveNext  
Data1.Recordset.MovePrevious
```

- In bound mode, you generally do not know the format, or *semantics*, of a bookmark, so do not attempt to read the details of a bookmark or construct a bookmark yourself. The only legitimate operations to perform on a bookmark are saving it to a variable, assigning it to an appropriate property or method, and comparing it to another bookmark to determine if the two are identical:

```
' Saving a bookmark:  
Dim SomeBookmark as Variant  
SomeBookmark = Data1.Recordset.Bookmark  
  
' Assigning a bookmark:  
Data1.Recordset.Bookmark = SomeBookmark  
  
' To reliably compare bookmarks, you must first convert them  
' into strings:  
Dim Bk1 As String, Bk2 As String  
Bk1 = SomeBookmark1  
Bk2 = SomeBookmark2  
If Bk1 = Bk2 Then  
    ...  
End If
```

Note that to reliably compare two bookmarks in Visual Basic, you must first convert them into strings as

shown in the preceding example. For more information, see [Application Mode Bookmarks](#).

Tutorials

- {button ,Jl('',`Introduction')} Introduction
- {button ,Jl('',`Tutorial_1')} Tutorial 1 - Binding True DBGrid to a Data Control
- {button ,Jl('',`Tutorial_2')} Tutorial 2 - Using True DBGrid with SQL Query Results
- {button ,Jl('',`Tutorial_3')} Tutorial 3 - Linking Multiple True DBGrid Controls
- {button ,Jl('',`Tutorial_4')} Tutorial 4 - Interacting with Code and Other Bound Controls
- {button ,Jl('',`Tutorial_5')} Tutorial 5 - Selecting Multiple Rows Using Bookmarks
- {button ,Jl('',`Tutorial_6')} Tutorial 6 - Defining Unbound Columns in a Bound Grid
- {button ,Jl('',`Tutorial_7')} Tutorial 7 - Displaying Translated Data with the Built-in Combo
- {button ,Jl('',`Tutorial_8')} Tutorial 8 - Attaching a True DBDropdown Control to a Grid Cell
- {button ,Jl('',`Tutorial_9')} Tutorial 9 - Attaching an Arbitrary Drop-down Control to a Grid Cell
- {button ,Jl('',`Tutorial_10')} Tutorial 10 - Enhancing the User Interface with In-Cell Bitmaps
- {button ,Jl('',`Tutorial_11')} Tutorial 11 - Using Styles to Highlight Related Data
- {button ,Jl('',`Tutorial_12')} Tutorial 12 - Displaying Rows in Alternating Colors
- {button ,Jl('',`Tutorial_13')} Tutorial 13 - Implementing Drag-and-Drop in True DBGrid
- {button ,Jl('',`Tutorial_14')} Tutorial 14 - Creating a Grid with Fixed, Nonscrolling Columns
- {button ,Jl('',`Tutorial_15')} Tutorial 15 - Displaying Array Data in Unbound Mode
- {button ,Jl('',`Tutorial_16')} Tutorial 16 - Displaying Array Data in Unbound Extended Mode
- {button ,Jl('',`Tutorial_17')} Tutorial 17 - Displaying Array Data in Unbound Application Mode
- {button ,Jl('',`Tutorial_18')} Tutorial 18 - Displaying Array Data in Unbound Storage Mode

Introduction

Eighteen tutorials are presented in this chapter. The tutorials were written for users of True DBGrid Pro 5.0 for Visual Basic 5.0, although they are backward compatible with True DBGrid Pro 5.0 for Visual Basic 4.0 (both 16- and 32-bit versions). The only exception to this is the final tutorial, which runs on 32-bit platforms only.

The tutorials assume that you are familiar with programming in Visual Basic, know what a Data control is, and know how to use the Visual Basic built-in Data control with bound controls in general. The tutorials provide step-by-step instructions---no prior knowledge of True DBGrid is needed. By following the steps outlined in this chapter, you will be able to create projects demonstrating a variety of True DBGrid features, and get a good sense of what the grid can do and how to do it.

The tutorials use an Access database, TDBGDemo.MDB. The database files TDBGDemo.MDB, TDBGDemo.SAV, and the tutorial projects are in the TUTORIAL subdirectory of the True DBGrid installation directory. TDBGDemo.SAV is a backup copy of TDBGDemo.MDB. If you want to restore TDBGDemo.MDB after editing, adding, or deleting records while using the tutorials, make a new copy of TDBGDemo.MDB from TDBGDemo.SAV.

We encourage you to run the tutorial projects in Visual Basic, examine the code, and experiment with your own modifications. This is the best and quickest way to learn how to realize the full potential of True DBGrid. You will find that True DBGrid is very easy to use, and it enables you to create powerful database applications.

The tutorials assume that the database file TDBGDemo.MDB is in the C:\TDBG5\TUTORIAL directory, and refer to it by filename instead of the full pathname for the sake of brevity.

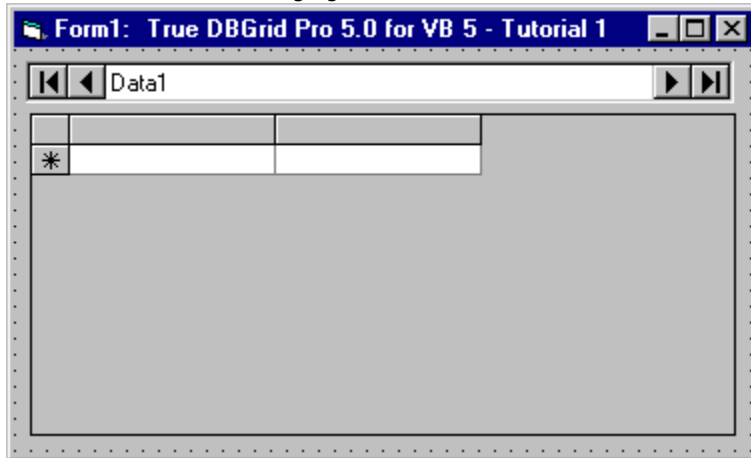
NOTE: Depending on where you store the projects and database files in the TUTORIAL subdirectory, you may need to change the **DatabaseName** property of the Data control in the tutorial projects in order for the projects to work properly.

Tutorial 1 - Binding True DBGrid to a Data Control

In this tutorial, you will learn how to bind True DBGrid to a Visual Basic Data control and create a fully functional database browser without writing a single line of code. You will also learn about the basic properties associated with the Data control and True DBGrid. You will then be able to run the program and observe the run-time features of the grid.

Step 1. Start a new project.

Step 2. Place a Data control (Data1) and a True DBGrid control (TDBGrid1) on the form (Form1) as shown in the following figure.



Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, and the **RecordSource** property to Composer.

Step 4. Set the **DataSource** property of TDBGrid1 to Data1.

Step 5. Set the **AllowAddNew** and **AllowDelete** properties of TDBGrid1 to True (note that the default value of **AllowUpdate** is True).

Run the program and observe the following:

- ⇒ True DBGrid retrieves the database schema information from the Data control and automatically configures itself to display all of the fields contained in the database table. Note that the field names are used as the default column headings.
- ⇒ True DBGrid automatically communicates with the Data control. Any actions taken on the Data control will be reflected in the grid. Click the navigation buttons on the Data control to move forward a record, back a record, to the last record, and to the first record. Note that the grid's current record stays in sync with the Data control.
- ⇒ You have created a fully functional database browser without writing a single line of code!

Refer to [Run Time Interaction](#) and try out the instructions for navigating, editing, and configuring the grid at run time.

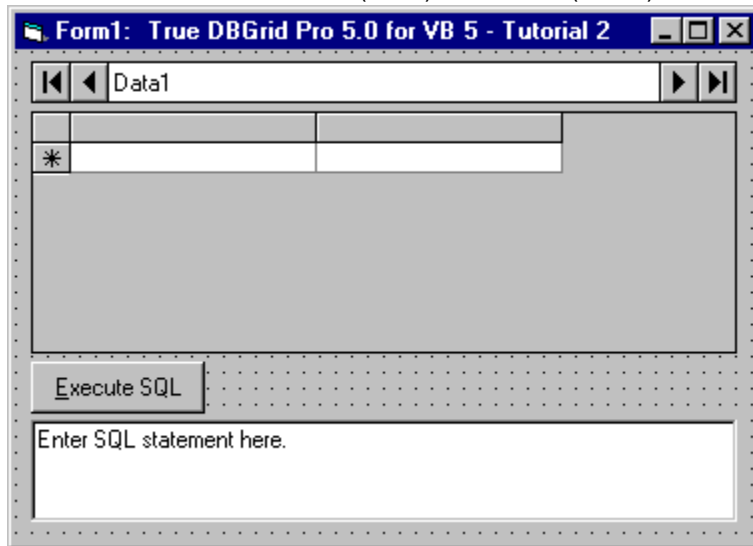
To end the program, press the End button on the Visual Basic toolbar. Congratulations, you have successfully completed Tutorial 1!

Tutorial 2 - Using True DBGrid with SQL Query Results

An important feature of True DBGrid is its ability to automatically sense changes to the database at run time. In this tutorial, you will learn how to use True DBGrid to display the results of ad-hoc SQL queries. Note that no code is necessary to tell the grid what to do---the grid will automatically change its field layout to match the new configuration of the query result. Also note that in order for the grid to automatically respond to field layout changes, you must not have defined any column properties at design time. If a layout is already defined, use the grid's **Clear Fields** context menu command to remove it. This will allow the grid to configure itself automatically.

Step 1. Start a new project.

Step 2. Place a Data control (Data1), a True DBGrid control (TDBGrid1), a command button (Command1) and a TextBox control (Text1) on the form (Form1) as shown in this figure.



Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, and the **RecordSource** property to Customer.

Step 4. Set the **DataSource** property of TDBGrid1 to Data1.

Step 5. Set the **Caption** property of Command1 to Execute SQL and the **MultiLine** property of Text1 to True.

Step 6. Add the following code to Command1:

```
Private Sub Command1_Click()  
  
    ' Execute the SQL statement in Text1, and trigger an error  
    ' message if something goes wrong.  
    On Error GoTo SQLErr  
    Data1.RecordSource = Text1.Text  
    Data1.Refresh  
    TDBGrid1.SetFocus  
    Exit Sub  
  
SQLErr:  
    MsgBox "Error Executing SQL Statement"  
    Exit Sub  
End Sub
```

Run the program and observe the following:

⇒ As in Tutorial 1, True DBGrid retrieves the database schema information from the Data control and automatically configures itself to display the data for all fields in the database table. Note that the field names are used as the default column headings.

Step 7. In the TextBox control, type the following SQL statement:

```
SELECT * FROM Customer
```

then press the Execute SQL command button. The grid display will not change. The above SQL statement displays all fields from the Customer table and is equivalent to the default display.

Step 8. In the TextBox control, type the following SQL statement:

```
SELECT Company FROM Customer
```

then press the Execute SQL command button. The grid responds by displaying only one column for the Company field.

Step 9. In the TextBox control, type the following SQL statement:

```
SELECT LastName, Company FROM Customer
```

then press the Execute SQL command button. This is similar to the previous SQL statement except that two columns (LastName and Company) are now displayed.

Step 10. In the TextBox control, type the following SQL statement:

```
SELECT Count(*) FROM Customer
```

then press the Execute SQL command button. The above SQL statement uses an aggregate function, **Count(*)**, to return the total number of records in the Customer table. Even though the SQL result is not a set of records, the grid faithfully responds by displaying the number of records in a single column. By default, Expr1000 is used as the column heading, indicating that the display is the result of an expression.

To display a more meaningful heading, you can type:

```
SELECT Count(*) AS Count FROM Customer
```

The column heading will display Count instead of Expr1000.

Step 11. In the TextBox control, type the following SQL statement:

```
SELECT UCase(LastName) AS ULAST, UCase(FirstName) AS UFIRST FROM Customer
```

then press the Execute SQL command button. The above SQL statement produces two calculated columns which display the LastName and FirstName fields in upper case. The grid also displays the (assigned) calculated column names, ULAST and UFIRST, as the column headings.

Step 12. In the TextBox control, type the following SQL statement:

```
SELECT * FROM Customer WHERE FirstName = "Jerry"
```

then press the Execute SQL command button. The above SQL statement displays only records with FirstName equal to Jerry.

Step 13. In the TextBox control, type the following SQL statement:

```
SELECT * FROM Customer ORDER BY LastName
```

then press the Execute SQL command button. The above SQL statement displays records in alphabetical order according to the LastName field.

You can also use an SQL statement to join two database tables, as demonstrated in [Tutorial 3](#).

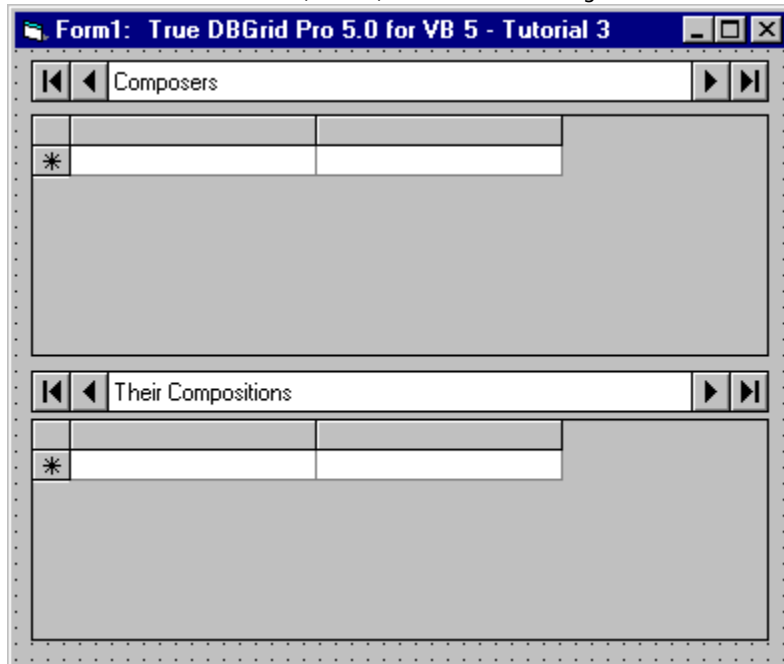
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 2.

Tutorial 3 - Linking Multiple True DBGrid Controls

This tutorial demonstrates how you can link multiple True DBGrid controls using the **RowColChange** event to trigger related actions. This technique is particularly useful for displaying master-detail relationships.

Step 1. Start a new project.

Step 2. Place two Data controls (Data1 and Data2) and two True DBGrid controls (TDBGrid1 and TDBGrid2) on the form (Form1) as shown in this figure.



Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, the **RecordSource** property to Composer, and the **Caption** property to Composers.

Step 4. Set the **DatabaseName** property of Data2 to TDBGDemo.MDB, the **RecordSource** property to Opus, and the **Caption** property to Their Compositions.

Step 5. Set the **DataSource** properties of TDBGrid1 and TDBGrid2 to Data1 and Data2, respectively.

Step 6. Add the following code to the **RowColChange** event of TDBGrid1:

```
Private Sub TDBGrid1_RowColChange(LastRow As Variant, ByVal  
    LastCol As Integer)  
  
    ' A query is performed by taking the "LAST" name field from  
    ' the Data1 control and building an SQL query on the LAST  
    ' name field in the Data2 (compositions) file.  
  
    ' The Second grid will respond automatically when the Data  
    ' Control causes the change. We put up an hourglass so that  
    ' there's a bit of feedback if Access is slow at finishing  
    ' the query.  
  
    Dim lastname$  
    Dim bk1 As String, bk2 As String  
  
    ' To reliably compare bookmarks, you must first convert them
```

```

' into strings. You will also need to test for Null
' Bookmarks being passed by LastRow. This will occur on the
' initial display of the grid and if the user places the
' cursor on the AddNewRow and then moves off.

If IsNull(LastRow) Then
    bk1 = ""
Else
    bk1 = LastRow
End If

bk2 = TDBGrid1.Bookmark

If bk1 <> bk2 Then
    Screen.MousePointer = vbHourglass

    lastname$ = Data1.Recordset("Last")
    Data2.RecordSource = "SELECT * FROM OPUS WHERE LAST = " _
        + Chr$(34) + lastname$ + Chr$(34)

    Data2.Refresh

    Screen.MousePointer = vbDefault
End If
End Sub

```

Run the program and observe the following:

- ⇒ When Form1 is loaded, TDBGrid1 and TDBGrid2 retrieve the database schema information from Data1 and automatically configure themselves to display all of the fields in the Composer and Opus tables, respectively.
- ⇒ However, when TDBGrid1 receives focus and sets the first row as the current row, the **RowColChange** event of TDBGrid1 will be fired. The **RecordSource** of Data2 will be modified and TDBGrid2 will reconfigure itself to display only compositions by Isaac Albeniz. If you observe carefully, when Form1 is first loaded, TDBGrid2 first displays all records in the Opus table, and then refreshes itself quickly to display only one record.
- ⇒ Change the current record position of Data1 by clicking on different rows of TDBGrid1. Observe that TDBGrid2 (the *detail* grid) will configure itself to display a new record set every time the row changes in TDBGrid1 (the *master* grid).

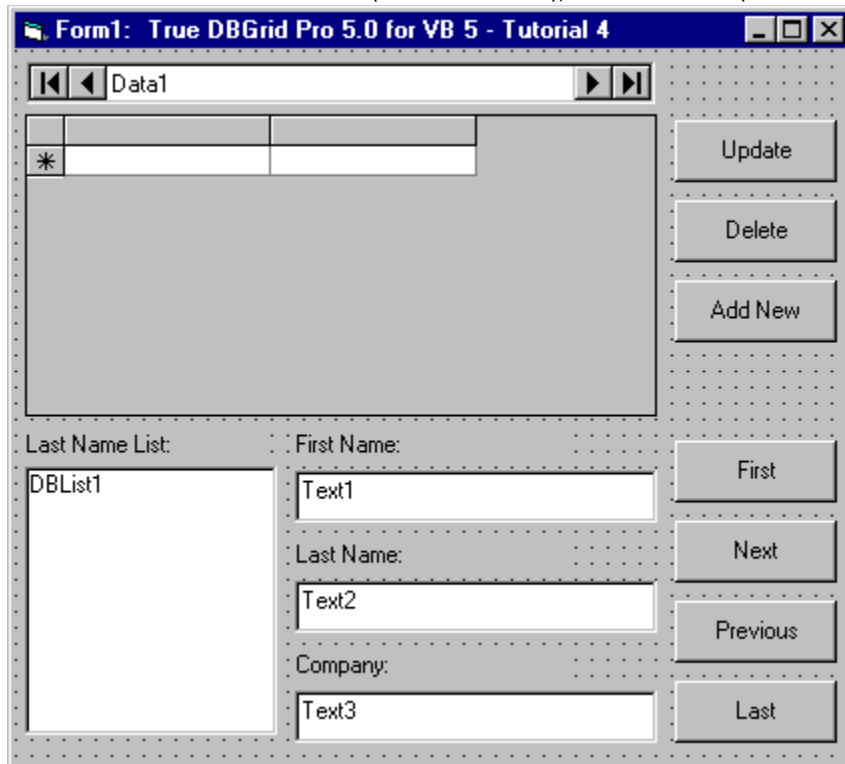
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 3.

Tutorial 4 - Interacting with Code and Other Bound Controls

In this tutorial, you will learn how True DBGrid interacts with other bound controls and with Visual Basic code that manipulates the same **Recordset** to which the grid is bound.

Step 1. Start a new project.

Step 2. Place the following controls on the form (Form1) as shown in the figure: a Data control (Data1), a True DBGrid control (TDBGrid1), a DBList control (DBList1), three text controls (Text1 to 3), seven command buttons (Command1 to 7), and four labels (Label1 to 4).



Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, and the **RecordSource** property to Customer.

Step 4. Set the **DataSource** property of TDBGrid1 to Data1, and the **AllowAddNew** and **AllowDelete** properties to True.

Step 5. Set the **DataSource** and **RowSource** properties of DBList1 to Data1, and the **ListField**, **DataField**, and **BoundColumn** properties to LastName.

Step 6. Set the **DataSource** properties of Text1, Text2 and Text3 to Data1, and the **DataField** properties to FirstName, LastName, and Company, respectively.

Step 7. Set the **Caption** properties of Label1-Label4 and Command1-Command7 as shown in the preceding figure.

We will be using code to affect the record position and data of the database. The seven buttons on this form will contain all of the code we use in this tutorial (the bound controls require no code).

Step 8. Add the following code to the Update button, Command1:

```
Private Sub Command1_Click()  
    ' True DBGrid will automatically respond to the  
    ' update and will clear the "modified indicator"
```

```

' (the pencil icon) on the record selector column
' to indicate that the modified data has been written
' to the database.

Data1.Recordset.Edit
Data1.Recordset.Update
TDBGrid1.SetFocus
End Sub

```

This button triggers an immediate update of all modified data in the bound controls (the grid and the three text controls) without moving the current row position.

Step 9. Add the following code to the Delete button, Command2:

```

Private Sub Command2_Click()
' When the current record is deleted, Jet Engine leaves
' the record pointer at the deleted record. Use MoveNext
' to move the current record to the row after the deleted
' record.

Data1.Recordset.Delete
Data1.Recordset.MoveNext

' If the last record is deleted, move the current record
' to the previous record

If Data1.Recordset.EOF = True Then
    Data1.Recordset.MovePrevious
End If

TDBGrid1.SetFocus
End Sub

```

When the current record is deleted from code, the Data control leaves the record pointer at the deleted record. Therefore, the preceding code uses the **MoveNext** method of the **Recordset** to move the current record to the row after the deleted record.

Step 10. Add the following code to the AddNew button, Command3:

```

Private Sub Command3_Click()
' This "Add New" button moves the cursor to the
' "new (blank) row" at the end so that user can start
' adding data to the new record.

' Move to last record so that "new row" will be visible
Data1.Recordset.MoveLast

' Move the cursor to the "addnew row"
TDBGrid1.Row = TDBGrid1.Row + 1
TDBGrid1.SetFocus

End Sub

```

The above code demonstrates how to move the current cell to the new (blank) row at the end so that the user can start adding data to the new record.

Step 11. Add the following code to the First button, Command4:

```

Private Sub Command4_Click()

```

```

' True DBGrid will follow the record movement.
    Data1.Recordset.MoveFirst
    TDBGrid1.SetFocus
End Sub

```

This button positions the record pointer to the first record in the **Recordset**.

Step 12. Add the following code to the Next button, Command5:

```

Private Sub Command5_Click()
' True DBGrid will follow the record movement.

    Data1.Recordset.MoveNext

' Keep the record away from EOF which is not
' a valid position
If Data1.Recordset.EOF = True Then
    Data1.Recordset.MovePrevious
End If

    TDBGrid1.SetFocus
End Sub

```

This button moves the current row to the next record.

Step 13. Add the following code to the Previous button, Command6:

```

Private Sub Command6_Click()
' True DBGrid will follow the record movement.

    Data1.Recordset.MovePrevious

' Keep the record away from BOF which is not
' a valid position
If Data1.Recordset.BOF = True Then
    Data1.Recordset.MoveNext
End If

    TDBGrid1.SetFocus
End Sub

```

This button moves the current row to the previous record.

Step 14. Add the following code to the Last button, Command7:

```

Private Sub Command7_Click()
' True DBGrid will follow the record movement.

    Data1.Recordset.MoveLast
    TDBGrid1.SetFocus
End Sub

```

This button positions the record pointer to the last record in the **Recordset**.

Run the program and observe the following:

- ⇒ Use the mouse or the keyboard to change the current row position in the grid, and observe the other bound controls (DBList and Text) changing their record positions along with the grid, even though they contain no code.
- ⇒ Click the Next, Previous, Last, and First buttons of the Data control and observe that the record

positions on all bound controls are automatically synchronized.

- ⇒ Click the Next, Previous, Last, and First command buttons and observe that they have the same effects as the corresponding buttons on the Data control.
- ⇒ Modify data in a few cells (in the same row) on the grid. Press the Update command button. Observe that the modified data has been updated to the database and the pencil icon on the record selector column disappears. Other bound controls on the form now display the modified data.
- ⇒ Modify data in one or more of the Text controls. Press the Update or the Next command button. The grid will automatically update its data to reflect the new changes.
- ⇒ Move the current cell of the grid to any record you wish to delete, then click the Delete command button. The record will be deleted and disappears from the grid. The grid automatically moves the current row to the record after the deleted record. Other bound controls on the form also respond by moving their record positions.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 4.

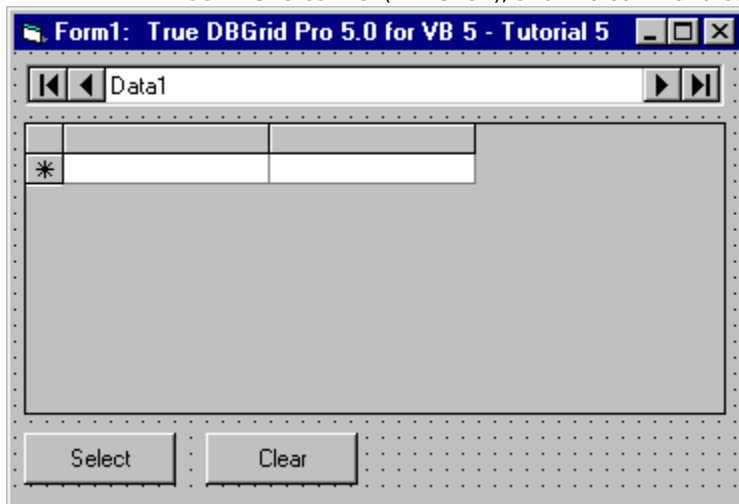
Tutorial 5 - Selecting Multiple Rows Using Bookmarks

In this tutorial, you will learn how to select and highlight records that satisfy specified criteria. A group of similar items is generally implemented as a collection in True DBGrid. When manipulating a group of items in True DBGrid, use techniques similar to those described here. In this case, a row or record is represented by a *bookmark* and a group of selected rows is represented by a **SelBookmarks** collection.

To make the project a bit more interesting, when setting up the **RecordSource** property of the Data control, you will also learn how to use an SQL statement to create a join between two tables in a database.

Step 1. Start a new project.

Step 2. Place the following controls on the form (Form1) as shown in the figure: a Data control (Data1), a True DBGrid control (TDBGrid1), and two command buttons (Command1 and Command2).



Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, and the **RecordSource** property to the following SQL statement:

```
SELECT * FROM composer, opus, composer INNER JOIN opus ON composer.last = opus.last
```

This will create a **Recordset** containing all records from Composer joined with Opus having the same values of the data field Last.

Step 4. Set the **DataSource** properties of TDBGrid1 to Data1.

Step 5. Set the **Caption** properties of Command1 and Command2 to Select and Clear, respectively.

Step 6. We can easily select and deselect rows in True DBGrid by manipulating the **SelBookmarks** collection. To select rows, place the following code in the **Click** event of Command1:

```
Private Sub Command1_Click()  
' This routine loops through the Recordset to find and  
' highlight all records with Country = "Germany"  
  
' We shall use a clone so that we do not move the actual  
' record position of the Data control  
Dim dclone As Recordset  
Set dclone = Data1.Recordset.Clone()  
  
' In case there is a large Recordset to search through  
Screen.MousePointer = vbHourglass
```

```

' For each matching record, add the bookmark to the
' SelBookmarks collection of the grid. The grid will
' highlight the corresponding rows. Note that the bookmarks
' of a clone are compatible with the original set.
' This is ONLY true of clones.
Dim SelBks As TrueDBGrid50.SelBookmarks
Set SelBks = TDBGrid1.SelBookmarks

Dim Criteria$
Criteria$ = "Country = " & Chr$(34) & "Germany" & Chr$(34)
dclone.FindFirst Criteria$
While Not dclone.NoMatch
    SelBks.Add dclone.Bookmark
    dclone.FindNext Criteria$
Wend

' Restore regular mouse pointer
Screen.MousePointer = vbDefault
End Sub

```

Step 7. To deselect rows, place the following code in the **Click** event of Command2:

```

Private Sub Command2_Click()
' Clear all selected rows by removing the selected records from
' the SelBookmarks collection.

Dim SelBks As TrueDBGrid50.SelBookmarks
Set SelBks = TDBGrid1.SelBookmarks

While SelBks.Count <> 0
    SelBks.Remove 0
Wend
End Sub

```

Run the program and observe the following:

- ⇒ TDBGrid1 retrieves the database schema information from the Data control and automatically configures itself to display all of the fields in the joined database tables. This is again similar to the behavior of the grid in Tutorial 1.
- ⇒ Click the Select command button and observe that all records with the Country field equal to Germany will be highlighted.
- ⇒ To deselect the highlighted records, click the Clear command button. Alternatively, clicking anywhere on a grid cell will also clear the selected rows.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 5.

Tutorial 6 - Defining Unbound Columns in a Bound Grid

In this tutorial, you will learn how to use the **UnboundColumnFetch** event to display two fields (FirstName and LastName) together in one column. You will also learn how to use an SQL statement to create a join between two tables in a database. The project we set up for this tutorial will also be used in Tutorials 7 through 12.

Step 1. Start a new project.

Step 2. Place a Data control (Data1) and a True DBGrid control (TDBGrid1) on the form (Form1).

Step 3. Set the **DatabaseName** property of Data1 to TDBGDemo.MDB, and the **RecordSource** property to the following SQL statement:

```
SELECT * FROM Contacts INNER JOIN Customers ON Contacts.UserCode =  
Customers.UserCode
```

The Contacts table contains records of recent customer contacts, but in the table, the customers contacted are recorded by their internal UserCode only, making the table difficult to use by itself. The Customers table contains customer data such as UserCode, FirstName, LastName, and so forth. Therefore, we create a join so that we can view the recent contact information along with the corresponding customer data.

Step 4. Set the **Caption** property of Data1 to Customer Contact.

Step 5. Set the **DataSource** property of TDBGrid1 to Data1.

Configuring the grid at design time

We shall configure the grid using its context menus and property pages. For more details, see [Design Time Interaction](#).

Step 6. Right-click the grid to display its context menu.

Step 7. Choose **Edit** from the context menu. The grid will enter its *visual editing mode*, enabling you to interactively change the grid's row and column layout.

Step 8. By default, the grid contains two columns. We are going to create three more. Right-click anywhere in the grid to display the visual editing menu. Choose the **Append** command to add a new column at the end of the grid. Execute this command two more times to create two more columns. A total of five columns are now in the grid.

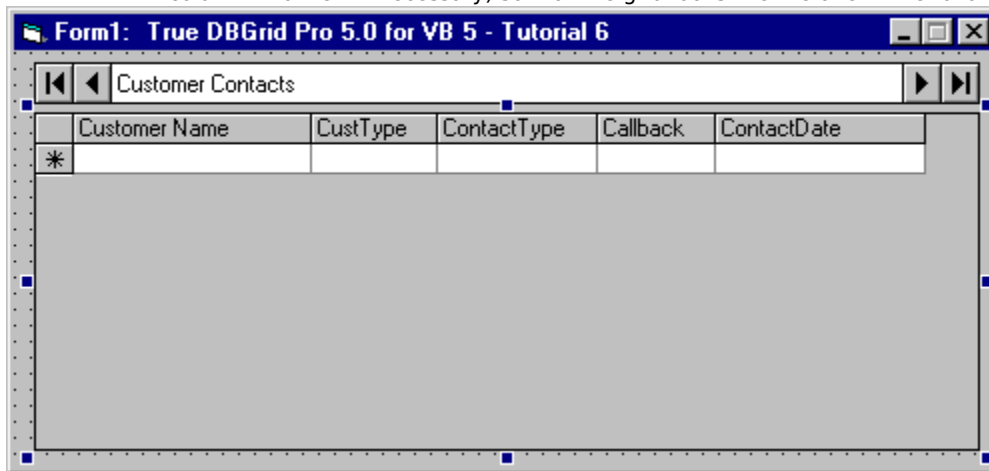
Step 9. Right-click again to display the visual editing menu. This time choose **Properties...** to display the Property Pages dialog. Select the Columns property page by clicking the Columns tab. The Column combo box will display Column0. We are going to configure Column0 as an unbound column. In the **Caption** text box, enter Customer Name. The **DataField** combo box and the other properties on the page will remain blank. If the **DataField** property of a column is blank (that is, equal to an empty string), but its **Caption** property is not, True DBGrid considers it an unbound column.

Step 10. Click the drop-down button of the Column combo box to display the default names of the five columns created in step 8 above: Column0, Column1, Column2, Column3, and Column4. Choose Column1 from the list to display its property values. Click the drop-down button of the **DataField** combo box to reveal a list of all the fields in the joined table. Choose CustType (the last item) from the list. The **Caption** property will default to the same name.

Step 11. Repeat the previous step with the remaining three columns. Column2: **DataField** = ContactType, Column3: **DataField** = Callback, Column4: **DataField** = ContactDate.

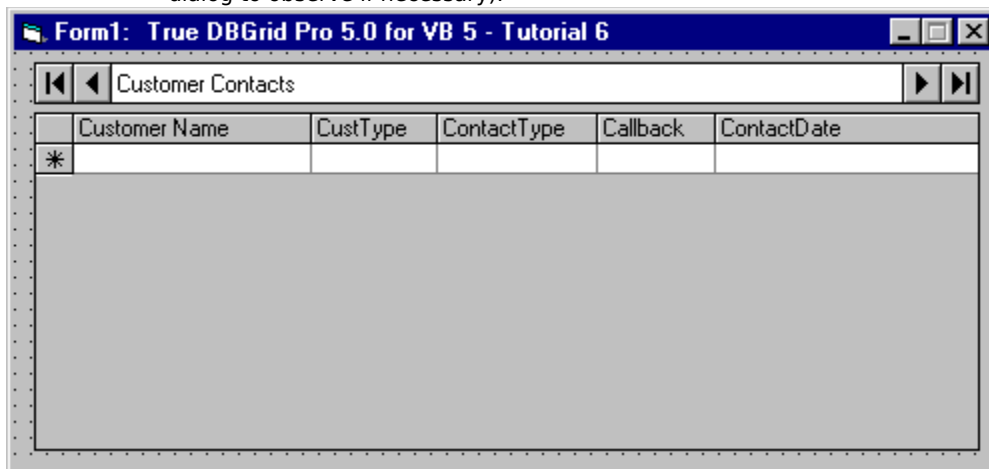
Step 12. After configuring the five columns, click the OK button at the bottom of the property page dialog to accept the changes.

Step 13. Note that you are still in the grid's visual editing mode. Place the mouse cursor over the column dividers within the column header area. It will turn into a horizontal double-arrow cursor, indicating that column resizing can now occur. Drag the dividers (use the horizontal scroll bar to bring a column into view if necessary) so that the grid looks like the one in the following figure.



Notice that there is a gray area between the rightmost column (Contact Date) and the right edge of the grid. We can eliminate this gray area by setting the **ExtendRightColumn** property to True.

Step 14. Bring up the Property Pages dialog again as in step 9. This time select the Splits property page by clicking the Splits tab. Check the **ExtendRightColumn** box and accept the change by clicking the Apply button, which is used to commit changes without closing the dialog. The rightmost column is now extended to the right edge of the grid, which now looks like this (move the Property Pages dialog to observe if necessary).



By default, the grid has one split. Although you have not created any additional splits, you are still working with the properties of the default split. The **ExtendRightColumn** property is on the Splits property page because each split in the grid can have a different value for this property. Properties such as this are referred to as *split-specific* properties.

Step 15. Finally, drop down the **MarqueeStyle** combo box and select 2 - Highlight Cell. For more information on this property, see [Highlighting the Current Row or Cell](#).

Step 16. Click Form1 anywhere outside TDBGrid1 to exit visual editing mode. You have now finished configuring the grid.

Displaying data in the unbound column

In step 9, Column0 of the grid was configured as an unbound column, so you must supply its data using the

UnboundColumnFetch event. When the grid needs to display data in an unbound column, it calls this event to get the necessary data. The following code shows how to display the combined FirstName and LastName fields in the unbound column. For more information on unbound columns, see [Unbound Columns](#).

Step 17. Declare RSClone as a **Recordset** in the General section of Form1 so that the RSClone variable will be available in all procedures in Form1:

```
Dim RSClone As Recordset
```

Step 18. In the **Form_Load** event, set RSClone to be a clone of Data1.Recordset. The Data1.Refresh statement is necessary to make sure Data1 is initialized before cloning its **Recordset**.

```
Private Sub Form_Load()  
    Data1.Refresh  
    Set RSClone = Data1.Recordset.Clone  
End Sub
```

Step 19. Finally, define data in the unbound column by combining the FirstName and LastName fields of the **Recordset** in the grid's **UnboundColumnFetch** event:

```
Private Sub TDBGrid1_UnboundColumnFetch( _  
    Bookmark As Variant, _  
    ByVal Col As Integer, Value As Variant)  
    RSClone.Bookmark = Bookmark  
    Value = RSClone("FirstName") & " " & RSClone("LastName")  
End Sub
```

When the **UnboundColumnFetch** event is called, the Bookmark argument specifies which row of data is being requested by the grid. Note that Bookmark does not usually refer to the current row, since the grid displays more than one row at a time. Hence we use a clone (RSClone) to get data from the **Recordset** so that we do not change the current row position of the Data control. In this example, we only have one unbound column, so we ignore the Col argument.

Run the program and observe the following:

- ⇒ TDBGrid1 displays data from the joined table according to the five columns configured at design time.
- ⇒ The first column displays the combined FirstName and LastName fields as defined in the **UnboundColumnFetch** event.
- ⇒ Since the **MarqueeStyle** is 2 - Highlight Cell, the entire cell is highlighted (not just the cell text) and there is no blinking cursor--the cell is not in *edit-ready* mode. If you click the current cell, it will enter edit mode with the blinking text cursor (caret) appearing at the beginning of the cell's contents. You can also initiate editing simply by typing, in which case the current cell contents will be replaced by what you type.

NOTE: The default **MarqueeStyle** is 6 - Floating Editor. The floating editor highlights the cell text (not the entire cell) as does the datasheet in Microsoft Access. The cell is in *edit-ready* mode with a blinking caret present at the beginning of the highlighted text. In this mode, you can click anywhere within the floating editor to position the insertion point.

- ⇒ The CustType, ContactType and Callback columns display numeric values which are quite cryptic to users. You might also comment that the data presentation is not so appealing. In the next three tutorials (7, 8, and 9), we will illustrate techniques to improve both the display and the user interface.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 6.

Tutorial 7 - Displaying Translated Data with the Built-in Combo

In this tutorial, you will learn how to use the **ValueItems** collection to display translated text and enable the grid's built-in drop-down combo for editing--all without writing a single line of code.

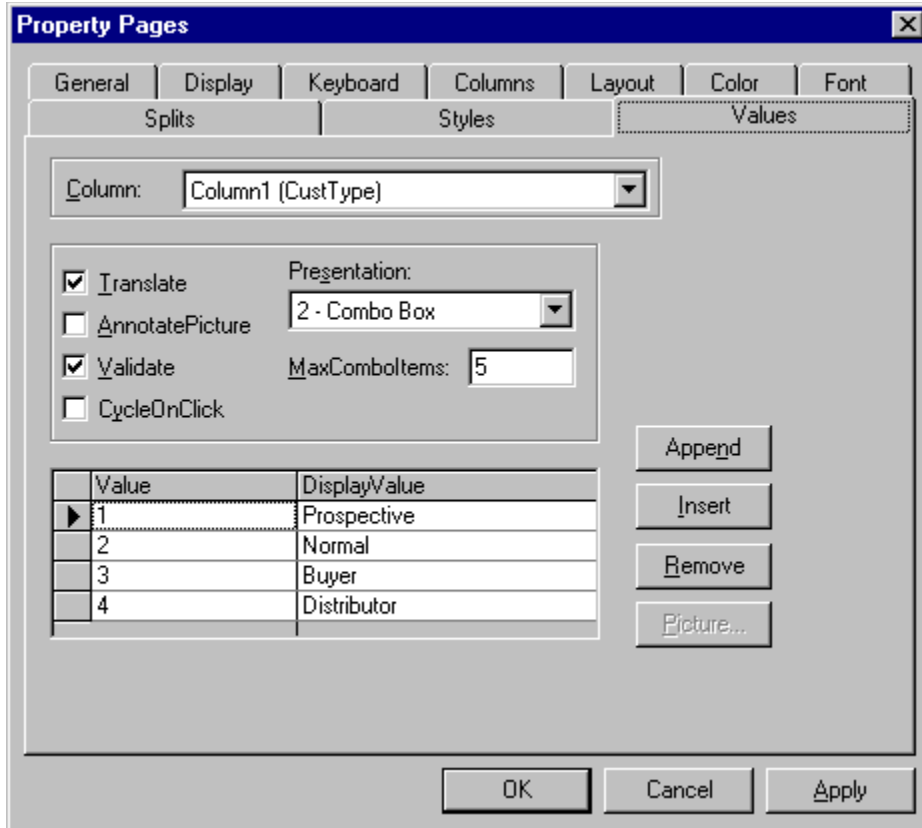
- Step 1.** Start with the project created in [Tutorial 6](#).
- Step 2.** Right-click TDBGrid1 to display its context menu.
- Step 3.** Choose **Properties...** to display the Property Pages dialog. Select the Values property page by clicking the Values tab. This property page is used to specify the **ValueItems** collection associated with a column.
- Step 4.** Drop down the Column combo box and select Column1 (CustType).
- Step 5.** Check the **Translate** box to instruct the grid to translate the data in Column1 before displaying it. Note that the grid at the bottom of the property page now displays two columns labeled Value and DisplayValue.
- Step 6.** Drop down the **Presentation** combo box and select 2 - Combo Box. This instructs the grid to display a combo box in Column1 when requested.
- Step 7.** Now enter the **Value - DisplayValue** pairs in the grid as follows:

Value	Display Value
1	Prospective
2	Normal
3	Buyer
4	Distributor
5	Other

Entries in the **Value** column are data values from the CustType field in the database table. The grid treats the **Value** property as a string. Entries in the **DisplayValue** column are translated values to be displayed in the CustType column of the grid. For example, a CustType of 1 will be displayed as *Prospective*, 2 will be displayed as *Normal*, and so forth.

NOTE: Some databases store numbers with a leading space character to hold the place of a minus sign, so it may be necessary to prefix **Value** column entries with a space.

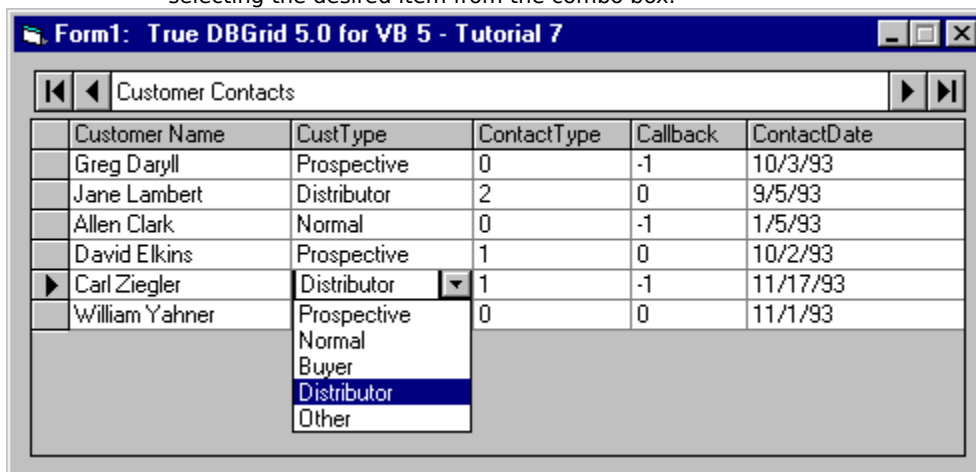
When you are finished entering data, the Values property page should look like this.



Step 8. Click the OK button at the bottom of the Property Pages dialog to accept the changes.

Run the program and observe the following:

- ⇒ TDBGrid1 displays data from the joined tables as in Tutorial 6.
- ⇒ The CustType column now displays the translated text instead of numeric values.
- ⇒ Click a cell in the CustType column to make it the current cell. Notice that a dropdown button appears at the right edge of the cell.
- ⇒ Click the dropdown button or press ALT+DOWN ARROW to display the built-in combo box containing translated values, as shown in the following figure. You can change the data in the current cell by selecting the desired item from the combo box.



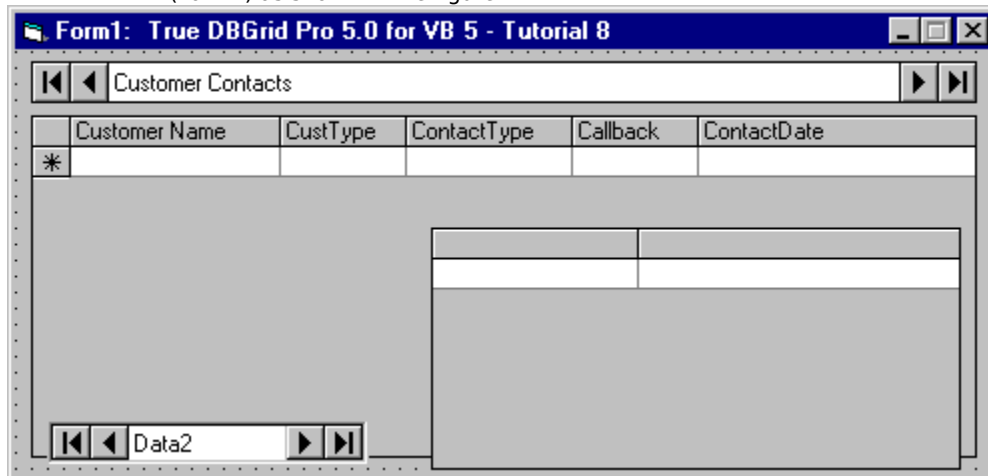
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 7.

Tutorial 8 - Attaching a True DBDropDown Control to a Grid Cell

In this tutorial, you will learn how to attach a multicolumn True DBDropDown control to a grid cell. Unlike the built-in combo demonstrated in [Tutorial 7](#), the **TDBDropDown** control can be bound to a Data control, which makes it ideal for data entry involving a secondary lookup table. The drop-down control appears whenever the user clicks a button within the current cell. This button appears automatically when the user gives focus to a column that has a drop-down control connected to it.

Step 1. Start with the project constructed in [Tutorial 6](#).

Step 2. Add a True DBDropDown control (TDBDropDown1) and another Data control (Data2) to the form (Form1) as shown in the figure:



Step 3. Set the **DatabaseName** property of Data2 to TDBGDemo.MDB, and the **RecordSource** property to CustType.

Step 4. Set the **DataSource** property of True DBDropDown to Data2, and the **ListField** property to Typeld.

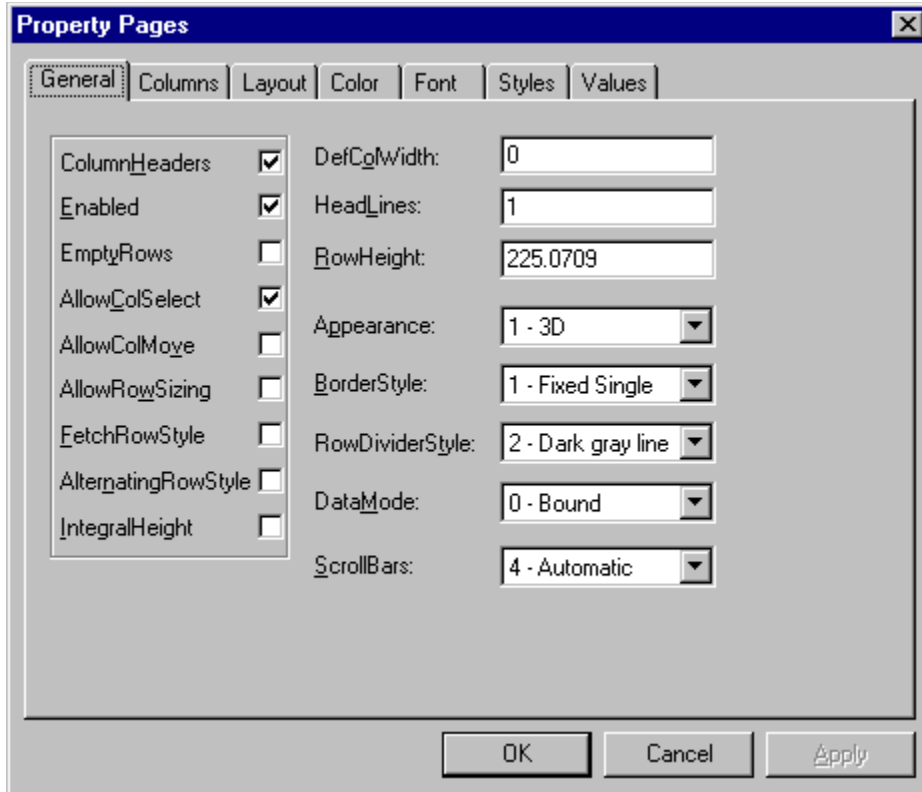
Modifying the True DBDropDown control

Step 5. Right-click the True DBDropDown control to display its context menu.

Step 6. Choose **Edit** to enter visual editing mode just as you would with a True DBGrid control. You can now interactively change the drop-down control's column layout.

Step 7. Resize the first column so that it is approximately 3/8 of an inch wide.

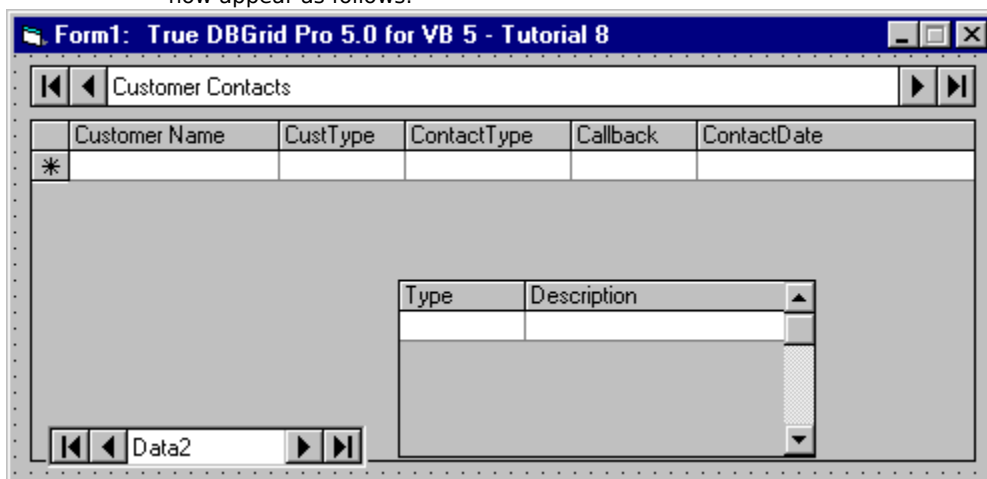
Step 8. Choose **Properties...** from the visual editing menu to display the Property Pages dialog for the True DBDropDown control.



Step 9. In the General property page, check **ColumnHeaders** and **IntegralHeight**. Clear **AllowColMove**, **AllowColSelect**, and **AllowRowSizing**. Set the **ScrollBars** property to 2 - Vertical. The **IntegralHeight** property is only supported by the **TDBDropDown** control. When set to True, it prevents the drop-down control from displaying partial rows.

Step 10. In the Columns property page, set the **DataField** property for Column0 to TypeId and the **DataField** property for Column1 to TypeDesc.

Step 11. Set the **Caption** properties for Column0 and Column1 to Type and Description. The form should now appear as follows.



Run the program and observe the following:

⇒ TDBGrid1 displays data from the joined table as in [Tutorial 6](#).

- ⇒ Click a cell in the CustType column to make it the current cell as indicated by the highlight. A button will be displayed at the right edge of the cell. Click the button to display the True DBDropDown control as shown in the following figure.

Customer Name	CustType	ContactType	Callback	ContactDate
Greg Daryll	1	0	-1	10/3/93
Jane Lambert	4	2	0	9/5/93
▶ Allen Clark	2	0	-1	1/5/93
David Elkins	Type	Description		10/2/93
Carl Ziegler	2	Normal		11/17/93
William Yahner	3	Buyer		11/1/93
*	4	Distributor		
	5	Other		

- ⇒ You can use the UP ARROW and DOWN ARROW keys to move the highlight bar of True DBDropDown. If you click another cell in the grid, True DBDropDown will lose focus and become invisible.
- ⇒ Select any item in True DBDropDown. The current cell in the grid will be updated with the selected item, and True DBDropDown will disappear until you initiate editing again.
- ⇒ If you move the current cell to another column, the button will disappear from the cell in the CustType column.

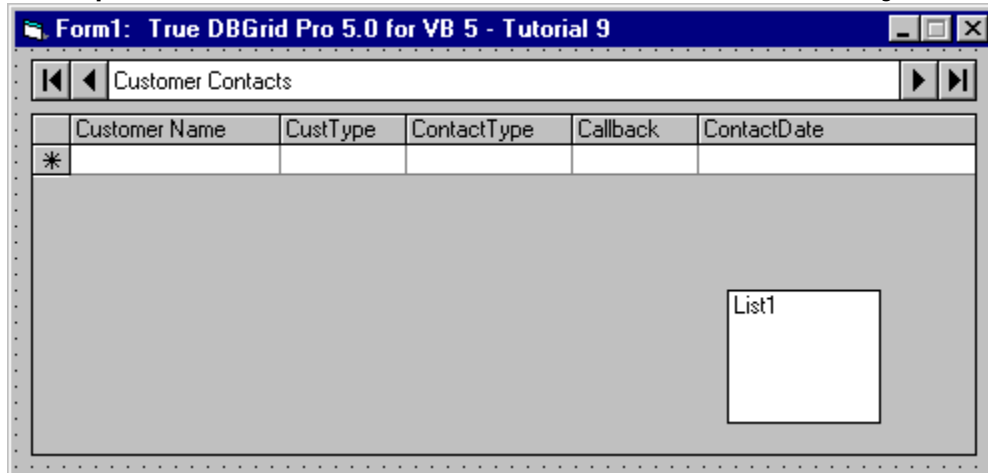
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 8.

Tutorial 9 - Attaching an Arbitrary Drop-down Control to a Grid Cell

In this tutorial, you will learn how to drop down an arbitrary control from a grid cell. This example uses a ListBox control containing pre-defined input values in order to facilitate user data entry. The list will drop down whenever the user initiates editing, such as by clicking the current cell. You will also learn how to place a button in the cell which, when clicked, will cause the ListBox control to appear. You can drop down any control from a grid cell using techniques similar to those described in this tutorial.

Step 1. Start with the project constructed in [Tutorial 6](#).

Step 2. Add a ListBox control (List1) to the form (Form1) as shown in the figure.



Step 3. Set the **Visible** property of List1 to False.

Adding code to drop down a ListBox control

The CustType field in the second column (Column1) of the grid displays numeric values ranging from 1 through 5 which represent the following customer types:

- 1 = Prospective
- 2 = Normal
- 3 = Buyer
- 4 = Distributor
- 5 = Other

We shall drop down List1, which will contain textual customer type descriptions, and allow users to double-click an item in order to enter the associated value into the grid.

Step 4. In the **Form_Load** event, we place code to add the customer types to List1. We also place a button in the CustType column using the **Button** property. The **Form_Load** event handler now looks like this:

```
Private Sub Form_Load()  
    ' Define RSClone as a clone  
    Data1.Refresh  
    Set RSClone = Data1.Recordset.Clone  
  
    ' Add customer types to List1  
    List1.AddItem "Prospective"  
    List1.AddItem "Normal"  
    List1.AddItem "Buyer"  
    List1.AddItem "Distributor"  
    List1.AddItem "Other"
```

```

        ' Place a button in the CustType column
        TDBGrid1.Columns("CustType").Button = True
    End Sub

```

Step 5. If a cell in the CustType column becomes current, a button will be placed at the right edge of the cell. Clicking the button will trigger the grid's **ButtonClick** event. We will drop down List1 whenever the button is clicked:

```

Private Sub TDBGrid1_ButtonClick(ByVal ColIndex As Integer)
    ' Assign the Column object to Co because it will be used
    ' more than once.
    Dim Co As Column
    Set Co = TDBGrid1.Columns(ColIndex)

    ' Position and drop down List1 at the right edge of the
    ' current cell.
    List1.Left = TDBGrid1.Left + Co.Left + Co.Width
    List1.Top = TDBGrid1.Top + TDBGrid1.RowTop(TDBGrid1.Row)
    List1.Visible = True
    List1.ZOrder 0
    List1.SetFocus
End Sub

```

Step 6. In the grid's **BeforeColEdit** event, we add the following code to drop down List1 if we are editing the CustType column (Column1). Note that the code below will not work if the **MarqueeStyle** property is set to 6 - Floating Editor. See [Highlighting the Current Row or Cell](#) for more details.

```

Private Sub TDBGrid1_BeforeColEdit( _
    ByVal ColIndex As Integer, _
    ByVal KeyAscii As Integer, Cancel As Integer)
    ' BeforeColEdit is called before the grid enters into
    ' edit mode. You can decide what happens and whether
    ' standard editing proceeds. This allows you to
    ' substitute different kinds of editing for the current
    ' cell, as is done here.

    If ColIndex = 1 Then
        ' Let the user edit by entering a key.
        If KeyAscii <> 0 Then Exit Sub

        ' Otherwise, cancel built-in editing and call the
        ' ButtonClick event to drop down List1.
        Cancel = True
        TDBGrid1_ButtonClick (ColIndex)
    End If
End Sub

```

Step 7. We allow the user to enter data into the CustType column of the grid by double-clicking the desired selection in List1:

```

Private List1_DblClick()
    ' When an item is selected in List1, copy its index to the
    ' proper column in TDBGrid1, then make List1 invisible.
    TDBGrid1.Columns(1).Text = List1.ListIndex + 1
    List1.Visible = False
End Sub

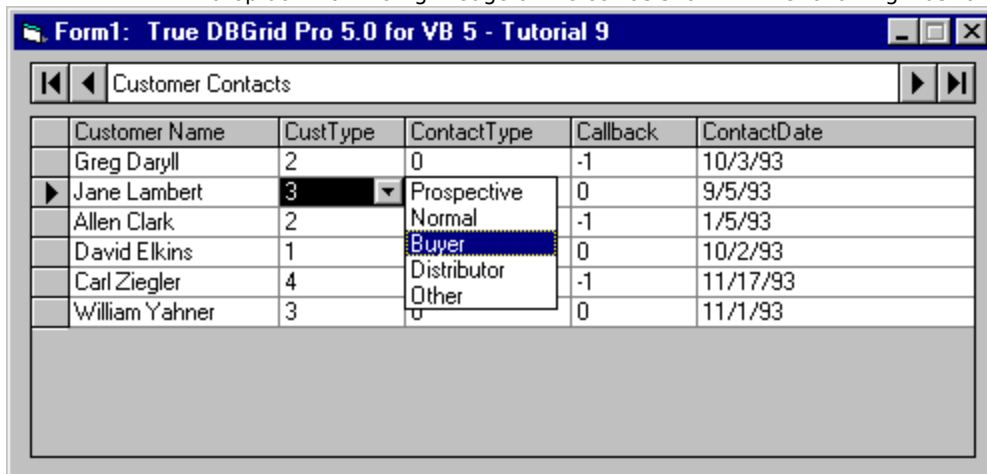
```

Step 8. Finally, we make List1 invisible whenever it loses focus or when the user scrolls the grid:

```
Private Sub List1_LostFocus()  
    ' Hide the list if it loses focus.  
    List1.Visible = False  
End Sub  
  
Private Sub TDBGrid1_Scroll(Cancel As Integer)  
    ' Hide the list if we scroll.  
    List1.Visible = False  
End Sub
```

Run the program and observe the following:

- ⇒ TDBGrid1 displays data from the joined table as in [Tutorial 6](#).
- ⇒ Click a cell in the CustType column to make it the current cell as indicated by the highlight. A button will be displayed at the right edge of the cell. Click the button to fire the **ButtonClick** event. List1 will drop down at the right edge of the cell as shown in the following illustration.



- ⇒ You can use the mouse or the UP ARROW and DOWN ARROW keys to move the highlight bar of List1. If you click another cell in the grid, List1 will lose focus and become invisible.
- ⇒ Double-click any item in List1. The current cell in the grid will be updated with the selected item, and List1 will disappear until you initiate editing again.
- ⇒ If you move the current cell to another column, the button will disappear from the cell in the CustType column.
- ⇒ Make a cell in the CustType column current again. This time, instead of clicking the button, click the text area of the current cell to put it in *edit mode*. Before the grid enters edit mode, it fires the **BeforeColEdit** event, and List1 appears at the right edge of the current cell as if you had clicked the in-cell button. You can use the list to select an item for data entry as in the previous steps.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 9.

Tutorial 10 - Enhancing the User Interface with In-Cell Bitmaps

In this tutorial, you will learn how to use the **ValueItems** collection to display bitmaps and check boxes in a cell--without writing a single line of code!

Step 1. Start with the project used in [Tutorial 9](#).

Step 2. First, we will change the captions of the ContactType and Callback columns. Right-click TDBGrid1 to display its context menu.

Step 3. Choose **Properties...** to display the Property Pages dialog. Select the Columns property page by clicking the Columns tab. Drop down the Column combo box and select Column2 (ContactType). Change the **Caption** property from ContactType to How. Repeat with Column3 (Callback) and change the **Caption** property from Callback to Call?

Step 4. Next, we assign bitmaps and check boxes to selected columns by populating the corresponding **ValueItems** collection. Select the Values property page by clicking the Values tab.

Step 5. Drop down the Column combo box and select Column2 (ContactType). Check the **Translate** box to instruct the grid to translate the data in Column2 before displaying it. Note that the grid at the bottom of the page now displays two columns labeled Value and DisplayValue. The **Value** column is for data values from the database. The **DisplayValue** column is for translated values you wish the grid to display.

Step 6. Check the **CycleOnClick** box so that when you click a cell in Column2 at run time, the cell will automatically cycle through all the values defined in the **Value - DisplayValue** table.

Step 7. The possible values of the ContactType field are 0, 1, and 2 which represent telephone, mail, and personal contact, respectively. We shall display bitmaps in the cell instead of these numeric values. If you installed the full product, you will find the following files in the BITMAPS subdirectory of the True DBGrid installation directory: PHONE.BMP, MAIL.BMP, and PERSON.BMP.

Click the first row within the Value column and enter 0 as the first value. Click the same row within the DisplayValue column to enable the Picture... button on the right. Click this button to show an open file dialog. To associate a bitmap with the value 0, choose PHONE.BMP and click the dialog's OK button to accept the selection. The phone bitmap will then appear in the DisplayValue column. Repeat this step with the following values and bitmaps:

Value	DisplayValue
1	MAIL.BMP
2	PERSON.BMP

Step 8. After defining the bitmap entries for Column2 (ContactType), drop down the Column combo box again and select Column3 (Callback). This column contains a boolean field with allowable values of 0 and -1 (False and True), which in this case represent whether a call needs to be returned. We shall display check boxes instead of boolean values. The bitmaps CHKOFF1.BMP and CHKON1.BMP are provided in the BITMAPS directory for this purpose.

Step 9. Check the **Translate**, and **CycleOnClick** boxes as with the ContactType column. Then enter the following Value - DisplayValue pairs as in step 7:

Value	DisplayValue
0	CHKOFF1.BMP
-1	CHKON1.BMP

Step 10. Click the OK button at the bottom of the Property Pages dialog to accept the changes. You should see the column captions on the grid modified according to the changes you made in step 3.

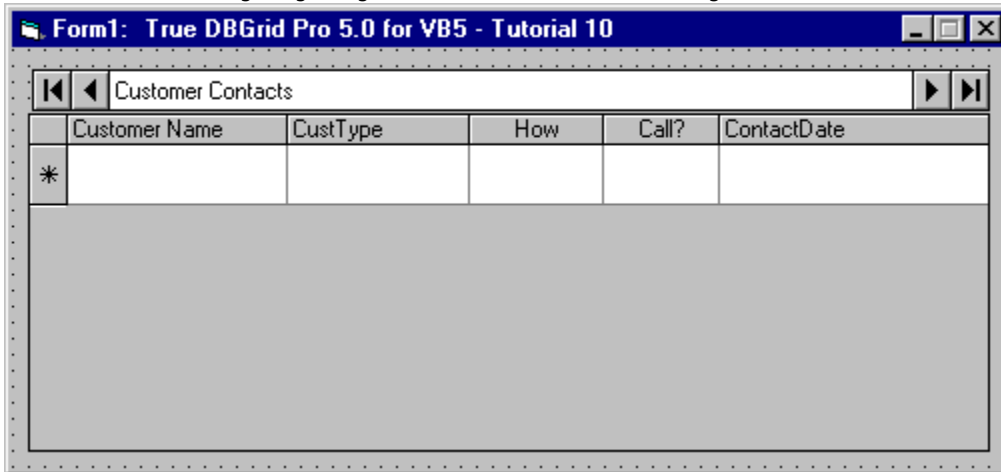
Step 11. Display the primary context menu again as in step 2, but this time choose the **Edit** option to enter

the grid's visual editing mode, which enables you to interactively change the grid's row height and column layout.

Step 12. Place the mouse cursor over a column divider in the column header area, changing it to a horizontal double-arrow resizing cursor. Drag the divider to the left to shorten the How and Call? columns.

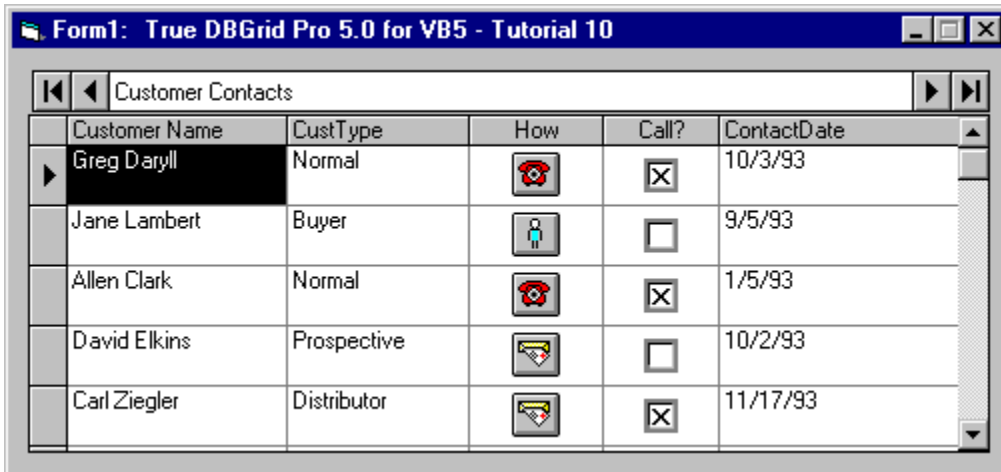
Step 13. Now place the mouse cursor over the row divider, changing it to a vertical double-arrow resizing cursor. Drag the divider downward to increase the row size to about double the current height.

Step 14. Click Form1 anywhere outside TDBGrid1 to exit visual editing mode. You have now completed reconfiguring the grid, which should look like the figure below.



Run the program and observe the following:

- ⇒ TDBGrid1 displays data from the joined table as in Tutorials 10 through 11.
- ⇒ The How and Call? columns now display bitmaps instead of numeric values as shown in the figure below.



- ⇒ Click a cell in the How column to make it the current cell. Then click it again several times and observe how the cell cycles through the PHONE, MAIL, and PERSON bitmaps.
- ⇒ Click a cell in the Call? column to make it the current cell. Then click it again several times and observe how the cell cycles through the CHKON1 and CHKOFF1 bitmaps. We have used the bitmap display to simulate check box behavior in a cell.

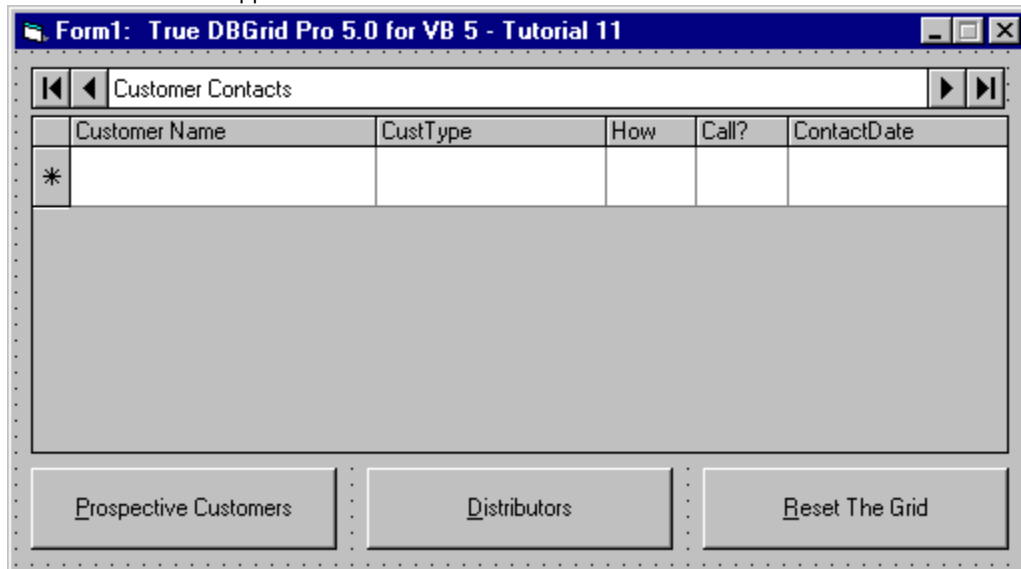
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 10.

Tutorial 11 - Using Styles to Highlight Related Data

In this tutorial, you will learn how to change the grid's display to highlight rows by defining new styles at run time. True DBGrid uses **Style** objects to apply font and color characteristics to rows, columns, and individual cells.

Step 1. Start with the project used in [Tutorial 10](#).

Step 2. Add three command buttons to the form. Change the **Caption** property for Command1 to Prospective Customers, Command2 to Distributors, and Command3 to Reset the Grid so that the form appears as follows.



Step 3. Add the following declarations to the General section of Form1:

```
Dim ButtonFlag As Variant
Dim RSClone As Recordset
Dim Col As TrueDBGrid50.Column
Dim Cols As TrueDBGrid50.Columns
Dim Prospective As New TrueDBGrid50.Style
Dim Buyers As New TrueDBGrid50.Style
```

Step 4. Enter the following code in the Click event of Command1:

```
Private Sub Command1_Click()
    ButtonFlag = 1

    Set Cols = TDBGrid1.Columns

    For Each Col In Cols
        Col.FetchStyle = True
    Next

    TDBGrid1.Refresh
End Sub
```

Step 5. Enter the following code in the Click event of Command2:

```
Private Sub Command2_Click()
    ButtonFlag = 2
```

```

Set Cols = TDBGrid1.Columns

For Each Col In Cols
    Col.FetchStyle = True
Next

TDBGrid1.Refresh
End Sub

```

Step 6. Enter the following code in the Click event of Command3:

```

Private Sub Command3_Click()
    ButtonFlag = 3

    Set Cols = TDBGrid1.Columns

    For Each Col In Cols
        Col.FetchStyle = True
    Next

    TDBGrid1.Refresh
End Sub

```

Step 7. Enter the following code in the **Form_Load** event:

```

Set Prospective = TDBGrid1.Styles.Add("Prospective")
Prospective.Font.Italic = True
Prospective.Font.Bold = True
Prospective.ForeColor = vbBlue

Set Buyers = TDBGrid1.Styles.Add("Distributors")
Distributors.BackColor = vbRed
Distributors.ForeColor = vbWhite

```

Step 8. Enter the following code in the **FetchCellStyle** event of TDBGrid1:

```

RSClone.Bookmark = Bookmark

If ButtonFlag = 1 And RSClone("CustType") = 1 Then
    CellStyle = Prospective
End If

If ButtonFlag = 2 And RSClone("CustType") = 4 Then
    CellStyle = Buyers
End If

If ButtonFlag = 3 Then
    CellStyle = TDBGrid1.Styles("Normal")
End If






```

Run the program and observe the following:

- ⇒ TDBGrid1 displays data as in [Tutorial 10](#).
- ⇒ Click the **Prospective Customers** button. The grid should appear as follows.

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 11

Customer Contacts






Customer Name	CustType	How	Call?	ContactDate
<i>Greg Daryll</i>	<i>Prospective</i>		<input checked="" type="checkbox"/>	<i>10/3/93</i>
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
<i>David Elkins</i>	<i>Prospective</i>		<input type="checkbox"/>	<i>10/2/93</i>
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93

Prospective Customers Distributors Reset The Grid

⇒ Click the **Distributors** button. The grid should now appear as follows:

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 11

Customer Contacts






Customer Name	CustType	How	Call?	ContactDate
Greg Daryll	Prospective		<input checked="" type="checkbox"/>	10/3/93
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
David Elkins	Prospective		<input type="checkbox"/>	10/2/93
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93

Prospective Customers Distributors Reset The Grid

⇒ Finally, click the **Reset The Grid** button. The grid should now appear as follows.

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 11

Customer Contacts

Customer Name	CustType	How	Call?	ContactDate
Greg Daryll	Prospective		<input checked="" type="checkbox"/>	10/3/93
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
David Elkins	Prospective		<input type="checkbox"/>	10/2/93
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93

Prospective Customers Distributors Reset The Grid

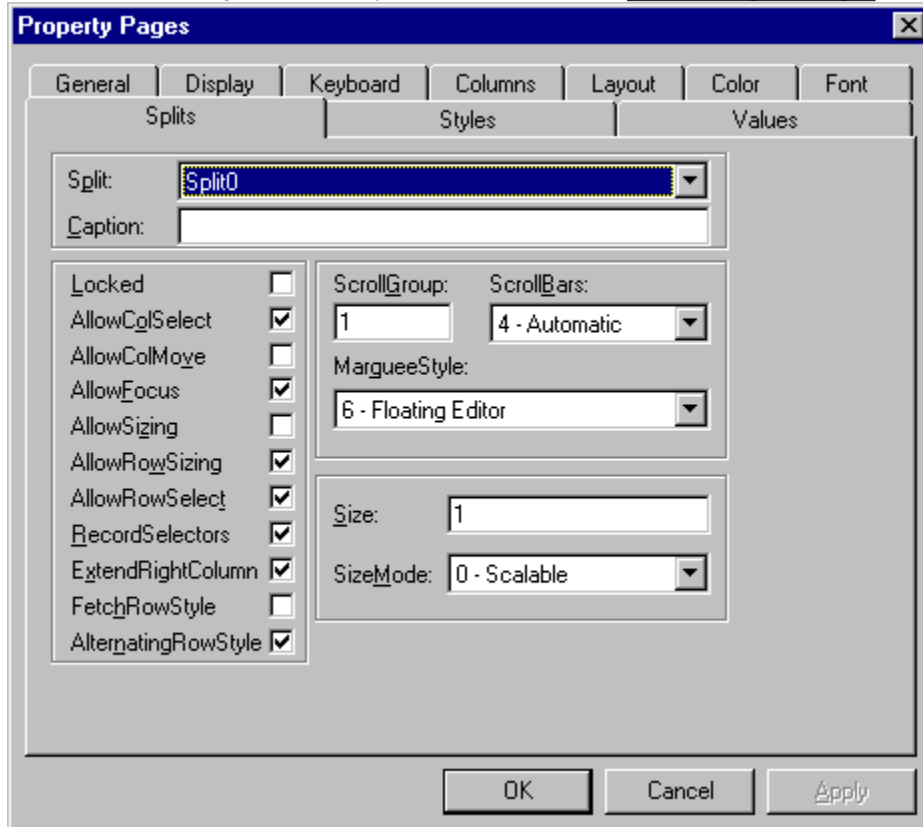
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 11.

Tutorial 12 - Displaying Rows in Alternating Colors

In this tutorial, you will learn how to use the **AlternatingRowStyle** property and built-in styles to apply alternating colors to grid rows to improve their readability.

Step 1. Start with the project used in [Tutorial 10](#).

Step 2. Right-click the grid and select **Properties...** from the context menu to display the Property Pages dialog. Click the Splits tab and select the **AlternatingRowStyle** check box.



The grid has default settings for both the EvenRow and OddRow styles. We will use the default settings first then change the settings for the EvenRow style.

Run the program and observe the following:

- ⇒ TDBGrid1 displays data as in [Tutorial 10](#), except that even-numbered rows have a light cyan background.

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 12

Customer Contacts

Customer Name	Customer Type	How	Call?	ContactDate
Greg Daryll	Prospective		<input checked="" type="checkbox"/>	10/3/93
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
David Elkins	Prospective		<input type="checkbox"/>	10/2/93
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93
William Yahner	Ruwer		<input type="checkbox"/>	11/1/93

Step 3. Right-click the grid and select **Properties...** from the context menu to display the Property Pages dialog. Click the Styles tab and from the Style Name combo box choose EvenRow.

Step 4. Next, select the Colors option button and change the **BackColor** of the EvenRow style by clicking the light gray color button and then clicking OK.

Run the program and observe the following:

⇒ TDBGrid1 displays data as in the previous figure, except that even-numbered rows now have a light gray background.

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 12

Customer Contacts

Customer Name	Customer Type	How	Call?	ContactDate
Greg Daryll	Prospective		<input checked="" type="checkbox"/>	10/3/93
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
David Elkins	Prospective		<input type="checkbox"/>	10/2/93
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93
William Yahner	Ruwer		<input type="checkbox"/>	11/1/93

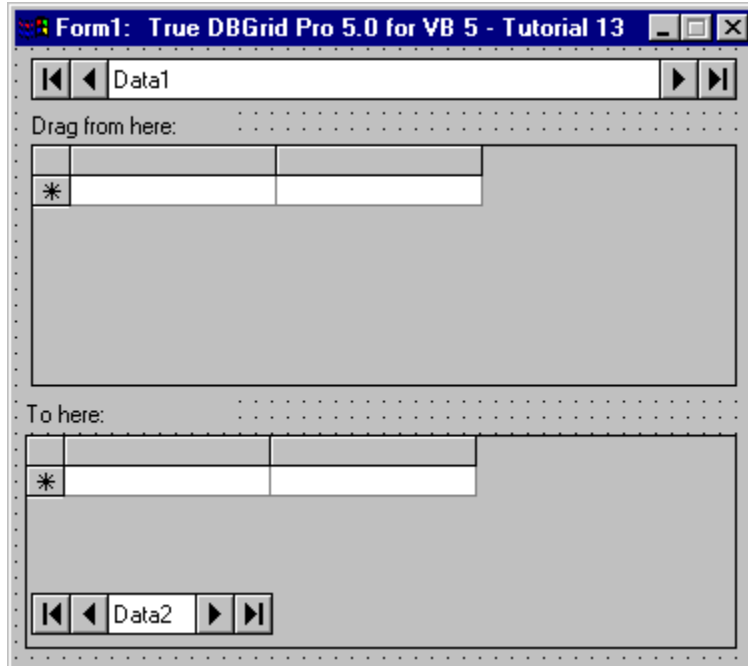
To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 12.

Tutorial 13 - Implementing Drag-and-Drop in True DBGrid

This tutorial demonstrates how you can use the drag-and-drop features of True DBGrid to drag data from one grid and drop it into another.

Step 1. Start a new project.

Step 2. Place two Data controls (Data1 and Data2), two True DBGrid controls (TDBGrid1 and TDBGrid2), and two Labels (Label1 and Label2) on the form (Form1) as shown in this figure:



Step 3. Set the **DatabaseName** property of Data1 and Data2 to TDBGDemo.MDB, the **RecordSource** property of Data1 to Customers, and the **RecordSource** property of Data2 to CallList.

Step 4. Set the **DataSource** property of TDBGrid1 to Data1, and the **DataSource** property of TDBGrid2 to Data2.

Step 5. Set the DragIcon property of TDBGrid1 to DRAG.ICO, which can be found in the BITMAPS subdirectory of the True DBGrid installation directory, or in the ICONS\ARROWS subdirectory of the Visual Basic installation directory.

Step 6. Set the **Visible** property of Data2 to False.

Step 7. Set the **Caption** property of Label1 to Drag from here: and the **Caption** property of Label2 to To here:.

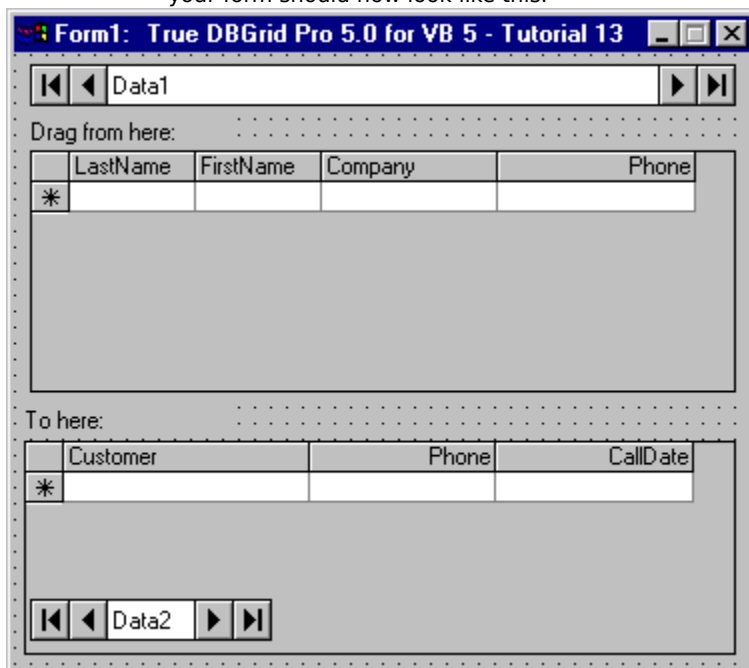
Configuring the grids at design time

We shall configure TDBGrid1 and TDBGrid2 at design time using techniques described in previous tutorials. We will only briefly outline the steps below. If you are not familiar with the basic techniques, please refer to [Tutorial 6](#) and [Design Time Interaction](#) before continuing.

Step 8. Right-click TDBGrid1 to display its context menu. Choose **Retrieve Fields** to configure TDBGrid1's layout to the **Recordset** defined by the **RecordSource** property of Data1.

Step 9. Right-click TDBGrid1 again to display its context menu. This time choose **Edit** to enter the grid's visual editing mode.

- Step 10.** Use the visual editing menu to delete the following columns from TDBGrid1: UserCode, Contacted, and CustType. To delete a column, first select it by clicking its header, then right-click the grid to display the visual editing menu, then select **Delete** from the menu.
- Step 11.** Adjust the column widths of the grid (by dragging the column dividers in the column header area) so that all columns will fit within the grid's display area.
- Step 12.** Right-click the grid to display the visual editing menu and choose **Properties...** to display the Property Pages dialog. On the Splits page, set the **MarqueeStyle** to 1 - Solid Cell Border. On the Columns page, select Column3 (Phone) and type (###)###-#### in the **NumberFormat** combo box. The grid's **NumberFormat** property provides the same functionality as Visual Basic's **Format\$** function.
- Step 13.** Repeat steps 8, 9, and 11 with TDBGrid2. Omit step 10 since we want to keep all three columns (Customer, Phone, and CallDate) in the grid.
- Step 14.** Right-click the grid to display the visual editing menu and choose **Properties...** to display the Property Pages dialog. On the Splits page, set the **MarqueeStyle** to 1 - Solid Cell Border. On the Columns page, select Column1 (Phone) and type (###)###-#### in the **NumberFormat** combo box. Then select Column2 (CallDate) and type MM/DD HH:NNa/p in the **NumberFormat** combo box.
- Step 15.** Click the OK button at the bottom of the Property Pages dialog to accept the changes. The grids on your form should now look like this.



If you choose, you can also set the **ExtendRightColumn** property of both grids to True (on the Splits property page) to eliminate the gray area between the rightmost column and the grid's right border.

Adding code to your project

This section describes the code needed to drag the contents of a cell or row from TDBGrid1 to TDBGrid2. The code assumes that if you drag from the Phone column, you want to drag only the phone number data to another cell in TDBGrid2. If you drag from any other column, however, the code assumes that you want to drag the entire row of data to TDBGrid2 in order to add a new record there.

- Step 16.** Add the following subroutine to your project to reset the **MarqueeStyle** property of each grid, which

is used to provide visual feedback while dragging is in progress. The reset routine will be called to perform clean-up after a drag-and-drop operation concludes.

```
Private Sub ResetDragDrop()  
' Turn off drag-and-drop by resetting the highlight and data  
' control caption.  
If TDBGrid1.MarqueeStyle = dbgSolidCellBorder Then Exit Sub  
TDBGrid1.MarqueeStyle = dbgSolidCellBorder  
TDBGrid2.MarqueeStyle = dbgSolidCellBorder  
Data1.Caption = "Drag a row, or just phone #"  
End Sub
```

Step 17. The **DragCell** event is called when dragging is initiated in a grid cell. The following code prepares for dragging data from the cell:

```
Private Sub TDBGrid1_DragCell(ByVal SplitIndex As Integer, _  
    RowBookmark As Variant, ByVal ColIndex As Integer)  
' DragCell is triggered when True DBGrid detects an attempt  
' to drag data from a cell.  
  
' Set the current cell to the one being dragged.  
TDBGrid1.Col = ColIndex  
TDBGrid1.Bookmark = RowBookmark  
  
' See if the starting cell is in the "Phone" column  
Select Case TDBGrid1.Columns(ColIndex).Caption  
    Case "Phone"  
        ' Highlight the phone number cell to indicate data  
        ' from the cell is being dragged.  
        TDBGrid1.MarqueeStyle = dbgHighlightCell  
        Data1.Caption = "Dragging phone number..."  
    Case Else  
        ' Highlight the entire row to indicate data from the  
        ' entire row is being dragged.  
        TDBGrid1.MarqueeStyle = dbgHighlightRow  
        Data1.Caption = "Create new call when dropped..."  
End Select  
  
' Use Visual Basic manual drag support  
TDBGrid1.Drag vbBeginDrag  
End Sub
```

Step 18. The following code provides different visual feedback to the user when dragging over TDBGrid2:

```
Private Sub TDBGrid2_DragOver(Source As Control, _  
    X As Single, Y As Single, State As Integer)  
' DragOver provides different visual feedback as we are  
' dragging a row, or just the phone number.  
  
Dim dragFrom As String  
Dim overCol As Integer  
Dim overRow As Long  
  
dragFrom = TDBGrid1.Columns(TDBGrid1.Col).DataField  
  
Select Case State  
    Case vbEnter
```

```

        If dragFrom = "Phone" Then
            TDBGrid2.MarqueeStyle = dbgHighlightCell
        Else
            TDBGrid2.MarqueeStyle = dbgNoMarquee
        End If
    Case vbLeave
        TDBGrid2.MarqueeStyle = dbgHighlightCell
    Case vbOver
        If dragFrom = "Phone" Then
            overCol = 1
        Else
            overCol = TDBGrid2.ColContaining(X)
        End If
        overRow = TDBGrid2.RowContaining(Y)
        If overCol >= 0 Then TDBGrid2.Col = overCol
        If overRow >= 0 Then TDBGrid2.Row = overRow
    End Select
End Sub

```

Step 19. When data is dropped on TDBGrid2, we either update the Phone column of TDBGrid2 or add a new record, as shown below:

```

Private Sub TDBGrid2_DragDrop(Source As Control, _
    X As Single, Y As Single)
    ' Allow phone drops right into the cell, other
    ' drops cause a new row to be added

    If TDBGrid1.Columns(TDBGrid1.Col).Caption = "Phone" Then
        TDBGrid2.Columns(TDBGrid2.Col).Value = _
            TDBGrid1.Columns(TDBGrid1.Col).Value
        Data2.UpdateRecord
    Else
        Data2.Recordset.AddNew
        Data2.Recordset!CallDate = Now
        Data2.Recordset!Phone = Data1.Recordset!Phone
        Data2.Recordset!Customer = Data1.Recordset!FirstName & _
            " " & Data1.Recordset!LastName & ", " & _
            Data1.Recordset!Company
        Data2.Recordset.Update
        Data2.Recordset.MoveLast
    End If

    ResetDragDrop
End Sub

```

Step 20. The following code performs clean-up when the mouse returns to TDBGrid1 with the button up:

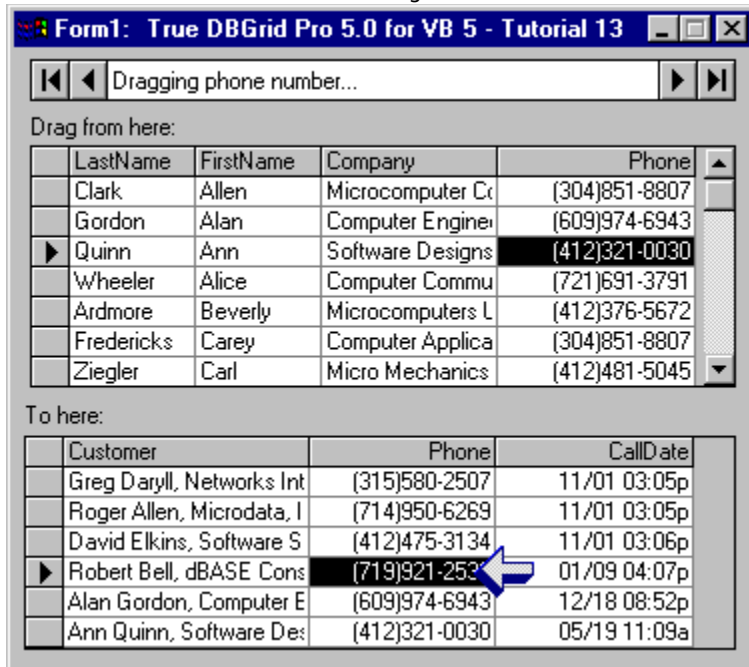
```

Private Sub TDBGrid1_MouseMove(Button As Integer, _
    Shift As Integer, X As Single, Y As Single)
    ' If the button is up and we get MouseMove, that means
    ' we exited the form and tried to drop elsewhere.
    ' Reset the drag upon returning.
    If Button = 0 Then ResetDragDrop
End Sub

```

Run the program and observe the following:

- ⇒ Hold down the left mouse button and drag from a cell in the Phone column of TDBGrid1. As you start dragging, the cell becomes current and is highlighted. The mouse pointer turns into the drag icon specified in step 5.
- ⇒ As you drag over TDBGrid2, the current cell in TDBGrid2 moves to the Phone column and is also highlighted. The current (highlighted) cell of TDBGrid2 stays in the Phone column and moves up and down with the drag motion as shown below:



- ⇒ If you drop (release the left mouse button) on a row in TDBGrid2, the phone number from the highlighted cell in TDBGrid1 will be copied to the phone number column of the row where the drop occurs.
- ⇒ If you start dragging from a column in TDBGrid1 other than the Phone column, the entire row in TDBGrid1 is highlighted, indicating that the entire row of data is being dragged.
- ⇒ As you drag over TDBGrid2, the current cell marquee (a solid border around the cell) disappears as in the following figure.

Form1: True DBGrid Pro 5.0 for VB 5 - Tutorial 13

« Create new call when dropped... »

Drag from here:

LastName	FirstName	Company	Phone
Clark	Allen	Microcomputer Co	(304)851-8807
Gordon	Alan	Computer Engine	(609)974-6943
Quinn	Ann	Software Designs	(412)321-0030
▶ Wheeler	Alice	Computer Commu	(721)691-3791
Ardmore	Beverly	Microcomputers L	(412)376-5672
Fredericks	Carey	Computer Applica	(304)851-8807
Ziegler	Carl	Micro Mechanics	(412)481-5045

To here:

Customer	Phone	CallDate
Greg Daryll, Networks Int	(315)580-2507	11/01 03:05p
Roger Allen, Microdata, I	(714)950-6269	11/01 03:05p
David Elkins, Software S	(412)475-3134	11/01 03:06p
▶ Robert Bell, dBASE Cons	(719)921-2	01/09 04:07p
Alan Gordon, Computer E	(609)974-6943	12/18 08:52p
Ann Quinn, Software Des	(412)321-0030	05/19 11:09a

⇒ If you drop the data on TDBGrid2, a new record is created using the data from the current row of TDBGrid1.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 13.

Tutorial 14 - Creating a Grid with Fixed, Nonscrolling Columns

Often, you would like to prevent one or more columns from scrolling horizontally so that they will always be in view. The **Splits** collection of True DBGrid provides a generalized mechanism for defining groups of adjacent columns, and can be used to implement any number of fixed, nonscrolling columns. In this tutorial, you will learn how to write code to create a grid with two splits, and then "fix" a pair of columns in the leftmost split.

Step 1. Follow steps 1 through 5 of [Tutorial 1](#) to create a project with a TDBGrid bound to a Data control.

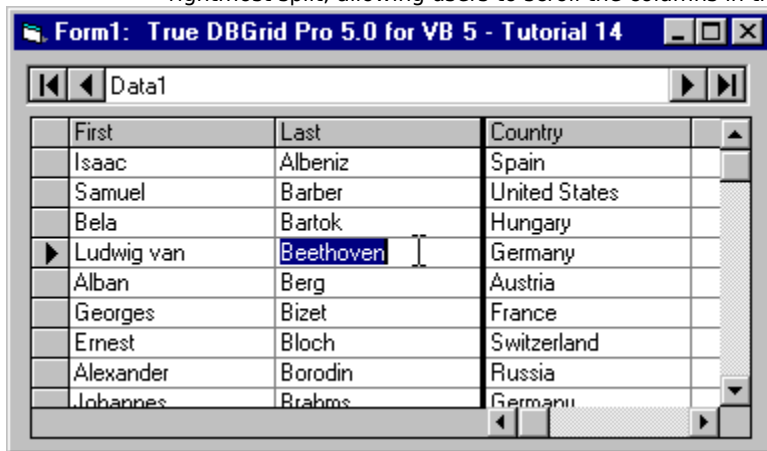
Step 2. In the **Form_Load** event, place the following code to create an additional split and to fix columns 0 and 1 in the leftmost split:

```
Private Sub Form_Load()  
    ' Before modifying the grid's properties, make sure the  
    ' grid is initialized by refreshing the Data control.  
    Data1.Refresh  
  
    ' Create an additional split:  
    Dim S As TrueDBGrid50.Split  
    Set S = TDBGrid1.Splits.Add(0)  
  
    ' Hide all columns in the leftmost split, Splits(0),  
    ' except for columns 0 and 1  
    Dim C As TrueDBGrid50.Column  
    Dim Cols As TrueDBGrid50.Columns  
    Set Cols = TDBGrid1.Splits(0).Columns  
    For Each C In Cols  
        C.Visible = False  
    Next C  
    Cols(0).Visible = True  
    Cols(1).Visible = True  
  
    ' Configure Splits(0) to display exactly two columns,  
    ' and disable resizing  
    With TDBGrid1.Splits(0)  
        .SizeMode = dbgNumberOfColumns  
        .Size = 2  
        .AllowSizing = False  
    End With  
  
    ' Usually, if you fix columns 0 and 1 from scrolling  
    ' in a split, you will want to make them invisible in  
    ' other splits:  
    Set Cols = TDBGrid1.Splits(1).Columns  
    Cols(0).Visible = False  
    Cols(1).Visible = False  
  
    ' Turn off the record selectors in Split 1:  
    TDBGrid1.Splits(1).RecordSelectors = False  
End Sub
```

Run the program and observe the following:

- ⇒ TDBGrid displays data from the Data control as in Tutorial 1.
- ⇒ The two columns (First and Last) in the leftmost split are fixed and cannot be scrolled. In fact, there

is no horizontal scroll bar present under the left split. A horizontal scroll bar appears under the rightmost split, allowing users to scroll the columns in this split.



To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 14.

You can use splits to create fixed, nonscrolling columns anywhere within the grid---even in the middle! You can also use splits to present different views of your data. For example, you can create splits which scroll independently (in the vertical direction) so that users may compare records at the beginning of the database with those at the end. For more information, see [How to Use Splits](#).

Tutorial 15 - Displaying Array Data in Unbound Mode

In this tutorial, you will learn how to use the *unbound mode* (**DataMode** property set to 1 - Unbound) of True DBGrid to display an array of strings.

NOTE: This unbound mode has been retained for backward compatibility with DBGrid and earlier versions of True DBGrid. APEX recommends using unbound extended mode ([Tutorial 16](#)), application mode ([Tutorial 17](#)), or storage mode ([Tutorial 18](#)) instead. For detailed instructions on how to use unbound mode 1, see [Unbound Mode](#).

For simplicity, this tutorial does not cover updating, adding, or deleting records. However, the UNBOUND1.VBP project provides a complete sample that you can use as a template for implementing unbound mode 1. This project is located in the TUTORIAL\UNBOUND1 subdirectory of the True DBGrid installation directory.

Step 1. Start a new project.

Step 2. Place a True DBGrid control (TDBGrid1) on the form (Form1).

Step 3. Set the **DataMode** property of TDBGrid1 to 1 - Unbound (the default value of this property is 0 - Bound).

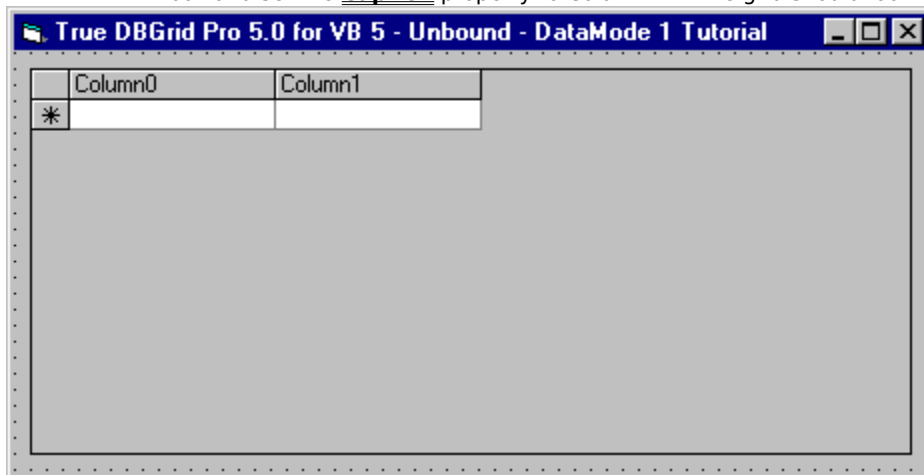
Configuring the grid at design time

This example uses the grid to display an array with two columns. Since True DBGrid displays two columns by default, you do not have to add or delete columns at design time.

Step 4. Right-click TDBGrid1 to display its context menu.

Step 5. Choose **Properties...** to display the Property Pages dialog. On the General property page, clear the **AllowUpdate** check box so that data displayed in the grid will be read-only.

Step 6. Select the Columns properties page by clicking the Columns tab. The Column combo box will display Column0. Set the **Caption** property to Column 0. Select Column1 from the Column combo box and set the **Caption** property to Column 1. The grid should look like this.



Initializing the array data

We first create and initialize a two-dimensional array to hold the data to be displayed in the grid.

Step 7. In the General section of Form1, insert the following declarations:

```
' General declarations
Option Explicit
```



```
' Special Note: Variables that store Row indices will be
' of data type Long (It is reasonable to assume there may
' be larger than 32,767 rows). Those storing Column
' indices will be of type Integer. (It is very unlikely
' there will be more than 32,767 columns.)
```

```
Dim MaxCol As Integer      ' Number of columns
Dim MaxRow As Long        ' Number of rows
Dim GridArray() As Variant ' Array to store the data
```

Step 8. In the **Form_Load** event, initialize the elements of GridArray and set the **ApproxCount** property of the grid accordingly. The **ApproxCount** property is optional, but setting this value will enable the grid to position the vertical scroll bar accurately. If you want to initialize the current cell to a specific location, the **Form_Activate** event is a good place to do it.

```
Private Sub Form_Load()
    ' Initialize the data array. Data must be ready
    ' before the grid is loaded. Form_Load or Main is a
    ' good place to initialize the data.

    Dim I As Integer      ' column index
    Dim J As Long         ' row index
    Dim C As TrueDBGrid50.Column
    Dim Col0 As TrueDBGrid50.Column
    Dim Col1 As TrueDBGrid50.Column
    Dim Col2 As TrueDBGrid50.Column
    Dim Col3 As TrueDBGrid50.Column

    ' Use a 4 columns by 22 rows array as data source.
    ' Since user will be allowed to add data to the grid,
    ' the array may grow in size.

    ' This tutorial project assumes there are 3 columns in
    ' the grid for simplicity. If you wish to change this
    ' number, you will need to modify the code below to
    ' add the correct number of columns.
    MaxCol = 4

    ' You can change MaxRow to show a different number of
    ' rows when the program loads. MaxRow must be >= 0.
    MaxRow = 22

    If MaxRow > 0 Then
        ' If MaxRow = 0, then (MaxRow - 1) equals -1. This
        ' causes an error in the statement below, so we
        ' handle this special case in the Else clause.
        ReDim GridArray(0 To MaxCol - 1, 0 To MaxRow - 1)
    Else
        ReDim GridArray(0 To MaxCol - 1, 0)
    End If

    For I = 0 To MaxCol - 1
        For J = 0 To MaxRow - 1
            GridArray(I, J) = "Row" + Str$(J) + ", Col" _
                + Str$(I)
        Next J
    Next I
End Sub
```

Next I

```
' Allow the user to add and delete rows in the grid.
' (By default, the grid already allows update.)
TDBGrid1.AllowAddNew = True
TDBGrid1.AllowDelete = True

' You can configure the grid either at design time or
' run time. Assuming you have not done this at
' design-time, the following code configures a grid
' with 4 columns. By default, the grid already has 2
' columns, so we just need to add 4 more (don't forget
' to make the new column visible). The existing
' columns are numbered 0 and 1, so we will add new
' columns at positions 2 and 3.
Set C = TDBGrid1.Columns.Add(2)
TDBGrid1.Columns(2).Visible = True
Set C = TDBGrid1.Columns.Add(3)
TDBGrid1.Columns(3).Visible = True

' For the sake of efficiency, we use Column objects
' to reference column properties instead of repeatedly
' going through the grid's Columns collection object.
Set Col0 = TDBGrid1.Columns(0)
Set Col1 = TDBGrid1.Columns(1)
Set Col2 = TDBGrid1.Columns(2)
Set Col3 = TDBGrid1.Columns(3)

' Set column heading text
Col0.Caption = "Column 0"
Col1.Caption = "Column 1"
Col2.Caption = "Column 2 - Locked"
Col3.Caption = "Column 3"

' Set column display widths (in container units)
Col0.Width = 1500
Col1.Width = 1500
Col2.Width = 1500
Col3.Width = 1500

' Set column default values
Col0.DefaultValue = "Default-0"
Col1.DefaultValue = "Default-1"
Col2.DefaultValue = "Default-2"
Col3.DefaultValue = "Default-3"

' Set column text alignment (left-, center-, or
' right-justified)
Col0.Alignment = 0      ' 0 - Left
Col1.Alignment = 2      ' 2 - Center
Col2.Alignment = 1      ' 1 - Right
Col3.Alignment = 1      ' 1 - Right

' Set column locking, which specifies if a column is
' read-only (i.e., the user cannot edit that column)
```

```

Col0.Locked = False      ' Column 0 is editable
Col1.Locked = False      ' Column 1 is editable
Col2.Locked = True       ' Column 2 is read-only
Col3.Locked = False      ' Column 3 is editable

' Inform the grid of how many rows are in the data set.
' This helps with scroll bar positioning.
TDBGrid1.ApproxCount = MaxRow
End Sub

Private Sub Form_Activate()
' Initialize current cell position to upper left corner
TDBGrid1.Row = 0
TDBGrid1.Col = 0
End Sub

```

Displaying data in the unbound grid

When using the Listbox control in Visual Basic, you store all of the data in the control using its **AddItem** method. This *storage* method is neither adequate nor efficient when you have a large amount of data or when the data source continuously changes.

Unlike the Listbox control, True DBGrid's unbound modes do not store your data. You manage the data while the grid handles all display and end-user interactions. Whenever the grid needs to display more rows of data, such as during vertical scrolling, it will fire the **UnboundReadData** event to ask for data from your data source. The grid generally asks for only what it needs to display, but may cache some data for efficiency considerations. You cannot predict when the grid will ask for data, nor can you assume data will be requested in any particular order. Furthermore, since the grid does not store the data, any data that has been requested once may be requested again.

Step 9. Place the following code in the **UnboundReadData** event of TDBGrid1. This example shows how data is provided to the grid via the **RowBuffer** object.

```

Private Sub TDBGrid1_UnboundReadData ( _
    ByVal RowBuf As RowBuffer, StartLocation As Variant, _
    ByVal ReadPriorRows As Boolean)

' UnboundReadData is fired by an unbound grid whenever
' it requires data for display. This event will fire
' when the grid is first shown, when Refresh or ReBind
' is used, when the grid is scrolled, and after a
' record in the grid is modified and the user commits
' the change by moving off of the current row. The
' grid fetches data in "chunks", and the number of rows
' the grid is asking for is given by RowBuf.RowCount.

' RowBuf is the row buffer where you place the data and
' the bookmarks for the rows that the grid is requesting
' to display. It will also hold the number of rows that
' were successfully supplied to the grid.

' StartLocation is a bookmark which specifies the row
' before or after the desired data, depending on the
' value of ReadPriorRows. If we are reading rows after
' StartLocation (ReadPriorRows = False), then the first
' row of data the grid wants is the row after
' StartLocation, and if we are reading rows before

```

```

' StartLocation (ReadPriorRows = True) then the first
' row of data the grid wants is the row before
' StartLocation.

' ReadPriorRows is a boolean value indicating whether
' we are reading rows before (ReadPriorRows = True) or
' after (ReadPriorRows = False) StartLocation.

Dim Bookmark As Variant
Dim I As Long, RelPos As Long
Dim J As Integer, RowsFetched As Integer

' Get a bookmark for the start location
Bookmark = StartLocation

If ReadPriorRows Then
    RelPos = -1 ' Requesting data in rows prior to
                ' StartLocation
Else
    RelPos = 1 ' Requesting data in rows after
              ' StartLocation
End If

RowsFetched = 0
For I = 0 To RowBuf.RowCount - 1
    ' Get the bookmark of the next available row
    Bookmark = GetRelativeBookmark(Bookmark, RelPos)

    ' If the next row is BOF or EOF, then stop
    ' fetching and return any rows fetched up to this
    ' point.
    If IsNull(Bookmark) Then Exit For

    ' Place the record data into the row buffer
    For J = 0 To RowBuf.ColumnCount - 1
        RowBuf.Value(I, J) = GetUserData(Bookmark, J)
    Next J

    ' Set the bookmark for the row
    RowBuf.Bookmark(I) = Bookmark

    ' Increment the count of fetched rows
    RowsFetched = RowsFetched + 1
Next I

' Tell the grid how many rows we fetched
RowBuf.RowCount = RowsFetched
End Sub

```

Step 10. The **UnboundReadData** event handler listed in the previous step calls the following support functions to manage the array data and bookmarks: **MakeBookmark**, **IndexFromBookmark**, **GetRelativeBookmark**, and **GetUserData**.

```

Private Function MakeBookmark(Index As Long) As Variant
    ' This support function is used only by the remaining
    ' support functions. It is not used directly by the

```

```

' unbound events.  It is a good idea to create a
' MakeBookmark function such that all bookmarks can be
' created in the same way.  Thus the method by which
' bookmarks are created is consistent and easy to
' modify.  This function creates a bookmark when given
' an array row index.

' Since we have data stored in an array, we will just
' use the array index as our bookmark.  We will convert
' it to a string first, using the CStr function.

MakeBookmark = CStr(Index)
End Function

Private Function IndexFromBookmark(Bookmark As Variant, _
    ReadPriorRows As Boolean) As Long

' This support function is used only by the remaining
' support functions.  It is not used directly by the
' unbound events.

' This function is the inverse of MakeBookmark.  Given
' a bookmark, IndexFromBookmark returns the row index
' that the given bookmark refers to.  If the given
' bookmark is Null, then it refers to BOF or EOF.  In
' such a case, we need to use ReadPriorRows to
' distinguish between the two.  If ReadPriorRows = True,
' the grid is requesting rows before the current
' location, so we must be at EOF, because no rows exist
' before BOF.  Conversely, if ReadPriorRows = False,
' we must be at BOF.

Dim Index As Long

If IsNull(Bookmark) Then
    If ReadPriorRows Then
        ' Bookmark refers to EOF.  Since (MaxRow - 1)
        ' is the index of the last record, we can use
        ' an index of (MaxRow) to represent EOF.
        IndexFromBookmark = MaxRow
    Else
        ' Bookmark refers to BOF.  Since 0 is the
        ' index of the first record, we can use an
        ' index of -1 to represent BOF.
        IndexFromBookmark = -1
    End If
Else
    ' Convert string to long integer
    Index = Val(Bookmark)

    ' Check to see if the row index is valid:
    ' (0 <= Index < MaxRow).
    ' If not, set it to a large negative number to
    ' indicate that the bookmark is invalid.
    If Index < 0 Or Index >= MaxRow Then Index = -9999

```

```

        IndexFromBookmark = Index
    End If
End Function

Private Function GetRelativeBookmark(Bookmark As Variant, _
    RelPos As Long) As Variant
    ' GetRelativeBookmark is used to get a bookmark for a
    ' row that is a given number of rows away from the given
    ' row. This specific example will always use either -1
    ' or +1 for a relative position, since we will always be
    ' retrieving either the row previous to the current one,
    ' or the row following the current one.

    ' IndexFromBookmark expects a Bookmark and a Boolean
    ' value: this Boolean value is True if the next row to
    ' read is before the current one [in this case,
    ' (RelPos < 0) is True], or False if the next row to
    ' read is after the current one [(RelPos < 0) is False].
    ' This is necessary to distinguish between BOF and EOF
    ' in the IndexFromBookmark function if our bookmark is
    ' Null. Once we get the correct row index from
    ' IndexFromBookmark, we simply add RelPos to it to get
    ' the desired row index and create a bookmark using
    ' that index.

    Dim Index As Long

    Index = IndexFromBookmark(Bookmark, RelPos < 0) + RelPos
    If Index < 0 Or Index >= MaxRow Then
        ' Index refers to a row before the first or after
        ' the last, so just return Null.
        GetRelativeBookmark = Null
    Else
        GetRelativeBookmark = MakeBookmark(Index)
    End If
End Function

Private Function GetUserData(Bookmark As Variant, _
    Col As Integer) As Variant
    ' In this example, GetUserData is called by
    ' UnboundReadData to ask the user what data should be
    ' displayed in a specific cell in the grid. The grid
    ' row the cell is in is the one referred to by the
    ' Bookmark parameter, and the column it is in is given
    ' by the Col parameter. GetUserData is called on a
    ' cell-by-cell basis.

    Dim Index As Long

    ' Figure out which row the bookmark refers to
    Index = IndexFromBookmark(Bookmark, False)

    If Index < 0 Or Index >= MaxRow Or _
        Col < 0 Or Col >= MaxCol Then
        ' Cell position is invalid, so just return null
        ' to indicate failure
    End If
End Function

```

```
        GetUserData = Null
    Else
        GetUserData = GridArray(Col, Index)
    End If
End Function
```

Run the program and observe the following:

- ⇒ The grid displays the elements of GridArray and otherwise behaves as if it were bound to a Data control.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 15.

Tutorial 16 - Displaying Array Data in Unbound Extended Mode

In this tutorial, you will learn how to use the unbound extended mode (**DataMode** property set to 2 - Unbound Extended) of True DBGrid to display an array of strings. As its name implies, the unbound extended mode is an extension of the unbound mode introduced in [Tutorial 15](#). Unbound mode 2 is both easier to use and more efficient than unbound mode 1. For detailed instructions on how to use unbound mode 2, see [Unbound Mode](#).

For simplicity, this tutorial does not cover updating, adding, or deleting records. However, the UNBOUND2.VBP project provides a complete sample that you can use as a template for implementing unbound mode 2. This project is located in the TUTORIAL\UNBOUND2 subdirectory of the True DBGrid installation directory.

Step 1. Start a new project.

Step 2. Place a True DBGrid control (TDBGrid1) on the form (Form1).

Step 3. Set the **DataMode** property of TDBGrid1 to 2 - Unbound Extended (the default value of this property is 0 - Bound).

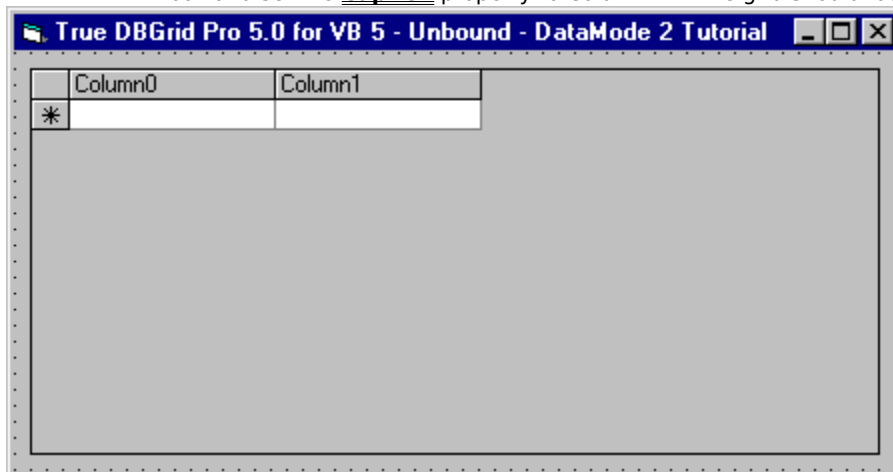
Configuring the grid at design time

We shall configure the grid as in [Tutorial 15](#).

Step 4. Right-click TDBGrid1 to display its context menu.

Step 5. Choose **Properties...** to display the Property Pages dialog. On the General property page, clear the **AllowUpdate** check box so that data displayed in the grid will be read-only.

Step 6. Select the Columns properties page by clicking the Columns tab. The Column combo box will display Column0. Set the **Caption** property to Column 0. Select Column1 from the Column combo box and set the **Caption** property to Column 1. The grid should look like this.



Initializing the array data

We first create and initialize a two-dimensional array to hold the data to be displayed in the grid.

Step 7. In the General section of Form1, insert the following declarations:

```
' General declarations
Option Explicit

' Use a 2 columns by 100 rows array as data source,
Const MaxCol = 2
```



```

Const MaxRow = 100
Dim GridArray(MaxCol, MaxRow) As Variant

```

Step 8. In the **Form_Load** event, initialize the elements of GridArray and set the **ApproxCount** property of the grid accordingly. The **ApproxCount** property is optional, but setting this value will enable the grid to position the vertical scroll bar accurately. If you want to initialize the current cell to a specific location, the **Form_Activate** event is a good place to do it.

```

Private Sub Form_Load()
    ' Initialize the data array. Data must be ready
    ' before the grid is loaded. Form_Load or Main is a
    ' good place to initialize the data.
    Dim I As Integer, J As Long
    For I = 0 To MaxCol - 1
        For J = 0 To MaxRow - 1
            GridArray(I, J) = "Row" + Str$(J) + ", Col" _
                + Str$(I)
        Next J
    Next I

    ' Inform the grid of how many rows are in the data set.
    ' This helps with scroll bar positioning.
    TDBGrid1.ApproxCount = MaxRow
End Sub

Private Sub Form_Activate()
    ' Initialize current cell position to upper left corner
    TDBGrid1.Row = 0
    TDBGrid1.Col = 0
End Sub

```

Displaying data in the unbound grid

As explained in [Tutorial 15](#), True DBGrid does not store your data in any of its unbound modes. In unbound mode 2, whenever the grid needs to display more rows of data, it will fire the **UnboundReadDataEx** event (instead of **UnboundReadData**) to ask for data from your data source.

Step 9. Place the following code in the **UnboundReadDataEx** event of TDBGrid1. This example shows how data and the row's ordinal position are provided to the grid via the **RowBuffer** object and the **ApproximatePosition** arguments, respectively:

```

Private Sub TDBGrid1_UnboundReadDataEx( _
    ByVal RowBuf As RowBuffer, StartLocation As Variant, _
    ByVal Offset As Long, ApproximatePosition As Long)

    ' UnboundReadData is fired by an unbound grid whenever
    ' it requires data for display. This event will fire
    ' when the grid is first shown, when Refresh or ReBind
    ' is used, when the grid is scrolled, and after a
    ' record in the grid is modified and the user commits
    ' the change by moving off of the current row. The
    ' grid fetches data in "chunks", and the number of rows
    ' the grid is asking for is given by RowBuf.RowCount.

    ' RowBuf is the row buffer where you place the data
    ' the bookmarks for the rows that the grid is
    ' requesting to display. It will also hold the number
    ' of rows that were successfully supplied to the grid.

```

' StartLocation is a bookmark which, together with
' Offset, specifies the row for the programmer to start
' transferring data. A StartLocation of Null indicates
' a request for data from BOF or EOF.

' Offset specifies the relative position (from
' StartLocation) of the row for the programmer to start
' transferring data. A positive number indicates a
' forward relative position while a negative number
' indicates a backward relative position. Regardless
' of whether the rows to be read are before or after
' StartLocation, rows are always fetched going forward
' (this is why there is no ReadPriorRows parameter to
' the procedure).

' If you page down on the grid, for instance, the new
' top row of the grid will have an index greater than
' the StartLocation (Offset > 0). If you page up on
' the grid, the new index is less than that of
' StartLocation, so Offset < 0. If StartLocation is
' a bookmark to row N, the grid always asks for row
' data in the following order:
' (N + Offset), (N + Offset + 1), (N + Offset + 2)...

' ApproximatePosition is a value you can set to indicate
' the ordinal position of (StartLocation + Offset).
' Setting this variable will enhance the ability of the
' grid to display its vertical scroll bar accurately.
' If the exact ordinal position of the new location is
' not known, you can set it to a reasonable,
' approximate value, or just ignore this parameter.

```
Dim ColIndex As Integer, J As Integer  
Dim RowsFetched As Integer, I As Long  
Dim NewPosition As Long, Bookmark As Variant
```

```
RowsFetched = 0
```

```
For I = 0 To RowBuf.RowCount - 1  
    ' Get the bookmark of the next available row  
    Bookmark = GetRelativeBookmark(StartLocation, _  
        Offset + I)  
  
    ' If the next row is BOF or EOF, then stop fetching  
    ' and return any rows fetched up to this point.  
    If IsNull(Bookmark) Then Exit For  
  
    ' Place the record data into the row buffer  
    For J = 0 To RowBuf.ColumnCount - 1  
        ColIndex = RowBuf.ColumnIndex(I, J)  
        RowBuf.Value(I, J) = GetUserData(Bookmark, _  
            ColIndex)  
    Next J
```

```

        ' Set the bookmark for the row
        RowBuf.Bookmark(I) = Bookmark

        ' Increment the count of fetched rows
        RowsFetched = RowsFetched + 1
    Next I

    ' Tell the grid how many rows were fetched
    RowBuf.RowCount = RowsFetched

    ' Set the approximate scroll bar position. Only
    ' nonnegative values of IndexFromBookmark() are valid.
    NewPosition = IndexFromBookmark(StartLocation, Offset)
    If NewPosition >= 0 Then _
        ApproximatePosition = NewPosition
End Sub

```

Step 10. The **UnboundReadDataEx** event handler used in the previous step calls the following support functions to manage the array data and bookmarks: **MakeBookmark**, **IndexFromBookmark**, **GetRelativeBookmark**, and **GetUserData**.

```

Private Function MakeBookmark(Index As Long) As Variant
    ' This support function is used only by the remaining
    ' support functions. It is not used directly by the
    ' unbound events. It is a good idea to create a
    ' MakeBookmark function such that all bookmarks can be
    ' created in the same way. Thus the method by which
    ' bookmarks are created is consistent and easy to
    ' modify. This function creates a bookmark when given
    ' an array row index.

    ' Since we have data stored in an array, we will just
    ' use the array index as our bookmark. We will convert
    ' it to a string first, using the CStr function.

    MakeBookmark = CStr(Index)
End Function

Private Function IndexFromBookmark(Bookmark As Variant, _
    Offset As Long) As Long
    ' This support function is used only by the remaining
    ' support functions. It is not used directly by the
    ' unbound events.

    ' IndexFromBookmark computes the row index that
    ' corresponds to a row that is (Offset) rows from the
    ' row specified by the Bookmark parameter. For example,
    ' if Bookmark refers to the index 50 of the dataset
    ' array and Offset = -10, then IndexFromBookmark will
    ' return 50 + (-10), or 40. Thus to get the index of
    ' the row specified by the bookmark itself, call
    ' IndexFromBookmark with an Offset of 0. If the given
    ' Bookmark is Null, it refers to BOF or EOF. If
    ' Offset < 0, the grid is requesting rows before the
    ' row specified by Bookmark, and so we must be at EOF
    ' because prior rows do not exist for BOF. Conversely,

```

```

' if Offset > 0, we are at BOF.

Dim Index As Long

If IsNull(Bookmark) Then
    If Offset < 0 Then
        ' Bookmark refers to EOF. Since (MaxRow - 1)
        ' is the index of the last record, we can use
        ' an index of (MaxRow) to represent EOF.
        Index = MaxRow + Offset
    Else
        ' Bookmark refers to BOF. Since 0 is the index
        ' of the first record, we can use an index of
        ' -1 to represent BOF.
        Index = -1 + Offset
    End If
Else
    ' Convert string to long integer
    Index = Val(Bookmark) + Offset
End If

' Check to see if the row index is valid:
' (0 <= Index < MaxRow).
' If not, set it to a large negative number to
' indicate that it is invalid.
If Index >= 0 And Index < MaxRow Then
    IndexFromBookmark = Index
Else
    IndexFromBookmark = -9999
End If
End Function

Private Function GetRelativeBookmark(Bookmark As Variant, _
    Offset As Long) As Variant
    ' GetRelativeBookmark is used to get a bookmark for a
    ' row that is a specified number of rows away from the
    ' given row. Offset specifies the number of rows to
    ' move. A positive Offset indicates that the desired
    ' row is after the one referred to by Bookmark, and a
    ' negative Offset means it is before the one referred
    ' to by Bookmark.

    Dim Index As Long

    ' Compute the row index for the desired row
    Index = IndexFromBookmark(Bookmark, Offset)
    If Index < 0 Or Index >= MaxRow Then
        ' Index refers to a row before the first or after
        ' the last, so just return Null.
        GetRelativeBookmark = Null
    Else
        GetRelativeBookmark = MakeBookmark(Index)
    End If
End Function

Private Function GetUserData(Bookmark As Variant, _

```

```

        Col As Integer) As Variant
' In this example, GetUserData is called by
' UnboundReadData to ask the user what data should be
' displayed in a specific cell in the grid. The grid
' row the cell is in is the one referred to by the
' Bookmark parameter, and the column it is in is given
' by the Col parameter. GetUserData is called on a
' cell-by-cell basis.

Dim Index As Long

' Figure out which row the bookmark refers to
Index = IndexFromBookmark(Bookmark, 0)

If Index < 0 Or Index >= MaxRow Or _
   Col < 0 Or Col >= MaxCol Then
    ' Cell position is invalid, so just return null to
    ' indicate failure
    GetUserData = Null
Else
    GetUserData = GridArray(Col, Index)
End If
End Function

```

Run the program and observe the following:

- ⇒ The grid displays the elements of GridArray and otherwise behaves as if it were bound to a Data control.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 16.

Tutorial 17 - Displaying Array Data in Unbound Application Mode

In this tutorial, you will learn how to use the unbound application mode (**DataMode** property set to 3 - Application) of True DBGrid to display an array of strings. When the data source is an array, application mode is easier to use than either unbound mode (1) or unbound extended mode (2). In application mode, the grid fetches data one cell at a time, and you must program the **UnboundGetRelativeBookmark** event in order for the grid to obtain the bookmark of a row. For detailed instructions on how to use unbound mode 3, see [Application Mode](#).

For simplicity, this tutorial does not cover updating, adding, or deleting records. However, the UNBOUND3.VBP project provides a complete sample that you can use as a template for implementing application mode. This project is located in the TUTORIAL\UNBOUND3 subdirectory of the True DBGrid installation directory.

Step 1. Start a new project.

Step 2. Place a True DBGrid control (TDBGrid1) on the form (Form1).

Step 3. Set the **DataMode** property of TDBGrid1 to 3 - Application (the default value of this property is 0 - Bound).

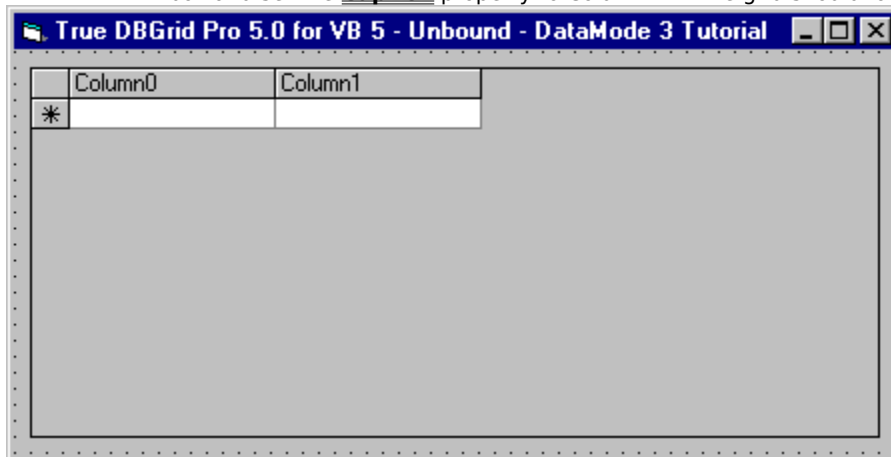
Configuring the grid at design time

We shall configure the grid as in [Tutorial 15](#).

Step 4. Right-click TDBGrid1 to display its context menu.

Step 5. Choose **Properties...** to display the Property Pages dialog. On the General property page, clear the **AllowUpdate** check box so that data displayed in the grid will be read-only.

Step 6. Select the Columns properties page by clicking the Columns tab. The Column combo box will display Column0. Set the **Caption** property to Column 0. Select Column1 from the Column combo box and set the **Caption** property to Column 1. The grid should look like this.



Initializing the array data

We first create and initialize a two-dimensional array to hold the data to be displayed in the grid.

Step 7. In the General section of Form1, insert the following declarations:

```
' General declarations
Option Explicit

' Use a 2 columns by 100 rows array as data source,
Const MaxCol = 2
```

```
Const MaxRow = 100
Dim GridArray(MaxCol, MaxRow) As Variant
```

Step 8. In the **Form_Load** event, initialize the elements of GridArray and set the **ApproxCount** property of the grid accordingly. The **ApproxCount** property is optional, but setting this value will enable the grid to position the vertical scroll bar accurately. If you want to initialize the current cell to a specific location, the **Form_Activate** event is a good place to do it.

```
Private Sub Form_Load()
    ' Initialize the data array. Data must be ready
    ' before the grid is loaded. Form_Load or Main is a
    ' good place to initialize the data.
    Dim I As Integer, J As Long
    For I = 0 To MaxCol - 1
        For J = 0 To MaxRow - 1
            GridArray(I, J) = "Row" + Str$(J) + ", Col" _
                + Str$(I)
        Next J
    Next I

    ' Inform the grid of how many rows are in the data set.
    ' This helps with scroll bar positioning.
    TDBGrid1.ApproxCount = MaxRow
End Sub

Private Sub Form_Activate()
    ' Initialize current cell position to upper left corner
    TDBGrid1.Row = 0
    TDBGrid1.Col = 0
End Sub
```

Displaying data in the unbound grid

As explained in [Tutorial 15](#), True DBGrid does not store your data in any of its unbound modes. Whenever the grid needs to display data in a cell, it will fire the **ClassicRead** event to ask for data from your data source. Unlike unbound modes 1 and 2, the **ClassicRead** event does not use the **RowBuffer** object to transfer data several rows at a time. Instead, the **ClassicRead** event fetches data one cell at a time. When using application mode, you must also program the **UnboundGetRelativeBookmark** event in order for the grid to obtain the bookmark of a row.

Step 9. Place the following code in the **UnboundGetRelativeBookmark** and the **ClassicRead** events of TDBGrid1. These events show how to provide bookmarks and data to the grid, respectively:

```
Private Sub TDBGrid1_UnboundGetRelativeBookmark( _
    StartLocation As Variant, ByVal Offset As Long, _
    NewLocation As Variant, ApproximatePosition As Long)

    ' TDBGrid1 calls this routine each time it needs to
    ' reposition itself. StartLocation is a bookmark
    ' supplied by the grid to indicate the "current"
    ' position -- the row we are moving from. Offset is
    ' the number of rows we must move from StartLocation
    ' in order to arrive at the desired destination row.
    ' A positive offset means the desired record is after
    ' the StartLocation, and a negative offset means the
    ' desired record is before StartLocation.

    ' If StartLocation is NULL, then we are positioning
```

```

' from either BOF or EOF. Once we determine the
' correct index for StartLocation, we will simply add
' the offset to get the correct destination row.
' GetRelativeBookmark already does all of this, so we
' just call it here.
NewLocation = GetRelativeBookmark(StartLocation, Offset)

' If we are on a valid data row (i.e., not at BOF or
' EOF), then set the ApproximatePosition (the ordinal
' row number) to improve scroll bar accuracy. We can
' call IndexFromBookmark to do this.
If Not IsNull(NewLocation) Then
    ApproximatePosition = IndexFromBookmark(NewLocation, 0)
End If
End Sub

Private Sub TDBGrid1_ClassicRead(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

' ClassicRead is analogous to the Fetch event of the
' TrueGrid Pro VBX control. When the grid needs data
' in DataMode 3, it fires a ClassicRead event for
' each visible cell on the grid to retrieve the data
' that will be shown there, so it fires on a
' cell-by-cell basis. The cell that this event is
' firing for is specified by the Bookmark (which
' tells which row to fetch the data from) and the
' Col parameter (which gives the column index). The
' only difference from the Fetch event of the VBX is
' that the row to fetch is specified by a Bookmark
' and not an integral row index. Thus, you must
' determine which row in your data source the bookmark.
' GetUserData uses the IndexFromBookmark function to
' do that.

' Assume that a function GetUserData(Bookmark, Col,
' Value) takes a row bookmark, a column index, and
' a variant which will hold the appropriate data to
' be fetched from the array or database. The function
' returns the fetched data in the Value parameter if
' the fetch is successful, otherwise, it returns Null.

Value = GetUserData(Bookmark, Col)
End Sub

```

Step 10. The **UnboundGetRelativeBookmark** and the **ClassicRead** event handlers used in the previous step call the following support functions to manage the array data and bookmarks: MakeBookmark, IndexFromBookmark, GetRelativeBookmark, and GetUserData.

```

Private Function MakeBookmark(Index As Long) As Variant
' This support function is used only by the remaining
' support functions. It is not used directly by the
' unbound events. It is a good idea to create a
' MakeBookmark function such that all bookmarks can
' be created in the same way. Thus the method by
' which bookmarks are created is consistent and easy

```



```
' to modify. This function creates a bookmark when
' given an array row index.
```

```
' Since we have data stored in an array, we will just
' use the array index as our bookmark. We will convert
' it to a string first, using the Str$ function. Thus,
' if Index = 27, the Bookmark that is created is the
' string " 27". (Str$ always leaves a leading space
' for the sign of the number.)
```

```
MakeBookmark = Str$(Index)
```

```
End Function
```

```
Private Function IndexFromBookmark(Bookmark As Variant, _
    Offset As Long) As Long
```

```
' This support function is used only by the remaining
' support functions. It is not used directly by the
' unbound events.
```

```
' IndexFromBookmark computes the row index that
' corresponds to a row that is (Offset) rows from the
' row specified by the Bookmark parameter. For
' example, if Bookmark refers to the index 50 of the
' dataset array and Offset = -10, then
' IndexFromBookmark will return 50 + (-10), or 40.
' Thus to get the index of the row specified by the
' bookmark itself, call IndexFromBookmark with an
' Offset of 0. If the given Bookmark is Null, it
' refers to BOF or EOF. If Offset < 0, the grid is
' requesting rows before the row specified by
' Bookmark, and so we must be at EOF because prior
' rows do not exist for BOF. Conversely, if
' Offset > 0, we are at BOF.
```

```
Dim Index As Long
```

```
If IsNull(Bookmark) Then
```

```
    If Offset < 0 Then
```

```
        ' Bookmark refers to EOF. Since (MaxRow - 1)
        ' is the index of the last record, we can use
        ' an index of (MaxRow) to represent EOF.
```

```
        Index = MaxRow + Offset
```

```
    Else
```

```
        ' Bookmark refers to BOF. Since 0 is the index
        ' of the first record, we can use an index of
        ' -1 to represent BOF.
```

```
        Index = -1 + Offset
```

```
    End If
```

```
Else
```

```
    ' Convert string to long integer
```

```
    Index = Val(Bookmark) + Offset
```

```
End If
```

```
' Check to see if the row index is valid:
```

```
' (0 <= Index < MaxRow).
```

```

' If not, set it to a large negative number to
' indicate that it is invalid.
If Index >= 0 And Index < MaxRow Then
    IndexFromBookmark = Index
Else
    IndexFromBookmark = -9999
End If
End Function

Private Function GetRelativeBookmark(Bookmark As Variant, _
    Offset As Long) As Variant
' GetRelativeBookmark is used to get a bookmark for
' a row that is a specified number of rows away from
' the given row. Offset specifies the number of rows
' to move. A positive Offset indicates that the
' desired row is after the one referred to by Bookmark,
' and a negative Offset means it is before the one
' referred to by Bookmark.

Dim Index As Long

' Compute the row index for the desired row
Index = IndexFromBookmark(Bookmark, Offset)
If Index < 0 Or Index >= MaxRow Then
    ' Index refers to a row before the first or
    ' after the last, so just return Null.
    GetRelativeBookmark = Null
Else
    GetRelativeBookmark = MakeBookmark(Index)
End If
End Function

Private Function GetUserData(Bookmark As Variant, _
    Col As Integer) As Variant
' In this example, GetUserData is called by
' UnboundReadData to ask the user what data should
' be displayed in a specific cell in the grid. The
' grid row the cell is in is the one referred to by
' the Bookmark parameter, and the column it is in it
' given by the Col parameter. GetUserData is called
' on a cell-by-cell basis.

Dim Index As Long

' Figure out which row the bookmark refers to
Index = IndexFromBookmark(Bookmark, 0)

If Index < 0 Or Index >= MaxRow Or Col < 0 Or _
    Col >= MaxCol Then
    ' Cell position is invalid, so just return null
    ' to indicate failure
    GetUserData = Null
Else
    GetUserData = GridArray(Col, Index)
End If
End Function

```

Run the program and observe the following:

- ⇒ The grid displays the elements of GridArray and otherwise behaves as if it were bound to a Data control.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 17.

Tutorial 18 - Displaying Array Data in Unbound Storage Mode

In this tutorial, you will learn how to use the unbound *storage mode* (**DataMode** property set to 4 - Storage) of True DBGrid to display an array of strings. Unlike unbound (extended) and application modes, storage mode does not fire data-gathering events. Instead, it uses an **XArray** ActiveX object to store, access, and maintain data.

In code, you create, re-dimension, and populate an **XArray** object with your data just as you would a Visual Basic array, then assign the **XArray** object to the **Array** property of the grid. The data will then be maintained and exchanged between the grid and the XArray object automatically. There are no unbound events to write, making this mode the easiest to use.

NOTE: Storage mode is not available in the 16-bit version of True DBGrid, since there is currently no 16-bit version of **XArray**.

Step 1. Start a new project.

Step 2. Place a True DBGrid control (TDBGrid1) on the form (Form1).

Step 3. Set the **DataMode** property of TDBGrid1 to 4 - Storage (the default value of this property is 0 - Bound).

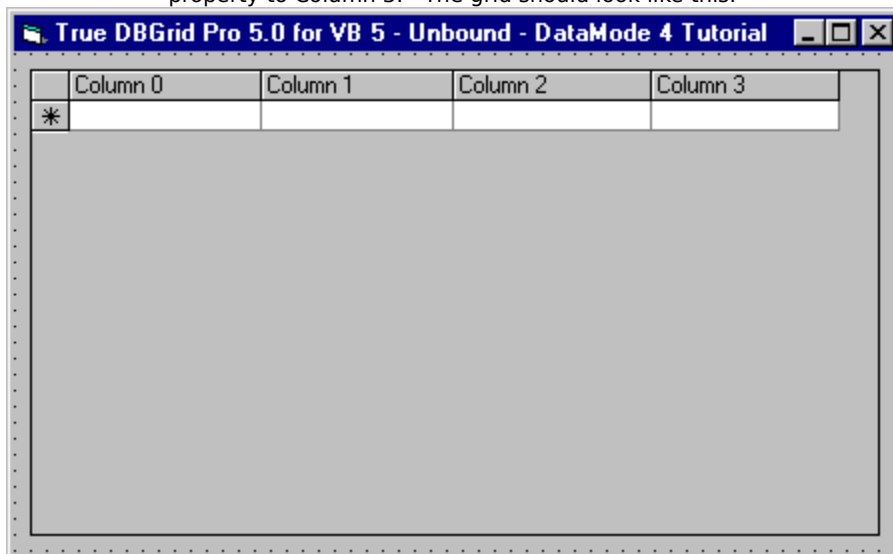
Configuring the grid at design time

We shall configure the grid to display four columns.

Step 4. Right-click TDBGrid1 to display its context menu.

Step 5. Choose **Properties...** to display the Property Pages dialog. On the General property page, check **AllowUpdate**, **AllowAddNew**, and **AllowDelete**.

Step 6. Select the Columns property page by clicking the Columns tab. The Column combo box will display Column0. Set the **Caption** property to Column 0. Select Column1 from the Column combo box and set the **Caption** property to Column 1. Select Column2 from the Column combo box and set the **Caption** property to Column 2. Select Column3 from the Column combo box and set the **Caption** property to Column 3. The grid should look like this.



Adding XArray to the project

Before writing the code, we need to add the APEX XArray Object to the project. The exact steps depend on whether you are using Visual Basic 4.0 or 5.0.

Step 7. If you are using Visual Basic 4.0, select **References...** from the **Tools** menu to display a list of available type library references. If you are using Visual Basic 5.0, select **References...** from the **Project** menu to display a list of available type library references. Select the check box labeled APEX XArray Object (XARRAY32.OCX), then press the **OK** button.

Initializing the array data

Next, we create the **XArray** object in code.

Step 8. In the General section of Form1, insert the following declarations:

```
' General declarations
Option Explicit

Dim x As New XArray
```

Step 9. In the **Form_Load** event, we **ReDim** the **XArray** to contain 100 rows and 4 columns. After populating the **XArray**, we assign it to the grid's **Array** property.

```
Private Sub Form_Load()

    ' Allocate space for 100 rows, 4 columns
    x.ReDim 0, 99, 0, 3

    Dim row, col As Long

    ' The LowerBound and UpperBound properties correspond
    ' to the LBound and UBound functions in Visual Basic.
    ' Hard-coded dimensions can be used instead, if known.
    For row = x.LowerBound(1) To x.UpperBound(1)
        For col = x.LowerBound(2) To x.UpperBound(2)
            x(row, col) = "Row " & row & ", Col " & col
        Next col
    Next row

    ' Bind True DBGrid Control to this XArray instance
    Set TDBGrid1.Array = x

End Sub
```

Run the program and observe the following:

⇒ The grid displays the data assigned to the **XArray** and appears as follows.

	Column 0	Column 1	Column 2	Column 3
▶	Row 0, Col 0	Row 0, Col 1	Row 0, Col 2	Row 0, Col 3
	Row 1, Col 0	Row 1, Col 1	Row 1, Col 2	Row 1, Col 3
	Row 2, Col 0	Row 2, Col 1	Row 2, Col 2	Row 2, Col 3
	Row 3, Col 0	Row 3, Col 1	Row 3, Col 2	Row 3, Col 3
	Row 4, Col 0	Row 4, Col 1	Row 4, Col 2	Row 4, Col 3
	Row 5, Col 0	Row 5, Col 1	Row 5, Col 2	Row 5, Col 3
	Row 6, Col 0	Row 6, Col 1	Row 6, Col 2	Row 6, Col 3
	Row 7, Col 0	Row 7, Col 1	Row 7, Col 2	Row 7, Col 3
	Row 8, Col 0	Row 8, Col 1	Row 8, Col 2	Row 8, Col 3
	Row 9, Col 0	Row 9, Col 1	Row 9, Col 2	Row 9, Col 3
	Row 10, Col 0	Row 10, Col 1	Row 10, Col 2	Row 10, Col 3
	Row 11, Col 0	Row 11, Col 1	Row 11, Col 2	Row 11, Col 3
	Row 12, Col 0	Row 12, Col 1	Row 12, Col 2	Row 12, Col 3
	Row 13, Col 0	Row 13, Col 1	Row 13, Col 2	Row 13, Col 3
	Row 14, Col 0	Row 14, Col 1	Row 14, Col 2	Row 14, Col 3

- ⇒ Type a different value into any cell. When you click another cell, the new data is saved to the array.
- ⇒ Select an entire row by clicking its record selector. Press the DEL key. The grid will remove the record from the display (and from the array).
- ⇒ Scroll down until the AddNew row appears. Enter any data into the cells. When you click a cell in another row, the newly added row is saved.

To end the program, press the End button on the Visual Basic toolbar. You have successfully completed Tutorial 18.

Congratulations, you have successfully completed all 18 tutorials!

Object Model

True DBGrid was developed using the latest ActiveX and data binding technologies. The True DBGrid control and its programmable components are all ActiveX objects designed according to Microsoft specifications. If you are already familiar with the Visual Basic 5.0 object and collection models, you will have no problem using True DBGrid.

If you are new to Visual Basic 5.0, please read [Working with Objects and Collections](#), which illustrates how to manipulate True DBGrid objects in code. Although individual objects are designed to perform different tasks, the techniques used to manipulate them are the same. Once you have mastered these common programming constructs, using Visual Basic 5.0 and ActiveX controls will be quite easy and intuitive.

Regardless of your experience level, please read the following section, as it provides a thumbnail sketch of all True DBGrid objects and collections.

{button ,JI(`,`True_DBGrid_Objects_and_Collections')} True DBGrid Objects and Collections
{button ,JI(`,`TDBGrid_Control')} TDBGrid Control
{button ,JI(`,`TDBDropDown_Control')} TDBDropDown Control
{button ,JI(`,`Column_Object')} Column Object, Columns Collection
{button ,JI(`,`Layouts_Collection')} Layouts Collection
{button ,JI(`,`RowBuffer_Object')} RowBuffer Object
{button ,JI(`,`SelBookmarks_Collection')} SelBookmarks Collection
{button ,JI(`,`Split_Object')} Split Object, Splits Collection
{button ,JI(`,`Style_Object')} Style Object, Styles Collection
{button ,JI(`,`ValueItem_Object')} ValueItem Object, ValueItems Collection
{button ,JI(`,`XArray_Object')} XArray Object
{button ,JI(`,`Working_with_Objects_and_Collections')} Working with Objects and Collections

True DBGrid Objects and Collections

True DBGrid provides a rich set of properties, methods, and events that enable you to develop sophisticated database applications. The organization imposed by True DBGrid's object model makes it easier to work with such a large feature set.

Objects and collections that refer to visual entities, such as columns, can be customized at design time or run time. Objects and collections that refer to abstract entities, such as arrays and bookmarks, are only available in code at run time.

When you add True DBGrid file to a Visual Basic project, the following controls are added to the Toolbox:

<u>TDBGrid</u>	True DBGrid ActiveX control.
<u>TDBDropDown</u>	True DBDropDown ActiveX control.

The type library for True DBGrid also contains definitions for the following objects:

<u>Column</u>	Represents a column of data within a grid or split.
<u>Split</u>	Represents a group of adjacent columns that scroll as a unit.
<u>Style</u>	Encapsulates font, color, and formatting information.
<u>ValueItem</u>	Allowable input value for a column, with optional translation.
<u>RowBuffer</u>	Transfers data to and from row-based unbound mode events.

A *collection* is an object used to group similar data items, such as bookmarks, or visual objects, such as grid columns. In general, a group of similar items in True DBGrid is implemented as a collection. Since a collection is an object, you can manipulate it in code just as you would any other object. The type library for True DBGrid contains definitions for the following collections:

<u>Columns</u>	Contains zero or more <u>Column</u> objects in a grid or split.
<u>Layouts</u>	Contains zero or more named grid layouts.
<u>SelBookmarks</u>	Contains zero or more selected row bookmarks.
<u>Splits</u>	Contains one or more <u>Split</u> objects in a grid.
<u>Styles</u>	Contains built-in and user-defined <u>Style</u> objects for a grid.
<u>ValueItems</u>	Contains zero or more <u>ValueItem</u> objects for a column.

When using True DBGrid's *storage mode* (**DataMode 4**, available in 32-bit versions only), you also need to add a reference to the APEX XArray Object to your project. This is not a control, but a reference that defines a single nongraphical object:

<u>XArray</u>	Variant array used as a data source in storage mode.
----------------------	--

The following sections provide a brief overview of True DBGrid's objects and collections.

TDBGrid Control

The **TDBGrid** control is the primary object of True DBGrid. When you place a True DBGrid control on a Visual Basic form at design time, an instance of the **TDBGrid** control object is created. **TDBGrid** objects created in Visual Basic will be given default names of TDBGrid1, TDBGrid2, and so forth. You can change the **TDBGrid** object name in the Visual Basic Properties window at design time.

TDBDropDown Control

The **TDBDropDown** control, which is a subset of the **TDBGrid** control, is used as a multicolumn drop-down list box for a grid column. You cannot use it as a standalone control.

At design time, you can place a **TDBDropDown** control on a Visual Basic form just as you would a **TDBGrid** control. However, the drop-down control will be invisible at run time unless it is attached to a **Column** object of a **TDBGrid** control.

To use the drop-down control, set the **DropDown** property of a grid column to the name of a **TDBDropDown** control at either design time or run time. At run time, when the user clicks the in-cell button for that column, the **TDBDropDown** control will appear below the grid's current cell. If the user selects an item from the drop-down control, the grid's current cell is updated.

The **TDBDropDown** control supports incremental search as well as all of the **DataMode** settings of the **TDBGrid** control. For more information, see [TDBDropDown at Design Time](#) and [Using the TDBDropDown Control](#).

Column Object, Columns Collection

Each column within a **TDBGrid** control, **TDBDropDown** control, or **Split** object is represented by a **Column** object. When a grid or drop-down control is bound to a database, each **Column** object is usually associated with a database field, although True DBGrid also supports unbound columns in bound mode. When a control is first created, it contains two **Column** objects by default.

The **TDBGrid** control, the **TDBDropDown** control, and the **Split** object all maintain a **Columns** collection to hold and manipulate **Column** objects. This collection is accessed using the **Columns** property.

For more information, see [Configuring Columns at Run Time](#).

Layouts Collection

In True DBGrid, the term *layout* refers to the complete set of persistent properties for a **TDBGrid** or **TDBDropDown** control. In earlier versions of True DBGrid, each control had only one layout, which was stored by Visual Basic in an .FRX file when the containing form was saved to disk.

With version 5.0 of True DBGrid, you can store multiple grid layouts in arbitrary files, then recall them later. You can even take advantage of this feature in code to enable your end-users to save their layout preferences.

The **Layouts** collection facilitates switching between multiple named layouts at either design time or run time. Both the **TDBGrid** and **TDBDropDown** controls support this collection.

For more information, see [Reusable Layouts](#).

RowBuffer Object

The **RowBuffer** object is only used when the **DataMode** property is set to 1 - Unbound or 2 - Unbound Extended. It exists only to transfer data to and from the grid in the row-based unbound mode events. You cannot create a standalone **RowBuffer** object.

For more information, see [Unbound Mode](#).

SelBookmarks Collection

When multiple rows are selected and highlighted, the grid uses the **SelBookmarks** collection to store and manipulate the bookmarks of the selected rows. This collection is accessed using the **SelBookmarks** property.

For more information, see [Selecting and Highlighting Records](#).

Split Object, Splits Collection

True DBGrid supports Excel-like splits that divide the grid into vertical panes to provide users with different views of the data source. Each split is represented by a **Split** object and contains a group of adjacent columns that scroll as a unit.

When a **TDBGrid** control is created, it contains one **Split** object by default. Many of the properties of the **Split** object also apply to the **TDBGrid** control as a whole, so you do not need to concern yourself with splits until you actually need to use them, such as when creating fixed, nonscrolling columns.

The **TDBGrid** control maintains a **Splits** collection to hold and manipulate **Split** objects. A grid has one split by default, but may contain multiple splits. This collection is accessed using the **Splits** property.

For more information, see [How to Use Splits](#).

Style Object, Styles Collection

Style objects encapsulate font, color, and formatting information for a **TDBGrid**, **TDBDropDown**, **Split**, or **Column** object. The **Style** object is a very flexible and powerful tool that provides Excel- and Word-like formatting capabilities for controlling the appearance of the grid's display.

When a **TDBGrid** or **TDBDropDown** control is created, it contains seven built-in styles. You can modify the built-in styles or add your own styles at either design time or run time. Both controls also support several optional events that use **Style** objects to convey formatting information on a per-cell or per-row basis.

The **TDBGrid** and **TDBDropDown** controls store all built-in and user-defined **Style** objects in the **Styles** collection. You can access the members of this collection by name at run time, then apply them to a grid, column, or split in order to control the appearance of the object in question. This collection is accessed using the **Styles** property.

For more information, see [How to Use Styles](#).

ValueItem Object, ValueItems Collection

A **ValueItem** object is used to simplify data access for the user. It specifies an allowable input value for a given **Column** object, and can also be used to translate raw data values into alternate text or graphics for display. For example, you may want to display Balance Due and Paid in Full instead of the numeric data values 0 and 1.

Each **Column** object within a **TDBGrid** or **TDBDropDown** control stores these items, if specified, in a **ValueItems** collection, which is accessed using the **ValueItems** property.

For more information, see [Automatic Data Translation with ValueItems](#).

XArray Object

The **XArray** object implements an array of arbitrary variants. It supports up to 10 dimensions, and automatically shifts its contents when elements (or dimensions) are inserted and deleted. Unlike the **RowBuffer** object, you can create a standalone **XArray** object, and even use it outside the context of True DBGrid.

XArray is implemented as a separate .OCX file; it is not contained in any of the grid .OCX files.

The **XArray** object is used as a data source for a **TDBGrid** or **TDBDropDown** control in storage mode, which corresponds to a **DataMode** property setting of 4 - Storage. For more information, see [Storage Mode](#).

Working with Objects and Collections

This section describes how to work with objects and collections in Visual Basic code, with an emphasis on efficiency. Although the concepts are illustrated with True DBGrid objects and collections, you can apply the same fundamentals to all Visual Basic objects and collections.

A **TDBGrid** object is created when you place a True DBGrid control on a Visual Basic form. **TDBGrid** objects created in Visual Basic will have default names of TDBGrid1, TDBGrid2, and so forth. You can change the control name in the Visual Basic Properties window at design time. You can also change the control's properties using the property pages at design time and Visual Basic code at run time.

A **TDBGrid** object has the following collections: **Splits**, **Columns**, **SelBookmarks**, **Styles**, and **Layouts**. By default, the **Splits** collection contains one **Split** object, and the **Columns** collection contains two **Column** objects. The **Styles** collection contains seven default **Style** objects: Normal, Heading, Selected, Caption, HighlightRow, EvenRow, and OddRow. The **SelBookmarks** and **Layouts** collections are initially empty.

You can reference an object in a collection using its zero-based index. For example, the default **Split** object in a grid has an index value of 0. You can read or set the **Split** object's properties as follows:

```
' Read a Split object property
variable = TDBGrid1.Splits(0).Property

' Set a Split object property
TDBGrid1.Splits(0).Property = variable
```

You can create a reference to an object in a collection using the collection's **Item** method. The following code creates a reference to a grid's default **Split** object:

```
' Declare Split0 as a Split object
Dim Split0 As TrueDBGrid50.Split

' Set Split0 to reference the first Split in the collection
Set Split0 = TDBGrid1.Splits.Item(0)
```

Note the use of the type library qualifier `TrueDBGrid50` in the preceding example. Using the type library qualifier is recommended in order to resolve potential naming conflicts with other controls. For example, if you use another control in the same project that also defines an object named **Split**, the `TrueDBGrid50` type library qualifier is required, as is the type library qualifier for the other control.

Since the **Item** method is implicit for collections, you can omit it:

```
' Declare Split0 as a Split object
Dim Split0 As TrueDBGrid50.Split

' Set Split0 to reference the first Split in the collection
Set Split0 = TDBGrid1.Splits(0)
```

You can now use `Split0` to read or set the **Split** object's properties or to execute its methods:

```
variable = Split0.Property      ' Read a Split object property
Split0.Property = variable      ' Set a Split object property
Split0.Method arg1, arg2, ...   ' Execute a Split object method
```

Very often, you need to read and set more than one of an object's properties. For example:

```
' Read a Split object's properties
variable1 = TDBGrid1.Splits(0).Property1
variable2 = TDBGrid1.Splits(0).Property2

' Set a Split object's properties
```

```
TDBGrid1.Splits(0).Property1 = variable1
TDBGrid1.Splits(0).Property2 = variable2
```

This code is very inefficient because each time the object `TDBGrid1.Splits(0)` is accessed, Visual Basic creates a reference to the object and then discards it after the statement is completed. It is more efficient to create a single reference to the object up front and use it repeatedly:

```
' Declare Split0 as a Split
Dim Split0 As TrueDBGrid50.Split

' Set Split0 to reference the first Split in the collection
Set Split0 = TDBGrid1.Splits(0)

' Read a Split object's properties
variable1 = Split0.Property1
variable2 = Split0.Property2

' Set a Split object's properties
Split0.Property1 = variable1
Split0.Property2 = variable2
```

This code is much more efficient and also easier to read. If your Visual Basic application accesses collection objects frequently, you can improve the performance of your code significantly by adhering to these guidelines.

Similarly, you can apply this technique to other objects and collections of True DBGrid, and of Visual Basic in general. Of particular importance to the grid are the **Column** object and **Columns** collection:

```
' Declare Cols as a Columns collection object, then set it to
' reference TDBGrid1's Columns collection object.
Dim Cols As TrueDBGrid50.Columns
Set Cols = TDBGrid1.Columns

' Declare Col0 as a Column object, then set it to reference the
' first Column object in the collection.
Dim Col0 As Column
Set Col0 = Cols(0)

' Read and set the Column object's Property1
variable1 = Col0.Property1
Col0.Property1 = variable1

' Execute the Column object's Method1 (declared as a Sub)
Col0.Method1 arg1, arg2, ...

' Execute the Column object's Method2 (declared as a Function)
variable2 = Col0.Method2(arg1)
```

Visual Basic also provides an efficient `With...End With` statement for setting multiple properties of an object without explicitly assigning it to a variable. For example, the following code sets multiple properties of the first column of a grid (recall that collections are zero-based):

```
With TDBGrid1.Columns(0)
    .Property1 = variable1
    .Property2 = variable2
End With
```

Some collections allow you to reference their members by name. For example, you can reference a **Column** object using either its index, the name of the database field the column is associated with, or the column's

heading caption. Thus, the following statements are equivalent:

```
' Declare Col0 as a Column object
Dim Col0 As TrueDBGrid50.Column

' Reference by numeric index
Set Col0 = TDBGrid1.Columns.Item(0)

' Reference by numeric index (Item method is implicit)
Set Col0 = TDBGrid1.Columns(0)

' Reference by database field name
Set Col0 = TDBGrid1.Columns("LAST")

' Reference by column header text (Caption property)
Set Col0 = TDBGrid1.Columns("Last Name")
```

A True DBGrid **Style** object can also be referenced by name:

```
' Declare S as a Style object
Dim S As TrueDBGrid50.Style

' Set S to the grid's built-in Normal style
Set S = TDBGrid1.Styles("Normal")

' Set S to the programmer-defined style MyStyle
Set S = TDBGrid1.Styles("MyStyle")
```

To create and add an object to a collection, use the collection's **Add** method. For example, you can create more splits in the grid by adding new **Split** objects to the **Splits** collection:

```
' Create a Split object with index 0
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(0)
```

This code adds a **Split** object with index 0 to the **Splits** collection of TDBGrid1. The original **Split** object now has an index of 1. Alternatively, you can create a **Split** object with index 1:

```
' Create a Split object with index 1
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(1)
```

Note that the **Add** method of the **Splits** collection is used like a function, with its arguments (here, the split index) enclosed in parentheses. Also, since the **Add** method always returns a reference to the **Split** object that was just created, you must precede the assignment statement with the Visual Basic **Set** keyword.

However, not all collections define their **Add** method to return a value. If a collection does nothing more than maintain a list of the arguments passed to its **Add** method, then there is no need for it to return the same item that was just added. In True DBGrid, the **Layouts**, **SelBookmarks**, and **ValueItems** collections are designed this way. For example, you can use the following code to select the current record in a **TDBGrid** control:

```
TDBGrid1.SelBookmarks.Add TDBGrid1.Bookmark
```

Since the **SelBookmarks** collection manages a list of variants corresponding to selected grid rows, its **Add** method does not return a value, and no assignment statement is needed. This example could also be coded as:

```
With TDBGrid1
    .SelBookmarks.Add .Bookmark
End With
```

Regardless of how a collection implements the **Add** method, the syntax for removing items is the same. To remove an existing item from a collection, use the **Remove** method:

```
' Remove the Split object with index 1
TDBGrid1.Splits.Remove 1
```

After this statement is executed, all splits with collection indexes greater than 1 will be shifted down by 1 to fill the place of the removed split. Note that the **Remove** method is called like a subroutine---its argument is not enclosed in parentheses.

You can determine the number of objects in a collection using the collection's **Count** property:

```
' Set a variable equal to the number of Splits in TDBGrid1
variable = TDBGrid1.Splits.Count
```

You can also iterate through all objects in a collection using the **Count** property as in the following example, which prints the **Caption** string of each **Column** object in a grid:

```
For n = 0 To TDBGrid1.Columns.Count - 1
    Debug.Print TDBGrid1.Columns(n).Caption
Next n
```

The **Count** property is also useful for appending and removing columns:

```
' Determine how many columns there are
Dim NumCols As Integer
NumCols = TDBGrid1.Columns.Count

' Append a column to the end of the Columns collection
Dim C As TrueDBGrid50.Column
Set C = TDBGrid1.Columns.Add(NumCols)

' Make the new column visible, since columns created
' at run time are invisible by default
TDBGrid1.Columns(NumCols).Visible = True

' The following loop removes all columns from the grid
While TDBGrid1.Columns.Count
    TDBGrid1.Columns.Remove 0
Wend
```

Visual Basic also provides an efficient **For Each...Next** statement that you can use to iterate through the objects in a collection without using the **Count** property:

```
Dim C As TrueDBGrid50.Column
For Each C In TDBGrid1.Columns
    Debug.Print C.Caption
Next S
```

In fact, using the **For Each...Next** statement is the preferred way to iterate through the objects in a collection.

Design Time Interaction

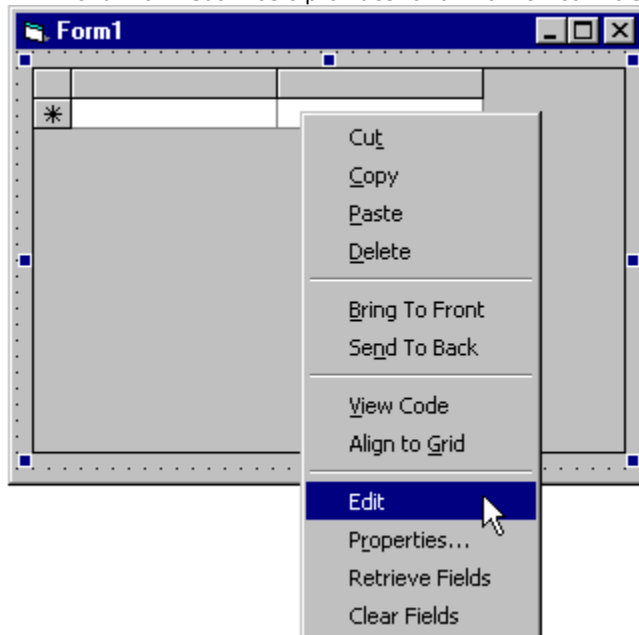
You can easily configure True DBGrid at design time using its context menu, visual editing mode, property pages, reusable layout facility, and add-in Design Assistant. These features enable you to see the grid's run-time appearance at design time and eliminate the need to write customization code for most applications.

The following sections describe how to use True DBGrid's design-time environment to configure the **TDBGrid** control. Most of the following material also applies to the **TDBDropDown** control since it is a subset of **TDBGrid**. Specific differences between the two controls are discussed at the end of this chapter.

{button ,JI('',`Context_Menu')}} Context Menu
{button ,JI('',`Visual_Editing_Mode')}} Visual Editing Mode
{button ,JI('',`Property_Pages')}} Property Pages
{button ,JI('',`Reusable_Layouts')}} Reusable Layouts
{button ,JI('',`Add-in_Design_Assistant')}} Add-in Design Assistant
{button ,JI('',`TDBDropDown_at_Design_Time')}} TDBDropDown at Design Time

Context Menu

Right-click anywhere on the grid to display the True DBGrid context menu, which is a superset of the context menu that Visual Basic provides for all ActiveX controls.



The first eight commands are controlled by Visual Basic; the last four commands are controlled by True DBGrid. The context menu commands operate as follows.

Cut, Copy, Paste, Delete

These commands are identical to those on the Visual Basic **Edit** menu. **Cut** (CTRL+X) moves the grid from the Visual Basic form to the Clipboard. **Copy** (CTRL+C) moves a copy of the grid to the Clipboard while leaving the grid on the form intact. **Paste** (CTRL+V) copies the grid from the Clipboard to the form. **Delete** (the DEL key) removes the grid but does not move it to the Clipboard. You can undo the **Delete** command by selecting **Undo** (CTRL+Z) from the Visual Basic **Edit** menu.

Bring To Front, Send To Back

These commands control the z-order of the grid relative to the other objects on the Visual Basic form. **Bring To Front** (CTRL+J) places the grid in front of other objects; **Send To Back** (CTRL+K) places it behind other objects. These commands are also available from the Visual Basic **Edit** menu. The **ZOrder** method can be used to change the z-order of controls at run time.

View Code

This command displays the grid's code window, which enables you to view and edit the grid's event handling code.

Align to Grid

This command automatically aligns the outer edges of the grid control to the design-time grid lines on the form.

Edit

This command switches to True DBGrid's visual editing mode, in which you can interactively change the grid's column layout and row height. Within visual editing mode, you can right-click anywhere on the grid to display a different context menu called the visual editing menu. Using the visual edit menu, you can

manipulate individual columns and splits directly on the surface of the grid. For convenience, the visual editing menu also contains some of the context menu commands. For details, see [Visual Editing Mode](#).

Properties...

This command displays the grid's property pages, which enable you to customize the layout and appearance of the grid at design time. You can also display the property pages by selecting **(Custom)** from the Visual Basic Properties window. For details, see [Property Pages](#).

Retrieve Fields

This command automatically configures the grid's columns according to the schema information obtained from the Data control's **Recordset**. If you have already changed the grid's default layout, True DBGrid will ask for confirmation before discarding your changes. By default, the grid will use the database field names, or SQL aliases, as the column headings. If all of the columns do not fit in the visible portion of the grid, a horizontal scroll bar will appear. The scroll bar is usable only when the grid is in visual editing mode as described in the next section.

Clear Fields

This command clears all column layouts and field names. Use this command to force a bound grid to use automatic layouts at run time.

Visual Editing Mode

To enter True DBGrid's visual editing mode, right-click anywhere on the grid to display the context menu, then click **Edit**. The appearance of the grid does not change; however, when you move the mouse over the column headers, column dividers, or row dividers, the mouse pointer changes to indicate that the object can be selected or resized.

To exit visual editing mode, select another control or click anywhere on the form outside the grid. The next time you right-click the grid, the context menu will appear, not the visual editing menu.

You can use visual editing mode to interactively perform any of the following tasks:

- Create and delete columns and splits.
- Move and resize columns within the grid.
- Move columns to and from the Clipboard.
- Adjust the grid's row height.
- Save the current grid layout to a file.
- Load and remove grid layouts stored in a file.

The following sections provide step-by-step instructions for using visual editing mode.

{button ,JI(`,`Sizing_columns_and_rows')} Sizing columns and rows
{button ,JI(`,`Creating_and_sizing_splits')} Creating and sizing splits
{button ,JI(`,`Selecting_and_highlighting_columns')} Selecting and highlighting columns
{button ,JI(`,`Moving_selected_columns')} Moving selected columns
{button ,JI(`,`Using_the_visual_editing_menu')} Using the visual editing menu

Sizing columns and rows

If necessary, use the horizontal scroll bar to bring the desired column into view. Move the pointer over a column divider in the column header area. The pointer will become a horizontal double arrow.



Simply drag the divider left or right to adjust the width of the corresponding column. Dragging the divider all the way to the left has the same effect as setting the column's **Visible** property to False. To redisplay an invisible column, move the pointer over the preceding column divider until it changes to a horizontal right arrow.



Drag the divider to the right to make the column visible and adjust its width.

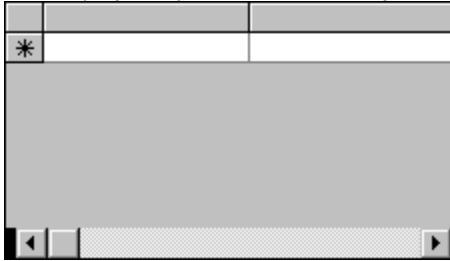
To adjust the height of all grid rows, move the pointer over a row divider in the record selector column. The pointer will become a vertical double arrow.



Drag the divider up or down to adjust the row height. All rows will be adjusted to the same height; you cannot have different heights for individual rows.

Creating and sizing splits

If the horizontal scroll bar is visible, and the **AllowSizing** property of the leftmost split is True, the grid displays a split box immediately to the left of the horizontal scroll bar.



To clone the current split, point to the split box. The pointer will change to a double vertical bar with a down arrow.



Drag the pointer to the right to display and move a pair of dividing lines that will mark the right edge of the new split.



The leftmost split box is always used to create a new split. If other split boxes are present, you can use them to reposition the split dividers. Point to the split box you want to move. The pointer will change to a double vertical bar with horizontal arrows.



Drag the dividers left or right to adjust the widths of the adjacent splits. As with columns, you can drag the dividers all the way to the left to hide a split. However, since splits do not have a **Visible** property, hiding a split in this manner actually deletes it.

If the horizontal scroll bar is not visible, or the **AllowSizing** property of the leftmost split is False, you can still create a new split with the **Split** command on the visual editing menu.

Selecting and highlighting columns

In order to move or delete columns in visual editing mode, you need to select them first.

To select and highlight an individual column, simply click the column header. If one or more columns are already selected, they will be deselected. To select multiple columns, use one of the following methods:

- Select a column by clicking its header. Then hold the `SHIFT` key and click another column's header. All of the columns between the first selected column and the current one (inclusive) will be selected.
- Press the mouse button and drag within the column header area to extend the selection until all desired columns are highlighted.

Note that you can only select multiple columns if they are adjacent.

To deselect all selected columns, click a cell in the grid's display area.

Moving selected columns

To move the selected columns as a unit to a different location, press the mouse button within the header area of any selected column. The pointer will change to an arrow with a small box at its lower right corner.

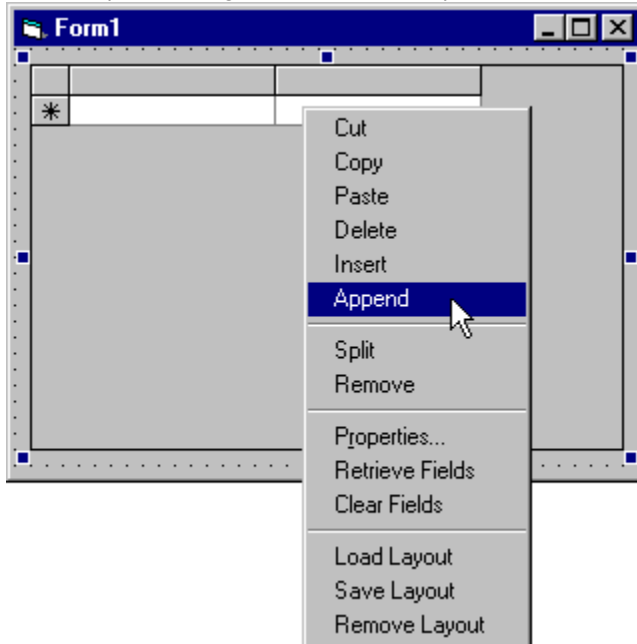


The divider at the left edge of the column being pointed to will be enlarged and highlighted. Drag the divider to the desired location and release the mouse button to move the selected columns immediately to the left of the divider. The moved columns remain selected.

If you drag the divider to a position within the currently selected range, no movement occurs. If a column is not selected, you cannot move it.

Using the visual editing menu

While the grid is in visual editing mode, right-click anywhere on the grid to display the visual editing menu. The visual editing menu contains several commands that are also present in the grid's context menu. However, while the context menu operates on the grid as a whole, the visual editing menu is used to manipulate the grid's columns and splits.



When you right-click a column to display the visual editing menu, the column you are pointing to becomes the current column. This enables you to quickly execute commands that depend on the current column, such as **Cut** and **Insert**.

Some commands, such as **Copy** and **Delete**, can operate on multiple selected columns. Follow the procedures described earlier for selecting columns, then choose the desired command from the visual editing menu.

For convenience, the visual editing menu also contains some of the commands from the grid's context menu. The visual editing menu commands operate as follows.

Cut, Copy, Paste, Delete

These commands are similar to their context menu counterparts except that they apply to a column, or selected columns, not to the entire grid. Using the **Cut**, **Copy**, and **Paste** commands, you can move columns to the Clipboard and paste them to another grid, or within the same grid. The ability to copy columns to the Clipboard provides an easy way to set up grid columns, since all of the properties that define a column are copied as a unit. You can set the properties for multiple columns, copy them to the Clipboard, then paste them to a grid on another form.

Cut moves the selected columns from the grid to the Clipboard. **Copy** moves a copy of the selected columns to the Clipboard while leaving the grid intact. If there are selected columns, then **Cut** and **Copy** operate on the entire selection. If there are no selected columns, then these commands operate on the current column only.

The **Paste** command is available only after a **Cut** or **Copy** command has been performed. If there are selected columns, then **Paste** replaces the selection with the Clipboard contents. If there are no selected columns, then **Paste** inserts the Clipboard contents to the left of the current column. In either case, the newly pasted columns are selected.

Delete removes columns without saving them to the Clipboard. To prevent accidental deletions, the **Delete** command is available only when one or more columns are selected.

Insert

This command creates and inserts a new column to the left of the current column.

Append

This command creates and adds a new column to the right of the rightmost column in the grid.

Split

This command creates and inserts a new split to the left of the current split. You can then use the Splits and Layout property pages to configure the splits and the columns they contain.

Remove

This command removes the current split, that is, the split where you clicked to display the visual editing menu.

Properties...

This command is identical to the context menu command with the same name. It displays the grid's property pages, which enable you to customize the grid's layout and appearance.

Retrieve Fields

This command is identical to the context menu command with the same name. It automatically configures the grid columns according to the schema information from the Data control's **Recordset**.

Clear Fields

This command is identical to the context menu command with the same name. All field and column layout properties set in the grid are cleared.

Load Layout

This command loads the grid layout whose name is given by the **LayoutName** property from the binary layout storage file specified in the **LayoutFileName** property. A layout comprises all persistent property settings for the entire grid, not just its columns and splits. This command is unavailable if either of the aforementioned properties have not been set.

Save Layout

This command saves the current grid layout using the name specified in the **LayoutName** property to the binary layout storage file specified in the **LayoutFileName** property. This command is unavailable if either of these properties have not been set.

Remove Layout

This command removes the grid layout whose name is given by the **LayoutName** property from the binary layout storage file specified by the **LayoutFileName** property. This command is unavailable if either of these properties have not been set.

Property Pages

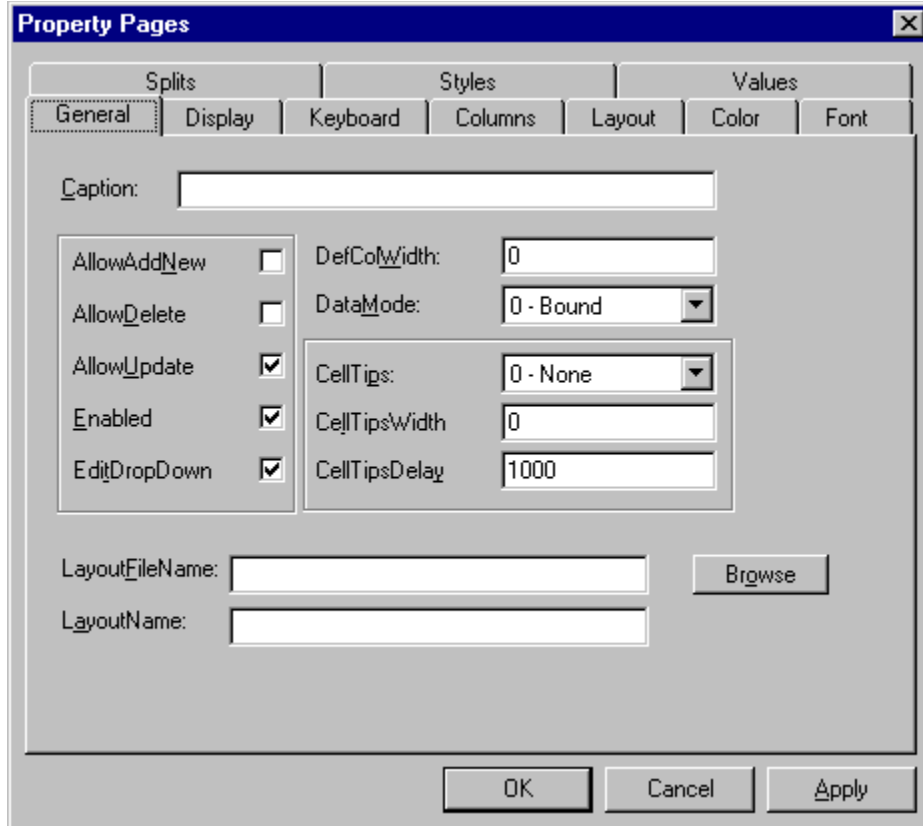
Select **Properties...** from the grid's context menu to display the property page dialog. You can also display the property pages by selecting **(Custom)** from the Visual Basic Properties window. To accept the changes made on any page, click the **OK** button at the bottom of the property page dialog. Click **Cancel** to discard any changes. The property page dialog will be closed after you click **OK** or **Cancel**.

You can also click the **Apply** button to commit your changes without closing the dialog. Any changes you have made will be reflected in the grid immediately. You can continue to make additional changes after clicking **Apply**. Note that selecting a different tab implicitly applies any changes made to the current tab.

```
{button ,JI(`,`General_property_page')}  General property page
{button ,JI(`,`Display_property_page')}  Display property page
{button ,JI(`,`Keyboard_property_page')}  Keyboard property page
{button ,JI(`,`Columns_property_page')}  Columns property page
{button ,JI(`,`Layout_property_page')}  Layout property page
{button ,JI(`,`Color_property_page')}  Color property page
{button ,JI(`,`Font_property_page')}  Font property page
{button ,JI(`,`Splits_property_page')}  Splits property page
{button ,JI(`,`Styles_property_page')}  Styles property page
{button ,JI(`,`Values_property_page')}  Values property page
```

General property page

The General property page defines database related permissions, cell tip behavior, and the grid's data access mode. These are global properties that apply to the grid as a whole (that is, the **TDBGrid** object). Using the General property page, you can also specify a grid layout file containing one or more named layouts.



The screenshot shows the 'Property Pages' dialog box with the 'General' tab selected. The dialog has a title bar with a close button. Below the title bar are three tabs: 'Splits', 'Styles', and 'Values'. Under 'Splits', there are sub-tabs: 'General', 'Display', 'Keyboard', 'Columns', 'Layout', 'Color', and 'Font'. The 'General' sub-tab is active. The main area contains the following controls:

- Caption: [Text box]
- AllowAddNew:
- AllowDelete:
- AllowUpdate:
- Enabled:
- EditDropDown:
- DefColWidth: [Text box with value 0]
- DataMode: [Dropdown menu with value 0 - Bound]
- CellTips: [Dropdown menu with value 0 - None]
- CellTipsWidth: [Text box with value 0]
- CellTipsDelay: [Text box with value 1000]
- LayoutFileName: [Text box] [Browse button]
- LayoutName: [Text box]

At the bottom are three buttons: 'OK', 'Cancel', and 'Apply'.

The General property page is used to set the following properties:

<u>Caption</u>	Text entered in the Caption text box will appear in the grid heading, above the column headers. If the Caption is empty (the default), the grid's title bar will not be displayed. A space will show a blank header.
<u>AllowAddNew</u>	Determines if an empty row appears after the last row. The user can type in this row to initiate an AddNew operation at run time. The default value is False.
<u>AllowDelete</u>	Determines if users can delete a selected record by pressing the DEL key. The default value is False.
<u>AllowUpdate</u>	Determines if users can update records in the grid at run time. The default value is True.
<u>Enabled</u>	Determines if the grid responds to user interactions. The default value is True. If False, data can be changed using code or another bound control.
<u>EditDropDown</u>	Determines whether editing takes place in a pop-up window or within cell boundaries. If True (the default), a pop-up editing window is displayed when the current cell's contents will not fit within its boundaries.
<u>DefColWidth</u>	Sets the default column width for newly created columns, in terms of the coordinate system of the grid's container. The default value is 0,

meaning that the column width will be determined by the field's schema information (that is, the field width in the database). The user can override this setting for individual columns by changing the column's width at design time when the grid is in visual editing mode, or by setting the column's **Width** property at run time.

DataMode

This property defines the grid's data access mode. The allowable values for this property are 0 - Bound, 1 - Unbound, 2 - Unbound Extended, 3 - Application, and 4 - Storage. The default value is 0 - Bound.

CellTips

Determines whether the grid displays a pop-up text window when the cursor is idle. The allowable values for this property are 0 - None, 1 - Anchored, and 2 - Floating. The default value is 0 - None.

CellTipsWidth

Sets the width of the cell tip window in terms of the coordinate system of the grid's container. The default value is zero, which causes the cell tip window to grow or shrink to accommodate the cell tip text.

CellTipsDelay

Controls the amount of time, in milliseconds, that must elapse before the cell tip window is displayed. The default is 1000 (one second).

LayoutFileName

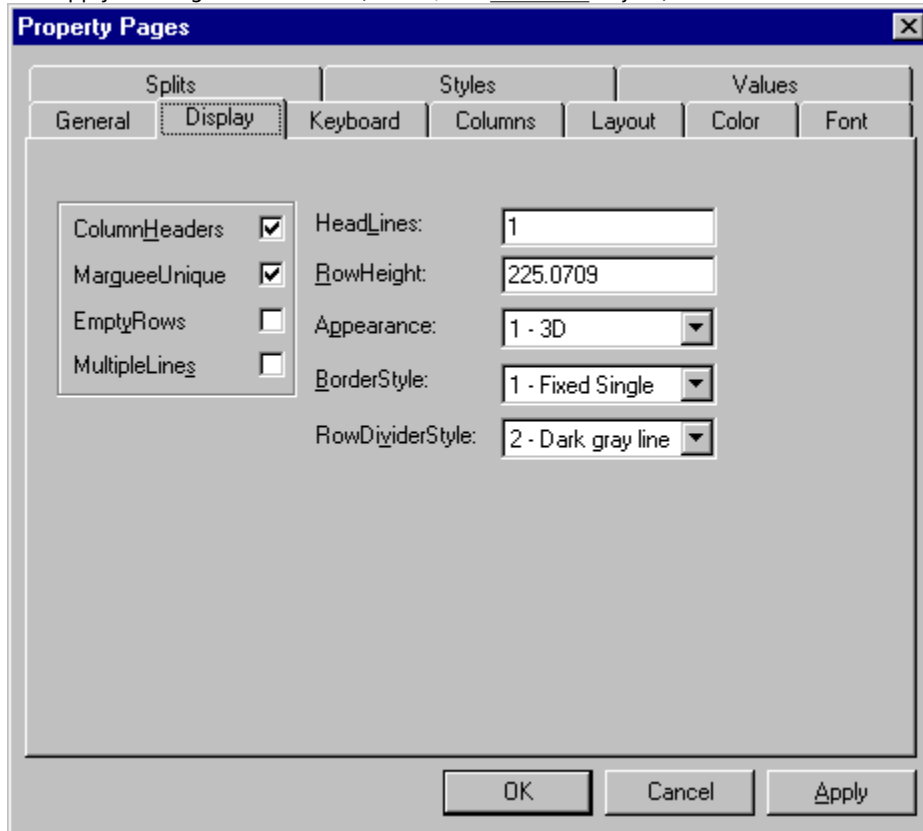
Sets the filename used to save and retrieve grid layouts. The Browse button displays an open file dialog for this property.

LayoutName

Sets the name of the current layout. This value is used by the layout commands on the visual editing menu.

Display property page

The Display property page defines general visual characteristics of the grid. These are global properties that apply to the grid as a whole (that is, the **TDBGrid** object).



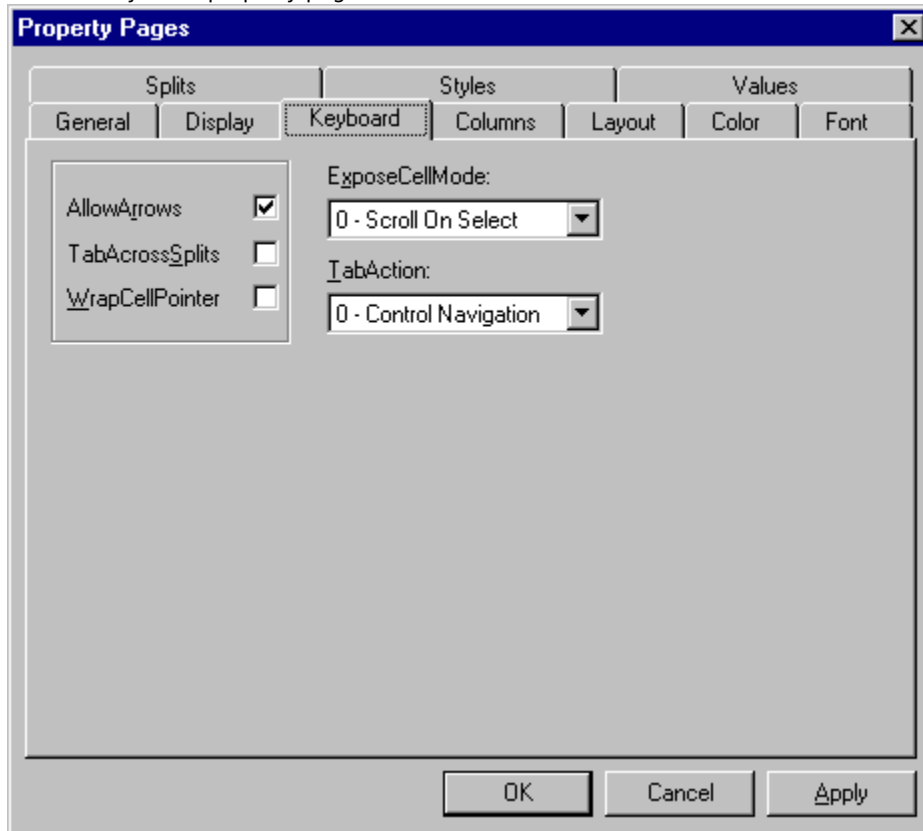
The Display property page is used to set the following properties:

<u>ColumnHeaders</u>	Determines if column headers are displayed. The default value is True.
<u>MarqueeUnique</u>	This property controls the display of the current cell marquee when there is more than one split. If the property is set to True, the marquee will be displayed in only one split. Default is True.
<u>EmptyRows</u>	Determines whether the grid displays empty rows below the last data row. The default value is False.
<u>MultipleLines</u>	Determines whether a single row can span multiple lines. The default value is False.
<u>HeadLines</u>	Sets the number of lines for column header text. This can be set to a non-integral value such as 1.5. The default value is 1.
<u>RowHeight</u>	Sets the height of all grid rows, in terms of the container's coordinates.
<u>Appearance</u>	This property determines the appearance of the grid's caption bar, column headings, and record selector columns. Values are 0 - Flat and 1 - 3D. Default is 1.
<u>BorderStyle</u>	Determines the style of the grid's border. The default value is 1 - Fixed Single.
<u>RowDividerStyle</u>	Determines the style of the lines between rows. The default value is 2 -

Dark gray line.

Keyboard property page

The Keyboard property page is used to customize the run-time behavior of the navigation keys.



The Keyboard property page is used to set the following properties:

AllowArrows

By setting this property to True, you can use the arrow keys to navigate through grid cells. If this property is set to False, the arrow keys will move focus from control to control, rather than from cell to cell. Default is True.

TabAcrossSplits

When this property is False, keyboard navigation is limited to movement within a split. To use the arrow keys or the tab key to move to an adjacent split, this property must be set to True. Default is False.

WrapCellPointer

If the **AllowArrows** property is True, this property determines the behavior of the tab and arrow keys at row boundaries (i.e., the first and last columns). If this property is False, you cannot move to the next (or previous) row using the tab or arrow keys. Default is True.

ExposeCellMode

If the rightmost column of the grid is only partially visible, this property determines whether the grid will scroll so that the rightmost column will be displayed in its entirety when it is clicked. The default is 0 - Scroll on Select, which causes the grid to automatically scroll to make the rightmost column visible when it is clicked. If set to 1 - Scroll On Edit, the grid scrolls only when the user starts to edit. If set to 2 - Never Scroll, the grid will never make the rightmost column visible, even if editing is attempted.

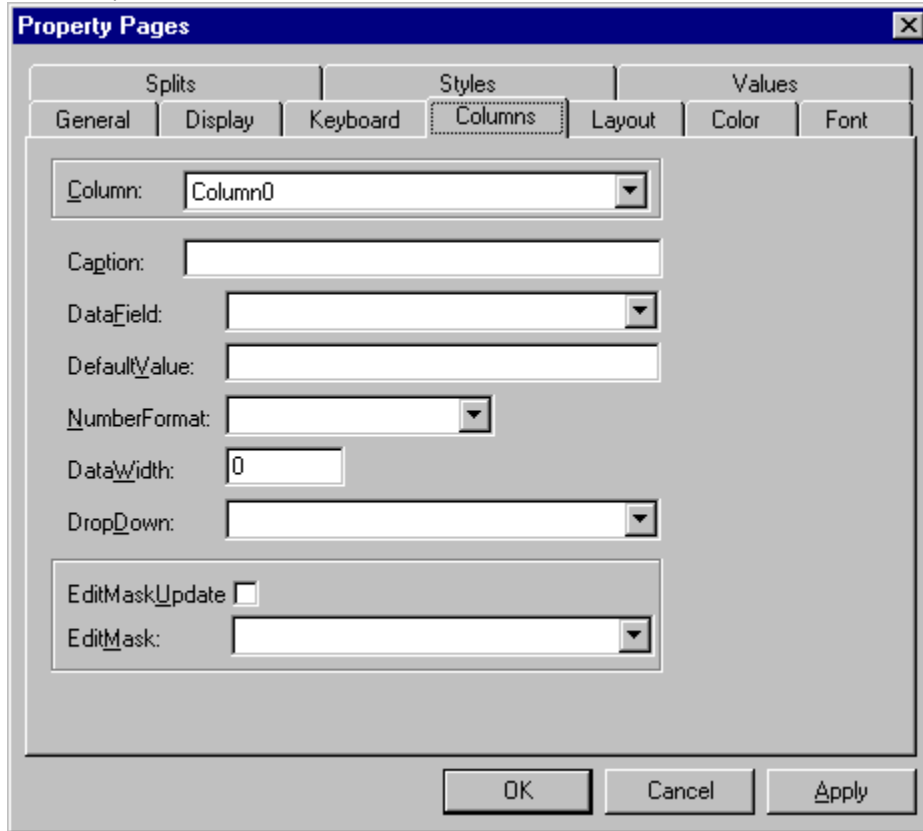
TabAction

This property defines the behavior of the tab key. The default is 0 -

Control Navigation, which allows the tab key to move focus from control to control. If set to 1 - Column Navigation, the tab key will move the current cell to the next column until it reaches the end of the row, then focus will be moved to the next control on the form. If set to 2 - Grid Navigation, the tab key will never move focus to another control; when the current cell reaches the end of the row, the action will be determined by the **WrapCellPointer** property.

Columns property page

The splits of a grid provide users with different views of the same data source. Therefore, corresponding columns in different splits must be bound to the same data fields. A few other column properties are global to all splits as well. The Columns property page is used to set **Column** object properties that cannot vary from one split to another.



The screenshot shows the 'Property Pages' dialog box with the 'Columns' tab selected. The dialog has a title bar with a close button. Below the title bar are three main sections: 'Splits', 'Styles', and 'Values'. Under 'Splits', there are sub-tabs: 'General', 'Display', 'Keyboard', 'Columns' (selected), 'Layout', 'Color', and 'Font'. The 'Columns' tab contains the following controls:

- Column:** A dropdown menu with 'Column0' selected.
- Caption:** An empty text input field.
- DataField:** A dropdown menu.
- DefaultValue:** An empty text input field.
- NumberFormat:** A dropdown menu.
- DataWidth:** A text input field containing '0'.
- DropDown:** A dropdown menu.
- EditMaskUpdate:** A checkbox that is unchecked.
- EditMask:** A dropdown menu.

At the bottom of the dialog are three buttons: 'OK', 'Cancel', and 'Apply'.

The Columns property page is used to set the following properties:

- | | |
|----------------------------|---|
| <u>Column</u> | Selects the current column to be modified. At run time, columns can be identified by a zero-based column index, their <u>Caption</u> string, or their <u>DataField</u> name. |
| <u>Caption</u> | Sets the text that appears in the column header. If you do not set this property, the <u>DataField</u> name will be used as the column caption immediately after it is defined. |
| <u>DataField</u> | Sets the database field, or SQL alias, to which a column is bound. The grid will automatically display all fields from the bound Recordset , if available, in this drop-down combo. |
| <u>DefaultValue</u> | This property applies only to unbound columns. It sets the default value of an unbound column within a new record. For bound columns or columns of an unbound grid, the grid does not use this property itself, but provides it as a placeholder for you to associate default values with the columns. This property can also be used as a tag for a column (whether it is bound or unbound). Arbitrary values can be stored and retrieved. |
| <u>NumberFormat</u> | Determines the display format of data in the column. True DBGrid supports the same functionality as the Format\$ function in Visual Basic. |

In fact, True DBGrid does not do the formatting itself, it simply passes the format string to Visual Basic to perform the appropriate formatting. Search the Visual Basic Help for user-defined formats for more information. The last item in the dropdown list contains an additional option called FormatText Event. If you choose this option, the grid will fire the **FormatText** event, which enables you to format (or even replace) column data before it is displayed.

DataWidth

This property defines the maximum number characters allowed when entering data into a column. The grid will only allow the user to enter this many characters. This has no effect on data that already exceeds this value. To impose no limits on the amount of text the user can enter, set this property to 0 (the default).

DropDown

Associates the name of a **TDBDropDown** control with a column in a **TDBGrid** control.

EditMaskUpdate

Determines whether formatted text is updated to the underlying database. The default value is False, which means that the modified cell text is stripped of literal characters before it is passed on to the underlying database.

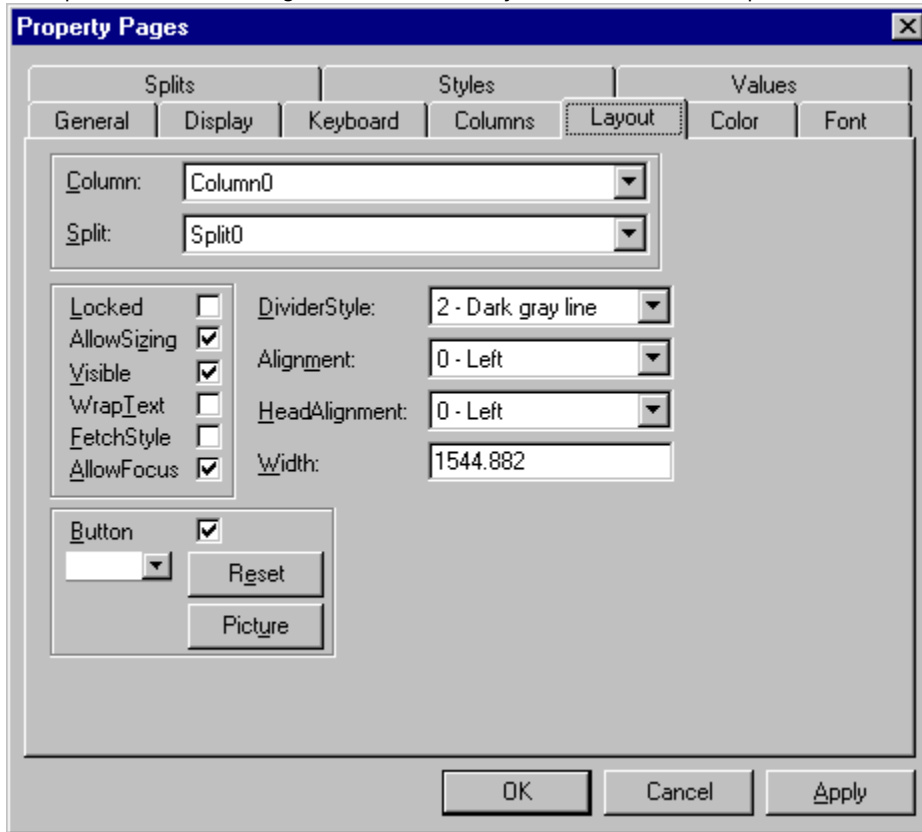
EditMask

Specifies an input mask template for end-user data entry.

Layout property page

The properties on the Columns property page, together with the properties on the Layout property page, define the column (field) layouts for the entire grid.

The properties on the Layout page are said to be *split-specific*. This means that they apply to columns, *not* to splits, but their settings are not necessarily the same across all splits.



The Layout property page is used to set the following properties:

- | | |
|---------------------------|---|
| <u>Column</u> | Selects the current column to be modified. At run time, columns can be identified by a zero-based column index, their <u>Caption</u> string, or their <u>DataField</u> name. |
| <u>Split</u> | Selects which split contains the column specified in the Column combo box. At run time, splits can only be identified by a zero-based split index. |
| <u>Locked</u> | Determines if the user will be able to edit the column. This property affects user interaction with the grid only. The column may still be updatable from code or through another bound control, even if <u>Locked</u> is set to True. Default is False. |
| <u>AllowSizing</u> | Determines if the user can interactively resize the column at run time. Default is True. |
| <u>Visible</u> | Determines if the column will be visible. The default value is True for columns created at design time, False for columns created at run time. |
| <u>WrapText</u> | If True, a line break occurs before words that would otherwise be partially displayed. If False, no line break occurs and text is clipped at the cell's right edge. Default is False. |

FetchStyle

If True, the **FetchCellStyle** event is fired before data is displayed in any cell of the specified column. This event allows the programmer to override the normal font and color attributes used for display. Default is False.

AllowFocus

If False, the column cannot receive focus at run time. Default is True.

Button

If True, a dropdown button will be displayed in the upper right corner of the current cell when it resides in the specified column. When clicked, the **ButtonClick** event is fired. You can use this feature to initiate any action in a cell, such as displaying a popup control for editing. Default is False.

ButtonPicture

Specifies an alternate bitmap for the in-cell button. You can use the command button labeled Picture to display an open file dialog for bitmap files. The Reset button restores the default button picture.

DividerStyle

This property sets the style of the divider at the right edge of the column. The default value is 2 - Dark gray line.

Alignment

Sets the horizontal alignment of data displayed in the column. For bound columns, the default value depends on the data type of the field (assuming that the layout was not modified at design time). For example, the default for a text or memo field is 0 - Left.

HeadAlignment

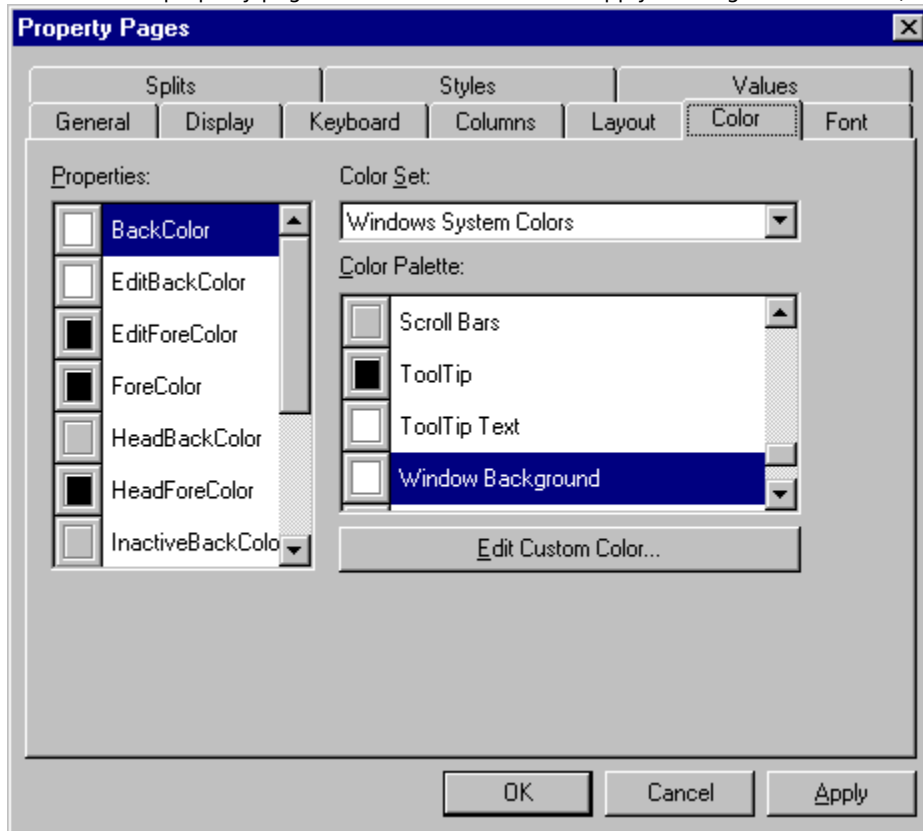
Sets the horizontal alignment of the column header text.

Width

This property specifies the column width in terms of the coordinate system of the grid's container.

Color property page

The Color property page defines color values that apply to the grid as a whole (that is, the **TDBGrid** object).

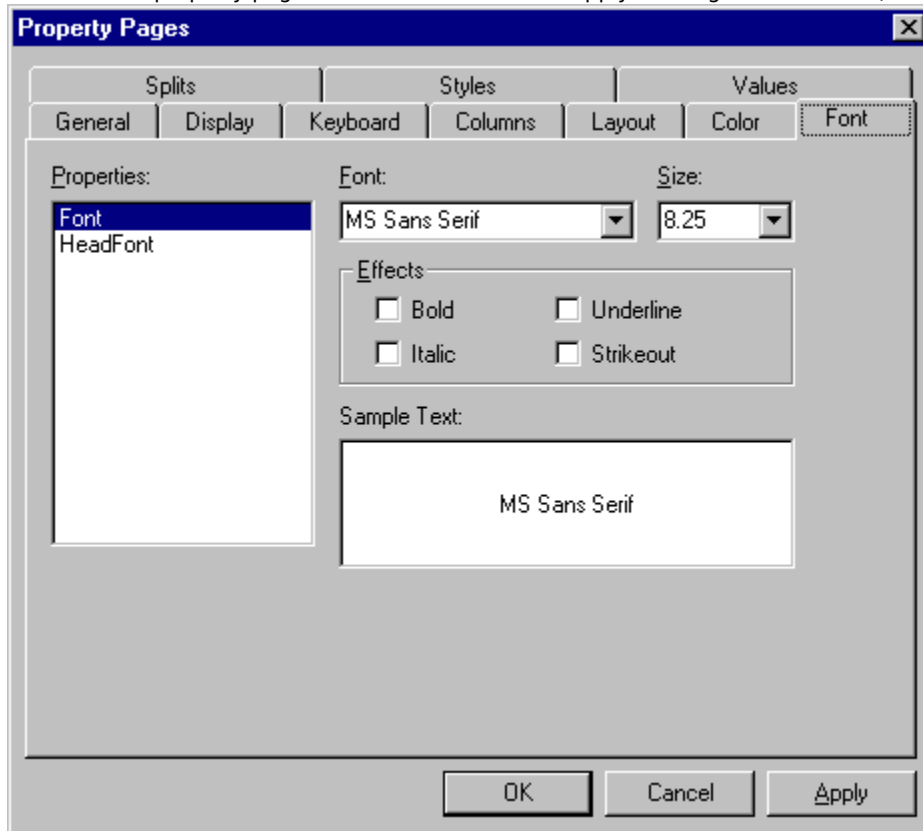


The Properties list box specifies which color property you are setting. You can set the color either by choosing a value from the Color Palette list box or clicking the Edit Custom Color button. The Color Set combo box toggles between windows system colors and standard RGB colors. The color properties you can set are:

<u>BackColor</u>	The background color of the grid's display area.
<u>ForeColor</u>	The foreground color of the grid's display area.
<u>EditBackColor</u>	The background color of the grid's edit window.
<u>EditForeColor</u>	The foreground color of the grid's edit window.
<u>HeadBackColor</u>	The background color of the column headers.
<u>HeadForeColor</u>	The foreground color of the column headers.
<u>InactiveBackColor</u>	The background color of the column headers when the grid is not active (does not have focus).
<u>InactiveForeColor</u>	The foreground color of the column headers when the grid is not active (does not have focus).
<u>SelectedBackColor</u>	The background color of selected rows and columns.
<u>SelectedForeColor</u>	The foreground color of selected rows and columns.

Font property page

The Font property page defines font values that apply to the grid as a whole (that is, the **TDBGrid** object).



The Properties list box specifies which font property you are setting. You can set either of the following font properties:

Font The font to be used for grid's display area.

HeadFont The font to be used for the column headers.

A **Font** object is associated with the font property specified in the Properties list box. The following controls correspond to **Font** object properties (with the exception of Sample Text):

Font Specifies the typeface name of the font---the **Name** property of the **Font** object.

Size Specifies the point size of the font---the **Size** property of the **Font** object.

Bold Specifies whether the font has the **bold** attribute enabled---the **Bold** property of the **Font** object.

Italic Specifies whether the font has the *italic* attribute enabled---the **Italic** property of the **Font** object.

Underline Specifies whether the font has the underline attribute enabled---the **Underline** property of the **Font** object.

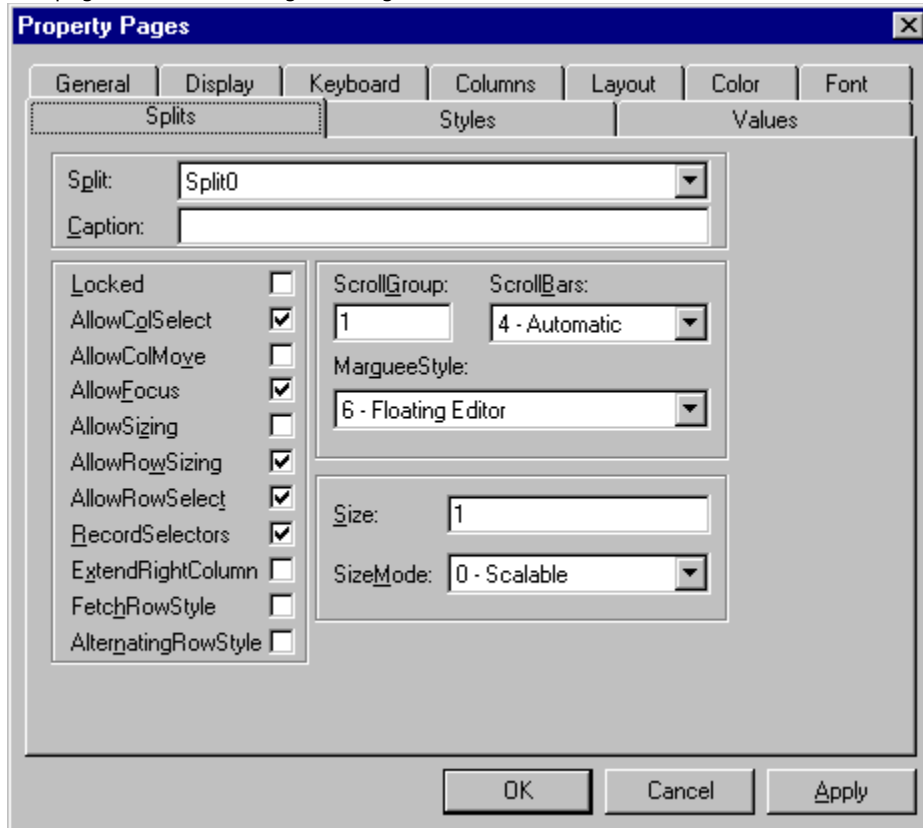
Strikeout Specifies whether the font has the ~~strikeout~~ attribute enabled---the **Strikethrough** property of the **Font** object.

Sample Text This static area displays sample text that shows how text will appear when the selected font is applied. Whenever you change a font setting,

the Sample area is updated so that you can see the results of the change before committing it with either the **OK** or **Apply** button.

Splits property page

This Splits property page defines the appearance and behavior of the splits in the grid. The grid has one split by default, so even if you do not create a grid with multiple splits, you may still need to set properties on this page in order to configure the grid's behavior.



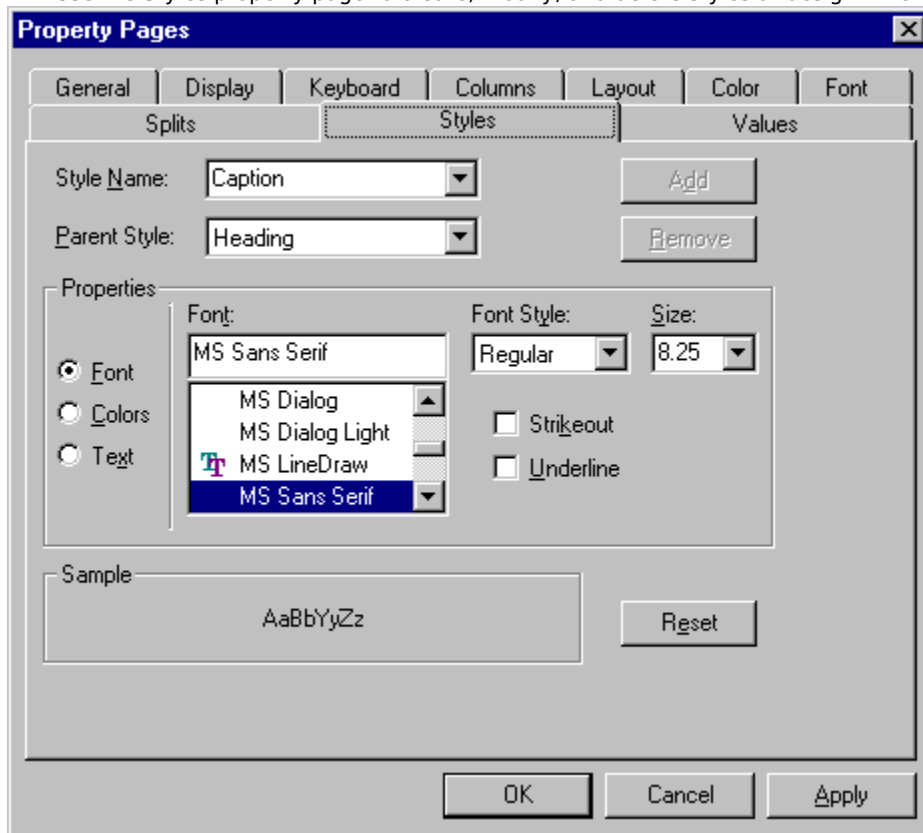
The Splits property page is used to set the following properties:

- | | |
|------------------------------|--|
| <u>Split</u> | Selects the current split to be modified. At run time, splits can only be identified by a zero-based split index. |
| <u>Caption</u> | Sets the caption string displayed above the column headers in a split. |
| <u>Locked</u> | Determines if the user will be able to edit cells in this split. This property affects user interaction with the split only. Columns within the split may still be updatable from code, through another bound control, or by the user in another split, even if <u>Locked</u> is set to True. Default is False. |
| <u>AllowColSelect</u> | When this property is True, the user may select columns at run time. This property affects user interaction at run time only. You can always select columns in code using the <u>SelStartCol</u> and <u>SelEndCol</u> properties. Default is True. |
| <u>AllowColMove</u> | When this property is True, the user may move selected columns at run time. This property affects user interaction at run time only. You can always move a column in code using its <u>Order</u> property. Default is False. |
| <u>AllowFocus</u> | Use this property to determine whether cells in the split can receive focus for user interaction at run time. If the property is False, the split cannot not receive focus. Default is True. |

<u>AllowSizing</u>	When this property is True, the user may resize the split or create a new split at run time. This property affects user interaction at run time only. You can always resize a split or create a split using code. Default is False.
<u>AllowRowSizing</u>	If True, the user can adjust the row height at run time by dragging a row divider in the record selector column. Default is True.
<u>AllowRowSelect</u>	If True, the user can select rows at run time by clicking in the record selector column. This property affects user interaction at run time only. You can always select rows in code by adding their bookmarks to the <u>SelBookmarks</u> collection. Default is True.
<u>RecordSelectors</u>	If this property is True, the record selectors will appear at the left edge of the split. Default is True.
<u>ExtendRightColumn</u>	If this property is True, the width of the rightmost column of the split will be extended to cover the split's entire display area, otherwise a blank area may exist between the rightmost column and the right edge of the split. Default is False.
<u>FetchRowStyle</u>	If this property is True, the <u>FetchRowStyle</u> event will be fired whenever the grid is about to display a row of data. Default is False.
<u>AlternatingRowStyle</u>	This property determines whether a grid or split displays odd-numbered rows in one style and even-numbered rows in another. Default is False.
<u>ScrollGroup</u>	You can assign any positive integer to the <u>ScrollGroup</u> property of a split. Splits with the same <u>ScrollGroup</u> values will scroll vertically simultaneously. Splits with different <u>ScrollGroup</u> values scroll independently of each other. Default is 1.
<u>ScrollBars</u>	This property determines whether scroll bars exist in the split. Default is 4 - Automatic.
<u>MarqueeStyle</u>	This property determines how the current row and cell are highlighted within the split. Default is 6 - Floating Editor.
<u>Size</u>	This property sets the width of the split according to the <u>SizeMode</u> property. Default is 1.
<u>SizeMode</u>	This property determines how the <u>Size</u> property is used to determine the actual size of a split. Default is 0 - Scalable.

Styles property page

Use the Styles property page to create, modify, and delete styles at design time.



The Styles property page contains the following controls:

- | | |
|---------------------|--|
| Style Name | Specifies which style is being edited. You can either select an existing style from the dropdown list or type in the name of a new or existing style. When a grid is first created, it contains the following built-in styles: Caption, EvenRow, Heading, HighlightRow, Normal, OddRow, and Selected. This corresponds to the style's Name property. |
| Parent Style | Specifies the name of the style from which the selected style inherits. For styles with no parent, such as the built-in Normal style, this combo box displays (no style). This corresponds to the style's Parent property. |
| Add | Creates a new style with the name specified in the Style Name combo box. Style names must be unique, so this button will be disabled if the text in the Style Name combo box matches the name of an existing style. This button corresponds to the Add method of the Styles collection. |
| Remove | This button deletes the selected style. The built-in styles cannot be deleted, so this button will be disabled when the text in the combo box matches one of the built-in styles. This button will also be disabled when you have entered text into the Style Name combo box that does not match the name of an existing style. This button corresponds to the Remove method of the Styles collection. |
| Reset | This button resets the selected style so that it inherits all of its font, color, and formatting attributes from its parent, if any. For styles with no |

parent, the Reset button causes the selected style to revert to the default settings held by the Normal style when the grid was first created. This button corresponds to the **Reset** method of the **Style** object.

Sample

This static area displays sample text that shows how a grid cell will appear when the selected style is applied. Whenever you change a font, color, or alignment setting, the Sample area is updated so that you can see the results of the change before committing it with either the **OK** or **Apply** button.

Font, Colors, Text

These radio buttons govern which properties appear in the Properties frame. Since not all controls will fit on the Styles property page at one time, these radio buttons are provided so that you can easily switch between control groups.

When the Styles property page is first displayed, the font controls are shown as in the preceding figure.

Font

Specifies the typeface name of the selected style's font. This corresponds to the **Name** property of the **Font** object associated with the style.

Font Style

Specifies the attributes of the selected style's font. This corresponds to the **Bold** and **Italic** properties of the **Font** object associated with the style.

Size

Specifies the point size of the selected style's font. This corresponds to the **Size** property of the **Font** object associated with the style.

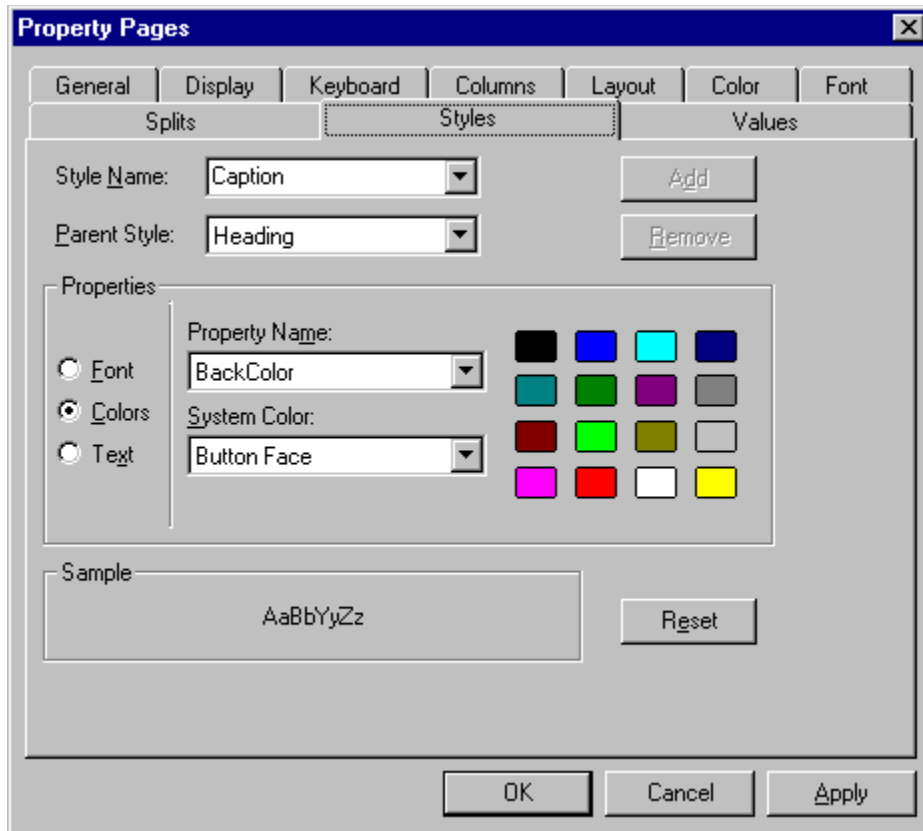
Strikeout

Specifies whether the selected style's font has the ~~strikeout~~ attribute enabled. This corresponds to the **Strikethrough** property of the **Font** object associated with the style.

Underline

Specifies whether the selected style's font has the underline attribute enabled. This corresponds to the **Underline** property of the **Font** object associated with the style.

When the Colors radio button is selected, the following controls are displayed in the Properties frame.



Property Name

This combo box specifies the name of the style property being modified. It always contains two items corresponding to the style's **BackColor** and **ForeColor** properties.

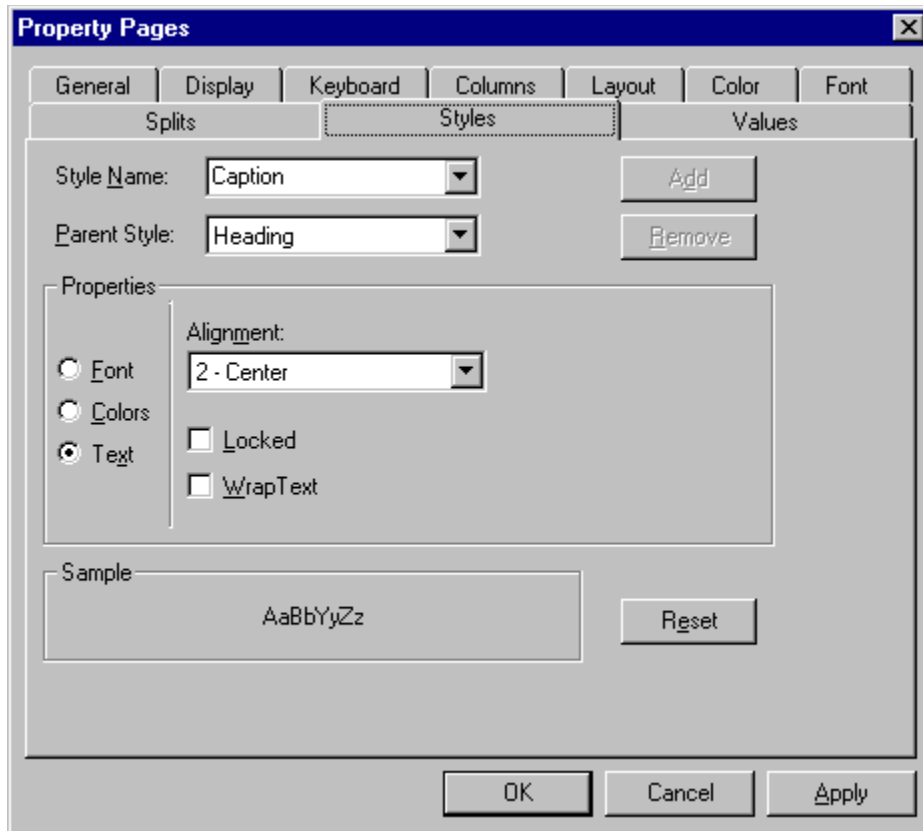
System Color

This combo box allows you to specify a system color value (instead of a physical color) for the property shown in the Property Name combo box. Whenever you select an item in this combo box, any color button selection is removed.

Color Buttons

These 16 buttons allow you to specify a physical color value (instead of a system color) for the property shown in the Property Name combo box. Whenever you click one of these buttons, its border is highlighted and any system color selection is cleared.

When the Text radio button is selected, the following controls are displayed in the Properties frame.



Alignment

Specifies the horizontal text alignment (left, center, right, or general) for the selected style. This control corresponds to the style's **Alignment** property.

Locked

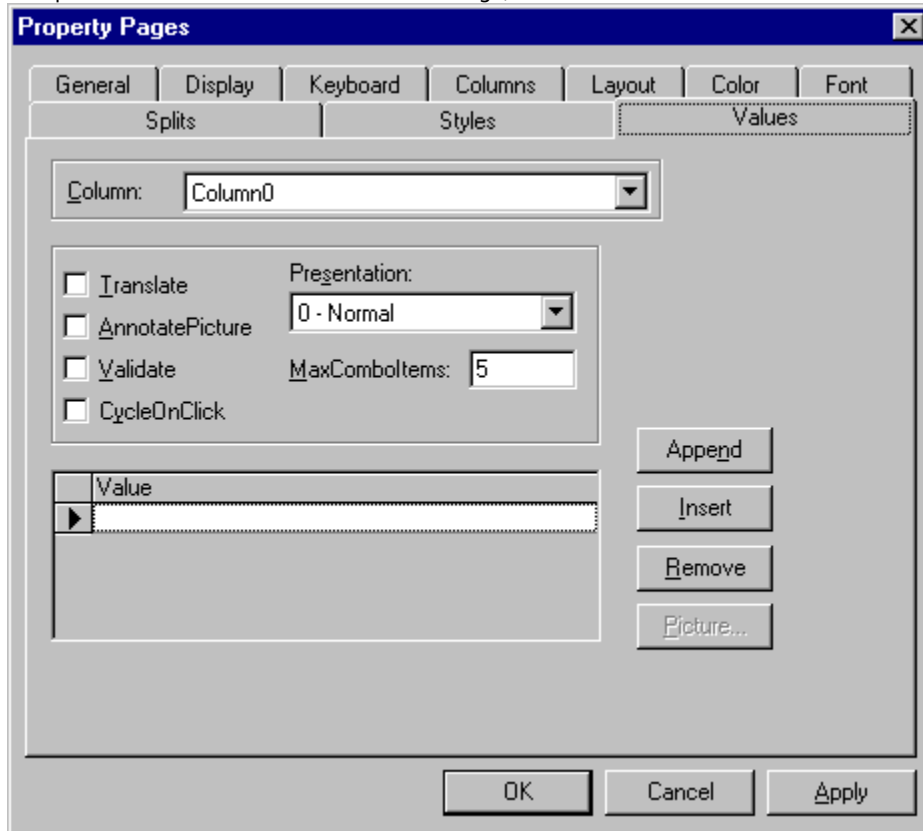
Specifies whether the selected style inhibits in-cell editing. If checked, editing is disallowed; if unchecked, editing is permitted. This control corresponds to the style's **Locked** property.

WrapText

Specifies whether the selected style causes cell text to be word wrapped. If checked, a line break occurs before words that would otherwise be partially displayed; if unchecked, no line break occurs and text is clipped at the cell's right edge. This control corresponds to the style's **WrapText** property.

Values property page

The Values property page defines alternate text or graphics for underlying data values, as well as special data presentation and user interaction settings, such as built-in combo boxes and radio buttons.



The Values property page contains the following controls:

- | | |
|-------------------------------|---|
| <u>Column</u> | Selects the current column to be modified. Like the properties in the Columns property page, the ValueItems collection is <i>global</i> , which means that it cannot vary from one split to another. |
| <u>Translate</u> | Determines whether data will be translated before it is displayed in the column. This control corresponds to the Translate property of the ValueItems collection. Default is False. |
| <u>AnnotatePicture</u> | Determines whether both text and graphics can be displayed in the same cell. This control corresponds to the AnnotatePicture property of the ValueItems collection. Default is False. |
| <u>Validate</u> | If this property is True, users can only enter data that is defined in the translation table. Data not defined in the translation table will not be accepted by the column. This corresponds to the Validate property of the ValueItems collection. Default is False. |
| <u>CycleOnClick</u> | If True, users can click on the cell to cycle through the data items defined in the translation table. This control corresponds to the CycleOnClick property of the ValueItems collection. Default is False. |
| <u>Presentation</u> | This property allows you to display value items as a set of radio buttons, a dropdown combo box, or text. This control corresponds to the Presentation property of the ValueItems collection. Default is 0 - |

Normal (normal text display).

MaxComboItems

If the built-in combo box is used for display (**Presentation** is set to 2 - Combo Box or 3 - Sorted Combo Box), this property defines the maximum number of items that can be displayed in the combo box. This control corresponds to the **MaxComboItems** property of the **ValueItems** collection. Default is 5.

Value

(First column of translation table) Matches the underlying data value from the database. This will be the value stored in the database. When translating, this value must match the string value of the data source. Visual Basic arrays and some databases store a space, for a possible minus sign, ahead of an integer value. If your data source does this, you may need to pad entries in this column with a single space. This column corresponds to the **Value** property of the **ValueItem** object.

DisplayValue

(Second column of translation table) Contains the translated display value, if desired. For example, to display 1 as Yes, enter 1 in the Value column, and Yes in the DisplayValue column. This column corresponds to the **DisplayValue** property of the **ValueItem** object.

Append

This button moves the current row of the translation table to a new blank row for entering additional data values.

Insert

This button creates a new row above the current row of the translation table for entering additional data values.

Remove

This button deletes the current row from the translation table.

Picture

To display a bitmap in the DisplayValue column, press this button to pop up a file selection dialog. Locate the bitmap you want to display, then press the **OK** button in the dialog to load the bitmap into the DisplayValue column.

Record Selectors

Select a row in the translation table to specify a default value to display whenever a data value not present in the **ValueItems** collection is encountered. The selected row corresponds to the **DefaultItem** property of the **ValueItems** collection.

Reusable Layouts

True DBGrid provides a reusable layout facility that enables you to store one or more grid layouts in a binary file, then recall them as needed at design time or run time. With this feature, you can:

- Create repositories of grid layouts that you can reuse in future projects.
- Reduce the number of grids on a form by associating multiple layouts with a single grid control.
- Change the layout at run time with very little coding.
- Save end-user layout preferences to a file, then reload them the next time the application is run.

At design time, use the **LayoutFileName** and **LayoutName** properties to specify the current layout. Then, use the appropriate visual editing menu commands to load, save, or remove the current layout.

At run time, use the **LayoutName** property and **LoadLayout** method to restore a layout from the file specified by the **LayoutFileName** property. To add a new layout to the current layout file, or replace an existing layout in the file, use the **Add** method of the **Layouts** collection. To remove a layout from the current layout file, use the **Remove** method of the **Layouts** collection.

```
{button ,JI(`,`Saving_the_current_layout')}} Saving the current layout  
{button ,JI(`,`Loading_a_saved_layout')}} Loading a saved layout  
{button ,JI(`,`Removing_a_saved_layout')}} Removing a saved layout
```

Saving the current layout

To save a grid layout to a file at design time, do the following:

1. Configure the grid to your liking as you normally would.
2. In the General property page or Visual Basic Properties window, set the **LayoutFileName** property to the name of a layout file. If the file does not exist, you will be asked if you want to create it.
3. In the General property page or Visual Basic Properties window, set the **LayoutName** property to the string that you will use to identify the layout.
4. From the visual editing menu, choose the **Save Layout** command. This is analogous to calling the **Add** method of the **Layouts** collection, which uses the current value of the **LayoutFileName** property at run time.

Loading a saved layout

To load a grid layout from a file at design time, do the following:

1. In the General property page or Visual Basic Properties window, set the **LayoutFileName** property to the name of an existing layout file.
2. In the General property page or Visual Basic Properties window, set the **LayoutName** property to the string that identifies the layout to be loaded.
3. From the visual editing menu, choose the **Load Layout** command. This is analogous to calling the **LoadLayout** method, which uses the current values of the **LayoutName** and **LayoutFileName** properties at run time.

Removing a saved layout

To remove a grid layout from a file at design time, do the following:

1. In the General property page or Visual Basic Properties window, set the **LayoutFileName** property to the name of an existing layout file.
2. In the General property page or Visual Basic Properties window, set the **LayoutName** property to the string that identifies the layout to be removed.
3. From the visual editing menu, choose the **Remove Layout** command. This is analogous to calling the **Remove** method of the **Layouts** collection, which uses the current value of the **LayoutFileName** property at run time.

Add-in Design Assistant

If you are developing in Visual Basic 5.0, you can use the True DBGrid Pro 5.0 Design Assistant to automate repetitive tasks and perform customizations that would otherwise require coding. For example, you can use the Design Assistant to:

- Customize color, font, and style properties for individual columns.
- Quickly create fixed, nonscrolling columns using splits.
- Apply a property value to all columns at once.

The Design Assistant toolbar contains the following commands:

Fix	Toggles the current column's nonscrolling state.
Apply	Applies the current cell value to all columns.
Refresh	Refreshes property values from the selected control.
Help	Displays this help topic.

The Design Assistant was written using the extensibility model of Visual Basic 5.0 and is therefore not available in Visual Basic 4.0.

{button ,JI(`,`Using_the_Design_Assistant')} Using the Design Assistant

{button ,JI(`,`Setting_column_properties_with_the_Design_Assistant')} Setting column properties with the Design Assistant

{button ,JI(`,`Creating_fixed_nonscrolling_columns_with_the_Design_Assistant')} Creating fixed, nonscrolling columns with the Design Assistant

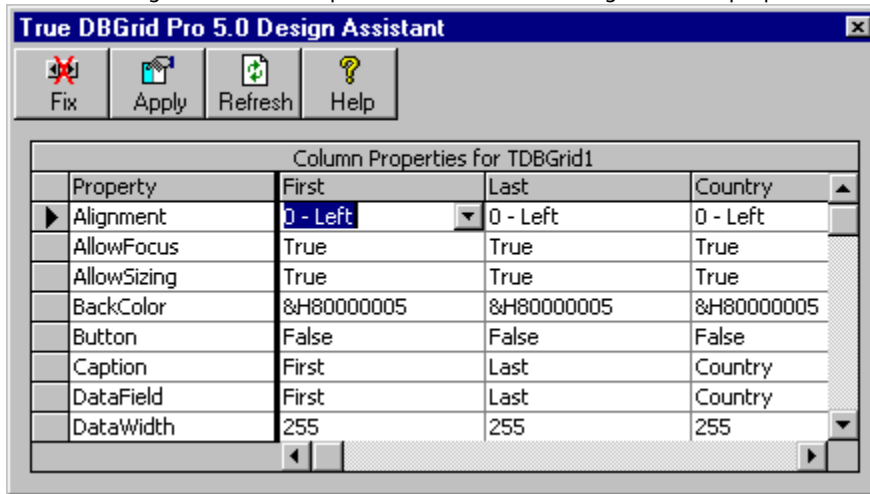
{button ,JI(`,`Refreshing_properties_in_the_Design_Assistant')} Refreshing properties in the Design Assistant

Using the Design Assistant

The True DBGrid Pro 5.0 Design Assistant is copied to your system during installation. To make it available within the design-time environment, run Visual Basic 5.0 and select **Add-In Manager...** from the **Add-Ins** menu. The Add-In Manager dialog will appear. Check the box labeled **True DBGrid Pro 5.0 Design Assistant**, then press the **OK** button. The True DBGrid Pro 5.0 Design Assistant icon will be placed on the toolbar. Click the icon to open the Design Assistant window.

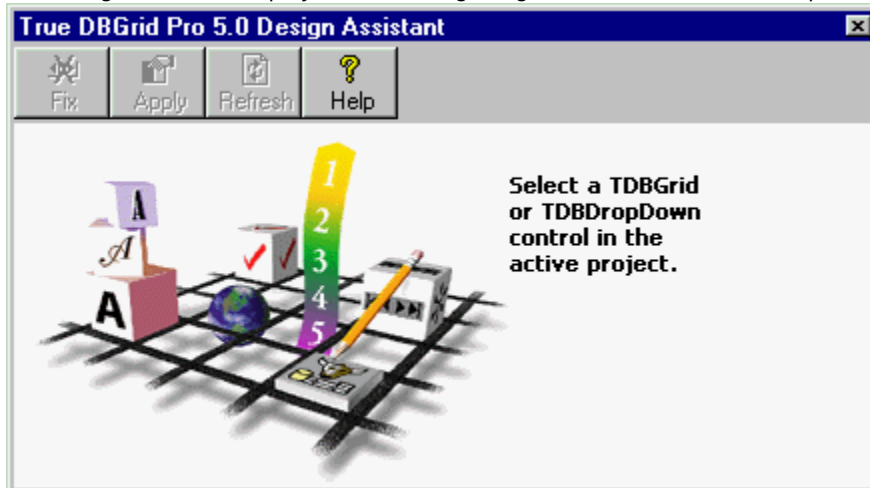


When you select a **TDBGrid** or **TBDDropDown** control in a form, the Design Assistant displays a grid similar to the Visual Basic Properties window. The first column contains a list of **Column** object properties; the remaining columns correspond to the current settings of these properties in the control that you selected.



As you navigate through the columns in the Design Assistant, the corresponding column in the selected control is scrolled into view.

When you select a control that is neither a **TDBGrid** nor a **TBDDropDown**, or if no controls are selected, the Design Assistant displays the following image, and all commands except **Help** are unavailable.



Setting column properties with the Design Assistant

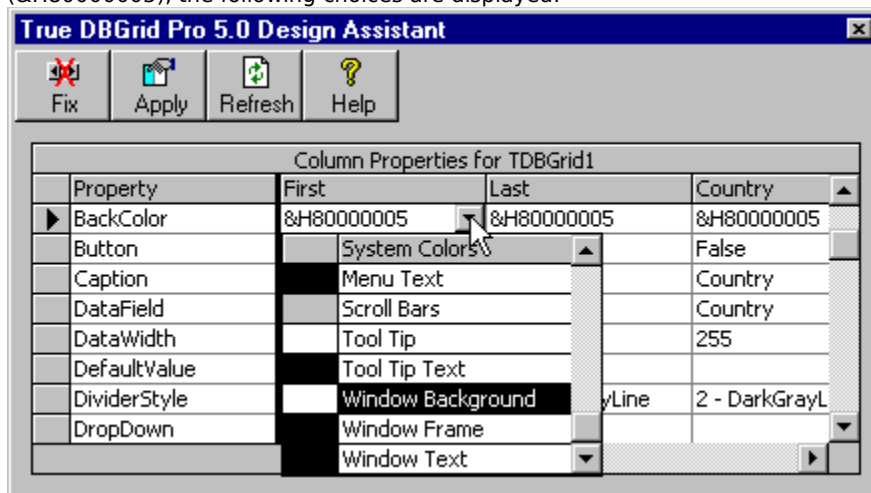
You can change the properties of an individual column with the Design Assistant just as you would with the grid's property pages. Unlike the property pages, however, the Design Assistant provides the following benefits:

- You can view the settings for multiple columns at once.
- You can change column properties such as **BackColor**, **Font**, and **Style**, which are not available in the property pages.
- You can easily replicate a value across all columns using the **Apply** button, which is ideal for properties such as **DividerStyle** and **HeadAlignment**.

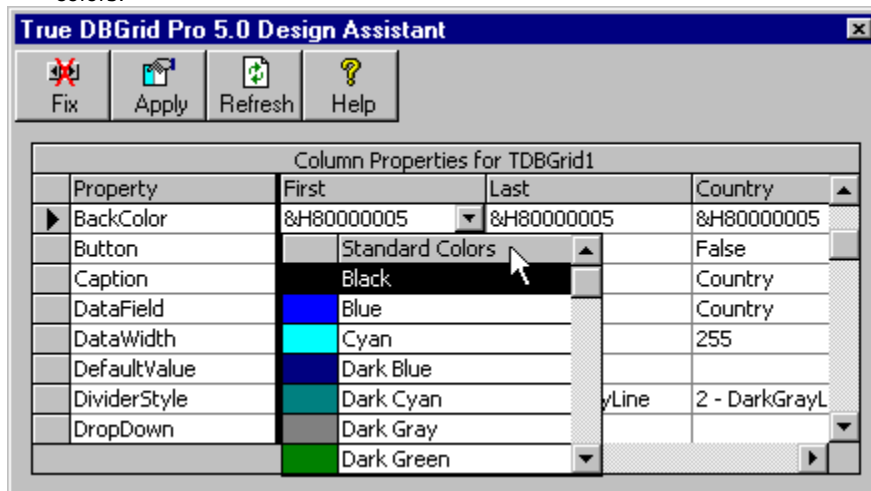
To change the value of a property, type the new value into the appropriate grid cell, then press ENTER or move to a different cell. Or, if a column button appears within a cell, you can click it to display a drop-down list or dialog box from which you can select a new value.



To close the drop-down list or dialog box without making a selection, hit the ESC key. For color properties, the drop-down control will display either a list of system colors or standard colors, depending upon the current value. For example, if the **BackColor** property is set to the system window background color (&H80000005), the following choices are displayed.



If you click the heading of the System Colors column, the color list toggles between system and standard colors.



For font properties, a standard font dialog appears when you click the in-cell button. Although you will generally want to use the same font for all grid columns, you can use the Design Assistant to render individual columns in bold or italic.

To apply a property value to all columns, navigate to the desired cell, then click the **Apply** button on the Design Assistant toolbar.

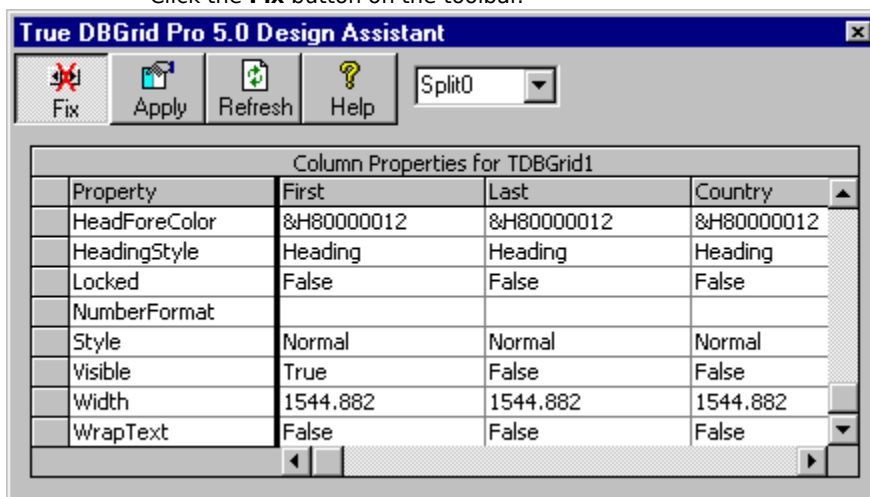
Creating fixed, nonscrolling columns with the Design Assistant

To create a fixed, nonscrolling column using the property pages and visual editing menu, you need to perform the following steps:

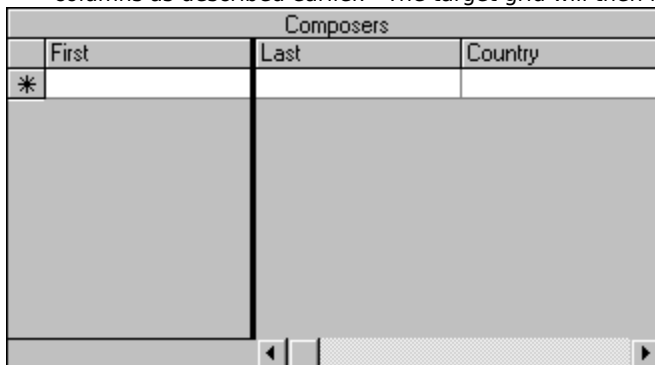
- Create a new split with the **Split** command on the visual editing menu.
- Set the **Visible** property of the column to be fixed to True in split 0, False in split 1.
- Set the **Visible** property of all other columns to False in split 0, True in split 1.

This can be a time-consuming process if there are many columns. Fortunately, the Design Assistant makes it easier:

- Navigate to the column to be fixed in the Design Assistant.
- Click the **Fix** button on the toolbar.



The Design Assistant will automatically create a new split in the target grid and set the **Visible** property of all columns as described earlier. The target grid will then look something like this.



You can repeat this procedure to make other columns nonscrolling. Similarly, to make a nonscrolling column scroll again:

- Navigate to the column in the Design Assistant. Note that the **Fix** push button is depressed, which indicates that the column is nonscrolling.
- Click the **Fix** button, which returns to its normal state.

The Design Assistant will automatically change the **Visible** property of the current column so that it is no longer visible in the leftmost split, but is visible in all other splits. If the last fixed column is moved in this manner, the Design Assistant will also remove the empty split from the target grid.

Note that the **Fix** command is unavailable in the following cases:

- The selected control is a **TBDropDown**, which does not support splits.
- A column is visible in more than one split. In this case, the Design Assistant assumes that you want to manage splits and column visibility yourself.

When the target grid contains more than one split, the Design Assistant toolbar displays a combo box for selecting the current split, as shown in the previous illustration. This is analogous to the split combo box on the Layout property page.

Refreshing properties in the Design Assistant

If you change the value of a column property using the property pages, or add or remove columns in visual editing mode, the Design Assistant does not update its display accordingly. However, you can use the **Refresh** button to ensure that the property values displayed are accurate.

TDBDropDown at Design Time

Since the **TDBDropDown** control is a subset of **TDBGrid**, the two controls have many properties in common and are similar to work with at design time. There are some differences, however, due to the reduced functionality and user interface of **TDBDropDown**.

First, **TDBDropDown** does not support multiple splits. Therefore, it does not have a **Splits** collection, which means that:

- There is no Splits property page. Properties that still make sense for a single-split control, such as **ScrollBars**, were moved to the General property page.
- The split combo box was removed from the Layout property page.
- The visual editing menu commands **Split** and **Remove** were removed.
- You cannot create a new split interactively in visual editing mode.

Second, **TDBDropDown** does not support cell editing. Therefore, the following features are no longer accessible from the property pages:

- In-cell buttons.
- Some **ValueItems** properties, such as **CycleOnClick** and **Validate**.
- User permission properties, such as **AllowAddNew**, **AllowDelete**, and **AllowUpdate**.

Third, **TDBDropDown** does not support reusable layouts. Therefore, it does not have a **Layouts** collection, which means that:

- The layout controls were removed from the General property page.
- The layout commands were removed from the visual editing menu.

Finally, the Display property page is not used since the General property page is large enough to accommodate the remaining properties.

Run Time Interaction

This chapter describes how users of your application interact with True DBGrid at run time. You can give your users the ability to perform any or all of the following:

- Navigate within the grid using the mouse or keyboard.
- Select rows or columns.
- Add, update, and delete records.
- Configure the grid's layout.

In the following sections, the properties and events associated with a particular user action are noted where applicable.

{button ,JI(`,`Navigation_and_Scrolling')}` Navigation and Scrolling
{button ,JI(`,`Selection_and_Movement')}` Selection and Movement
{button ,JI(`,`Sizing_and_Splitting')}` Sizing and Splitting
{button ,JI(`,`Database_Operations')}` Database Operations
{button ,JI(`,`Drag-and-Drop_Behavior')}` Drag-and-Drop Behavior

Navigation and Scrolling

The following sections describe the grid's default navigation and scrolling behavior. You always have complete control over the behavior of the TAB and arrow keys as well as the position of the current cell when a row or split boundary is reached.

{button ,JI(`',`Mouse_navigation')}} Mouse navigation
{button ,JI(`',`Clicking_the_rightmost_column')}} Clicking the rightmost column
{button ,JI(`',`IntelliMouse_support')}} IntelliMouse support
{button ,JI(`',`Keyboard_navigation')}} Keyboard navigation
{button ,JI(`',`Navigation_at_row_boundaries')}} Navigation at row boundaries
{button ,JI(`',`Navigation_at_split_boundaries')}} Navigation at split boundaries
{button ,JI(`',`Restricting_cell_navigation')}} Restricting cell navigation

Mouse navigation

When the user clicks a cell, that cell becomes current, and the **RowColChange** event is fired. The only exceptions to this are:

- If the user clicks a cell in a column or split that has the **AllowFocus** property set to False, and the cell belongs to the current row, then the current cell does not change.
- If the user clicks a cell in a column or split that has the **AllowFocus** property set to False, and the cell does not belong to the current row, then the current row changes, but the column with the focus retains it.
- If the current cell has been modified, and the **BeforeColUpdate** event is canceled, then the current cell does not change.
- If the current row has been modified, and the user clicks a cell in a different row, and the **BeforeUpdate** event is canceled, then the current cell does not change.

The user can also use the mouse to manipulate the grid's scroll bars, bringing cells that lie outside the grid's display area into view. The vertical scroll bar governs rows; the horizontal scroll bar governs columns. The **ScrollBars** property controls which scroll bars are displayed, if any.

Scrolling always occurs in discrete cell units; the user cannot scroll on a per-pixel basis in either direction.

Note that the scroll bars do not change the current cell. Therefore, the current cell may not always be visible.

To respond to vertical scrolling operations in code, use the **FirstRowChange** event. To respond to horizontal scrolling operations in code, use the **LeftColChange** event.

Clicking the rightmost column

The grid always displays the leftmost column (the first visible column) in its entirety. The rightmost column, however, is usually clipped. The behavior of the last partially visible column when clicked by the user is controlled by the grid's **ExposeCellMode** property.

The default value for the **ExposeCellMode** property is 0 - Scroll On Select. If the user clicks the rightmost column when it is partially visible, the grid will scroll to the left to display this column in its entirety. This may be less desirable for users who commonly click on the grid to begin editing, as the grid will always shift to the left when the user clicks on a partially visible rightmost column.

If **ExposeCellMode** is set to 1 - Scroll On Edit, the grid will not scroll when the rightmost visible column is clicked. However, if the user attempts to edit the cell, then the grid will scroll to the left to display the column in its entirety. This is how Microsoft Excel works and is probably the most familiar setting to users.

If **ExposeCellMode** is set to 2 - Never Scroll, the grid will not scroll to make the rightmost column visible, even if the user subsequently attempts to edit the cell. Note that editing may be difficult if only a small portion of the column is visible. The chief reason to use this setting is if you know there will always be enough space available for editing (or if you disallow editing) and you never want the grid to shift accidentally.

Note that the **ExposeCellMode** property controls the behavior of the rightmost visible column only when the user clicks it with the mouse. If the rightmost column becomes visible by code (setting the grid's **Col** property) or by keyboard navigation, then the grid will always scroll to make it totally visible.

IntelliMouse support

True DBGrid responds to Microsoft IntelliMouse activity as follows:

- If the user turns the wheel, the grid scrolls vertically by one row for each click of the wheel.
- If the user holds down the `SHIFT` key while turning the wheel, the grid scrolls vertically by one page for each click of the wheel.
- If a horizontal scroll bar is present, and the user holds down the `CTRL` key while turning the wheel, the grid scrolls horizontally by one column for each click of the wheel. If there is no vertical scroll bar, the user need not hold down the `CTRL` key.
- If the user holds down the `SHIFT` key while performing a horizontal scrolling operation, the grid scrolls horizontally by one page for each click of the wheel.

In summary:

- Vertical scrolling takes precedence over horizontal scrolling, unless overridden with the `CTRL` key.
- The default scrolling increment is one row (or column), unless overridden with the `SHIFT` key, in which case the grid scrolls by one page.

Keyboard navigation

By default, the user can navigate the grid with the arrow keys, the TAB key, the PGUP and PGDN keys, and the HOME and END keys.

UP/DOWN ARROWS	These keys move the current cell to adjacent rows.
LEFT/RIGHT ARROWS	<p>If the AllowArrows property is True (the default), these keys move the current cell to adjacent columns.</p> <p>If the AllowArrows property is False, then these keys move focus from control to control and cannot be used to move between cells.</p>
TAB	<p>If the TabAction property is 0 - Control Navigation (the default), the TAB key moves focus to the next control on the form, as determined by the tab order.</p> <p>If the TabAction property is 1 - Column Navigation or 2 - Grid Navigation, the TAB key moves the current cell to the next column, while SHIFT + TAB moves to the previous column. The differences between column and grid navigation are discussed in the next section.</p>
PGUP, PGDN	These keys scroll the grid up or down an entire page at a time. Unlike the vertical scroll bar, the PGUP and PGDN keys change the current row by the number of visible rows in the grid's display. When paging up, the current row becomes the first row in the display area. When paging down, the current row becomes the last row in the display area, including the AddNew row. The current column does not change.
HOME, END	These keys move the current cell to the first or last column. If necessary, the grid will scroll horizontally so that the current cell becomes visible. The current row does not change. If the current cell is being edited, HOME and END move the insertion point to the beginning or end of the cell's text.

Navigation at row boundaries

At row boundaries, namely the first and last column, grid navigation depends on the **WrapCellPointer** property. The following explanation assumes that the **AllowArrows** property is True, and that the **TabAction** property is set to either 1 - Column Navigation or 2 - Grid Navigation.

LEFT/RIGHT ARROWS

If the **WrapCellPointer** property is True, the current cell wraps across row boundaries. If the current cell is in the last column, the RIGHT ARROW key moves it to the first column of the next row. If the current cell is in the first column, the LEFT ARROW key moves it to the last column of the previous row.

If the **WrapCellPointer** property is False (the default), these keys cannot move the current cell at row boundaries.

TAB

If the **TabAction** property is 1 - Column Navigation, the cell pointer does not wrap to an adjacent row, and the **WrapCellPointer** property is ignored. If the current cell is in the last column, TAB moves focus to the next control in the tab order. If the current cell is in the first column, SHIFT+TAB moves focus to the previous control in the tab order.

If the **TabAction** property is 2 - Grid Navigation and **WrapCellPointer** is True, TAB and SHIFT+TAB move the current cell to the next or previous row. The current cell will not cross row boundaries if **WrapCellPointer** is False.

Navigation at split boundaries

At split boundaries, grid navigation depends on the **TabAcrossSplits** property as follows:

LEFT/RIGHT ARROWS

If the **TabAcrossSplits** property is True, these keys move the current cell across split boundaries to the next or previous split.

If the **TabAcrossSplits** property is False (the default), the behavior of these keys at split boundaries will be the same as their behavior at row boundaries. Note that a split's **AllowFocus** property must be True in order for these keys to move the current cell to that split.

TAB

The TAB and SHIFT+TAB keys honor **TabAcrossSplits** as previously described for the arrow keys.

Restricting cell navigation

If the current cell has been modified, you can use the **BeforeColUpdate** event to examine its value before moving to another grid cell. If the value entered is invalid, you can set the **Cancel** argument to True to prevent the current cell from changing, and optionally beep or display an error message for the user. The **BeforeColUpdate** event provides a flexible way to validate user input and restrict cell navigation.

```
Private Sub TDBGrid1_BeforeColUpdate( _
    ByVal ColIndex As Integer, _
    OldValue As Variant, _
    Cancel As Integer)

    Dim CharCode As Integer
    If ColIndex = 1 Then
        ' Data in Column 1 must start with upper case
        CharCode = Asc(TDBGrid1.Columns(1).Text)
        If CharCode > 64 And CharCode < 91 Then Exit Sub

        ' Display warning message for user
        MsgBox "Last name must start with upper case"

        ' Data validation fails, prohibit user from moving to
        ' another cell
        Cancel = True
    End If
End Sub
```

Selection and Movement

The following sections describe how users can select columns, move selected columns, and select rows. You can always restrict any or all of these operations at design time or in code.

{button ,JI(`,`Selecting_columns') } Selecting columns

{button ,JI(`,`Moving_columns') } Moving columns

{button ,JI(`,`Selecting_rows') } Selecting rows

Selecting columns

If the **AllowColSelect** property is True, the user can select an individual column or a range of columns with the mouse. Nonadjacent column selections are not supported.

When the user clicks a column header, that column is selected and highlighted, and any columns or rows that were previously selected are deselected. There are two ways for the user to select a range of columns:

1. After selecting the first column in the range by clicking its header, the user can select the last column in the range by holding down the SHIFT key and clicking another column header. If necessary, the horizontal scroll bar can be used to bring additional columns into view.
2. Alternatively, the user can hold and drag the mouse pointer within the column headers to select multiple columns.

The **SelStartCol** and **SelEndCol** properties will be adjusted to reflect the columns selected by the user.

You can prevent a column selection from occurring at run time by setting the **Cancel** argument to True in the grid's **SelChange** event.

Moving columns

If the **AllowColMove** property is True, the user can move previously selected columns as a unit to a different location by pressing the mouse button within the header area of any selected column. The pointer will change to an arrow with a small box at its lower right corner, and the divider at the left edge of the column being pointed to will be enlarged and highlighted. Dragging the divider to the desired location and releasing the mouse button will move the selected columns immediately to the left of the divider. The moved columns remain selected.

If the user drags the divider to a position within the currently selected range, no movement occurs. Columns that are not selected cannot be moved interactively.

When a move occurs, the **Order** property is adjusted for all affected columns. You can always rearrange columns in code by modifying the **Order** property yourself.

You can prevent interactive column movement from occurring at run time by setting the **Cancel** argument to True in the **ColMove** event.

Selecting rows

If the **AllowRowSelect** and **RecordSelectors** properties are True, the user can select one or more records with the mouse. Unlike column selections, nonadjacent row selections are supported.

When the user clicks the record selector for a row, that row is selected and highlighted, and any rows or columns that were previously selected are deselected. The newly selected row also becomes the current row.

However, if the user holds down the CTRL key while making the selection, the current row does not change, and any previously selected rows remain selected. This technique also enables the user to select multiple rows, one at a time. Since selected rows do not have to be adjacent, the user can also operate the vertical scroll bar to bring other rows into view if desired.

The user can also select a range of contiguous rows by clicking the record selector of the first row in the range, then holding down the SHIFT key and clicking the record selector of the last row in the range. If necessary, the vertical scroll bar can be used to bring additional rows into view.

The user can deselect all rows by clicking a data cell or selecting columns. Clicking the record selector of a selected row does not deselect it.

The **SelBookmarks** collection will always be updated to reflect which rows are currently selected by the user. You can always select rows in code by adding bookmarks to the **SelBookmarks** collection. Similarly, you can deselect rows by removing bookmarks from this collection.

You can prevent a row selection from occurring at run time by setting the **Cancel** argument to True in the grid's **SelChange** event.

Sizing and Splitting

The following sections describe how users can resize rows, columns, and splits. You can always restrict any or all of these operations at design time or in code.

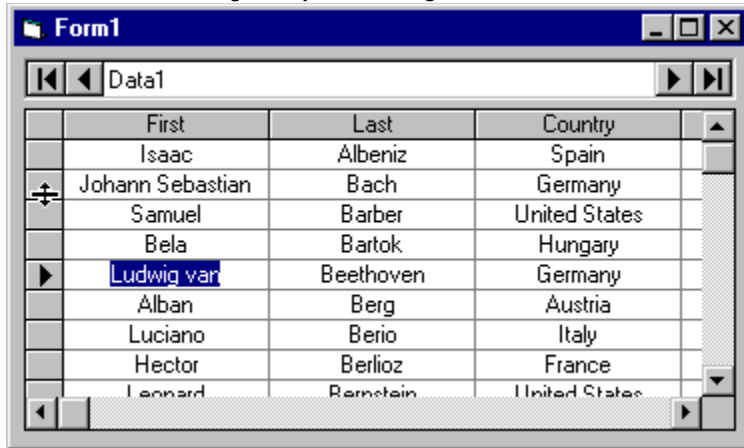
{button ,JI(`,`Sizing_rows')} Sizing rows

{button ,JI(`,`Sizing_columns')} Sizing columns

{button ,JI(`,`Sizing_splits')} Sizing splits

Sizing rows

If the **AllowRowSizing** property is True, the user can change the row height at run time. When the user points to a row divider in the record selector column, the pointer changes to a vertical double arrow, which the user can drag to adjust the height of all rows.



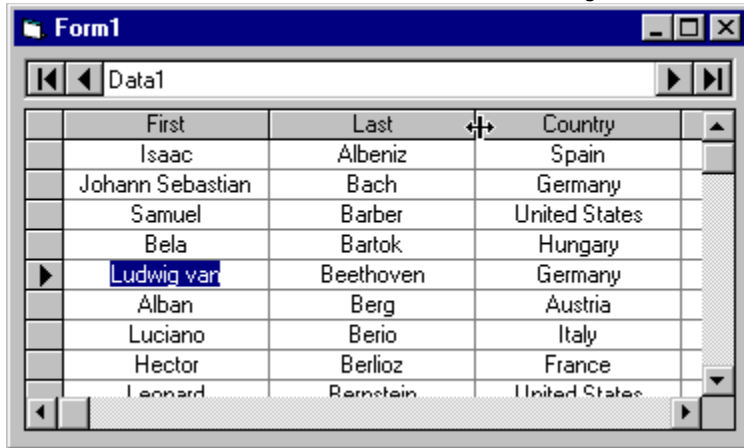
Dragging the pointer upward makes the rows smaller; dragging it downward makes the rows larger. All rows in the grid will be resized to the same height; it is not possible to resize individual rows. If the grid does not display the record selector column (that is, the **RecordSelectors** property is False), users cannot interactively change the row height.

The **RowHeight** property of the grid will be adjusted when the user completes the resize operation.

You can prevent row resizing from occurring at run time by setting the **Cancel** argument to True in the **RowResize** event. You can always change the **RowHeight** of the grid in code, even if **AllowRowSizing** is False or you cancel the **RowResize** event.

Sizing columns

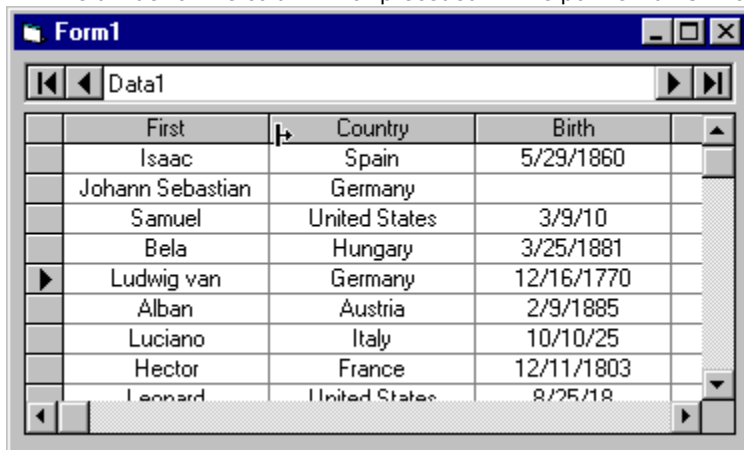
If the **AllowSizing** property is True for a column, the user can adjust the width of the column individually at run time. When the user points to the divider at the right edge of a column's header, the pointer changes to a horizontal double arrow, which the user can drag to resize the column in question.



Dragging the pointer to the left makes the column smaller; dragging it to the right makes the column larger. The column's **Width** property will be adjusted when the user completes the resize operation.

If the grid does not display column headers (that is, the **ColumnHeaders** property is False), the horizontal double arrow will appear when the pointer is over the column divider within the grid's data area.

If the user drags the pointer all the way to the left, the column retains its original **Width** property setting, but its **Visible** property is set to False. To make the column visible again, the user can point to the right side of the divider of the column that preceded it. The pointer turns into a vertical bar with a right arrow.



Dragging the pointer to the right establishes a new column width and sets the column's **Visible** property back to True.

You can prevent column resizing from occurring at run time by setting the **Cancel** argument to True in the **ColResize** event. You can always change the width of a column in code, even if **AllowSizing** is False for that column.

Sizing splits

If the **AllowSizing** property is True for a split, the user can reposition its split bar. If the split is the leftmost sizable split of the grid, the user can also create a new split. For details, see [Creating and sizing splits](#).

Database Operations

The editing, deleting, and adding permissions granted to the user at run time are controlled by the **AllowUpdate**, **AllowDelete**, and **AllowAddNew** properties. The default values of these properties are:

```
AllowUpdate = True
AllowDelete = False
AllowAddNew = False
```

Note that these properties only control user interaction with the grid at run time. They do not control whether database operations can be performed by the Data control or other bound controls, or by your application code.

```
{button ,JI(`,`Editing_data')} Editing data
{button ,JI(`,`Adding_a_new_record')} Adding a new record
{button ,JI(`,`Deleting_a_record')} Deleting a record
```

Editing data

True DBGrid's **AllowUpdate** property must be True in order for the user to edit data in the grid. The default value is True.

If the user moves to a cell and starts typing, the cell's data will be replaced by what is typed. Alternatively, clicking within the current cell will put the grid into edit mode (its **EditActive** property becomes True), enabling the user to modify the cell's data.

While editing, the LEFT ARROW and RIGHT ARROW keys move the insertion point within the cell. If the insertion point is at the beginning or end of the cell's text, the LEFT ARROW and RIGHT ARROW keys will terminate editing by moving to the adjacent cell. The UP ARROW and DOWN ARROW keys terminate editing by moving the current cell to the row above or below the current one. The user can also end editing without moving the current cell by pressing the ENTER key.

When one or more cells in a row have been modified, a pencil icon will appear in the record selector column to indicate that data in the row has been changed. The pencil icon does **not** mean that the grid's **EditActive** property is True; it means that the grid's **DataChanged** property is *True*. To cancel the changes made to the current cell, the user can press the ESC key. In fact, before moving to another row, the user can revisit any column within the current row and press the ESC key to restore the cell to its original value. If the user repeats this procedure for all modified cells in the row, the pencil icon in the record selector will disappear.

Moving to another row by clicking it, using the UP ARROW or DOWN ARROW keys, or by clicking the navigation buttons of the Data control will update the modified record to the database. If the update is successful, the pencil icon will disappear. If no grid columns have been modified, no update will occur when changing rows.

Adding a new record

True DBGrid's **AllowAddNew** property must be True in order for the user to add new records to the grid interactively. The default value is False.

If the **AllowAddNew** property is True, an empty *AddNew row*, marked by an asterisk in the record selector column, will be displayed after the last record. The user can initiate an add operation by navigating to the AddNew row, either by clicking it or by using the DOWN ARROW key, then typing new data. The first character typed will cause the grid to insert a blank row before the AddNew row. The newly inserted blank row becomes the current row.

At this point, the new row exists only in the grid---it does not have a bookmark, and it does not yet represent a physical database record. The new row is added to the underlying data source when the user navigates to another data row or the AddNew row.

Deleting a record

True DBGrid's **AllowDelete** property must be True in order for the user to delete records through the grid. The default value is False.

To delete a record, the user selects the row to be deleted by clicking its record selector, then pressing the DEL key. Only one record can be deleted at a time. The user cannot select multiple records and press the DEL key to delete them all.

NOTE: In order for the record to be deleted, the grid must have focus so it can receive the DEL key. Clicking the grid's record selector column does **not** set focus to the grid. However, if you always want the grid to receive focus when the user clicks the record selector column, set focus to the grid in the grid's **SelChange** event:

```
Private Sub TDBGrid1_SelChange(Cancel As Integer)
    TDBGrid1.SetFocus
End Sub
```

Drag-and-Drop Behavior

Typically, implementing a drag-and-drop interface using a grid is a painstaking task, since you normally want to drag data from a particular cell or row to another. Visual Basic's drag-and-drop facilities work with entire *controls*, but do not provide features for detecting which element of a control is involved.

True DBGrid has a special event, **DragCell**, designed to simplify the initiation of a drag operation. **DragCell** is called whenever the user attempts to drag data from a cell to another location; your code can respond accordingly. **DragCell** informs you of the split index, row bookmark, and column index of the cell being dragged. Typically, you will save the information so that it is available when the drag-and-drop operation terminates. At design time, be sure to change the grid's **DragIcon** property to a meaningful icon, since the default behavior is to drag an outline of the grid.

NOTE: The **DragCell** event will not fire for the current cell when the grid's **MarqueeStyle** property is set to the default value of 6 - Floating Editor. This is because the floating editor processes mouse events itself, as it must handle insertion point movement and text selection. For more information, see [Highlighting the Current Row or Cell](#).

For example, assume that you want to be able to drag a row of data elsewhere. The following code in the **DragCell** event handler starts the drag-and-drop operation:

```
Private Sub TDBGrid1_DragCell(ByVal SplitIndex As Integer, _
    RowBookmark As Variant, ByVal ColIndex As Integer)

    ' Set the current cell to the one being dragged
    TDBGrid1.Col = ColIndex
    TDBGrid1.Bookmark = RowBookmark

    ' Set up drag operation, such as creating visual effects by
    ' highlighting the cell or row being dragged.

    ' Use VB manual drag support (put TDBGrid1 into drag mode)
    TDBGrid1.Drag = vbBeginDrag
End Sub
```

Note that the **Col** and **Bookmark** properties of the grid are set to reflect the cell that was dragged. Once this event is completed, Visual Basic takes over the drag operation (see the Visual Basic documentation for the **Drag** method). Place code in the **DragDrop** event of the destination to perform the actions related to the drop.

If the destination of a drag operation is another True DBGrid control, and you want to drop data into a row or cell, you need to consider the following:

- You may want to provide feedback for the user about which row or cell will be the target of the drop.
- You may want to respond to the **DragDrop** event by actually entering the data into the target cell or row.
- The limitations of your data source may preclude some operations. For example, you cannot insert a row into the middle of a recordset or resultset---you can only modify existing records or append new ones. However, if your grid is populated from an array in unbound mode, you can insert a new row into the array.

Often, the difficulty with implementing such operations on grids is that, given the mouse location, it is difficult to find out which cell, row, or column you are about to drop into. True DBGrid solves this problem by providing the **SplitContaining**, **ColContaining**, and **RowContaining** methods, which translate mouse coordinates into a grid location.

Suppose that you want to provide feedback to the user about which cell they are over. The easiest way to do

this is in the **DragOver** event, which fires as the mouse moves over the destination grid. Here's how you would set the current cell pointer so that it tracks the dragging object:

```
Private Sub TDBGrid2_DragOver(Source As Control, _
    X As Single, Y As Single, State As Integer)
    ' Set current cell to "track" the dragging object
    Dim overCol As Integer
    Dim overRow As Long
    overCol = TDBGrid2.ColContaining(X)
    overRow = TDBGrid2.RowContaining(Y)
    If overCol >= 0 Then TDBGrid2.Col = overCol
    If overRow >= 0 Then TDBGrid2.Row = overRow
End Sub
```

When the drop occurs (detected in the **DragDrop** event), you can move the appropriate data into the destination grid, or perform whatever action you want the drop to trigger. For example, you can copy the contents of the dragged cell (which was made current in the [DragCell](#) example presented earlier) to the current cell in the destination grid:

```
Private Sub TDBGrid2_DragDrop(Source As Control, _
    X As Single, Y As Single)
    TDBGrid2.Columns(TDBGrid2.Col).Value = _
        TDBGrid1.Columns(TDBGrid1.Col).Value
End Sub
```

You should also perform some clean-up when the drag-and-drop operation fails or the user completes the drop outside the boundaries of the destination control. [Tutorial 14](#) demonstrates how to implement a drag-and-drop interface from one grid to another.

Bound Mode

The easiest way to use True DBGrid is in *bound mode*, in which it communicates directly with an intrinsic or external data control to retrieve and update data. Visual Basic provides two data controls that work with True DBGrid in this fashion:

- The built-in Data control, which is included in the standard toolbox.
- The Remote Data Control (RDC), which you add to a project just as you would any other ActiveX control.

You can also use any other data control that adheres to the Microsoft data binding specifications, such as Microsoft's ActiveX Data Connector (ADC), or APEX's MyData Control.

To use bound mode, set the **DataMode** property of the grid to 0 - Bound at design time, then set the **DataSource** property of the grid to reference an intrinsic or external data control on the same Visual Basic form. You do not need to fully configure the data control at design time since True DBGrid automatically responds to changes in the data source at run time. Therefore, you can defer specifying a database table or query until the application is running.

{button ,JI(`,`Binding_True_DBGrid_to_a_Data_Control')} Binding True DBGrid to a Data Control
{button ,JI(`,`Visual_Basic_Data_Control_Considerations')} Visual Basic Data Control Considerations
{button ,JI(`,`Remote_Data_Control_RDC_Considerations')} Remote Data Control (RDC) Considerations
{button ,JI(`,`Unbound_Columns')} Unbound Columns

Binding True DBGrid to a Data Control

You can make an association between a True DBGrid data bound control and a Visual Basic Data control by setting the **DataSource** property of the grid after the Data control has been created on the same form. This is all that is required to make True DBGrid fully aware of the database in your application.

Once such a link exists, True DBGrid and the Data control automatically notify and respond to all operations performed by the other, simplifying your application development:

- Once the grid has been associated with a Data control, you can retrieve the field layout of the database at design time and use the property pages to customize the appearance of the grid. For more information, see [Design Time Interaction](#).
- When you run your application, True DBGrid will automatically respond by displaying the contents of the **Recordset** defined by the Data control's **RecordSource** property. Data will be fully available at run time, and can be edited as well.

True DBGrid will automatically update its display to reflect any changes made to the Data control. However, True DBGrid waits until the system is idle before performing such display updates, since your program may need to perform other actions before the display is synchronized.

Visual Basic Data Control Considerations

This section gives an overview of how True DBGrid interacts with the built-in Data control of Visual Basic. Generally speaking, True DBGrid responds to external data controls in a similar manner; however, the terminology and programming interface used in this section is specific to Visual Basic's intrinsic Data control.

{button ,JI(`,`How_True_DBGrid_reacts_to_Recordset_changes')}} How True DBGrid reacts to Recordset changes

{button ,JI(`,`Interactions_between_True_DBGrid_and_the_Data_control')}} Interactions between True DBGrid and the Data control

{button ,JI(`,`Using_True_DBGrid_with_a_Data_control_on_a_different_form')}} Using True DBGrid with a Data control on a different form

{button ,JI(`,`Using_True_DBGrid_to_display_SQL_query_results')}} Using True DBGrid to display SQL query results

How True DBGrid reacts to Recordset changes

When you bind a grid to a Data control, you are actually linking the grid to the underlying **Recordset** object, which is managed by the Data control. The **Recordset** object can be accessed directly using the **Recordset** property of the Data control:

```
Data1.Recordset.MoveFirst
```

This statement positions the record pointer to the first record in the **Recordset**.

NOTE: As of Visual Basic 4.0, the Data control does not support the outdated **Dynaset** object. You can create either a Table, Snapshot, or Dynaset-type **Recordset** using the Data control's **RecordsetType** property. By default, the Data control creates a Dynaset-type **Recordset** that is completely compatible with the **Dynaset** object, so you should not have to modify any existing code. Consult the Visual Basic Help for more information.

You need not worry about keeping the grid synchronized with changes made to the **Recordset**, as this happens automatically. Modifying the **Recordset** is the preferred way to effect changes to the grid's display.

Here is how the grid responds to various operations performed on the **Recordset** associated with the Data control:

Positioning	The grid's current row will change in response to a successful invocation of any of the following Recordset methods: Seek , Move , MoveFirst , MoveLast , MoveNext , MovePrevious , FindFirst , FindLast , FindNext , and FindPrevious . If necessary, the grid will scroll to make the current row visible. The grid optimizes its response to these messages. For example, if your code performs a MoveFirst followed by 20 consecutive MoveNext calls, the grid will position only once, to the 21st row. Consult the Visual Basic Help for more information on the Recordset object's methods.
Update	The Recordset object's Update method causes the grid to write any changed data to the database. The update may be canceled in the Data control's Validate event. After the database is updated, the grid will clear the pencil icon in the record selector column.
Delete	True DBGrid reacts to the Recordset object's Delete method by removing the current row from the grid display. However, the Delete method does not change the current record, therefore the Bookmark property of both the grid and the Recordset still refers to the record that was just deleted. At this point, there is no current row in the grid, and its Row property returns -1.
Requery	This method causes the grid to refetch and redisplay all data. Any modifications made to the grid's current row that have not been updated to the database will be lost. After the Requery method is executed, the current cell will be the leftmost visible column of the first record, which will be displayed at the upper left corner of the grid.

Interactions between True DBGrid and the Data control

The Data control's First, Last, Next, and Previous buttons can be used to navigate through the grid or any other bound control. When a Data control button is clicked, the new current row is indicated in the record selector column of the grid:

- The First and Last buttons have the same effect as the **MoveFirst** and **MoveLast** methods of the **Recordset**. The grid will position to the first or last row.
- The Next and Previous buttons have the same effect as the **MoveNext** and **MovePrevious** methods of the **Recordset**. The grid will move forward or backward by one row.

If grid data has been edited, moving the record pointer using the Data control will automatically trigger an update to the database.

The grid responds to the Data control's **Refresh**, **UpdateRecord**, and **UpdateControls** methods as follows:

Refresh	The grid refetches and redisplay all data. Any modifications made to the grid's current row that have not been updated to the database will be lost. After the refresh, the current cell will be the leftmost visible column of the first record, which will be displayed at the upper left corner of the grid.
UpdateRecord	If data in the current row has changed, the modified data in the grid will be written to the database without firing the Data control's Validate event. The pencil icon in the record selector column will disappear. The current cell and the grid display are not changed.
UpdateControls	Any modifications made to the grid's current row that have not been updated to the database will be discarded and data will be refreshed from the database. The current cell and the grid display are not changed. This method can be used to cancel an AddNew or Edit operation.

Using True DBGrid with a Data control on a different form

Using the Data control's **Recordset** property, you can effectively bind True DBGrid or any other bound control on one form to a Data control on another form. Strictly speaking, you cannot *directly* bind a grid to a Data control on another form. For example, assume you have a grid on Form2 and you would like it to display data from the **Recordset** of Data1, which is on Form1. You need to first bind the grid to a Data control (Data2) on Form2. Form2.Data2 is not connected to a database. Instead, the **Recordset** of Form2.Data2 is set to the **Recordset** of Form1.Data1:

```
Form2.Data2.DatabaseName = Form1.Data1.DatabaseName  
Set Form2.Data2.Recordset = Form1.Data1.Recordset
```

The grid on Form2 will work as if it were directly bound to Form1.Data1. When you move or update records through Form1.Data1, the grid on Form2 will respond accordingly. Conversely, if you move or update records in the grid, all controls on Form1 that are bound to Form1.Data1 will respond.

Using True DBGrid to display SQL query results

True DBGrid can automatically configure itself to changes in the data control's **Recordset**. This feature is very useful for displaying the results of ad-hoc SQL queries. If the grid does not have a layout defined, it will detect any change in the Data control's **Recordset** and display the new query result set automatically---no code is necessary to tell the grid what to do. You can create very powerful user interfaces using these automatic features of the grid. [Tutorial 2](#) describes this feature in detail and also illustrates a few interesting and useful SQL statements.

Remote Data Control (RDC) Considerations

The Remote Data Control (RDC), which is used to connect to ODBC-compliant data sources, can also be used in True DBGrid's bound mode. Alternatively, you can work with the Remote Data Objects (RDO), and use the **rdoResultset** object to populate a True DBGrid control in one of the unbound modes.

APEX provides several sample programs that demonstrate how to use RDC and RDO with True DBGrid. You can download these programs from the APEX Web site at <http://www.apexsc.com>.

Unbound Columns

Normally, True DBGrid automatically displays data from bound database fields. However, you may need to augment the set of fields present in your layouts with columns derived from database fields, or columns which are unrelated (or only loosely related) to database information. For example, if your database contains a Balance field, you may instead want to display two columns, Credit and Debit, to show positive and negative numbers separately. Or, you may want to look up data in another database, or convert field data to some other form, such as mapping numeric codes to textual descriptions.

To accomplish such tasks you can use unbound columns. The term *unbound column* refers to a column that is part of a *bound grid*, but is not tied directly to a database field. A bound grid has its **DataMode** property set to 0 - Bound, and its **DataSource** property set to the name of a Visual Basic Data control (or a Remote Data Control). Unbound columns are not used in any of the unbound modes.

Columns that do not have the **DataField** property set (that is, the **DataField** property is equal to an empty string), but do have the column **Caption** property set are considered unbound columns. The grid will request data for these columns through the **UnboundColumnFetch** event.

Columns with their **DataField** property set will be bound to the underlying **Recordset** if the **DataField** property is the same as one of the fields of the **Recordset**.

Columns with their **DataField** property set to a value that is not in the **Recordset** are ignored for the purposes of fetching data. Similarly, columns that have no value for both the **DataField** and **Caption** properties set are also ignored. Values entered into the grid for these columns will not be cached by the grid, and will therefore disappear when the row is scrolled out of view.

```
{button ,JI(`,`Creating_unbound_columns')}} Creating unbound columns
{button ,JI(`,`Implementing_unbound_columns_using_Recordset_clones')}} Implementing unbound columns
using Recordset clones
{button ,JI(`,`Implementing_unbound_columns_using_cell_access_methods')}} Implementing unbound
columns using cell access methods
{button ,JI(`,`Implementing_multiple_unbound_columns')}} Implementing multiple unbound columns
{button ,JI(`,`Updating_unbound_columns')}} Updating unbound columns
{button ,JI(`,`Editing_unbound_columns')}} Editing unbound columns
```

Creating unbound columns

The first step in using an unbound column is creating the column itself. This may be done at design time by choosing either the **Append** or **Insert** command from the grid's visual editing menu. At run time, unbound columns may be added using the **Add** method of the **Columns** collection. The column must be given a name by setting its **Caption** property. At design time, this is done using the Columns property page. At run time, the **Caption** property of the appropriate **Column** object is set. When these actions are performed at run time, new columns are not considered as unbound columns until the grid's **ReBind** method is executed.

NOTE: If you attempt to insert an unbound column in code, you may need to use the **HoldFields** method to ensure that the column appears at the desired position within the grid:

```
Dim Col As TrueDBGrid50.Column

With TDBGrid1
    Set Col = .Columns.Add(1)
    Col.Visible = True
    Col.Caption = "Unbound"
    .HoldFields
    .ReBind
End With
```

When the grid needs to display the value of an unbound column, it fires the **UnboundColumnFetch** event. This event supplies the user with a bookmark and a column index as the means of identifying the grid cell being requested. The third argument to the event is a Variant which by default is Null, but can be changed to any desired value, and will be used to fill the contents of the cell specified by the given bookmark and column index.

```
Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)
```

When **Recordset** data is retrieved through the data control, the data is cached by the grid to allow smooth scrolling operations and rapid display. As a result, many rows must be fetched at one time so that they are readily available for display. Internally, the grid uses a **Recordset** clone when it requests data, thus allowing data to be retrieved without changing the current row of the bound **Recordset** managed by the Data control. Why is this important? The reason is that **UnboundColumnFetch** may need to fetch data unrelated to the current row position.

For example, suppose a row is being edited and the user scrolls the grid vertically or horizontally. To update the display, the grid will need to fetch the new data that is scrolled into view for all rows and columns on the face of the grid. However, changing the current row would cause an unwanted update to occur. For this reason, the grid **will not allow** the current row of the grid or **Recordset** to be changed during the **UnboundColumnFetch** event, even through implicit means such as the **FindFirst** method of the **Recordset**. Similarly, other **Recordset** operations are prohibited as well during the course of this event.

Given these restrictions, how do you obtain **Recordset** data in order to set the values of the unbound column? There are several ways, all of which involve the use of a **Recordset clone**.

Implementing unbound columns using Recordset clones

The simplest way to gather the data from other columns is to clone the **Recordset**, move to the specified bookmark, and get the data from the clone, as in the following example:

```
Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

    Dim myclone As Recordset
    Set myclone = Data1.Recordset.Clone()

    myclone.Bookmark = Bookmark
    Value = myclone.Fields(Col)
End Sub
```

Although this example is functional, it would be more efficient to make the clone a global object, as it would no longer be necessary for Visual Basic to create it with each call to the event. A good place to do this is in the **Form_Load** event, which fires before the grid is displayed:

```
Dim ucfClone As Recordset ' Global UnboundColumnFetch clone

Private Sub Form_Load()
    Data1.Refresh ' Make sure the recordset is created first
    Set ucfClone = Data1.Recordset.Clone()
End Sub

Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

    ucfClone.Bookmark = Bookmark
    Value = ucfClone.Fields(Col)
End Sub
```

You can speed things up even more by using a **Field** object, creating it from the clone's **Fields** collection. This is faster because Visual Basic does not need to locate the correct field each time the event is called:

```
Dim ucfClone As Recordset ' Global UnboundColumnFetch clone
Dim ucfField As Field ' Global UnboundColumnFetch field

Private Sub Form_Load()
    Data1.Refresh
    Set ucfClone = Data1.Recordset.Clone()
    Set ucfField = ucfClone.Fields(1)
End Sub

Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

    ucfClone.Bookmark = Bookmark
    Value = ucfField
End Sub
```

After the `ucfField` object is initialized in **Form_Load**, it will always contain the data in Field 1 of the current row of the clone. If the underlying database field allows null values, you should test the **Field** object first before assigning its data to the Value argument:

```
Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
```

```

        ByVal Col As Integer, Value As Variant)

    ucfClone.Bookmark = Bookmark
    If Not IsNull(ucfField) Then Value = ucfField
End Sub

```

Using **Field** objects is an elegant approach. Not only is it more efficient, but it frees you from keeping track of collection indexes throughout your code. For example, given a Rectangle table containing Length and Width fields, the following code implements an unbound column that uses **Field** objects to calculate the area:

```

Dim ucfClone As Recordset ' Global UnboundColumnFetch clone
Dim ucfLength As Field
Dim ucfWidth As Field

Private Sub Form_Load()
    Data1.Refresh
    Set ucfClone = Data1.Recordset.Clone()
    Set ucfLength = ucfClone.Fields("Length")
    Set ucfWidth = ucfClone.Fields("Width")
End Sub

Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

    ucfClone.Bookmark = Bookmark
    Value = ucfLength * ucfWidth ' Calculate "Area" column
End Sub

```

Implementing unbound columns using cell access methods

Using **Recordset** clones is the preferred way to handle the **UnboundColumnFetch** event. However, if the grid is bound to a data control that does not support clones, such as the Microsoft Remote Data Control (RDC), you can derive cell values using the **Column** object methods **CellText** and **CellValue**.

```
Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _  
    ByVal Col As Integer, Value As Variant)  
  
    Value = TDBGrid1.Columns(1).CellText(Bookmark)  
End Sub
```

Note that these methods are not as efficient as using your own clone. This is because they always create a new clone (internal to the grid), get the value, then destroy the clone. However, at times using **CellText** or **CellValue** may be preferable for the sake of simplicity.

Using a global clone can become complicated when the data control is refreshed frequently. Refreshing the data control rebuilds the **Recordset**, meaning that the data control's bookmarks are no longer the same as the bookmarks of the clone. Thus, re-cloning and re-establishing the field variables is necessary, or else the clone will continue to access the data of the old **Recordset**. As this may be a cumbersome process, you may find that the simplicity of **CellText** and **CellValue** is a workable alternative.

Finally, please note that **CellText** and **CellValue** cannot be used to retrieve the values of other unbound columns within the context of the **UnboundColumnFetch** event. Attempts to do so will always return an empty string (**CellText**) or Null (**CellValue**). The grid has been designed this way to avoid infinite recursions of **UnboundColumnFetch** events when two unbound columns reference one another.

Implementing multiple unbound columns

So far, our examples have demonstrated the **UnboundColumnFetch** event using only a single unbound column. Suppose you want to have more than one? Since the **UnboundColumnFetch** is fired for each unbound column of each row, only one column value may be set at a time, and each column must be identified for the value to be properly determined. The second **UnboundColumnFetch** argument, Col, is used to identify the column of the grid for which the value is required.

Returning to the Rectangle example, the following code also displays the perimeter of the rectangle in addition to its area:

```
Dim ucfClone As Recordset ' Global UnboundColumnFetch clone
Dim ucfLength As Field
Dim ucfWidth As Field

Private Sub Form_Load()
    Data1.Refresh
    Set ucfClone = Data1.Recordset.Clone()
    Set ucfLength = ucfClone.Fields("Length")
    Set ucfWidth = ucfClone.Fields("Width")
End Sub

Private Sub TDBGrid1_UnboundColumnFetch(Bookmark As Variant, _
    ByVal Col As Integer, Value As Variant)

    ucfClone.Bookmark = Bookmark

    Select Case TDBGrid1.Columns(Col).Caption
        Case "Area"
            ' Calculate "Area" column of grid
            Value = ucfLength * ucfWidth
        Case "Perimeter"
            ' Calculate "Perimeter" column of grid
            Value = 2 * (ucfLength + ucfWidth)
    End Select
End Sub
```

Please note the use of the column captions to identify the actual column for which the value is to be set. The **Caption** property (instead of the index number) is sometimes necessary to identify the proper column. If columns are added or removed at run time, each column's index number could change, resulting in different values of Col for the Area column. Since the caption text is much less ambiguous than the column number, code is easier to read, more reliable, and self-adjusting to column layout changes.

Updating unbound columns

In most cases, you will want unbound columns to be read-only, as the values are derived from other data in the grid. In these cases, you should set the **Locked** property of the column to True.

If **Locked** is False and updates are allowed, the user can edit the values of an unbound column, even if the underlying **Recordset** is not updatable. However, if the underlying **Recordset** is not updatable, the unbound column's **Locked** property will automatically be set to True at runtime, regardless of the design time setting in the property pages. Therefore, for editing to be allowed, you must set the unbound column object's **Locked** property to False at run time using code.

If editing of an unbound column occurs, the row will be marked as dirty (a pencil icon will be shown in the record selector column) and the update sequence will be performed as usual. However, the grid does not know what to do with the modified data, since there is no database field in which to store it. Therefore, you must put code in the **BeforeUpdate** and **AfterUpdate** events (or **BeforeInsert** and **AfterInsert** for AddNew operations) to properly store the edited values. These values may be stored in any manner desired, including another database table.

BeforeUpdate can be used to cancel the update operation. Therefore, if the unbound column is to be used in cooperation with another database, the update of the unbound column should be performed in **BeforeUpdate**. If the operation fails, then the event should be canceled. However, if the operation succeeds, then the bound update should be allowed to proceed. The bound update may then fail, hence any database actions associated with unbound columns would best be handled on a transactional basis.

If the bound update succeeds, the **AfterUpdate** event is fired, and the unbound column transaction should be committed. If the bound update fails, the unbound column transaction should be rolled back in either the grid's or the Data control's **Error** event, or within a trappable error handler, depending on how the update was initiated. If transactions are not available, then you must store the original unbound column values prior to the update, then perform another update to restore these values should the bound update fail.

Editing unbound columns

Another technique for updating an unbound column is to use the **AfterColUpdate** event to adjust the value of other (bound) columns. For example, imagine a pair of columns for Debit and Credit, as shown in this portion of a grid display:

Debit	Credit
\$13,677.13	
\$3,288.50	
\$5,466.78	
	\$2,208.00
	\$1,513.00
\$287.30	
\$2,344.50	

Assume that there is no database field for these, but that they are unbound columns which derive their value from a single Balance column, which is either positive or negative. From the user's perspective, it would be desirable to edit these values directly; from your perspective, it would be desirable to have the grid update the dependent Balance column automatically.

True DBGrid makes such tasks easy. Here's the code you would put in the grid's **AfterColUpdate** event to cause either column to change the Balance column when updated:

```
Private Sub TDBGrid1_AfterColUpdate(ByVal ColIndex As Integer)
    Dim Cols As Columns
    Set Cols = TDBGrid1.Columns

    Select Case Cols(ColIndex).Caption
        Case "Debit"
            Cols("Balance").Value = -Cols(ColIndex).Value
        Case "Credit"
            Cols("Balance").Value = Cols(ColIndex).Value
    End Select
End Sub
```

Notice that, when updating these columns, the code actually changes the value of the Balance column, which is both bound and invisible.

Storage Mode

True DBGrid supports an array-based unbound mode, or *storage mode*, that does not rely upon the Visual Basic Data control or any other data provider that follows the Microsoft data binding specifications. Instead, storage mode uses an APEX **XArray** object as a data source. A 32-bit version of **XArray** is included with True DBGrid (and also with APEX's MyData Control product).

To use storage mode, set the **DataMode** property of the grid to 4 - Storage at design time. In code, redimension and populate an **XArray** object with your data just as you would a Visual Basic array, then assign the **XArray** object to the **Array** property of the grid. The data will then be maintained and exchanged between the grid and the **XArray** object automatically. There are no unbound events to write, making this mode the easiest to use.

Storage mode was created to deliver ease of use without sacrificing the power and flexibility that you expect from True DBGrid. When using this mode, the index of the first dimension of the **XArray** object serves as a bookmark to uniquely identify rows. This means that all of the grid's bookmark-related properties, methods, and events (**Bookmark**, **FirstRow**, **GetBookmark**, **FetchCellStyle**, and others) work the same in storage mode as in any other **DataMode**, either bound or unbound.

NOTE: **DataMode 4 - Storage** is not available in any of the 16-bit versions of True DBGrid (since there is no 16-bit version of the **XArray** object).

{button ,JI(`,`When_to_Use_Storage_Mode')}} When to Use Storage Mode

{button ,JI(`,`Using_the_XArray_Object')}} Using the XArray Object

{button ,JI(`,`Interactions_between_True_DBGrid_and_XArray')}} Interactions between True DBGrid and XArray

{button ,JI(`,`Storage_Mode_Example')}} Storage Mode Example

When to Use Storage Mode

If you need to display and manipulate two-dimensional array data, either one of the following **DataMode** settings is ideally suited to the task:

- 3 - Application, which fires events on a cell-by-cell basis,
- 4 - Storage, which communicates directly with an **XArray** object.

The choice of which one to use depends upon the target operating system. If you need to deliver both 16- and 32-bit versions of your programs, application mode is recommended, since storage mode is only available in the 32-bit versions of True DBGrid. If you do not need to support 16-bit platforms, storage mode is recommended, since it is easier to use.

Using the XArray Object

XArray is an ActiveX object designed as a drop-in replacement for Visual Basic variant arrays. The **XArray** object, which can be used outside the scope of True DBGrid, is functionally similar to a standard array but provides additional flexibility. For example, data is automatically preserved when redimensioning, inserting and removing indexes, and inserting and removing dimensions (up to 10).

Once created, an **XArray** object is assigned to a **TDBGrid** control via the grid's **Array** property (usually in the **Form_Load** event). You can create as many **XArray** objects as you want, then attach them to one or more grids as needed.

Future versions of **XArray** will provide additional features, such as sorting, while retaining the same interface with True DBGrid and other APEX bound controls.

```
{button ,JI(`,`Adding_XArray_to_a_Visual_Basic_project')} Adding XArray to a Visual Basic project
{button ,JI(`,`Creating_an_XArray_object')} Creating an XArray object
{button ,JI(`,`Redimensioning_an_XArray_object')} Redimensioning an XArray object
{button ,JI(`,`Populating_an_XArray_object')} Populating an XArray object
{button ,JI(`,`Attaching_an_XArray_object_to_a_True_DBGrid_control')} Attaching an XArray object to a True
DBGrid control
```

Adding XArray to a Visual Basic project

From the **Project** menu in Visual Basic 5.0 (or the **Tools** menu in Visual Basic 4.0), select **References...** to display a list of available type library references. Select the check box labeled APEX XArray Object (XARRAY32.OCX), then press the **OK** button.

Creating an XArray object

XArray has no design time interface or persistent properties. All **XArray** operations are performed in code at run time.

To create an **XArray** object at the start of an application, add the following line to the general declarations section of a form:

```
Dim MyArray As New XArray
```

To declare an **XArray** object variable without creating it, omit the `New` keyword. Use the `Set` statement to create the **XArray** in code:

```
Dim MyArray As XArray  
Set MyArray = New XArray
```

Redimensioning an XArray object

Before an **XArray** object can be used, you must define its dimensions in code with the **ReDim** method, which is similar to its counterpart in Visual Basic. For example, the following line of code sets up a two-dimensional array with 20 rows (indexed from 1 to 20) and 4 columns (indexed from 0 to 3):

```
MyArray.ReDim 1, 20, 0, 3
```

You can use the **Count** property to determine the number of elements in a given dimension:

```
Debug.Print MyArray.Count(1) ' prints 20  
Debug.Print MyArray.Count(2) ' prints 4
```

Note that the index passed to the **Count** property is one-based. To determine the valid indexes for a given dimension, you can use the **LowerBound** and **UpperBound** properties:

```
Debug.Print MyArray.LowerBound(1) ' prints 1  
Debug.Print MyArray.UpperBound(1) ' prints 20  
Debug.Print MyArray.LowerBound(2) ' prints 0  
Debug.Print MyArray.UpperBound(2) ' prints 3
```

When an **XArray** object is connected to a **TDBGrid** control, its first dimension always specifies the row index from the grid's perspective. Or, to put it another way, the set of allowable bookmarks ranges from `LowerBound(1)` to `UpperBound(1)`.

Likewise, the second dimension of an **XArray** object always specifies the column index from the grid's perspective. Or, to put it another way, the grid addresses data columns in the **XArray** using indexes that range from `LowerBound(2)` to `UpperBound(2)`. Therefore, if your application manipulates columns in code, it is a good idea to make the second **XArray** dimension zero-based instead of one-based. That way, you can use the same indexes to refer to both grid columns and **XArray** columns.

Although **XArray** supports up to 10 dimensions, when used in conjunction with True DBGrid's storage mode (**DataMode** 4), only two-dimensional and one-dimensional arrays make sense.

Populating an XArray object

To set or retrieve an element of an **XArray** object, use the **Value** property:

```
MyArray.Value(x, y) = "A string"  
s$ = MyArray.Value(x, y)
```

Since the **Value** property is the default property for **XArray**, the preceding statements can be shortened to:

```
MyArray(x, y) = "A string"  
s$ = MyArray(x, y)
```


Attaching an XArray object to a True DBGrid control

Since **XArray** objects can only exist at run time, you must write code to associate them with a **TDBGrid** or **TDBDropDown** control. This is done by setting the **Array** property:

```
TDBGrid1.Array = MyArray
```

Once this association is made, the grid will store a reference to the **XArray** object as query it as needed to determine the contents of individual cells. When the **Array** property is set, the grid also determines the number of rows (the first dimension) in an **XArray** object and calibrates the vertical scroll bar accordingly.

However, the grid will never adjust its column layout to match the number of columns (the second dimension) in an **XArray** object. As with other unbound data modes, you must define the column layout yourself at design time, run time, or a combination of both. At run time, you can use the following code to clear the existing columns from a grid, then create a new one for each **XArray** column:

```
Dim C As TrueDBGrid50.Column
With TDBGrid1.Columns
    While .Count > 0
        .Remove 0
    Wend
    While .Count < MyArray.Count(2)
        Set C = .Add(0)
        C.Visible = True
    Wend
End With
```

Note that newly created columns are invisible by default, so you must explicitly set their **Visible** property to True.

Interactions between True DBGrid and XArray

Add, update, and delete operations performed through True DBGrid's user interface or its properties and methods are automatically reflected in the attached **XArray** object. For example, if the following code is executed:

```
With TDBGrid1
    .Text = "New value"
    .Update
End With
```

then the current cell of the grid and the corresponding element of the associated **XArray** object are automatically updated, and the following expression will be True (assuming that the second array dimension is zero-based):

```
MyArray(TDBGrid1.Bookmark, TDBGrid1.Col) = "New value"
```

However, the reverse is not true. If you insert or delete **XArray** rows or columns directly in code, or even change the value of a single element, the grid does not receive any notifications from the **XArray**. Therefore, you must either **Refresh**, **ReBind**, or **ReOpen** the grid in order to update the display.

You can follow two rules of thumb to ensure that the display is updated properly:

1. If you insert or delete **XArray** columns in code, you are changing the structure of the underlying data source. Therefore, you should invoke the grid's **ReBind** or **ReOpen** method.
2. If you insert or delete **XArray** rows in code, or change the value of an array element, you are changing the underlying data source without altering its structure. Therefore, you should invoke the grid's **Refresh** method.

{button ,JI(`,`Updating_XArray_elements')} Updating XArray elements

{button ,JI(`,`Inserting_and_removing_XArray_rows')} Inserting and removing XArray rows

{button ,JI(`,`Inserting_and_removing_XArray_columns')} Inserting and removing XArray columns

Updating XArray elements

With the **XArray** object, you always have instant access to all "records" in your "database," so you can update cells in different rows directly without having to move the current record pointer, initiate edit mode, modify one or more cells, then update the changed record. For this reason, it is usually more convenient to make changes to **XArray** directly, then use the grid's **Refresh** method to update the display.

It is important to note that when you change one or more elements in an **XArray**, you **must** refresh the grid, or else the display will not reflect the correct data. Consider the following example, which implements a command button that clears the contents of the current grid row, then sets focus to the grid:

```
Private Sub Command1_Click()  
    Dim row As Long, col As Integer  
    With TDBGrid1  
        row = .Bookmark  
        With MyArray  
            For col = .LowerBound(2) To .UpperBound(2)  
                .Value(row, col) = ""  
            Next col  
        End With  
        .Refresh  
        .EditActive = True  
        .EditActive = False  
        .SetFocus  
    End With  
End Sub
```

Note that the **Bookmark** property of the grid is used as a row index for the **XArray** object. The loop that clears the current row also uses the **LowerBound** and **UpperBound** properties to iterate over the columns (second dimension) of the array. This technique will work with any **XArray**, although you can substitute integer constants if the array bounds are known in advance.

After the array elements are cleared, the grid's **Refresh** method is invoked, causing the non-current cells in the current row to be repainted as empty. Note that if the **MarqueeStyle** property is set to its default value of 6 - Floating Editor, the **Refresh** method does not clear the text within the floating editor window. However, by setting the **EditActive** property to True, then False, you can clear the floating editor as well.

Inserting and removing XArray rows

When inserting rows, as when updating cells, you must **Refresh** the grid afterwards, since it does not receive any notifications from the **XArray** object. You do not need to perform a **ReBind**, since the underlying database structure has not changed.

The following example implements a command button that inserts a new row before the current grid row, then sets focus to the grid:

```
Private Sub Command1_Click()  
    With TDBGrid1  
        MyArray.Insert 1, .Bookmark  
        .Refresh  
        .EditActive = True  
        .EditActive = False  
        .SetFocus  
    End With  
End Sub
```

As with the previous example, which clears the contents of the current grid row, the **EditActive** property is used to clear the text within the floating editor window when the **MarqueeStyle** property is set to its default value of 6 - Floating Editor.

To delete the current row, you could use the **Delete** method of **XArray**, then refresh the grid. However, the **Delete** method of the grid provides a more direct way of accomplishing the same task.

Inserting and removing XArray columns

If you use the **Insert** or **Delete** methods of **XArray** to add or remove columns (that is, the second dimension), the grid will not automatically insert or delete its own columns. You must write code to do this, as in the following example, which implements a command button that inserts a new column before the current one:

```
Private Sub Command1_Click()  
    Dim C As TrueDBGrid50.Column  
    With TDBGrid1  
        MyArray.Insert 2, .Col  
        Set C = .Columns.Add(.Col)  
        C.Visible = True  
        .ReBind  
        .SetFocus  
    End With  
End Sub
```

Note the use of the grid's **ReBind** method rather than the **Refresh** method. This is necessary because the addition or deletion of a column constitutes a change in the underlying database structure as opposed to a change in data values.

Storage Mode Example

For an example of how to use True DBGrid in storage mode (**DataMode** 4) using an **XArray** object as the data source, see [Tutorial 18](#) or examine the UNBOUND4.VBP sample, which can be found in the TUTORIAL\UNBOUND4 subdirectory of the True DBGrid installation directory.

Application Mode

True DBGrid supports a cell-based unbound mode, or *application mode*, that does not rely upon either the Visual Basic Data control or the APEX **XArray** object. Instead, application mode fires events whenever the grid needs to retrieve or update the value of an individual cell. Your application is in total control of the data; no intermediate objects are involved.

To use application mode, set the **DataMode** property of the grid to 3 - Application at design time. In code, write handlers for the **ClassicRead** and **UnboundGetRelativeBookmark** events. If users of your application need the ability to add, update, and delete records, you will have to write handlers for the **ClassicAdd**, **ClassicWrite**, and **ClassicDelete** events as well.

Application mode was modeled after the **Fetch** and **Update** events of APEX's TrueGrid Pro (VBX) product. However, unlike TrueGrid Pro, which identifies rows with absolute row numbers, True DBGrid identifies rows with bookmarks. This subtle change provides uniformity across data access modes, and ensures that all of the grid's bookmark-related properties, methods, and events (**Bookmark**, **FirstRow**, **GetBookmark**, **FetchCellStyle**, and others) work the same in application mode as in any other **DataMode**, either bound or unbound.

{button ,JI(`,`When_to_Use_Application_Mode')} When to Use Application Mode
{button ,JI(`,`How_Application_Mode_Works')} How Application Mode Works
{button ,JI(`,`Application_Mode_Bookmarks')} Application Mode Bookmarks
{button ,JI(`,`Application_Mode_Events')} Application Mode Events
{button ,JI(`,`Application_Mode_Programming_Considerations')} Application Mode Programming Considerations
{button ,JI(`,`Application_Mode_Example')} Application Mode Example

When to Use Application Mode

If you need to display and manipulate two-dimensional array data, either one of the following **DataMode** settings is ideally suited to the task:

- 3 - Application, which fires events on a cell-by-cell basis,
- 4 - Storage, which communicates directly with an **XArray** object.

The choice of which one to use depends upon the target operating system. If you need to deliver both 16- and 32-bit versions of your programs, application mode is recommended, since storage mode is only available in the 32-bit versions of True DBGrid. If you do not need to support 16-bit platforms, storage mode is recommended, since it is easier to use.

If you are working with a database instead of an array, the choice comes down to efficiency versus ease of implementation. If you are concerned about efficiency, and would like to minimize the number of events that fire, you should consider using **DataMode 2 - Unbound Extended**. Mode 2 is also recommended when using database APIs that support multiple-row fetches, such as ODBC.

If you are migrating from DBGrid and find **DataMode 1 - Unbound** difficult to use, you should consider switching to application mode, as it provides the same benefits, but is much easier to implement.

If you are familiar with the **Fetch** and **Update** callback events of TrueGrid Pro (TRUEGRID.VBX), then application mode is recommended, as the style of coding is very similar. In fact, the "classic" events are so named because they were patterned after the callback mode events of TrueGrid Pro.

How Application Mode Works

When True DBGrid runs in application mode, it is not connected to a data control. Instead, your application must supply and maintain the data, while the grid handles all user interaction and data display. For example, when a user covers the grid with another window, then uncovers it later, the grid is completely responsible for repainting the exposed area. Your application does not need to deal with any low-level display operations.

With the grid in control of low-level display, you need to concentrate solely on maintaining your data. The grid fires the **ClassicRead** event as needed to determine the value of individual cells. It is up to your application to provide the requested value on demand. Similarly, when the user repositions the scroll box, the grid may need to determine the bookmark of a row that has yet to be displayed. In this case, it fires the **UnboundGetRelativeBookmark** event, and your application needs to respond accordingly.

In this respect, programming True DBGrid in application mode is very similar to writing the event-handling code for a Visual Basic form. You cannot predict when the user will click a button or select an item from a combo box, so your application must be prepared to handle these events at all times. Similarly, you cannot predict when the grid will request the value of a particular cell, or provide a new value to be written to the underlying data source. Therefore, the code that handles application mode events such as **ClassicRead** and **UnboundGetRelativeBookmark** should be written so that it performs as little work as possible.

The grid generally limits its data requests to visible cells, although it may also cache other rows in anticipation of paging and scrolling operations. You cannot predict when the grid will ask for data, nor can you assume that data will be requested in any particular order. Furthermore, since the grid does not permanently store the data, data that has been requested once from your application may be requested again.

Compare this event-driven approach with the storage mode used by the intrinsic ListBox control of Visual Basic, which is populated by repeated calls to its **AddItem** method at run time. Although this storage mode is convenient for small datasets, it is neither adequate nor efficient when there is a large volume of data or when the data source changes dynamically.

When running in application mode, True DBGrid translates user interactions into events that enable you to keep your data source synchronized. For example, when the user updates a cell, then attempts to move to another row, the grid fires the **ClassicWrite** event. If the cell was modified as part of a pending AddNew operation, the grid fires the **ClassicAdd** event instead. If the user deletes an entire row through the grid's user interface, your application receives notification through the **ClassicDelete** event.

Conversely, if your application code manipulates the data source directly, you need to tell the grid to update its display by using either the **Refresh** or **ReBind** method.

To summarize, True DBGrid's application mode is a useful tool for dealing with dynamic data. Since it has no inherent storage capability, the grid handles frequently changing data fluidly and easily. A common use of application mode is to provide an interface for viewing and updating the contents of a Visual Basic array. Application mode can also be used with proprietary database formats not supported by the Visual Basic Data control.

Application Mode Bookmarks

In application mode, a bookmark is a variant supplied by your application and used by the grid as a means of uniquely identifying a row of data to be displayed or modified.

Just as your application provides, stores, and maintains the data for the unbound grid, you must deal similarly with the bookmarks. The bookmarks themselves must be supplied by your code in the **UnboundGetRelativeBookmark**, **ClassicRead**, and **ClassicAdd** events as variant data. You are free to use whatever you choose for the purpose of identifying a row, but keep in mind that the bookmarks **must** be unique for each row. In general, you will also want to be able to search for the associated record quickly when given a bookmark. Three common examples of what to use for bookmarks are:

1. If you use the unbound grid with a proprietary database, you can use the values of a unique key field as bookmarks. That way, when given a bookmark, you can search and retrieve the associated record quickly.
2. If the database you use supports unique row IDs or record numbers, these can be conveniently used as bookmarks.
3. If you use the grid to display an array, the array's row index is an obvious choice for bookmarks.

Bookmarks cannot exceed 255 characters in length. Since Visual Basic 5.0 uses 2 bytes per character, this means that string bookmarks cannot exceed 127 characters.

True DBGrid's application mode supports string, integral, and floating point bookmarks. All other data types must be converted to strings before they are passed to the grid as variant bookmarks.

Since True DBGrid and Visual Basic differ in their treatment of bookmarks, some restrictions apply when manipulating them in code, as discussed in the following sections.

{button ,JI(`,`Bookmarks_in_True_DBGrid')} Bookmarks in True DBGrid

{button ,JI(`,`Bookmarks_in_Visual_Basic')} Bookmarks in Visual Basic

Bookmarks in True DBGrid

True DBGrid treats bookmarks as packets of binary information that cannot be interpreted. To the grid, a bookmark is a piece of data containing a specific number of bytes (ASCII codes) in a specific order. Thus, pieces of different lengths, or pieces with different bytes, are considered different bookmarks. For example, if the grid were to compare the following string bookmarks:

```
bm1 = "1"  
bm2 = " 1"  
bm3 = "01"
```

it would consider each bookmark to be different from the others, although they could be considered equivalent numerically. Similarly, a 2-byte integer and a 4-byte integer would also be considered different, even if both contained the same numeric value.

Bookmarks in Visual Basic

Visual Basic, on the other hand, treats bookmarks as true variants. That is, they are quantities that can be converted from one form to another without loss of equality, unless they are both in the form of a string. In addition, bookmarks are often passed in Visual Basic as byte arrays, both by the grid and by the Data control.

In Visual Basic, two bookmarks should not be compared for equality unless they are first converted to strings. This rule holds true regardless of whether the bookmark comes from a grid (bound or unbound) or from a Data control.

Another important consideration regarding bookmarks is their length. You should take care to ensure that all bookmarks in your application are created in the same way. For example, the Visual Basic functions **Format\$** and **Str\$** do **not** generate the same string, even if they are passed the same numeric value. The **Str\$** function always generates a *leading space* character for the sign of the numeric value, while **Format\$** does not:

```
Str$(1) = " 1"  
Format$(1) = "1"
```

Remember that since these strings are of different length, they constitute different bookmarks.

To avoid difficulties of this nature, we suggest writing a single Visual Basic function, like the **MakeBookmark** function used in the unbound tutorial projects, and use it consistently whenever a bookmark must be generated.

Application Mode Events

In application mode, there are five events that may fire, depending upon end-user permission settings and run-time interactions. You must write handlers for these two events:

UnboundGetRelativeBookmark Fired when the control needs to retrieve a bookmark.

ClassicRead Fired when the control requires unbound data for display.

The following three events are optional:

ClassicWrite Fired when an unbound row needs to be modified. Required if **AllowUpdate** is True.

ClassicAdd Fired when a new row is added to the unbound dataset. Required if **AllowAddNew** is True.

ClassicDelete Fired when an unbound row needs to be deleted. Required if **AllowDelete** is True.

{button ,JI(`,`Handling_the_UnboundGetRelativeBookmark_event_in_mode_3')}` Handling the UnboundGetRelativeBookmark event in mode 3

{button ,JI(`,`Handling_the_ClassicRead_event_in_mode_3')}` Handling the ClassicRead event in mode 3

{button ,JI(`,`Handling_the_ClassicWrite_event_in_mode_3')}` Handling the ClassicWrite event in mode 3

{button ,JI(`,`Handling_the_ClassicAdd_event_in_mode_3')}` Handling the ClassicAdd event in mode 3

{button ,JI(`,`Handling_the_ClassicDelete_event_in_mode_3')}` Handling the ClassicDelete event in mode 3

Handling the **UnboundGetRelativeBookmark** event in mode 3

You must always provide a handler for the **UnboundGetRelativeBookmark** event in application mode. True DBGrid fires this event whenever it needs to determine the bookmark that identifies a row given a starting bookmark and a long integer offset. The starting bookmark may be Null, which denotes BOF if the offset is positive, EOF if the offset is negative. The offset may be positive to denote forward movement, or negative to denote backward movement. The syntax for this event is as follows:

```
Private Sub TDBGrid1_UnboundGetRelativeBookmark( _  
    StartLocation As Variant, ByVal Offset As Long, _  
    NewLocation As Variant, ApproximatePosition As Long)
```

StartLocation is a bookmark which, together with Offset, specifies the row to be returned in NewLocation.

Offset specifies the relative position (from StartLocation) of the row to be returned in NewLocation. A positive number indicates a forward relative position while a negative number indicates a backward relative position.

NewLocation is a variable which receives the bookmark of the row which is specified by StartLocation plus Offset. If the row specified is beyond the first or the last row (or beyond BOF or EOF), then NewLocation should be set to Null.

ApproximatePosition is a variable which receives the ordinal position of NewLocation. Setting this variable will enhance the ability of the grid to display its vertical scroll bar accurately. If the exact ordinal position of NewLocation is not known, you can set it to a reasonable, approximate value, or just ignore it altogether.

Before returning from the **UnboundGetRelativeBookmark** event, you must set NewLocation to a valid bookmark. For example, if Offset is 1 (or -1), then you must return in NewLocation the bookmark of the row that follows (or precedes) StartLocation. However, if the requested row is beyond the first or last row, then you should return Null in NewLocation to inform the grid of BOF/EOF conditions.

Similarly, a StartLocation of Null indicates a request for a row from BOF or EOF. For example, if StartLocation is Null and Offset is 2 (or -2), then you must return in NewLocation the bookmark of the second (or second to last) row. The following code template provides the basis for a typical implementation of the **UnboundGetRelativeBookmark** event:

```
If IsNull(StartLocation) Then  
    If Offset < 0 Then  
        ' StartLocation indicates EOF, because the grid is  
        ' requesting data in rows prior to the StartLocation,  
        ' and prior rows only exist for EOF.  
        ' There are no rows prior to BOF.  
    Else  
        ' StartLocation indicates BOF, because the grid is  
        ' requesting data in rows after the StartLocation,  
        ' and rows after only exist for BOF.  
        ' There are no rows after EOF.  
    End If  
Else  
    ' StartLocation is an actual bookmark passed to the grid  
    ' via one of the unbound events. It is up to the VB  
    ' programmer to ensure the bookmark is valid, and to take  
    ' the appropriate action if it is not.  
End If
```

The **UnboundGetRelativeBookmark** event is also used to improve performance in **DataMode 1 - Unbound**. For more information, see [Unbound Mode](#).

Handling the ClassicRead event in mode 3

In application mode, the grid uses the **UnboundGetRelativeBookmark** event to determine the bookmark of the row it is about to display. To determine the contents of an individual cell, the grid passes a known bookmark and a column index to the **ClassicRead** event, which has the following syntax:

```
Private Sub TDBGrid1_ClassicRead(Bookmark As Variant, _  
    ByVal Col As Integer, Value As Variant)
```

Bookmark is a bookmark which specifies the row of the desired data.

Col is a value which specifies the column index of data to be retrieved.

Value is a variable used to return the data corresponding to the cell location identified by **Bookmark** and **Col**.

Before returning from this event, you must set **Value** to the actual data, or else the grid will display an empty cell. Note that **Value** need not contain a string. It can also hold numeric, boolean, or date data, and the grid will convert it to a string automatically, just as it does with such data types in bound mode.

Handling the ClassicWrite event in mode 3

If the **AllowUpdate** property of the grid is True, and the user has edited data in one or more cells of the current row, then moving to another row will trigger an update. The grid will then fire the **ClassicWrite** event once for each updated cell, which enables you to apply the changed value to the underlying data stored and maintained by your application. The syntax of this event is as follows:

```
Private Sub TDBGrid1_ClassicWrite(Bookmark as Variant, _  
    ByVal Col as Integer, Value As Variant)
```

Bookmark is a bookmark which specifies the row that needs to be updated.

Col is a value which specifies the column index of data that has been modified.

Value is the data that corresponds to the cell location identified by **Bookmark** and **Col**.

Before returning from this event, you must update the underlying data source with **Value**. If, for some reason, the update cannot be completed, you can set the **Bookmark** argument to Null in order to force the grid to maintain the previous value of the current cell.

Handling the **ClassicAdd** event in mode 3

If the **AllowAddNew** and **AllowUpdate** properties of the grid are True, then users can add new records to the grid, and you must implement the **ClassicAdd** event. The syntax of this event is as follows:

```
Private Sub TDBGrid1_ClassicAdd(NewRowBookmark As Variant, _  
    ByVal Col As Integer, Value As Variant)
```

NewRowBookmark is a variable which receives the bookmark for the newly added row. Initially, the **NewRowBookmark** argument is Null. However, before returning from this event, you must set it to a bookmark that uniquely identifies the newly added row. If you do not set the value of **NewRowBookmark**, the add operation will fail and the grid will not allow its current row to change.

Col is a value which specifies the column index of data that has been added.

Value is the data that corresponds to the newly added cell.

Before returning from this event, you must update the underlying data source with **Value** and set the **NewRowBookmark** argument to the bookmark of the newly added row. If, for some reason, the add cannot be completed, you can set **NewRowBookmark** to Null in order to force the grid to cancel the AddNew operation initiated by the user.

Note that the **ClassicAdd** event will fire once for each modified cell in the new row. This does not pose a problem if the underlying data source permits you to add blank records. However, if this is not the case, and you are using unique key field values as bookmarks, you cannot assume that the event will fire for the key field column first. Since you must still supply a valid bookmark (even if you don't know what it is yet), you can set **NewRowBookmark** to a special key that will never refer to a real record. Then, when the **ClassicAdd** event fires for the key field column (and subsequent columns), you can return the actual bookmark. The grid will use the last bookmark it receives and discard the special key that you supplied earlier.

Handling the **ClassicDelete** event in mode 3

If the **AllowDelete** property of the grid is True, then the user can delete rows from the grid, and you must implement the **ClassicDelete** event:

```
Private Sub TDBGrid1_ClassicDelete(Bookmark As Variant)
```

When fired, the **Bookmark** argument specifies the row being deleted. If the deletion fails, you should set the **Bookmark** argument to Null before returning from the event. Note that following execution of the **ClassicDelete** event, the grid is automatically refreshed.

Application Mode Programming Considerations

When the user updates, adds, or deletes data through the grid's user interface, the grid fires the **ClassicWrite**, **ClassicAdd**, or **ClassicDelete** event so that your application can take the appropriate action. Conversely, when you modify the underlying data source in code, you need to notify the grid so that it can update its display to synchronize with your data.

{button ,JI(`,`Refreshing_the_display_in_mode_3')} Refreshing the display in mode 3

{button ,JI(`,`Reinitializing_the_grid_in_mode_3')} Reinitializing the grid in mode 3

{button ,JI(`,`Updating_and_deleting_rows_in_mode_3')} Updating and deleting rows in mode 3

Refreshing the display in mode 3

You can refresh the display with the grid's **Refresh** and **ReBind** methods, which work in the same manner as documented for bound mode. When these methods are called, the grid will discard its current display and call the **ClassicRead** event to retrieve new data.

Please note that during a **ReBind** operation, the grid attempts to maintain the current record position. Therefore, if the current record does not exist after the **ReBind** operation, it is up to you to position the grid to a valid record afterwards. Or, you can avoid potential problems by reinitializing the grid before performing the **ReBind**.

Reinitializing the grid in mode 3

Setting the grid's **Bookmark** property equal to Null before issuing the grid's **ReBind** or **Refresh** method will cause the **UnboundGetRelativeBookmark** event to fire with a Null StartLocation just as if the grid were first being displayed:

```
TDBGrid1.Bookmark = Null  
TDBGrid1.ReBind
```

Updating and deleting rows in mode 3

In application mode, or in any other mode, you can update a row in code and force the **ClassicWrite** (or **ClassicAdd**) event to fire by applying the grid's **Update** method:

```
TDBGrid1.Update
```

Similarly, you can delete a row in code and force the **ClassicDelete** event to fire by applying the grid's **Delete** method:

```
TDBGrid1.Delete
```

Application Mode Example

For an example of how to use True DBGrid in application mode (**DataMode** 3) using a Visual Basic array as the data source, see [Tutorial 17](#) or examine the UNBOUND3.VBP sample, which can be found in the TUTORIAL\UNBOUND3 subdirectory of the True DBGrid installation directory.

Unbound Mode

True DBGrid supports a row-based unbound mode that does not rely upon either the Visual Basic Data control or the APEX **XArray** object. Instead, unbound mode events fire whenever the grid needs to retrieve a group of adjacent rows or update, add, or delete an individual row. Unlike application mode, an intermediate **RowBuffer** object serves as the liaison between the grid and your data source.

There are actually two row-based unbound modes: **DataMode 1** - Unbound, is the original unbound mode of DBGrid; **DataMode 2** - Unbound Extended, uses a slightly different event syntax that simplifies coding and is more efficient. Therefore, if you are writing a new application, you should use mode 2. Mode 1 is included for backward compatibility with DBGrid.

To use the newer row-based unbound mode, set the **DataMode** property of the grid to 2 - Unbound Extended at design time. In code, write a handler for the **UnboundReadDataEx** event. If users of your application need the ability to add, update, and delete records, you will have to write handlers for the **UnboundAddData**, **UnboundWriteData**, and **UnboundDeleteRow** events as well.

If you are converting an application that uses **DataMode 1**, you can improve its performance dramatically by writing a handler for the **UnboundGetRelativeBookmark** event.

Regardless of whether you use mode 1 or 2, all of the grid's bookmark-related properties, methods, and events (**Bookmark**, **FirstRow**, **GetBookmark**, **FetchCellStyle**, and others) work the same in the row-based unbound modes as they do in any other **DataMode**, either bound or unbound.

{button ,JI(`,`When_to_Use_Unbound_Mode')} When to Use Unbound Mode
{button ,JI(`,`How_Unbound_Mode_Works')} How Unbound Mode Works
{button ,JI(`,`Unbound_Mode_Bookmarks')} Unbound Mode Bookmarks
{button ,JI(`,`Using_the_RowBuffer_Object')} Using the RowBuffer Object
{button ,JI(`,`Unbound_Mode_Events')} Unbound Mode Events
{button ,JI(`,`Unbound_Mode_Programming_Considerations')} Unbound Mode Programming Considerations
{button ,JI(`,`Unbound_Mode_Examples')} Unbound Mode Examples

When to Use Unbound Mode

The row-based unbound modes were designed to provide a workable interface between database APIs and the grid. Therefore, if you simply need to display and manipulate two-dimensional array data, the row-based unbound modes can be cumbersome to work with, and either one of the following **DataMode** settings would be a better choice:

- 3 - Application, which fires events on a cell-by-cell basis,
- 4 - Storage, which communicates directly with an **XArray** object.

Modes 3 and 4 are specifically intended for an array-based data source, whereas the row-based modes 1 and 2 are more generalized and will work well with any data source.

If you are working with a database instead of an array, the choice comes down to efficiency versus ease of implementation. If you are concerned about efficiency, and would like to minimize the number of events that fire, you should consider using **DataMode 2 - Unbound Extended**. Mode 2 is also recommended when using database APIs that support multiple-row fetches, such as ODBC. You can also use mode 2 to bypass the overhead associated with an external bound control. For example, instead of binding to a Remote Data Control (RDC), you can use the Remote Data Objects (RDO) in unbound mode 2 to populate the grid with data from an **rdoResultSet** object.

Mode 3 is easier to implement than mode 2, but it can be less efficient because events fire on a per-cell rather than a per-row basis. If speed and efficiency are your primary concerns, mode 2 is preferred over mode 3.

If your application uses an unbound DBGrid, you should consider switching to mode 2, as it is more efficient. However, if you do not want to revise existing **UnboundReadData** handlers, you can achieve similar performance improvements in mode 1 by implementing the optional **UnboundGetRelativeBookmark** event, which the grid fires whenever it needs to determine the bookmark of a row given a starting position and an integral offset. Even if you do not implement this event, your existing application will continue to function properly.

If you are familiar with the **Fetch** and **Update** callback events of TrueGrid Pro (TRUEGRID.VBX), then mode 3 is recommended, as the style of coding is very similar. In fact, the "classic" events of application mode are so named because they were patterned after the callback mode events of TrueGrid Pro.

How Unbound Mode Works

When True DBGrid runs in unbound mode, it is not connected to a data control. Instead, your application must supply and maintain the data, while the grid handles all user interaction and data display. For example, when a user covers the grid with another window, then uncovers it later, the grid is completely responsible for repainting the exposed area. Your application does not need to deal with any low-level display operations.

With the grid in control of low-level display, you need to concentrate solely on maintaining your data. In **DataMode 1 - Unbound**, the grid fires the **UnboundReadData** event whenever it needs to fetch a row of data. It is up to your application to fill the **RowBuffer** object, passed as an event parameter, with the requested values on demand. Similarly, when the user repositions the scroll box, the grid may need to determine the bookmark of a row that has yet to be displayed. In this case, it fires the **UnboundGetRelativeBookmark** event, and your application needs to respond accordingly.

In **DataMode 2 - Unbound Extended**, the grid fires the **UnboundReadDataEx** event, which combines the functionality of **UnboundReadData** and **UnboundGetRelativeBookmark**. When the grid first loads, it fires the **UnboundReadDataEx** event to retrieve the bookmark (but not the data) for the first row. If your event handler provides a bookmark, the grid fires **UnboundReadDataEx** again, this time to fetch the actual data in a block of ten rows. This process continues until the grid has enough data to fill its display or your event handler informs the grid that the end of the dataset has been reached. As the user scrolls through the grid, the **UnboundReadDataEx** event is fired as needed to obtain bookmarks and data.

In this respect, programming True DBGrid in unbound mode is very similar to writing the event-handling code for a Visual Basic form. You cannot predict when the user will click a button or select an item from a combo box, so your application must be prepared to handle these events at all times. Similarly, you cannot predict when the grid will request a particular data row, or provide a new value to be written to the underlying data source. Therefore, the code that handles unbound mode events such as **UnboundReadData** and **UnboundWriteData** should be written so that it performs as little work as possible.

The grid generally limits its data requests to visible cells, although it may also cache other rows in anticipation of paging and scrolling operations. You cannot predict when the grid will ask for data, nor can you assume that data will be requested in any particular order. Furthermore, since the grid does not permanently store the data, data that has been requested once from your application may be requested again. Thus, you must provide and maintain your own data storage, as the grid will not do this for you.

Compare this event-driven approach with the storage mode used by the intrinsic ListBox control of Visual Basic, which is populated by repeated calls to its **AddItem** method at run time. Although this storage mode is convenient for small datasets, it is neither adequate nor efficient when there is a large volume of data or when the data source changes dynamically.

When running in unbound mode, True DBGrid translates user interactions into events that enable you to keep your data source synchronized. For example, when the user updates a cell, then attempts to move to another row, the grid fires the **UnboundWriteData** event. If the cell was modified as part of a pending AddNew operation, the grid fires the **UnboundAddData** event instead. If the user deletes an entire row through the grid's user interface, your application receives notification through the **UnboundDeleteRow** event.

Conversely, if your application code manipulates the data source directly, the grid will not know that the underlying data has changed, so you need to tell the grid to update its display by using either the **Refresh** or **ReBind** method.

To summarize, True DBGrid's unbound mode is a useful tool for communicating with third-party database APIs or avoiding the overhead associated with bound data controls. Although it is not as easy to implement as application mode, unbound mode can provide significant gains in performance, especially when there are many columns.

Unbound Mode Bookmarks

In unbound mode, as in application mode, a bookmark is a variant supplied by your application and used by the grid as a means of uniquely identifying a row of data to be displayed or modified.

Just as your application must provide, store, and maintain the data for the unbound grid, you must deal similarly with the bookmarks. The bookmarks themselves must be supplied by your code as variant data in the following events:

- **UnboundGetRelativeBookmark**, **UnboundReadData**, and **UnboundAddData** (for **DataMode 1** - Unbound)
- **UnboundReadDataEx** and **UnboundAddData** (for **DataMode 2** - Unbound Extended)

You are free to use whatever you choose for the purpose of identifying a row, but keep in mind that the bookmarks **must** be unique for each row. In general, you will also want to be able to search for the associated record quickly when given a bookmark. That is, when the grid gives you a bookmark, asking for information about a particular row in your dataset, you should be able to locate the row the grid is asking for quickly. An important concept to remember is that whatever you supply to the grid as a bookmark for a particular row, that is how the grid will refer to that row later on. Three common examples of what to use for bookmarks are:

1. If you use the unbound grid with a proprietary database, you can use the values of a unique key field as bookmarks. That way, when given a bookmark, you can search and retrieve the associated record quickly.
2. If the database you use supports unique row IDs or record numbers, these can be conveniently used as bookmarks.
3. If you use the grid to display an array, the array's row index is an obvious choice for bookmarks.

Bookmarks cannot exceed 255 characters in length. Since Visual Basic 5.0 uses 2 bytes per character, this means that string bookmarks cannot exceed 127 characters.

True DBGrid's unbound modes support string, integral, and floating point bookmarks. All other data types must be converted to strings before they are passed to the grid as variant bookmarks.

Since True DBGrid and Visual Basic differ in their treatment of bookmarks, some restrictions apply when manipulating them in code. For more information, see [Application Mode Bookmarks](#).

Using the RowBuffer Object

The **RowBuffer** is a programmable object used to exchange data between the grid and your data source via the unbound grid events. The **RowBuffer** object is passed into the unbound grid event handlers as an argument. In fact, the **RowBuffer** object can only exist within the scope of the unbound events; you cannot create a new one in code as you would a **Column** or **Split** object. Here is a thumbnail sketch of the properties of the **RowBuffer** object:

RowCount property

RowCount is a long integer that specifies the maximum number of rows that can be processed in an unbound event (read, write, or add). If the value of this property exceeds the number of rows that can be processed, such as when an end-of-file condition is detected, then your event handling code should change this property to reflect the actual number of rows processed.

```
RowBuffer.RowCount = Long
```

ColumnCount property

ColumnCount is an integer that specifies the number of columns that the unbound event should process. This property is read-only, and no attempt should be made to change it. The unbound event should process all columns requested.

```
Integer = RowBuffer.ColumnCount
```

ColumnName property

ColumnName is a string array that specifies the name of the grid column corresponding to a row buffer index. This property is read-only.

```
String = RowBuffer.ColumnName(ColIndex)
' where ColIndex = 0 to ColumnCount - 1
```

Bookmark property

Bookmark is a variant array used to specify unique row bookmarks when the **RowBuffer** is used to fetch data during an unbound read event.

```
RowBuffer.Bookmark(RowIndex) = Variant
' where RowIndex = 0 to RowCount - 1
```

Value property

Value is a variant array used to specify the data value associated with a **RowBuffer** row and column.

```
RowBuffer.Value(RowIndex, ColIndex) = Variant
' where RowIndex = 0 to RowCount - 1
' and ColIndex = 0 to ColumnCount - 1
```

ColumnIndex property

ColumnIndex is a variant array used to specify a grid column index associated with a **RowBuffer** row and column. This property is read-only. You can use it in the **UnboundReadDataEx** event to identify which data columns are being requested.

```
Col = RowBuffer.ColumnIndex(RowIndex, ColIndex)
' where RowIndex = 0 to RowCount - 1
' and ColIndex = 0 to ColumnCount - 1
```

Unbound Mode Events

In **DataMode 1** - Unbound, the grid fires one event when it needs to determine a relative bookmark, another when it needs to fetch data rows. The first event is optional and can be implemented to improve performance; the second event is mandatory:

UnboundGetRelativeBookmark Fired when the control needs to retrieve a bookmark.

UnboundReadData Fired when the control requires unbound data for display.

In **DataMode 2** - Unbound Extended, the grid fires a single event to acquire both data and relative bookmarks. This event is mandatory:

UnboundReadDataEx Fired when the control needs to retrieve a bookmark or requires unbound data for display.

In modes 1 and 2, the following three events are optional, depending upon end-user permission settings:

UnboundWriteData Fired when the current row of the grid has been modified and the user commits the changes by leaving the row. This event is your notification that the user wants to modify a row in the unbound dataset. It is also fired when the grid's **Update** method is executed in code. Required if **AllowUpdate** is True.

UnboundAddData Fired when the user types data into the grid's AddNew row and commits the changes by leaving the row. This event is your notification that the user wants to add a new row to the unbound dataset. Required if **AllowAddNew** is True.

UnboundDeleteRow Fired when the user deletes the current grid row by clicking its record selector and pressing the DEL key. This event is your notification that the user wants to delete a row from the unbound dataset. It is also fired when the grid's **Delete** method is executed in code. Required if **AllowDelete** is True.

{button ,Jl(`,`Handling_the_UnboundReadData_event_in_mode_1')} Handling the UnboundReadData event in mode 1

{button ,Jl(`,`Handling_the_UnboundGetRelativeBookmark_event_in_mode_1')} Handling the UnboundGetRelativeBookmark event in mode 1

{button ,Jl(`,`Handling_the_UnboundReadDataEx_event_in_mode_2')} Handling the UnboundReadDataEx event in mode 2

{button ,Jl(`,`UnboundReadDataEx_event_examples')} UnboundReadDataEx event examples

{button ,Jl(`,`Handling_the_UnboundWriteData_event_in_modes_1_and_2')} Handling the UnboundWriteData event in modes 1 and 2

{button ,Jl(`,`Handling_the_UnboundAddData_event_in_modes_1_and_2')} Handling the UnboundAddData event in modes 1 and 2

{button ,Jl(`,`Handling_the_UnboundDeleteRow_event_in_modes_1_and_2')} Handling the UnboundDeleteRow event in modes 1 and 2

Handling the UnboundReadData event in mode 1

The **UnboundReadData** event is fired only if the **DataMode** property is set to 1 - Unbound. This event is retained only for backward compatibility with DBGrid and earlier versions of True DBGrid. If you are writing a new unbound mode application, **DataMode 2 - Unbound Extended** is recommended, since it is more efficient and easier to use.

The **UnboundReadData** event is fired whenever the grid requires data for display. Its syntax is as follows:

```
Private Sub TDBGrid1_UnboundReadData( _  
    ByVal RowBuf As RowBuffer, _  
    StartLocation As Variant, _  
    ReadPriorRows As Boolean)
```

When this event is fired, the properties of the **RowBuf** argument are set as follows:

- **RowCount** specifies the number of rows of data requested from your data source.
- **ColumnCount** specifies the number of columns of data requested from your data source.
- The **ColumnName** array contains the names of the grid columns corresponding to the columns in **RowBuf**.
- The **Bookmark** and **Value** arrays contain all Null values.

StartLocation is a bookmark that specifies the row *before* or *after* the desired data, depending on the value of the **ReadPriorRows** argument.

ReadPriorRows indicates the direction in which the grid is requesting the data. If False, you should provide data in the forward direction, starting with the row immediately after the row specified by **StartLocation**. If True, you should provide data in the backward direction, starting with the row immediately before the row specified by **StartLocation**.

Before returning from the **UnboundReadData** event, you are expected to fill the **Bookmark** property array with unique row identifiers, and the **Value** property array with the actual data:

```
DimRowIndex As Long  
Dim ColIndex As Integer  
  
With RowBuf  
    ForRowIndex = 0 To .RowCount - 1  
        .Bookmark(RowIndex) = Variant Bookmark  
        For ColIndex = 0 To .ColumnCount - 1  
            .Value(RowIndex, ColIndex) = Variant Data  
        Next ColIndex  
    Next RowIndex  
End With
```

For example, if the grid specifies a **StartLocation** bookmark indicating the 46th row, the **ReadPriorRows** argument is False, and the row buffer's **RowCount** property is 10, then the grid is asking for the records **following** the 46th row, and your **UnboundReadData** event handler should populate the row buffer's **Value** array as follows:

```
RowBuf.Value(0, ColIndex) = Data for row 47  
RowBuf.Value(1, ColIndex) = Data for row 48  
RowBuf.Value(2, ColIndex) = Data for row 49  
...  
RowBuf.Value(9, ColIndex) = Data for row 56
```

However, if **ReadPriorRows** is False, then the grid is asking for the records **preceding** the 46th row, and a different set of values must be returned:

```

RowBuf.Value(0, ColIndex) = Data for row 45
RowBuf.Value(1, ColIndex) = Data for row 44
RowBuf.Value(2, ColIndex) = Data for row 43
...
RowBuf.Value(9, ColIndex) = Data for row 36

```

If you reach the beginning or end of the data, and have fewer than **RowCount** rows to provide, then you should fill the row buffer with the data you can provide, and change the **RowCount** property to the actual number of rows provided, which may be zero.

For example, if your dataset contains 50 records, the grid specifies a **StartLocation** bookmark indicating the 46th row, the **ReadPriorRows** argument is False, and the row buffer's **RowCount** property is 10, then your **UnboundReadData** event handler should populate the row buffer's **Value** array as follows:

```

RowBuf.Value(0, ColIndex) = Data for row 47
RowBuf.Value(1, ColIndex) = Data for row 48
RowBuf.Value(2, ColIndex) = Data for row 49
RowBuf.Value(3, ColIndex) = Data for row 50

RowBuf.RowCount = 4      ' Since only 4 rows were processed

```

At first glance, **StartLocation** and **ReadPriorRows** may seem unnecessarily cumbersome. However, they communicate the row boundaries to the grid simply and directly. The grid only caches a portion of the data, and it is with these two arguments that it can navigate from one bookmark to the next.

For example, suppose there are 100 rows of data, the current row is 75, and the grid is asked to move to row 3 using a previously obtained bookmark. The following sequence demonstrates what might happen in this situation:

1. The grid receives a bookmark for row 3. Since the data for this row is not in the grid's cache, the grid requests the data using the **UnboundReadData** event, which is called with the following parameters:

```

RowBuf.RowCount = 10
RowBuf.ColumnCount = Number of columns
StartLocation = Bookmark for row 3
ReadPriorRows = False

```

2. The event code responds as follows:

```

RowBuf.Value(0, ColIndex) = Data for row 4
RowBuf.Value(1, ColIndex) = Data for row 5
...
RowBuf.Value(9, ColIndex) = Data for row 13

RowBuf.RowCount = 10      ' Since all 10 rows were processed

```

3. The **UnboundReadData** event is called again, this time with:

```

RowBuf.RowCount = 10
RowBuf.ColumnCount = Number of columns
StartLocation = Bookmark for row 4
ReadPriorRows = True

```

4. The event code responds as follows:

```

RowBuf.Value(0, ColIndex) = Data for row 3
RowBuf.Value(1, ColIndex) = Data for row 2
RowBuf.Value(2, ColIndex) = Data for row 1

RowBuf.RowCount = 3      ' Since only 3 rows were processed

```

Note that after fetching row 1, the event code stops setting values since there is no more data available in the indicated direction from the starting bookmark. Also, `RowBuf.RowCount` is set to 3, since only 3 rows could be read before the beginning of the dataset was encountered. At this point, additional **UnboundReadData** events may be fired to obtain the data necessary to complete the display.

The preceding example demonstrates how the same event interface is called upon to handle both BOF and EOF conditions. When one of these special cases is encountered, the event handler simply exits the loop used to fill the row buffer and reports the number of rows actually processed in the **RowCount** property.

A **StartLocation** of Null indicates a request for BOF or EOF. Whether it indicates BOF or EOF depends upon the value of **ReadPriorRows**:

```
If IsNull(StartLocation) Then
  If ReadPriorRows Then
    ' StartLocation indicates EOF, because the grid is
    ' requesting data in rows prior to the StartLocation,
    ' and prior rows only exist for EOF.
    ' There are no rows prior to BOF.
  Else
    ' StartLocation indicates BOF, because the grid is
    ' requesting data in rows after the StartLocation,
    ' and rows after only exist for BOF.
    ' There are no rows after EOF.
  End If
Else
  ' StartLocation is an actual bookmark passed to the grid
  ' in the RowBuffer, an event argument (UnboundAddData), or
  ' the setting of a grid bookmark. You must ensure that
  ' the bookmark is valid, and take the appropriate action
  ' if it is not.
End If
```

NOTE: You cannot make any assumptions about when the grid will request data, or how many times it will request the same data. In short, it is the grid's responsibility to display the data properly, while the task of storing and maintaining the data falls to you. This division of labor frees you from worrying about when or how to display data in the grid.

Handling the **UnboundGetRelativeBookmark** event in mode 1

This event is mandatory when the **DataMode** property is set to 3 - Application, but optional when it is set to 1 - Unbound. It is not used at all when the **DataMode** property is set to 2 - Unbound Extended.

In **DataMode** 1, this event is used in conjunction with the **UnboundReadData** event when the grid needs to obtain positional information about your underlying data. If you are converting an existing project that uses DBGrid or an earlier version of True DBGrid, you can add a handler for this event to dramatically improve the grid's performance. However, if you choose to ignore this event, your project will continue to function properly. The syntax for this event is as follows:

```
Private Sub TDBGrid1_UnboundGetRelativeBookmark( _  
    StartLocation As Variant, _  
    ByVal Offset As Long, _  
    NewLocation As Variant, _  
    ApproximatePosition As Long)
```

For more information on the **UnboundGetRelativeBookmark** event, see [Application Mode](#).

Handling the **UnboundReadDataEx** event in mode 2

The **UnboundReadDataEx** event is used when the **DataMode** property is set to 2 - Unbound Extended, and is fired by the grid whenever it requires one of the following:

- A bookmark for a specific row.
- Data for display.

The syntax of the **UnboundReadDataEx** event is as follows:

```
Private Sub TDBGrid1_UnboundReadDataEx( _  
    ByVal RowBuf As RowBuffer, _  
    StartLocation As Variant, _  
    ByVal Offset As Long, _  
    ApproximatePosition As Long)
```

When this event is fired, the properties of the **RowBuf** argument are set as follows:

- **RowCount** specifies the number of rows of data requested from your data source.
- **ColumnCount** specifies the number of columns of data requested from your data source.
- The **ColumnName** array contains the names of the grid columns corresponding to the columns in **RowBuf**.
- The **ColumnIndex** array contains the indexes of the grid columns corresponding to the columns in **RowBuf**.
- The **Bookmark** and **Value** arrays contain all Null values.

You can examine the **RowCount** and **ColumnCount** properties to determine whether the grid is requesting a bookmark or data. If **RowCount** is 1 and **ColumnCount** is 0, the grid is asking for a bookmark only; if **ColumnCount** is nonzero, the grid is asking for **RowCount** rows of data (and the corresponding bookmarks).

StartLocation is a bookmark which, together with **Offset**, specifies the first row of data to be transferred to **RowBuf**.

Offset specifies the relative position (from **StartLocation**) of the first row of data to be transferred. A positive number indicates a forward relative position; a negative number indicates a backward relative position. Regardless of whether **Offset** is positive or negative, you should always return rows to the grid in the forward direction.

ApproximatePosition is a variable which optionally receives the ordinal position of the first row of data to be transferred. Setting this variable will enhance the ability of the grid to display its vertical scroll bar accurately. If the exact ordinal position of the row is not known, you can set it to a reasonable, approximate value, or just ignore this parameter.

Before returning from the **UnboundReadDataEx** event, you are expected to fill the **Bookmark** array of **RowBuf** with unique row identifiers, and the **Value** array with the actual data, if requested. For example, if **Offset** is 1 (or -1), then you must fill in **RowBuf** starting from the row that follows (or precedes) **StartLocation**:

```
DimRowIndex As Long  
Dim ColIndex As Integer, Col As Integer  
  
With RowBuf  
    ForRowIndex = 0 To .RowCount - 1  
        .Bookmark(RowIndex) = Variant Bookmark  
        For ColIndex = 0 To .ColumnCount - 1  
            Col = .ColumnIndex(RowIndex, ColIndex)  
            .Value(RowIndex, ColIndex) = Variant Data for Col
```

```
Next ColIndex
Next RowIndex
End With
```

Note that there is a subtle difference between this example and the one presented in the earlier discussion of the **UnboundReadData** event of mode 1. When programming the **UnboundReadDataEx** event, you must fill in the **Value** array with column data according to the **ColumnIndex** array of the **RowBuffer** object, since it is possible that the column indexes of the grid and the row buffer no longer match.

The grid generally asks for data according to the number of columns and the order of the columns as displayed on the grid. For example, if your data source has 20 columns, and the grid needs to display the first 5 columns on the screen, then the **UnboundReadDataEx** event will be called with **ColumnCount** equal to 5 and the **ColumnIndex** array equal to (0, 1, 2, 3, 4). However, if the user moves column 4 between column 0 and column 1, then the next **UnboundReadDataEx** event will be called with **ColumnCount** equal to 5 and the **ColumnIndex** property array equal to (0, 4, 1, 2, 3). Therefore, you must account for the new column order, as given by the **ColumnIndex** property, when filling the **Value** array.

Another important distinction between the two row-based unbound modes is that in mode 2, **UnboundReadDataEx** will not fetch data for columns whose **Visible** property is False, and may not fetch data for columns that are not physically displayed or have been scrolled out of view, even if their **Visible** property is True. In mode 1, however, **UnboundReadData** will always fetch data for all columns, even if they are not shown on the screen or have their **Visible** property set to False. This is one of the reasons why mode 2 generally outperforms mode 1.

When the grid first loads, it needs to determine if there is any data to display. It does this by firing the **UnboundReadDataEx** event to retrieve the bookmark (but not the data) for the first row. If your event handler provides a bookmark, the grid fires **UnboundReadDataEx** again, this time to fetch the actual data in a block of ten rows. This process continues until the grid has enough data to fill its display or your event handler informs the grid that the end of the dataset has been reached.

For example, if the grid specifies a **StartLocation** bookmark indicating the 46th row, the **Offset** argument is 3, and the row buffer's **RowCount** property is 10, then your **UnboundReadDataEx** event handler should populate the row buffer's **Value** array as follows:

```
RowBuf.Value(0, ColIndex) = Data for row 49
RowBuf.Value(1, ColIndex) = Data for row 50
RowBuf.Value(2, ColIndex) = Data for row 51
...
RowBuf.Value(9, ColIndex) = Data for row 58
```

However, if **Offset** is -3, a different set of values must be returned:

```
RowBuf.Value(0, ColIndex) = Data for row 43
RowBuf.Value(1, ColIndex) = Data for row 44
RowBuf.Value(2, ColIndex) = Data for row 45
...
RowBuf.Value(9, ColIndex) = Data for row 52
```

Note that you should always populate the **Value** array in the forward direction, regardless of whether **Offset** is positive or negative. This differs from the **UnboundReadData** event of mode 1, in which rows must be returned in reverse order if **ReadPriorRows** is True.

If you reach the beginning or end of the data, and have fewer than **RowCount** rows to provide, then you should fill the row buffer with the data you can provide, and change the **RowCount** property to the actual number of rows provided, which may be zero.

For example, if your dataset contains 50 records, the grid specifies a **StartLocation** bookmark indicating the 46th row, the **Offset** argument is 1, and the row buffer's **RowCount** property is 10, then your **UnboundReadDataEx** event handler should populate the row buffer's **Value** array as follows:

```
RowBuf.Value(0, ColIndex) = Data for row 47
```

```
RowBuf.Value(1, ColIndex) = Data for row 48
RowBuf.Value(2, ColIndex) = Data for row 49
RowBuf.Value(3, ColIndex) = Data for row 50
```

```
RowBuf.RowCount = 4      ' Since only 4 rows were processed
```

Since the value of **RowCount** was changed from 10 to 4, and **Offset** is positive, the grid determines that it has reached the end of your data and stops firing **UnboundReadDataEx** with a positive **Offset**.

When the user scrolls vertically, the grid computes the position of the new topmost visible row and fires **UnboundReadDataEx** to obtain a bookmark for that row. For example, if the user hits the PGUP key, the grid might fire **UnboundReadDataEx** with a **StartLocation** representing row 90 and **Offset** equal to -20. In this case, the grid is effectively asking for a bookmark for row 70. Once the grid has the bookmark it needs, it will fire **UnboundReadDataEx** again to fetch the data to be displayed.

At times, the arguments passed to **UnboundReadDataEx** may seem peculiar. For example, if **StartLocation** specifies row 2 and **Offset** equals -10, the grid is effectively asking for a bookmark for row -8, and you should set **RowCount** to 0 and exit the event. Although it may seem unnecessary for the grid to request the bookmark of a row that does not exist, such behavior is normal, for this is how the grid determines the boundaries of your data source. Also, since the grid is designed to work with multiuser data sources, it is very conservative about boundary conditions. As long as you respond to **UnboundReadDataEx** consistently and correctly, the grid will detect BOF and EOF conditions as fluidly as it does in bound mode.

A **StartLocation** of Null indicates a request for data from BOF or EOF. For example, if **StartLocation** is Null and **Offset** is 2 (or -2), then you should retrieve data starting from the second (or second to last) row:

```
If IsNull(StartLocation) Then
  If Offset < 0 Then
    ' StartLocation indicates EOF, because the grid is
    ' requesting data in rows prior to the StartLocation,
    ' and prior rows only exist for EOF.
    ' There are no rows prior to BOF.
  Else
    ' StartLocation indicates BOF, because the grid is
    ' requesting data in rows after the StartLocation,
    ' and rows after only exist for BOF.
    ' There are no rows after EOF.
  End If
Else
  ' StartLocation is an actual bookmark passed to the grid
  ' in the RowBuffer, an event argument (UnboundAddData), or
  ' the setting of a grid bookmark. You must ensure that
  ' the bookmark is valid, and take the appropriate action
  ' if it is not.
End If
```

NOTE: You cannot make any assumptions about when the grid will request data, or how many times it will request the same data. In short, it is the grid's responsibility to display the data properly, while the task of storing and maintaining the data falls to you. This division of labor frees you from worrying about when or how to display data in the grid.

UnboundReadDataEx event examples

The following examples assume a dataset containing 100 rows, numbered 0 to 99. Thus, when calculating bookmark positions, a negative row number denotes BOF, and a row number greater than or equal to 100 denotes EOF.

Example 1:

```
RowBuf.RowCount = 1
RowBuf.ColumnCount = 0
StartLocation = Bookmark for row 8
Offset = -1
```

In this example, 1 row and 0 columns are being requested, so the event handler must supply a bookmark only. Since Offset is -1, the grid is asking for the row before row 8, and the event handler should respond as follows:

```
RowBuf.Bookmark(0) = Bookmark for row 7
RowBuf.RowCount = 1
ApproximatePosition = 7
```

Example 2:

```
RowBuf.RowCount = 10
RowBuf.ColumnCount = 5
StartLocation = Bookmark for row 80
Offset = 15
```

In this example, the grid is asking for 10 rows of data starting with row 95 (80 + 15). Thus, the grid wants data from rows 95 to 104, in ascending order. However, rows 100 to 104 do not exist in the dataset, so the event handler returns as many rows as it can:

```
RowBuf.Bookmark(0) = Bookmark for row 95
RowBuf.Value(0, ColIndex) = Data for row 95
...
RowBuf.Bookmark(4) = Bookmark for row 99
RowBuf.Value(4, ColIndex) = Data for row 99
RowBuf.RowCount = 5
ApproximatePosition = 95
```

Example 3:

```
RowBuf.RowCount = 10
RowBuf.ColumnCount = 2
StartLocation = Null
Offset = -13
```

In this example, the grid is asking for 10 rows of data. Since StartLocation is Null, and since Offset is negative, the grid wants data starting at 13 rows before EOF. Since the last valid row is 99, row 100 denotes EOF, and the first requested row is 87 (100 - 13). Thus, the grid wants data from rows 87 to 96, in ascending order, and the event handler should respond as follows:

```
RowBuf.Bookmark(0) = Bookmark for row 87
RowBuf.Value(0, ColIndex) = Data for row 87
...
RowBuf.Bookmark(9) = Bookmark for row 96
RowBuf.Value(9, ColIndex) = Data for row 96
RowBuf.RowCount = 10
ApproximatePosition = 87
```

Example 4:

```
RowBuf.RowCount = 1
RowBuf.ColumnCount = 0
StartLocation = Null
Offset = 1
```

In this example, the grid is asking for a bookmark, since 1 row and 0 columns are being requested. Since StartLocation is Null and Offset is positive, the request is relative to BOF. Since Offset is 1, the bookmark requested is that of the first row of the dataset. This example corresponds to the initial firing of **UnboundReadDataEx** when the grid is first loaded, and the event handler should respond as follows:

```
RowBuf.Bookmark(0) = Bookmark for row 0
RowBuf.RowCount = 1
ApproximatePosition = 0
```

Example 5:

```
RowBuf.RowCount = 1
RowBuf.ColumnCount = 0
StartLocation = Bookmark for row 6
Offset = -15
```

In this example, the grid is asking for a bookmark, but the row requested ($6 - 15 = -9$) does not exist, since the first valid data row is 0. In this case, the event handler should respond as follows:

```
RowBuf.RowCount = 0
Exit Sub
```

This is also the correct response when there are no records in the dataset.

Handling the UnboundWriteData event in modes 1 and 2

This event applies to both **DataMode** 1 - Unbound and 2 - Unbound Extended.

If the **AllowUpdate** property of the grid is True, and the user has edited data in one or more cells of the current row, then moving to another row will trigger an update. The grid will then fire the **UnboundWriteData** event, which allows you to use the changed values, passed via a **RowBuffer** object, to update the data you are responsible for storing and maintaining. The syntax of this event is as follows:

```
Private Sub TDBGrid1_UnboundWriteData( _  
    ByVal RowBuf As RowBuffer, WriteLocation As Variant)
```

When this event is fired, the properties of the **RowBuf** argument are set as follows:

- **RowCount** is 1, since only one row can be updated at a time.
- **ColumnCount** specifies the number of columns of data in the **Value** array. This will always reflect the total number of columns in the grid's **Columns** collection and not just those that are visible on the screen. This ensures that data in invisible columns, which may have been modified by other event handlers such as **AfterColUpdate**, are also updated to the underlying data source.
- The **ColumnName** array contains the names of the grid columns corresponding to the columns in **RowBuf**.
- The **Bookmark** array is not used, since **WriteLocation** specifies the row being updated.
- The **Value** array contains a single row of data. Entries which are Null have not been changed. Entries which are not Null reflect the user's changes.

WriteLocation is a bookmark that specifies the *exact* row that needs to be updated. This differs from the **StartLocation** argument in the **UnboundReadData** event, which specifies the row *before* or *after* the desired data, depending on the value of the **ReadPriorRows** argument.

The following code sample demonstrates how to determine which cells were modified by the user:

```
For ColIndex = 0 To RowBuf.ColumnCount - 1  
    If IsNull(RowBuf.Value(0, ColIndex)) Then  
        ' Cell not modified by the user  
    Else  
        ' RowBuf.Value(0, ColIndex) contains updated value  
    End If  
Next ColIndex
```

The **RowCount** property is always set to 1 for this event, but you should change it to 0 if the update cannot be completed, such as when the underlying database reports an error. The contents of the current cell will be maintained if **RowCount** is set to zero.

NOTE: You can force the **UnboundWriteData** event to occur in code with the grid's **Update** method. This technique is particularly valuable when the unbound dataset contains a single row and **AllowAddNew** is False, since there is no way for the user to trigger the update by moving to another row in this case.

Handling the **UnboundAddData** event in modes 1 and 2

This event applies to both **DataMode** 1 - Unbound and 2 - Unbound Extended.

If the **AllowAddNew** and **AllowUpdate** properties of the grid are True, then users can add new records to the grid, and you must implement the **UnboundAddData** event. The syntax of this event is as follows:

```
Private Sub TDBGrid1_UnboundAddData( _  
    ByVal RowBuf As RowBuffer, NewRowBookmark As Variant)
```

As with the **UnboundWriteData** event, data from the new row is passed via a **RowBuffer** object. When this event is fired, the properties of the **RowBuf** argument are set as follows:

- **RowCount** is 1, since only one row can be added at a time.
- **ColumnCount** specifies the number of columns of data in the **Value** array.
- The **ColumnName** array contains the names of the grid columns corresponding to the columns in **RowBuf**.
- The **Bookmark** array is not used, since the new row does not yet have a valid bookmark.
- The **Value** array contains a single row of data. Entries which are Null have not been specified by the user. Entries which are not Null reflect the user's additions.

The **NewRowBookmark** argument is initially Null. However, before returning from this event, you must set it to a bookmark that uniquely identifies the newly added row. If you do not set the value of **NewRowBookmark**, the add operation will fail and the grid will not allow its current row to change.

Handling the **UnboundDeleteRow** event in modes 1 and 2

This event applies to both **DataMode** 1 - Unbound and 2 - Unbound Extended.

If the **AllowDelete** property of the grid is True, then users can delete rows from the grid, and you must implement the **UnboundDeleteRow** event. The syntax of this event is as follows:

```
Private Sub TDBGrid1_UnboundDeleteRow(Bookmark As Variant)
```

This is the simplest of all the unbound grid events. When fired, the **Bookmark** argument specifies the row being deleted. If the deletion fails, you should set the **Bookmark** argument to Null before returning from the event. Note that following execution of the **UnboundDeleteRow** event, the grid is automatically refreshed.

Unbound Mode Programming Considerations

When the user updates, adds, or deletes data through the grid's user interface, the grid fires the **UnboundWriteData**, **UnboundAddData**, or **UnboundDeleteRow** event so that your application can take the appropriate action. Conversely, when you modify the underlying data source in code, you need to notify the grid so that it can update its display to synchronize with your data.

```
{button ,JI(`,`Refreshing_the_display_in_mode_1') } Refreshing the display in mode 1  
{button ,JI(`,`Refreshing_the_display_in_mode_2') } Refreshing the display in mode 2  
{button ,JI(`,`Reinitializing_the_grid_in_modes_1_and_2') } Reinitializing the grid in modes 1 and 2  
{button ,JI(`,`Updating_and_deleting_rows_in_modes_1_and_2') } Updating and deleting rows in modes 1  
and 2  
{button ,JI(`,`Adding_rows_in_modes_1_and_2') } Adding rows in modes 1 and 2  
{button ,JI(`,`Calibrating_the_vertical_scroll_bar_in_modes_1_and_2') } Calibrating the vertical scroll bar in  
modes 1 and 2
```

Refreshing the display in mode 1

In the original DBGrid, the **Refresh** and **ReBind** methods behaved differently in bound and unbound modes. For backward compatibility, these differences were preserved in True DBGrid when the **DataMode** property is set to 1 - Unbound.

When a grid **Refresh** occurs, the grid refetches and redisplay all data by firing the **UnboundReadData** event. After the refresh, the current cell is the first column of the first record, which is displayed at the upper left corner of the grid. Any changed data will be lost.

When a grid **ReBind** occurs, the grid refetches data by firing the **UnboundReadData** event, but it maintains any changed data within the current row. When redisplaying data, the grid attempts to preserve the same current cell and top row, if possible.

Refreshing the display in mode 2

In **DataMode 2** - Unbound Extended, the grid's **Refresh** and **ReBind** methods work in the same manner as documented for bound mode. When these methods are called, the grid will discard its current display and call the **UnboundReadDataEx** event to retrieve new data.

Please note that during a **ReBind** operation, the grid attempts to maintain the current record position. Therefore, if the current record does not exist after the **ReBind** operation, it is up to you to position the grid to a valid record afterwards. Or, you can avoid potential problems by reinitializing the grid before performing the **ReBind**.

Reinitializing the grid in modes 1 and 2

In **DataMode 1** - Unbound, setting the grid's **Bookmark** property to Null before calling the grid's **Refresh** or **ReBind** method will cause the **UnboundGetRelativeBookmark** and **UnboundReadData** events to fire with a **StartLocation** of Null just as if the grid were first being displayed:

```
TDBGrid1.Bookmark = Null  
TDBGrid1.ReBind
```

Similarly, in **DataMode 2** - Unbound Extended, setting the grid's **Bookmark** property to Null before calling the grid's **Refresh** or **ReBind** method will cause the **UnboundReadDataEx** event to fire with a **StartLocation** of Null just as if the grid were first being displayed:

Updating and deleting rows in modes 1 and 2

In unbound mode, you can update a row in code and force the **UnboundWriteData** event to fire by calling the grid's **Update** method:

```
TDBGrid1.Update
```

Similarly, you can delete a row in code and force the **UnboundDeleteRow** event to fire by calling the grid's **Delete** method:

```
TDBGrid1.Delete
```

Adding rows in modes 1 and 2

True DBGrid does not provide an **AddNew** method to complement **Update** and **Delete** since it cannot anticipate the requirements of the underlying data source. However, you can use the **ApproxCount** property and **ReOpen** method to resynchronize the grid after adding one or more rows in code.

The following example assumes that the data source is a two-dimensional array. It allocates storage space for a new row, informs the grid that the total number of rows has changed, then calls the grid's **ReOpen** method to force the grid to refetch data and position to the newly added row:

```
' General declarations
Dim MaxRow As Long
Dim MaxCol As Integer
Dim MyArray() As String

' Assume that MyArray is one-based
MaxRow = MaxRow + 1
ReDim Preserve MyArray(1 To MaxRow, 1 To MaxCol)

' Update the number of rows
TDBGrid1.ApproxCount = MaxRow

' ReOpen the data source and make MaxRow the current row
TDBGrid1.ReOpen MaxRow
```

Calibrating the vertical scroll bar in modes 1 and 2

When the grid displays a vertical scroll bar, the scroll box indicates the ordinal positions of the records being displayed, and users expect to be able to drag the scroll box to quickly locate the desired records.

In order for the scroll box to function properly, both the total number of rows and the ordinal position of the first displayed row must be known, or at least approximated. Unfortunately, one or both of these quantities are often unavailable to the grid, especially in unbound modes.

Since data is supplied and maintained by the application code, the grid does not generally know the total number of rows in the data source. Also, when the grid is instructed to position to a particular record, as in an assignment to its **Bookmark** property, the grid does not generally know the ordinal position of the assigned bookmark, which may have been obtained through a find or seek operation.

Unless you compensate for these unknowns, the vertical scroll bar may behave unpredictably. To avoid this, you need to supply the grid with the total number of rows in the data source (or an approximate total), and the ordinal position of the first displayed row (or an approximate position).

To provide the total row count, you can set the grid's **ApproxCount** property:

```
TDBGrid1.ApproxCount = TotalRows ' Set approximate row count
```

To provide the ordinal position of the first displayed row in **DataMode 1** - Unbound, write a handler for the **UnboundGetRelativeBookmark** event.

To provide the ordinal position of the first displayed row in **DataMode 2** - Unbound Extended, set the **ApproximatePosition** argument within the handler for the **UnboundReadDataEx** event.

Unbound Mode Examples

For an example of how to use True DBGrid in unbound mode (**DataMode** 1) using a Visual Basic array as the data source, see [Tutorial 15](#) or examine the UNBOUND1.VBP sample, which can be found in the TUTORIAL\UNBOUND1 subdirectory of the True DBGrid installation directory.

For an example of how to use True DBGrid in unbound extended mode (**DataMode** 2) using a Visual Basic array as the data source, see [Tutorial 16](#) or examine the UNBOUND2.VBP sample, which can be found in the TUTORIAL\UNBOUND2 subdirectory of the True DBGrid installation directory.

APEX provides additional sample programs that demonstrate how to use the unbound modes of True DBGrid. You can download these programs from the APEX Web site at <http://www.apexsc.com>.

Database Programming Techniques

As [Tutorial 1](#) demonstrates, no code is necessary to create a fully functional database browser and editor using True DBGrid. However, in order to build more sophisticated applications, you can use the techniques outlined in this chapter. Except where noted, these techniques apply to all **DataMode** property settings.

If you haven't already, please read [Understanding Bookmarks](#), as the remainder of this chapter presupposes knowledge of bookmarks.

{button ,JI(`,`Changing_the_Current_Record_Position')}} Changing the Current Record Position
{button ,JI(`,`Accessing_and_Manipulating_Cell_Data')}} Accessing and Manipulating Cell Data
{button ,JI(`,`Validating_Cell_Data')}} Validating Cell Data
{button ,JI(`,`Selecting_and_Highlighting_Records')}} Selecting and Highlighting Records
{button ,JI(`,`Updating_and_Deleting_the_Current_Record')}} Updating and Deleting the Current Record
{button ,JI(`,`Refreshing_the_Display')}} Refreshing the Display
{button ,JI(`,`Coordinating_with_Other_Controls')}} Coordinating with Other Controls
{button ,JI(`,`Handling_Database_Errors')}} Handling Database Errors
{button ,JI(`,`Postponing_Illegal_Operations_in_Grid_Events')}} Postponing Illegal Operations in Grid Events

Changing the Current Record Position

True DBGrid enables you to manipulate the current record position directly in either bound or unbound modes. To do so, you can use the **Bookmark** property or one of the navigation methods.

{button ,JI(`,`Using_the_Bookmark_property')} Using the Bookmark property

{button ,JI(`,`Using_the_navigation_methods')} Using the navigation methods

{button ,JI(`,`Detecting_BOF_and_EOF_conditions')} Detecting BOF and EOF conditions

Using the **Bookmark** property

When you set the **Bookmark** property to a valid value in code, the row associated with that value becomes the current row, and the grid adjusts its display to bring the new current row into view if necessary.

Using the navigation methods

When the grid is bound to a Data control, you can manipulate the underlying **Recordset** using Data control properties and methods such as **EOF**, **MoveFirst**, and **MoveNext**. However, if you decide later on to switch to an unbound data source, these properties and methods will be unavailable.

True DBGrid solves this problem by providing properties and methods that mimic those supported by **Recordset** objects. These properties and methods work the same for all **DataMode** settings, thus eliminating the need to write separate implementations for bound and unbound modes.

<u>MoveFirst</u>	Moves the current record to the first record available.
<u>MoveLast</u>	Moves the current record to the last record available.
<u>MoveNext</u>	Moves the current record to the next record available.
<u>MovePrevious</u>	Moves the current record to the previous record available.
<u>MoveRelative</u>	Moves the current record according to a specified offset and optional bookmark.

Only the **MoveRelative** method accepts arguments (a long integer offset and an optional variant bookmark). A positive offset indicates forward movement; a negative offset indicates backward movement. If the bookmark argument is omitted, the current row's bookmark is used. Thus, the following statements are all equivalent:

```
TDBGrid1.MoveRelative -1, TDBGrid1.Bookmark  
TDBGrid1.MoveRelative -1  
TDBGrid1.MovePrevious
```

Detecting BOF and EOF conditions

In addition to the record navigation methods, True DBGrid provides **BOF** and **EOF** properties for determining beginning and end of file conditions. For example, the following loop iterates through all available records in any data mode:

```
With TDBGrid1
    .MoveFirst
    While Not .EOF
        ' Process current record
        .MoveNext
    Wend
End With
```

NOTE: This example is provided for illustration purposes only. In a real application, iterating through the entire grid would produce a flurry of screen activity, and is therefore not recommended. Such operations are best performed on the underlying data source, or better yet, a **Recordset** clone if one is available.

Accessing and Manipulating Cell Data

In order to access and manipulate cell data, you need to use the **Columns** collection, which contains zero or more **Column** objects. For more information, see [Configuring Columns at Run Time](#).

{button ,JI(`,`Reading_and_writing_cell_data_within_the_current_record')}} Reading and writing cell data within the current record

{button ,JI(`,`Reading_cell_data_from_non-current_records')}} Reading cell data from non-current records

{button ,JI(`,`Retrieving_a_bookmark_relative_to_the_current_record')}} Retrieving a bookmark relative to the current record

{button ,JI(`,`Retrieving_a_bookmark_relative_to_a_displayed_row')}} Retrieving a bookmark relative to a displayed row

Reading and writing cell data within the current record

You can read and write cell data within the current row by using the **Column** object's **Text** and **Value** properties. (The **CellText** and **CellValue** methods are used to read values from other, non-current rows.)

To examine cell data in the current row:

```
CurrentText$ = TDBGrid1.Columns(ColIndex).Text  
CurrentValue = TDBGrid1.Columns(ColIndex).Value
```

The **Text** and **Value** properties return the current contents of the specified column in the current row. Note that the contents may have been edited by the user. The **Text** property returns a formatted string (according to the column's **NumberFormat** property) exactly as it appears in the cell. The **Value** property returns the unformatted cell data as a string variant.

To change cell contents in the current row:

```
TDBGrid1.Columns(ColIndex).Text = NewText$  
TDBGrid1.Columns(ColIndex).Value = NewValue
```

Since the **Value** property accepts a variant, you can supply the new data using any appropriate data type. For example, you can write a null value to the database by setting **Value** to Null in the **BeforeUpdate** event. You do not need to format `NewText$` or `NewValue`, since the grid will perform the appropriate formatting before displaying the data.

Reading cell data from non-current records

You can read cell data from non-current rows by using the Column object's CellText and CellValue methods. Note that you can only read data from non-current rows; you cannot update them. (The Text and Value properties are used to read and write values within the current row.)

To examine cell data in any row, where `bkm` is the bookmark of the desired row:

```
RowText$ = TDBGrid1.Columns(ColIndex).CellText(bkm)
RowValue = TDBGrid1.Columns(ColIndex).CellValue(bkm)
```

The CellText method returns a formatted string (according to the column's `NumberFormat` property) exactly as it appears in the cell. The CellValue method returns the unformatted cell data as a string variant.

Retrieving a bookmark relative to the current record

To retrieve a bookmark relative to the current record, use the grid's **GetBookmark** method, which accepts an integer offset that specifies the number of records after the current record if positive, or before the current record if negative:

```
Dim Bmk As Variant
Bmk = TDBGrid1.GetBookmark(0) ' Current row
Bmk = TDBGrid1.GetBookmark(1) ' Next row
Bmk = TDBGrid1.GetBookmark(-1) ' Previous row
Bmk = TDBGrid1.GetBookmark(4) ' Fourth row after current row
```

Note that the record referred to by the **GetBookmark** method need not be visible on the screen.

Retrieving a bookmark relative to a displayed row

To retrieve a bookmark relative to a row that is currently displayed, use the **RowBookmark** method, which accepts an integer offset that specifies a zero-based row number ranging from 0 to **VisibleRows** - 1:

```
Dim Bmk As Variant
With TDBGrid1
    Bmk = .RowBookmark(0)           ' First displayed row
    Bmk = .RowBookmark(1)         ' Second displayed row
    Bmk = .RowBookmark(.VisibleRows - 1) ' Last displayed row
End With
```

Note that unlike the **GetBookmark** method, the **RowBookmark** method only returns bookmarks for visible rows.

Validating Cell Data

If the grid's **AllowUpdate** property is True, your users will be able to modify the underlying data interactively through the grid's user interface. To prevent users from updating the database with invalid entries, you can write data validation code that works on a per-cell or per-row basis. To validate individual cells, use the **BeforeColUpdate** event. To validate entire rows, use the **BeforeUpdate** event.

For certain kinds of data, such as dates and phone numbers, you can use the **EditMask** property to specify an input template that filters out illegal characters as they are typed. Even if you use True DBGrid's built-in input masking features, you may still want to write data validation code. For example, the user may enter a phone number that is syntactically legal, but contains a bogus area code. Since the grid knows nothing about area codes, you would have to write a handler for either **BeforeColUpdate** or **BeforeUpdate** in order to prevent the invalid number from being written to the underlying data source.

```
{button ,JI(`,`Validating_end-user_data_entries')} Validating end-user data entries
```

```
{button ,JI(`,`Validating_data_before_updating_to_the_database')} Validating data before updating to the database
```

Validating end-user data entries

When finished making changes to a cell, the user can terminate editing by pressing the ENTER key, moving to another cell using the arrow keys, or clicking another cell with the mouse. If the user stays in the same row, data is not updated to the database. At the end of each cell editing session, you have the opportunity to reject or change the user's modifications in the **BeforeColUpdate** event:

```
Private Sub TDBGrid1_BeforeColUpdate( _  
    ByVal ColIndex As Integer, _  
    OldValue As Variant, Cancel As Integer)
```

ColIndex is the index of the column just edited. **OldValue** is the value of the cell before any changes were made to it. It is possible to edit a cell many times before updating to the database, but **OldValue** will be the same for all the updating sessions.

You generally use **BeforeColUpdate** to validate data entered by the user before leaving the cell. You can examine or change the edited data using the **Column** object's **Text** and **Value** properties. In any data mode, you can compare the changes to the original cell data using the **OldValue** argument. In bound mode only, you can also examine the field value of the data control:

```
Data1.Recordset.Fields(ColIndex).Value
```

Note that this expression assumes that the grid's columns match the members of the **Fields** collection of the Data control's **Recordset**. This assumption holds true if you are using the grid's automatic layout feature. However, if you have defined your own layout, then you should reference the **Fields** collection by name:

```
Data1.Recordset.Fields("FieldName").Value
```

Note the similarity between the **Column** object and **Columns** collection and the **Field** object and the **Fields** collection. You can access the members of either collection by numeric index or name, whichever is more convenient.

You can cancel the update by setting **Cancel** to True in the **BeforeColUpdate** event. If canceled, the grid will restore the cell to its value before the current editing session. The restored value may not be the same as **OldValue** if the user has edited the cell multiple times. The cell movement will be canceled and the current cell remains in place.

If the update is not canceled, the **AfterColUpdate** event will fire:

```
Private Sub TDBGrid1_AfterColUpdate(ByVal ColIndex As Integer)
```

and the current cell moves to a new location if appropriate. The current cell does not change if the user has left editing by pressing the ENTER key. Note that unless the cell is moved to another row, the user has only changed the visible data in the grid; no change will be made to the database. Before the change has been updated to the database, the user can always restore the cell to **OldValue** by pressing the ESC key. For more information, see [Database Operations](#).

The following example prevents the user from leaving the current cell if the value entered is an empty string. Although your first inclination may be to display a message box for the user in **BeforeColUpdate**, this will not work as expected because the message box will cause the **BeforeColUpdate** event to fire again before it is finished, resulting in an endless series of message boxes. The solution is to post a message to the grid using the **PostMsg** method, then show the message box in the handler for the **PostEvent** event, which will not fire until the **BeforeColUpdate** event has finished.

```
Private Sub TDBGrid1_BeforeColUpdate( _  
    ByVal ColIndex As Integer, _  
    OldValue As Variant, Cancel As Integer)
```

```
    If TDBGrid1.Text = "" Then  
        ' Schedule the PostEvent event
```

```
TDBGrid1.PostMsg 1

' The following line is used to keep the cell blank
' (remove it to restore the old cell value)
OldValue = ""

' Cancel the update and keep focus on this cell
Cancel = True
End If
End Sub

Private Sub TDBGrid1_PostEvent(ByVal MsgId As Integer)

' 1 is the argument to PostMsg in BeforeColUpdate
If MsgId = 1 Then MsgBox "Cells may not be empty"

End Sub
```

Validating data before updating to the database

Any changes made to the current row are updated to the database when the user moves to another row or when your code executes the **Edit** and **Update** methods on the Data control's **Recordset**:

```
Data1.Recordset.Edit  
Data1.Recordset.Update
```

Before any changes are made to the database, the Data control's **Validate** event will fire, followed by True DBGrid's **BeforeUpdate** event. Either of these events can be used to cancel the update to the database by setting the **Cancel** argument to True.

```
Private Sub TDBGrid1_BeforeUpdate(Cancel As Integer)
```

If the update succeeds, the grid's **AfterUpdate** event will fire:

```
Private Sub TDBGrid1_AfterUpdate()
```

and the current cell moves to a new location when appropriate.

You can use similar True DBGrid events to validate the data before adding or deleting a row from the database:

```
Private Sub TDBGrid1_BeforeInsert(Cancel As Integer)  
Private Sub TDBGrid1_AfterInsert()  
Private Sub TDBGrid1_BeforeDelete(Cancel As Integer)  
Private Sub TDBGrid1_AfterDelete()
```

Alternatively, you can use the Data control's **Validate** event to accept or reject the changes to the database. Internally, True DBGrid uses bookmarks to navigate through the database. Thus, when a Data control's **Validate** event is fired in response to a row change in the grid, the **Action** argument will be **vbDataActionBookmark**, which indicates that the **Bookmark** property of the underlying **Recordset** has changed. You can cancel the movement by setting the **Action** argument to zero. In this case, the grid will keep the original row current, rather than changing to the newly selected row. The **Save** argument will be True if data in a bound control has changed and is about to be written to the database. If you set the **Save** argument to False, any user edits will remain, and the user will need to correct the data before continuing. This is how invalid data is typically handled. See the Visual Basic Help for more details on using the Data control's **Validate** event.

The following example demonstrates how to use the **BeforeUpdate** event to validate an entire row of data. Whenever the user attempts to update the current row, a message box is displayed, asking the user if the record should be saved. If the user chooses Yes, the update proceeds. If the user chooses No, the update is canceled, and the grid's current row does not change.

```
Dim UpdateFlag As Boolean  
  
Private Sub TDBGrid1_BeforeUpdate(Cancel As Integer)  
    Dim MsgText As String, MsgCaption As String  
    Dim Response As Integer, MsgBoxType As Integer  
  
    MsgText = "Do you want to save this record?"  
    MsgBoxType = vbYesNo + vbQuestion  
    MsgCaption = "Confirm Update"  
  
    Response = MsgBox(MsgText, MsgBoxType, MsgCaption)  
  
    If Response = vbNo Then  
        Cancel = True  
        UpdateFlag = True
```

```
End If
End Sub

Private Sub TDBGrid1_Error(ByVal DataError As Integer, _
    Response As Integer)

    If UpdateFlag Then      ' Error triggered in BeforeUpdate
        Response = 0      ' Don't display default message
        UpdateFlag = False ' Clear flag for next update
    End If
End Sub
```

Unlike the **BeforeColUpdate** event, displaying a message box in the **BeforeUpdate** event does not cause recursion problems.

The **Error** event is included in this example because it will fire as a result of the update being canceled. The boolean variable `UpdateFlag` is used to suppress the default error message for the **BeforeUpdate** case but not for other kinds of errors. For more information, see [Handling Database Errors](#).

Selecting and Highlighting Records

At run time, the user can select and highlight one or more records by clicking the record selector of the desired row. You can achieve the same effect in code by manipulating the grid's **SelBookmarks** collection, which maintains a list of bookmarks corresponding to the selected rows.

Like all other collections in True DBGrid, the **SelBookmarks** collection supports **Add**, **Item**, and **Remove** methods and a **Count** property. For example, to select the grid's current row, use the **Add** method:

```
TDBGrid1.SelBookmarks.Add TDBGrid1.Bookmark
```

After the **Add** method executes, the collection's **Count** property is incremented. You can use the **Add** method repeatedly to select and highlight additional rows. This is analogous to the user holding down the CTRL key while clicking a record selector.

To deselect a single record, use the **Remove** method, which takes a collection index, not a bookmark:

```
TDBGrid1.SelBookmarks.Remove 0
```

After the **Remove** method executes, the collection's **Count** property is decremented. If more than one record is selected, the previous example will only remove the first selected record; that is, the one that was added to the collection first, regardless of its position on the screen. To deselect all records, you need to write a loop like the following:

```
With TDBGrid1.SelBookmarks
  While .Count > 0
    .Remove 0
  Wend
End With
```

[Tutorial 5](#) demonstrates how to use the **SelBookmarks** collection to select records that satisfy specific criteria.

Updating and Deleting the Current Record

If the user finishes making changes to a cell, but stays in the same row, data is not updated to the database. You can force the row to be updated by applying the **Update** method of the grid. This method works for any **DataMode** setting:

```
TDBGrid1.Update
```

For a bound grid (**DataMode** property set to 0 - Bound), you can also force an update using the **Update** method of the Data control's **Recordset**:

```
Data1.Recordset.Edit  
Data1.Recordset.Update
```

You can delete the current row in any data mode using the grid's **Delete** method:

```
TDBGrid1.Delete
```

Refreshing the Display

True DBGrid defers screen updates until they are needed by waiting until Windows enters an *idle loop*. This generally occurs when your code stops executing and the system is waiting for user input. You can simulate an idle loop in code by calling the Visual Basic function **DoEvents**, which causes all pending events to be processed.

In most cases, you need not be concerned with the grid's display operations and can concentrate on writing code that works directly with the database. However, if the structure of your data source changes, or you need to temporarily keep the grid from responding to database events, you can use the **Refresh**, **ReBind**, or **ReOpen** methods. If you are **not** using **DataMode** 1 - Unbound, these methods behave as follows:

<u>Refresh</u>	This method simply forces the grid to repaint, and no database access occurs. The grid maintains all modified data in the current row, and the current cell position is unchanged.
<u>ReBind</u>	This method causes the grid to disconnect from and then reconnect to its data source. The grid rebinds all columns and refetches all data. Any data changed by the user (but not yet updated to the database) will be lost. The grid maintains the current row but not the current column. When data is redisplayed, the leftmost visible column becomes current, and the current row becomes the first row in the grid (unless all records are visible).
<u>ReOpen</u>	This method is typically used following a Close method to reconnect the grid to its data source. The grid is repopulated and the current row is positioned to the row identified by an optional bookmark argument. If no bookmark is specified, then the current row reflects the current row of the data source.

To maintain backward compatibility with the original unbound mode of DBGrid, the **Refresh** and **ReBind** methods behave as follows when the **DataMode** property is set to 1 - Unbound:

<u>Refresh</u>	The grid refetches and redisplayes all data by firing the UnboundReadData event. Any data changed by the user (but not yet updated to the database) will be lost. The grid maintains the current row but not the current column. When data is redisplayed, the leftmost visible column becomes current, and the current row becomes the first row in the grid (unless all records are visible).
<u>ReBind</u>	The grid refetches data by firing the UnboundReadData event, but it maintains any data changed by the user within the current row. When data is redisplayed, the current cell position and the grid display are unchanged.

Coordinating with Other Controls

You can link multiple True DBGrid controls using the **RowColChange** event to trigger related actions. Whenever the grid's current cell changes, the **RowColChange** event is fired, indicating that a new row and/or column has become current. **RowColChange** is triggered in the following cases:

- The user clicks on a new row and/or column in the grid.
- The current cell is changed by code. This includes changing the grid's **Bookmark**, **Col**, or **Row** properties and changing the current record through the **Recordset**.
- Data in the grid is refreshed.
- The user moves the record pointer using the navigation buttons on the Data control associated with the grid.

For example, you can use the **RowColChange** event to update the display of a status bar that provides additional information or instructions as the user navigates from column to column. By providing the user with clues during the data entry process, you can make your program more intuitive.

```
Private Sub TDBGrid1_RowColChange(LastRow As Variant, _
    ByVal LastCol As Integer)

    ' TDBGrid1.Col has the new current column index
    ' Display help string in Label1
    Select Case TDBGrid1.Col
        Case 0
            Label1.Caption = "Customer's credit card number"
        Case 1
            Label1.Caption = "Five-digit order number"
        Case 2
            Label1.Caption = "Total cost in US dollars"
    End Select
End Sub
```

The **RowColChange** event is also a convenient place to coordinate activities with other controls or databases. [Tutorial 3](#) provides an example of how you can use the **RowColChange** event to implement master-detail relationships using two True DBGrid controls.

Handling Database Errors

True DBGrid processes errors in a manner consistent with other custom controls, generating both trappable errors and error events. Trappable errors are reported whenever an error is generated by a Visual Basic command or statement. Error events are fired whenever an error is generated by a user action and there is no identifiable line of Visual Basic code causing the error.

When you develop error handling procedures, there are two key points to always keep in mind:

1. The source of the operation which ultimately results in an error.
2. The source of the actual error.

{button ,JI(`,`Trapping_errors')} Trapping errors

{button ,JI(`,`Processing_trapped_errors')} Processing trapped errors

{button ,JI(`,`Errors_caused_by_cancellation')} Errors caused by cancellation

Trapping errors

Trapping an error is the first step in handling an error. As an example, suppose there is a grid bound to a Data control, displaying a **Recordset** with a numeric field. Suppose also that a user enters non-numeric data in the grid column associated with the numeric field. When an attempt is made to change rows, the Data control automatically initiates the update process and tries to store the non-numeric data, resulting in an error.

How is this error to be handled? Will the error appear as a trappable error, or in an error event? The answer lies with the first key point given. The error will be reported by the control or code which initiates the row change (the source of the operation which ultimately leads to the error), since it is the row change operation which cannot be completed.

How many ways can you initiate the row change operation, and how is each trapped? Here are some examples:

1. Click a command button which performs a **Recordset** move operation. In this case, the row change operation is initiated by a line of Visual Basic code, and the error will be reported as a trappable error, which is handled with the `On Error` statement.
2. Click a button of the data control. In this case, there is no identifiable line of Visual Basic code which can be trapped, so the error will be reported in an error event. Because the row change is initiated by the data control, the data control's **Error** event will be fired.
3. Click any non-current row of True DBGrid. As in example 2, there is no identifiable line of Visual Basic code which can be trapped, so once again the error is reported in an error event. However, in this case, the row change is initiated by True DBGrid, so the error is reported in the grid's **Error** event.
4. Press the down arrow key. As in example 3, the error is reported in the grid's **Error** event for the same reasons.
5. Press the down arrow key, but perform a row change operation in the **KeyDown** event of the grid using a bookmark or **Recordset** move operation. In this case, the row change event is initiated by the code in the **KeyDown** event, and the error is trapped through the `On Error` statement.

Processing trapped errors

After trapping a reported error, the task remains to process the error appropriately within the context of your program. This requires analyzing the error code and error text to determine the problem and, of course, deciding what to do about it.

In many cases, you will find that simply reporting the error to the user is the most appropriate. When an error is trapped through an error event, reporting the error can be handled automatically by not coding the error event at all. By default, a message box is displayed indicating the error. Additional details of generalized **Error** events can be obtained from the Visual Basic help file.

Trappable errors, which originate from Visual Basic code, will result in program termination unless the `On Error` statement is used to set up an error handler. Appropriate messages for the errors can usually be obtained from the Visual Basic **Error** function, but sometimes require inspection of the object in which the original error occurred. This will be discussed later.

When it becomes necessary to do more than display an error message, the second key point (the source of the actual error) must be considered. In the example presented earlier, where non-numeric data is placed in a numeric field, the error "Data type conversion error" is usually issued. This error does not originate from the grid, or even the Data control. It comes from the database engine---DAO in the case of an Access database.

Database errors usually result in a cascade of errors, each a direct result of the other. In our example, the first error encountered is the "Data conversion error" generated by the database engine when attempting to update the numeric field. As a result, a general failure error occurs from the row update. The row update error causes the row change operation to fail, also with a general error code. This cascade of errors makes the problem difficult to analyze, since only the general error of the row change failure is reported to the grid from the data control, and finally passed on to your Visual Basic program as the trappable error code or the `DataError` argument of the grid's **Error** event. In order to discern the true nature of the problem, it is necessary to inspect the source of the original error---the **Errors** collection of the database engine. This collection object is discussed fully in the Visual Basic help file.

To simplify handling of errors in the grid's **Error** event, True DBGrid supports an **ErrorText** property. This property is read-only, and is available only during the execution of the **Error** event. It returns the error message string associated with the underlying data source error that ultimately caused the grid's **Error** event to fire.

Errors caused by cancellation

There are times when the source of an error is difficult to determine. The most common case of this is an error which reports "Action canceled by an associated object." In most cases, this will be the result of a database operation which has not been completed because a bound control has denied permission for the operation to complete. Commonly, this will be the result of a canceled event. For example, suppose the grid's BeforeUpdate event is canceled. The cancellation is reported to the Data control, which simply reports the cancellation back to the grid. The Data control does not have knowledge of the reason for cancellation; it only knows that it has occurred. Therefore, it issues the generic "Action canceled by an associated object" message. In this case, since the cancellation is performed by your code. It is your responsibility to display an informative error message and cancel the default error message generated by the Data control.

Postponing Illegal Operations in Grid Events

During most of the grid events, database and other system operations are still pending, and certain operations are not allowed within these grid events. To circumvent such limitations, you can use the **PostMsg** method in conjunction with the **PostEvent** event to postpone operations which are illegal within the grid events. If the **PostMsg** method is called, the grid will fire the **PostEvent** event with the **MsgId** of the corresponding **PostMsg** invocation after all pending operations are completed. You can then safely perform all desired operations in the **PostEvent** event.

For example, suppose it is necessary to keep the rows of a **Recordset** in sorted order based upon a particular column, say Column 2. Whenever a cell in Column 2 is edited, or a new row is added, it is then necessary to **Refresh** the Data control so the new data will be placed in the correct order in the **Recordset**. A reasonable approach to this problem is to include a flag variable to determine that a **Refresh** is needed in the grid's **BeforeUpdate** event:

```
Dim refreshNeeded As Boolean ' Global flag variable

Private Sub Form_Load()
    refreshNeeded = False ' Initialize flag variable
End Sub

Private Sub TDBGrid1_BeforeUpdate(Cancel As Integer)
    refreshNeeded = TDBGrid1.Columns(2).DataChanged Or _
        (TDBGrid1.AddNewMode = dbgAddNewPending)
End Sub
```

A convenient place to perform the Data control **Refresh** is within the grid's **AfterUpdate** event. However, it is not possible to perform the **Refresh** method within the **AfterUpdate** event because database operations are still pending, and the **Refresh** method will fail. Instead of performing the **Refresh**, you can call the **PostMsg** method with an arbitrary numeric argument (1, in the following example) in the **AfterUpdate** event. After all pending database operations are completed, the grid will fire the **PostEvent** event. You can then check for the numeric argument and perform the **Refresh** operation safely:

```
Private Sub TDBGrid1_AfterUpdate()
    If refreshNeeded Then
        refreshNeeded = False ' Reset flag variable
        TDBGrid1.PostMsg 1 ' Post a message with MsgId = 1
    End If
End Sub

Private Sub TDBGrid1_PostEvent(ByVal MsgId As Integer)
    Select Case MsgId
        Case 0
            Exit Sub
        Case 1
            Data1.Refresh
        Case 2
            ' Process other postponed operations
    End Select
End Sub
```

The **PostMsg** and **PostEvent** combination in the preceding example postpones the **Refresh** until the **PostEvent** event---after all pending database operations have completed.

If the **PostMsg** argument is zero, the grid will fire the **PostEvent** event with an argument of zero, but all other pending posted events will be discarded. Since refreshing the Data control makes all other posted events irrelevant anyway, the preceding example can be simplified as follows:

```

Private Sub TDBGrid1_AfterUpdate()
    If refreshNeeded Then
        refreshNeeded = False      ' Reset flag variable
        TDBGrid1.PostMsg 0         ' Post a message with Id = 0
    End If
End Sub

Private Sub TDBGrid1_PostEvent(ByVal MsgId As Integer)
    Select Case MsgId
        Case 0
            Data1.Refresh
        Case 1
            ' Process other postponed operations
    End Select
End Sub

```

When the **PostMsg** method is called with a numeric argument, the grid uses the Windows API function **PostMessage** to place a message in the Windows message queue. The numeric argument is passed along with it. When the grid retrieves the posted Windows message, it fires the **PostEvent** event with the numeric argument of the corresponding **PostMsg** method.

Please note that execution of the Visual Basic **DoEvents** function will cause the Windows message queue to be processed, thus causing execution of the **PostEvent** events before the **DoEvents** returns.

Customizing the Grid's Appearance

This chapter explains how to configure the non-interactive elements of True DBGrid's display, such as captions, headings, and dividing lines.

{button ,JI(`,`Captions_and_Headings')} Captions and Headings

{button ,JI(`,`Three-dimensional_versus_Flat_Display')} Three-dimensional versus Flat Display

{button ,JI(`,`Borders_and_Dividing_Lines')} Borders and Dividing Lines

{button ,JI(`,`Unpopulated_Regions')} Unpopulated Regions

{button ,JI(`,`Highlighting_the_Current_Row_or_Cell')} Highlighting the Current Row or Cell

{button ,JI(`,`Row_Height_and_Multiple-line_Displays')} Row Height and Multiple-line Displays

{button ,JI(`,`Alternating_Row_Colors')} Alternating Row Colors

{button ,JI(`,`Alignment_and_Wordwrap')} Alignment and Wordwrap

Captions and Headings

You can affix a title to a grid, column, or split by setting the **Caption** property of the appropriate object.

```
TDBGrid1.Caption = "Grid Caption"  
TDBGrid1.Columns(0).Caption = "Column 0 Caption"  
TDBGrid1.Splits(0).Caption = "Split 0 Caption"
```

```
{button ,JI(`,`Column_and_grid_captions')} Column and grid captions  
{button ,JI(`,`Multiple-line_captions')} Multiple-line captions  
{button ,JI(`,`Split_captions')} Split captions
```

Column and grid captions

For **Column** objects, the **Caption** property specifies the text that appears in each column's header area.

	Column0	Column1
*		

If you are using True DBGrid in bound mode with an automatic layout, the column captions are set automatically at run time. At design time, you can use the **Retrieve Fields** context menu item to initialize the grid layout according to the current **RecordSource** setting for the Data control to which the grid is bound.

You can also set column captions at design time using the Columns property page, or at run time by manipulating the **Columns** collection in code. For more information, see [Configuring Columns at Run Time](#).

The **Caption** property also applies to the **TDBGrid** control itself, which lets you provide a descriptive header for the entire grid.

TDBGrid		
	Column0	Column1
*		

By default, True DBGrid displays headings for each column, even if you never set the **Caption** property of an individual column explicitly. However, you can hide all column headings by setting the **ColumnHeaders** property to False.

TDBGrid		

Multiple-line captions

The **HeadLines** property controls the height of the column headers. By default, it is set to 1, which means that the column headers occupy a single row. If you need to display more than one line of text in a column header, you can increase the **HeadLines** property to accommodate additional lines, as in the following example:

```
With TDBGrid1
    .HeadLines = 2
    .Columns(0).Caption = "First line" + vbCr + "Second line"
End With
```

Note the use of the Visual Basic constant `vbCr` to specify a line break within the caption text. After this code executes, the first column's caption will contain two lines of text, and the second column's caption will be centered vertically.

TDBGrid	
First line Second line	Column1

NOTE: The **HeadLines** property only affects column headers; it has no effect on grid or split captions, which can only occupy a single row.

Split captions

Split objects can also have their own captions. For a grid with one split, a split caption can serve as a second grid caption.

TDBGrid	
Split0	
Column0	Column1
*	

Split captions are best used in grids with at least two splits, as they are ideal for categorizing groups of columns for end-users.

Composers		Vital Statistics			
	First	Last	Country	Birth	Death
▶	Isaac	Albeniz	Spain	05/29/1860	05/18/1909
	Bela	Bartok	Hungary	03/25/1881	09/26/1945
	Alban	Berg	Austria	02/09/1885	12/24/1935
	Ernest	Bloch	Switzerland	07/24/1880	07/15/1959
	Benjamin	Britten	England	11/22/1913	12/04/1976
	Max	Bruch	Germany	01/06/1838	10/02/1920
	Claude	Debussy	France	08/22/1862	03/05/1918
	Edward	Elgar	England	06/02/1857	02/23/1934
	Manuel de	Falla	Spain	11/23/1876	11/14/1946
	Gabriel	Faure	France	05/12/1845	11/04/1924

Three-dimensional versus Flat Display

True DBGrid supports a standard, "flat" control appearance, as well as the more attractive three-dimensional appearance used by many controls. By default, the grid's **Appearance** property is set so that the 3-D look is used. However, this property only controls whether 3-D effects are used within the grid's caption bar, column headings, and record selector column. It does not affect the grid's border, data cells, or row and column dividers.

When **Appearance** is set to 1 - 3D, the grid looks like this.



First	Last	Country
Isaac	Albeniz	Spain
Johann Sebastian	Bach	Germany
Samuel	Barber	United States
Bela	Bartok	Hungary
Ludwig van	Beethoven	Germany
Alban	Berg	Austria
Luciano	Berio	Italy
Hector	Berlioz	France
Leonard	Bernstein	United States

When **Appearance** is set to 0 - Flat, the grid looks like this.



First	Last	Country
Isaac	Albeniz	Spain
Johann Sebastian	Bach	Germany
Samuel	Barber	United States
Bela	Bartok	Hungary
Ludwig van	Beethoven	Germany
Alban	Berg	Austria
Luciano	Berio	Italy
Hector	Berlioz	France
Leonard	Bernstein	United States

To achieve a 3-D appearance for the entire grid, including its interior, set the following properties at either design time or run time:

- On the Display property page, set the **RowDividerStyle** property to 4 - Inset. Or, in code:

```
TDBGrid1.RowDividerStyle = dbgInset
```

- On the Layout property page, set the **DividerStyle** property to 4 - Inset for **all** columns. Or, in code:

```
Dim C As TrueDBGrid50.Column  
For Each C In TDBGrid1.Columns  
    C.DividerStyle = dbgInset  
Next
```

- On the Color property page, set the **BackColor** property to gray. Or, in code:

```
TDBGrid1.BackColor = &HC0C0C0
```

The resulting grid will look something like this.

Composers		
First	Last	Country
Isaac	Albeniz	Spain
Johann Sebastian	Bach	Germany
Samuel	Barber	United States
Bela	Bartok	Hungary
Ludwig van	Beethoven	Germany
Alban	Berg	Austria
Luciano	Berio	Italy
Hector	Berlioz	France

Note that changing the **RowDividerStyle** property from 2 - Dark gray line to 4 - Inset consumes an extra vertical pixel in each data row, resulting in fewer visible rows.

You can experiment to achieve different 3-D effects using other color combinations and divider styles, as explained in the next section.

Borders and Dividing Lines

The **RowDividerStyle** and **DividerStyle** properties enable you to choose different horizontal and vertical lines and their colors. Note that **RowDividerStyle** is a **TDBGrid** object property and **DividerStyle** is a **Column** object property. The allowable values for both properties are as follows:

- 0 - No dividers
- 1 - Black line
- 2 - Dark gray line
- 3 - Raised
- 4 - Inset
- 5 - ForeColor
- 6 - Light gray line

For example, setting the **RowDividerStyle** property to 0 - No dividers eliminates the dividing lines between rows and enables you to cram a bit more data into the available area.

Composers		
First	Last	Country
Isaac	Albeniz	Spain
Johann Sebastian	Bach	Germany
Samuel	Barber	United States
Bela	Bartok	Hungary
Ludwig van	Beethoven	Germany
Alban	Berg	Austria
Luciano	Berio	Italy
Hector	Berlioz	France
Leonard	Bernstein	United States
Georges	Rizet	France

Similarly, by setting the **DividerStyle** property to 0 - No dividers, you can visually group related columns. The column headers are not affected, however.

Composers		
First	Last	Country
Isaac	Albeniz	Spain
Johann Sebastian	Bach	Germany
Samuel	Barber	United States
Bela	Bartok	Hungary
Ludwig van	Beethoven	Germany
Alban	Berg	Austria
Luciano	Berio	Italy
Hector	Berlioz	France
Leonard	Bernstein	United States

Unpopulated Regions

Depending upon the number of rows and columns in the data source, a portion of the grid's interior may not contain data cells. However, you can eliminate these "dead zones" using the **ExtendRightColumn** and **EmptyRows** properties.

```
{button ,JI(``, `The_rightmost_column`)} The rightmost column
```

```
{button ,JI(``, `Unused_data_rows`)} Unused data rows
```

The rightmost column

As the grid scrolls horizontally until the last column is totally visible, there is usually a blank area between the last column and the right border of the grid.

Composers		
	First	Last
▶	Isaac	Albeniz
	Johann Sebastian	Bach
	Samuel	Barber
	Bela	Bartok
	Ludwig van	Beethoven
	Alban	Berg
	Luciano	Berio
	Hector	Berlioz
	Leonard	Bernstein
	Georges	Bizet

The color of this blank area depends on the setting of your system's 3D Objects color (or Button Face color), which is usually gray. You can eliminate this blank area with the **ExtendRightColumn** property. The default value of this property is False, but if you set it to True, then the last column will extend its width to the right edge of the grid.

Composers		
	First	Last
▶	Isaac	Albeniz
	Johann Sebastian	Bach
	Samuel	Barber
	Bela	Bartok
	Ludwig van	Beethoven
	Alban	Berg
	Luciano	Berio
	Hector	Berlioz
	Leonard	Bernstein
	Georges	Bizet

Unused data rows

If the data source contains fewer rows than the grid can display, the area below the AddNew row (or the last data row, if **AllowAddNew** is False) is left blank.

American Composers		
	First	Last
▶	Samuel	Barber
	Leonard	Bernstein
	Aaron	Copland
	George	Gershwin
	Charles	Ives
	Virgil	Thomson
*		

The color of this blank area depends on the setting of your system's 3D Objects color (or Button Face color), which is usually gray. You can eliminate this blank area with the **EmptyRows** property. The default value of this property is False, but if you set it to True, then the grid will display empty rows below the last usable data row.

American Composers		
	First	Last
▶	Samuel	Barber
	Leonard	Bernstein
	Aaron	Copland
	George	Gershwin
	Charles	Ives
	Virgil	Thomson
*		

Note that the empty rows cannot receive focus.

You can set both the **EmptyRows** and **ExtendRightColumn** properties to True to ensure that no blank areas appear within the interior of the grid.

American Composers		
	First	Last
▶	Samuel	Barber
	Leonard	Bernstein
	Aaron	Copland
	George	Gershwin
	Charles	Ives
	Virgil	Thomson
*		

Highlighting the Current Row or Cell

The term *marquee* refers to the highlighted area which represents the current grid cell or row. The **MarqueeStyle** property can be set to seven possible presentations, illustrated as follows.

{button ,JI(`,` MarqueeStyle_0')} MarqueeStyle 0 - Dotted Cell Border
{button ,JI(`,` MarqueeStyle_1')} MarqueeStyle 1 - Solid Cell Border
{button ,JI(`,` MarqueeStyle_2')} MarqueeStyle 2 - Highlight Cell
{button ,JI(`,` MarqueeStyle_3')} MarqueeStyle 3 - Highlight Row
{button ,JI(`,` MarqueeStyle_4')} MarqueeStyle 4 - Highlight Row, Raise Cell
{button ,JI(`,` MarqueeStyle_5')} MarqueeStyle 5 - No Marquee
{button ,JI(`,` MarqueeStyle_6')} MarqueeStyle 6 - Floating Editor

MarqueeStyle 0 - Dotted Cell Border

The current cell is highlighted by a dotted border.

UserCode	LastName	FirstName
ACMC	Clark	Allen
AGCE	Gordon	Alan
▶ AQSD	Quinn	Ann
AWCC	Wheeler	Alice
BAMU	Ardmore	Beverly

MarqueeStyle 1 - Solid Cell Border

This is a more distinctive form of cell highlighting, often useful when a different background color is used (since the dotted rectangle is often difficult to spot).

UserCode	LastName	FirstName
ACMC	Clark	Allen
AGCE	Gordon	Alan
▶ AQSD	Quinn	Ann
AWCC	Wheeler	Alice
BAMU	Ardmore	Beverly

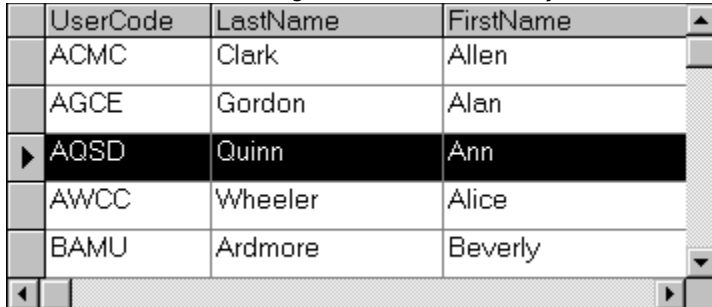
MarqueeStyle 2 - Highlight Cell

This style inverts the current cell completely, making it very visible. Values of the **EditBackColor** and **EditForeColor** properties should be chosen carefully to make a pleasing effect if your grid is editable.

UserCode	LastName	FirstName
ACMC	Clark	Allen
AGCE	Gordon	Alan
▶ AQSD	Quinn	Ann
AWCC	Wheeler	Alice
BAMU	Ardmore	Beverly

MarqueeStyle 3 - Highlight Row

The entire row will be highlighted, but it won't be possible to tell which cell is the current cell in the row. To change highlight colors, you can edit the built-in HighlightRow style on the Styles property page. This style is most useful when the grid is not editable and your users would view the data one record at a time.



UserCode	LastName	FirstName
ACMC	Clark	Allen
AGCE	Gordon	Alan
AQSD	Quinn	Ann
AWCC	Wheeler	Alice
BAMU	Ardmore	Beverly

MarqueeStyle 4 - Highlight Row, Raise Cell

This value should only be used if 3-D lines are used in the grid, since cell highlighting is accomplished using a "raised" appearance for the current cell.

	UserCode	LastName	FirstName	
	ACMC	Clark	Allen	
	AGCE	Gordon	Alan	
▶	AQSD	Quinn	Ann	
	AWCC	Wheeler	Alice	
	BAMU	Ardmore	Beverly	

MarqueeStyle 5 - No Marquee

This setting will make the marquee disappear completely. Often this setting is useful for cases where the current row is irrelevant, or where you don't want to draw the user's attention to the grid until necessary.

	UserCode	LastName	FirstName	
	ACMC	Clark	Allen	
	AGCE	Gordon	Alan	
▶	AQSD	Quinn	Ann	
	AWCC	Wheeler	Alice	
	BAMU	Ardmore	Beverly	

MarqueeStyle 6 - Floating Editor

This is the default marquee style of the grid. The cell text (the actual text only, **not** the entire cell) is highlighted and there is a blinking text cursor (caret) at the end of the text.

UserCode	LastName	FirstName
ACMC	Clark	Allen
AGCE	Gordon	Alan
AQSD	Quinn	Ann
AWCC	Wheeler	Alice
BAMU	Ardmore	Beverly

The color of the highlight is your system's highlight color. The floating editor style simulates the look and feel of the Microsoft Access datasheet. The blinking text cursor indicates that the cell is *edit-ready*, hence the name floating editor for this marquee style. Since no other marquee style places the cell in a similar edit-ready mode, the behavior of the grid with the floating editor is sometimes different from the other marquee styles. The following list summarizes the differences when the **MarqueeStyle** property is set to 6 - Floating Editor:

1. The following properties are ignored by the floating editor: **EditBackColor**, **EditDropDown**, **EditForeColor**, **EditorStyle**, and **MarqueeUnique**.
2. When using the **AddCellStyle** and **AddRegexCellStyle** methods with the floating editor, the grid ignores the current cell bit (`dbgCurrentCell`) and highlighted row bit (`dbgMarqueeRow`) of the **Condition** argument. For more details, see [Introduction to Cell Styles](#).
3. The floating editor will not be displayed in a cell with radio buttons or a picture, as described in [Automatic Data Translation with ValueItems](#). A dotted cell marquee will be used instead. The floating editor highlight will return when the current cell is changed to one with normal text display.
4. The **CycleOnClick** property (applies to **ValueItems** collection) has no effect when the **MarqueeStyle** property is set to 6 - Floating Editor.
5. The drag-and-drop features described in [Drag-and-Drop Behavior](#) will not work as well as they do with other marquee styles. Users will not be able to begin dragging a cell to trigger the **DragCell** event unless they manage to grab the narrow region between the floating editor and either column divider. This is because the floating editor is itself a control and is preventing the grid cell from detecting the drag.
6. The **DbiClick** event of the **TDBGrid** control does not fire when the user double-clicks a noncurrent cell within the grid. This is because the first click is used by the floating editor to begin editing, placing the cell into edit mode at the character on which the click occurred. Double-clicking the current cell of the grid fires the **DbiClick** event normally, however.

Row Height and Multiple-line Displays

The **RowHeight** property controls the height of all grid rows. The **MultipleLines** property controls whether a single row can span multiple lines.

{button ,JI(`,`Adjusting_the_height_of_all_grid_rows')} Adjusting the height of all grid rows

{button ,JI(`,`Displaying_a_single_record_on_multiple_lines')} Displaying a single record on multiple lines

{button ,JI(`,`Implications_of_multiple-line_mode')} Implications of multiple-line mode

Adjusting the height of all grid rows

You can configure the row height interactively at design time by placing the grid in its visual editing mode or by changing the grid's **RowHeight** property on the General property page. At run time, the user can adjust the row height interactively if **AllowRowSizing** is True. For more information, see [Run Time Interaction](#).

The **RowHeight** property is expressed in units of the container's coordinate system. However, a setting of 0 causes the grid to readjust its display so that each row occupies a single line of text in the current font. Therefore, you can use the following code to adjust the row height to display exactly three lines of text:

```
TDBGrid1.RowHeight = 0  
TDBGrid1.RowHeight = 3 * TDBGrid1.RowHeight
```

This technique is particularly effective when displaying multiple-line memo fields, as in this example:

Element	Description
Actinium	A radioactive chemical element found with uranium and radium in pitchblende and other minerals and formed in reactors by the neutron irradiation of radium.
Aluminum	A silvery, lightweight, easily worked metal that resists corrosion and is found abundantly, but only in combination.
Americium	One of the transuranic elements produced by the beta decay of an isotope of plutonium.
Antimony	A silvery-white, brittle, metallic chemical element of crystalline structure, found only in combination. It is used in alloys with other metals to harden them and increase

Note that the Description column must have its **WrapText** property set to True; otherwise, the memo field display will be truncated after the first line.

Displaying a single record on multiple lines

Normally, a record is displayed in a single row in the grid. If the grid is not wide enough to display all of the columns in the record, a horizontal scroll bar automatically appears to enable users to scroll columns in and out of view. For discussion purposes, we shall distinguish between the following:

- A **line** in a grid is a single *physical row* of cells displayed in the grid. Do not confuse this with a line of text inside a grid cell; depending upon the settings of the **RowHeight** and **WrapText** properties, data in a grid cell may be displayed in multiple lines of text.
- A **row** in a grid is used to display a single record. A row may contain multiple lines or multiple *physical rows*.

The **MultipleLines** property of the grid controls how records are displayed. The default value is False, which means that a single record or row cannot span multiple lines. If necessary, the end-user can operate the horizontal scroll bar to view all of the columns within a row. This is how the grid normally displays data.

However, if the **MultipleLines** property is set to True, then a single record may span multiple lines. This feature enables the end-user to view simultaneously all of the columns (fields) of a record within the width of the grid without scrolling horizontally, as in the following figure:

Element	Atomic No.	▲
Symbol	Atomic Wt.	
▶ Hydrogen	1	
H	1.0079	
Helium	2	
He	4.0026	
Lithium	3	
Li	6.941	
Beryllium	4	
Be	9.01218	
Boron	5	
B	10.81	
Carbon	6	
C	12.011	▼

This powerful feature requires very little work from the programmer's perspective. Just set the **MultipleLines** property to True, and the grid will do the rest. The horizontal scroll bar will be hidden (if one is present), and the grid will automatically span or wrap the columns to multiple lines so that all columns will be visible within the width of the grid. You can adjust the column layout at either design time or run time by changing the widths and orders of the columns. You can even turn the **MultipleLines** feature on and off with code at run time.

NOTE: If **MultipleLines** is True and the **ScrollBars** property is set to 4 - Automatic, the design time and run time layouts may not match because of the extra space taken up by the vertical scroll bar at run time. You can compensate for this by using a different **ScrollBars** setting when adjusting column widths at design time.

Implications of multiple-line mode

Existing row-related properties, methods, and events fit well with the earlier definitions of records, rows, and lines (with two exceptions to be described later). For example:







- The **VisibleRows** property returns the number of visible rows or records displayed on the grid—not the number of visible lines. If a row spans 2 lines, and the **VisibleRows** property is 5, then there are 10 visible lines displayed on the grid.
- The **RowTop** method accepts a row number argument ranging from 0 to **VisibleRows** - 1. If a row spans 2 lines, then **RowTop**(2) returns the position of the top of the third displayed row (that is, the fifth displayed line).
- The **RowResize** event will be fired whenever a row is resized by the user at run time. In fact, at the record selector column, only row divider boundaries are displayed; thus, the user can only resize rows, not lines.

Other row-related properties, methods, and events can be interpreted similarly. There are two exceptions:

1. The first is the **RowHeight** property. The **RowHeight** property returns the height of a cell or a line, **not** the height of a row. Changing this property would break users' existing code, so we decided to keep this property the same.
2. The second is more of a limitation than an exception. Currently the dividers between rows and lines are the same. When you change the **RowDividerStyle** property, all dividers between rows and lines change to the same style. That is, you cannot have different dividers for rows and for lines.

Alternating Row Colors

You can often improve the readability of the display by alternating the background colors of adjacent rows. When you set the **AlternatingRowStyle** property to True, the grid displays odd-numbered rows (the first displayed row is 1) using the built-in style OddRowStyle, and even-numbered rows using the built-in style EvenRowStyle.

Customer Name	Customer Type	How	Call?	ContactDate
Greg Daryl	Prospective		<input checked="" type="checkbox"/>	10/3/93
Jane Lambert	Distributor		<input type="checkbox"/>	9/5/93
Allen Clark	Normal		<input checked="" type="checkbox"/>	1/5/93
David Elkins	Prospective		<input type="checkbox"/>	10/2/93
Carl Ziegler	Distributor		<input checked="" type="checkbox"/>	11/17/93
William Yahner	Rover		<input type="checkbox"/>	11/1/93

[Tutorial 12](#) demonstrates how to change the default alternating colors at design time.

Alignment and Wordwrap

By default, a grid cell displays a single line of text, truncated at the cell's right boundary. You can display multiple lines of text in a cell by increasing the grid's row height and by setting the **WrapText** property of the desired columns to True. If **WrapText** is True (the default is False), a line break occurs before words that would otherwise be partially displayed in a cell. The cell contents will continue to display on the next line, assuming that the grid's row height accommodates multiple lines.

You can use the following loop to enable wordwrap for all grid columns:

```
Dim C As TrueDBGrid50.Column
For Each C In TDBGrid1.Columns
    C.WrapText = True
Next
```

Data Presentation Techniques

This chapter explains how to display cell data in a variety of textual and graphical formats. To learn how to customize the behavior of cell editing in True DBGrid, see [Cell Editing Techniques](#).

{button ,JI(`,`Text_Formatting')}` Text Formatting

{button ,JI(`,`Automatic_Data_Translation_with_ValueItems')}` Automatic Data Translation with ValueItems

{button ,JI(`,`Context-sensitive_Help_with_CellTips')}` Context-sensitive Help with CellTips

Text Formatting

In many cases, the raw numeric data that True DBGrid receives from its data source is not suitable for end-user display. For example, date fields may need to be converted to a specific international format; currency fields may contain too many insignificant digits after the decimal point. Therefore, True DBGrid provides access to the intrinsic formatting of Visual Basic on a per-column basis by means of the **NumberFormat** property.

For cases where Visual Basic's formatting is inadequate, or for other development environments such as Visual C++, True DBGrid provides an event, **FormatText**, that enables your application to override the default formatting on a per-column basis.

{button ,JI(`,`Using_Visual_Basic_built-in_formatting')}} Using Visual Basic's built-in formatting
{button ,JI(`,`Input_validation_with_built-in_formatting')}} Input validation with built-in formatting
{button ,JI(`,`Formatting_with_an_input_mask')}} Formatting with an input mask
{button ,JI(`,`Formatting_with_a_custom_event_handler')}} Formatting with a custom event handler

Using Visual Basic's built-in formatting

True DBGrid supports a variety of data formatting options through the **Column** object's **NumberFormat** property, which provides the same functionality as Visual Basic's **Format\$** function. For example, to display all date values within a column according to the form 26-Apr-97, you would use the Medium Date setting:

```
TDBGrid1.Columns("HireDate").NumberFormat = "Medium Date"
```

Note that if you change the **NumberFormat** property of a column at run time, you do not need to refresh the display, as True DBGrid handles this automatically.

At design time, you can set the **NumberFormat** property using the Columns property page. For numeric data, the following predefined options are available:

Standard	Display number with thousands separator, at least one digit to the left and two digits to the right of the decimal separator.
General Number	Display number as is, with no thousand separators.
Currency	Display number with thousand separator, if appropriate; display two digits to the right of the decimal separator. Note that output is based on system locale settings.
Percent	Display number multiplied by 100 with a percent sign (%) appended to the right; always display two digits to the right of the decimal separator.
Fixed	Display at least one digit to the left and two digits to the right of the decimal separator.
Scientific	Use standard scientific notation.
Yes/No	Display No if number is 0; otherwise, display Yes.
True/False	Display False if number is 0; otherwise, display True.
On/Off	Display Off if number is 0; otherwise, display On.
0%	Display number multiplied by 100, then rounded to the nearest integer, with a percent sign (%) appended to the right.
0.00%	Same as Percent.

For date and time data, the following predefined options are available:

General Date	Display a date and/or time. For real numbers, display a date and time (for example, 4/3/93 05:34 PM); if there is no fractional part, display only a date (for example, 4/3/93); if there is no integer part, display only a time (for example, 05:34 PM). Date display is determined by your system settings.
Long Date	Display a date using your system's long date format.
Medium Date	Display a date using the medium date format appropriate for the language version of Visual Basic.
Short Date	Display a date using your system's short date format.
Long Time	Display a time using your system's long time format: includes hours, minutes, seconds.
Medium Time	Display a time in 12-hour format using hours and minutes and the AM/PM designator.
Short Time	Display a time using the 24-hour format (for example, 17:45).

Input validation with built-in formatting

It is important to note that the **NumberFormat** property affects only the *display* of data in the grid. Unless you also specify a value for the **EditMask** property, True DBGrid does not enforce an input template, and the user is free to type anything into the formatted cell. When moving to another cell, the grid will reasonably interpret the user's input value, update to the database (if necessary), and redisplay the data according to the **NumberFormat** setting.

For example, if Medium Date formatting is in effect for a column, a date of Saturday, April 26, 1997, 12:00:00 AM will be displayed as 26-Apr-97 with the day of the week and time ignored. If a user enters July and moves to another row, the grid cannot reasonably interpret the input date value and a trappable error will occur. If the user enters oct 5 or 10/5, the grid will interpret the entered date as October 5, 1997 (that is, the current year is assumed). If the database update is successful, the entered date will be redisplayed as 05-Oct-97, since Medium Date formatting is in effect.

Formatting with an input mask

Since it is common for the input and display formats to be the same, the **NumberFormat** property has an Edit Mask option (note the space between words). If you select this option, then the **EditMask** property setting will be used for both data input and display. However, the input and display formats need not be the same, so you are free to select a **NumberFormat** option that differs from the **EditMask** property.

For example, the following code applies a phone number template to a column for both display and editing:

```
With TDBGrid1.Columns("Phone")
    .EditMask = "(###) ###-####"
    .NumberFormat = "Edit Mask"
End With
```

For more information on how to specify a data input mask, see [Input Masking](#).

Formatting with a custom event handler

On occasion, you may find that the Visual Basic formatting options do not suit your particular needs. Or, you may be using True DBGrid in a development environment that does not support Visual Basic formatting, such as Visual C++. In these cases, the `FormatText` Event option can be specified for the **NumberFormat** property. Choosing this option for a column will cause the **FormatText** event to fire each time data is about to be displayed in that column. The event allows you to reformat, translate, indent, or do anything you want to the data just prior to display:

```
Private Sub TDBGrid1_FormatText(ByVal ColIndex As Integer, _
    Value As Variant)
```

`ColIndex` contains the column number of the grid to be reformatted. `Value` contains the current value of the data and also serves as a placeholder for the formatted display value. For example, suppose the first column contains numeric values from 1 to 30, and you wish to display the data as Roman numerals:

```
Private Sub TDBGrid1_FormatText(ByVal ColIndex As Integer, _
    Value As Variant)
```

```
    Dim result As String

    If ColIndex = 0 Then
        ' Determine how many X's
        While Value >= 10
            result = result & "X"
            Value = Value - 10
        Wend

        ' Append "digits" 1-9
        Select Case Value
            Case 1
                result = result & "I"
            Case 2
                result = result & "II"
            Case 3
                result = result & "III"
            Case 4
                result = result & "IV"
            Case 5
                result = result & "V"
            Case 6
                result = result & "VI"
            Case 7
                result = result & "VII"
            Case 8
                result = result & "VIII"
            Case 9
                result = result & "IX"
        End Select

        ' Change the actual format
        Value = result
    End If
End Sub
```

Since the **FormatText** event is not restricted to a particular development environment, you can always use it

to gain full control over the textual content of any value displayed in the grid.

Automatic Data Translation with ValueItems

Although you can use the **FormatText** event to map data values into more descriptive display values, True DBGrid also provides a mechanism for performing such data translations automatically without code. Through the use of the **ValueItems** collection, you can specify alternate text or even pictures to be displayed in place of the underlying data values.

This feature is ideally suited for displaying numeric codes or cryptic abbreviations in a form that makes sense to end-users. For example, country codes can be rendered as proper names or even as pictures of their respective flags. Or, the numbers 0, 1, and 2 may be displayed as Yes, No, and Maybe. Either the actual values (0, 1, 2) or the translated values (Yes, No, Maybe) may be displayed as radio buttons in a cell or in a drop-down combo box.

{button ,JI(`,`What_are_ValueItems?')} What are ValueItems?
{button ,JI(`,`Specifying_text-to-text_translations')} Specifying text-to-text translations
{button ,JI(`,`Specifying_text-to-picture_translations')} Specifying text-to-picture translations
{button ,JI(`,`Displaying_both_text_and_pictures_in_a_cell')} Displaying both text and pictures in a cell
{button ,JI(`,`Simulating_check_boxes_with_in-cell_bitmaps')} Simulating check boxes with in-cell bitmaps
{button ,JI(`,`Displaying_allowable_values_as_radio_buttons')} Displaying allowable values as radio buttons

What are ValueItems?

A **ValueItem** object describes the association between an underlying data value and its visual representation. It supports only two properties: **Value**, the underlying data value, and **DisplayValue**, its visual representation. Both properties are of type Variant.

A **ValueItems** collection contains zero or more **ValueItem** objects. Each **Column** object owns one **ValueItems** collection, which is initially empty.

At design time, the Values property page can be used to build the **ValueItems** collection for a column. [Tutorial 7](#) and [Tutorial 10](#) provide step-by-step instructions for using the Values property page.

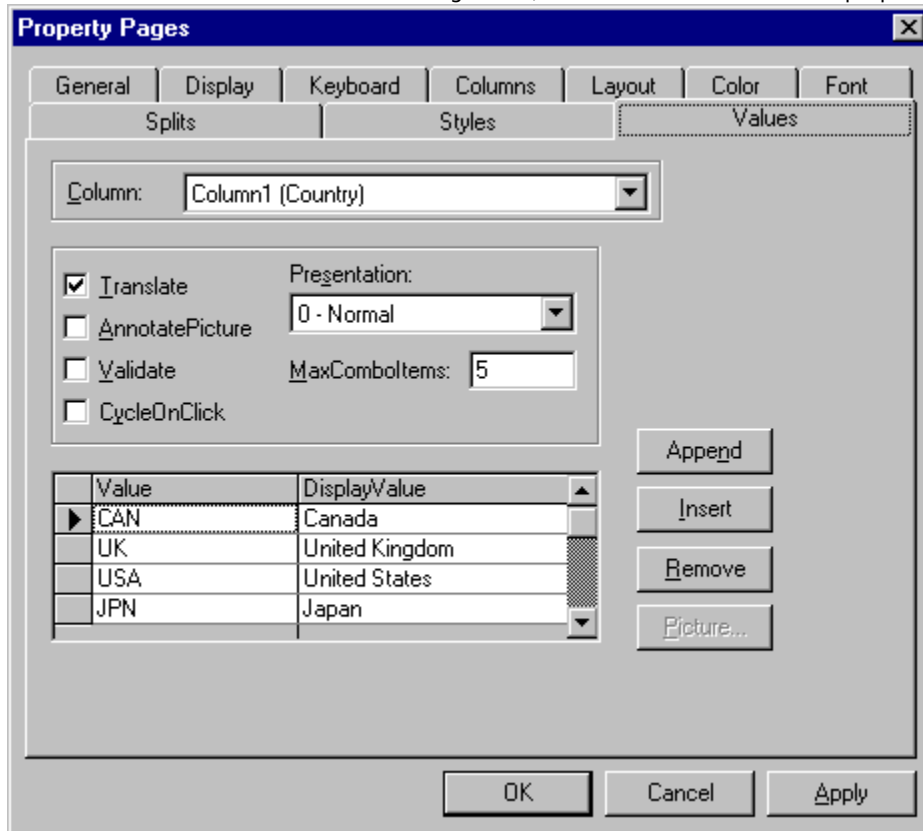
At run time, you can manipulate the **ValueItems** collection as you would any other True DBGrid or Visual Basic collection.

Specifying text-to-text translations

Consider the following example, in which the Country field is represented by a short character code.

Company	Country
Maple Leaf Systems	CAN
Her Majesty's Software	UK
Software Mart	USA
Far East Distributors	JPN
Outback Software, Inc.	AUS
Northwest Purchasing Agents, Inc.	USA

To display the character codes as proper names, you can use the column's **ValueItems** collection to specify automatic data translations. At design time, this is done with the Values property page.



The Values property page enables you to specify data translations on a per-column basis. To construct a list of data translations for an individual column, do the following:

1. Use the Column combo box to select the column for which automatic data translation is to be performed.
2. Select the **Translate** check box. This enables automatic data translation and causes the **DisplayValue** column to appear in the property page grid. If the **Translate** check box is not selected, you will only be able to enter items in the **Value** column of the property page grid.
3. Enter as many **Value/DisplayValue** pairs as necessary. Use the Append or Insert buttons to cause a new data entry row to appear.
4. Select OK or Apply to commit the changes.

When the program is run, Country field values that match an item in the **Value** column appear as the corresponding **DisplayValue** entry. For example, CAN becomes Canada, UK becomes United Kingdom, and

so forth.

	Company	Country
▶	Maple Leaf Systems	Canada
	Her Majesty's Software	United Kingdom
	Software Mart	United States
	Far East Distributors	Japan
	Outback Software, Inc.	Australia
	Northwest Purchasing Agents, Inc.	United States

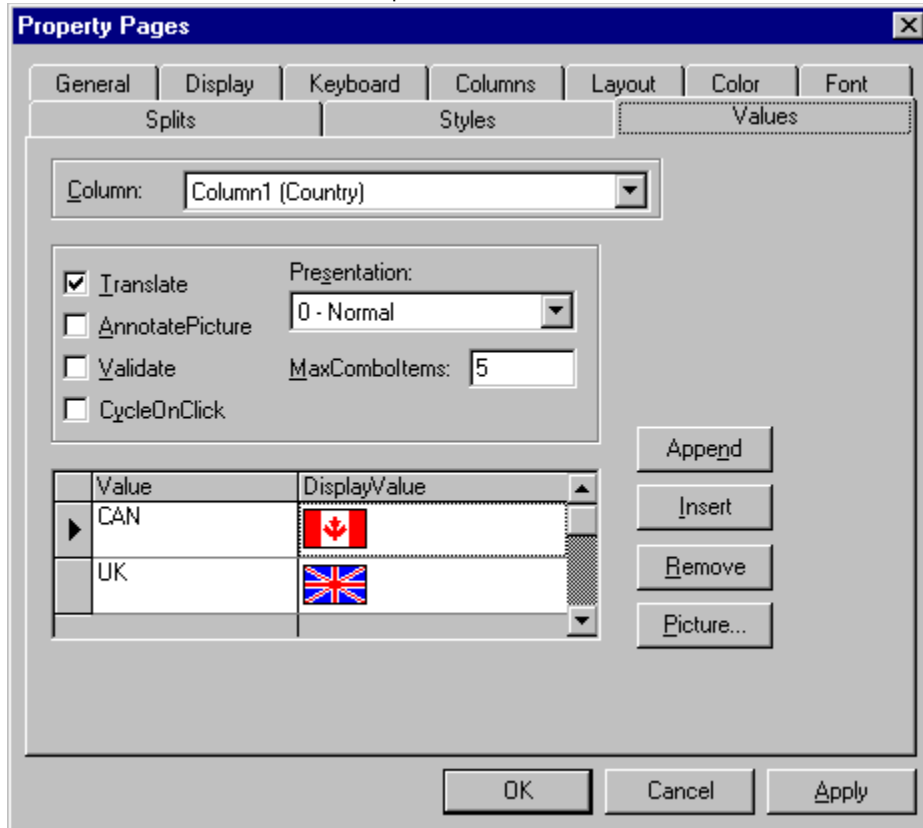
Note that the underlying database is not affected; only the presentation of the data value is different. The same effect can be achieved in code as follows:

```
Dim Item As New ValueItem

With TDBGrid1.Columns("Country").ValueItems
    Item.Value = "CAN"
    Item.DisplayValue = "Canada"
    .Add Item
    Item.Value = "UK"
    Item.DisplayValue = "United Kingdom"
    .Add Item
    Item.Value = "USA"
    Item.DisplayValue = "United States"
    .Add Item
    Item.Value = "JPN"
    Item.DisplayValue = "Japan"
    .Add Item
    Item.Value = "AUS"
    Item.DisplayValue = "Australia"
    .Add Item
    .Translate = True
End With
```

Specifying text-to-picture translations

The same techniques used to specify text-to-text translations can also be used for text-to-picture translations. Within the Values property page, instead of typing a string into the **DisplayValue** column, you can use the Picture button to select a bitmap to be used for data translations.



To make the Picture button available, move the current cell marquee to the **DisplayValue** column. Note that the **Translate** check box must also be selected. Depending upon the height of the bitmaps, you may need to adjust the value of the **RowHeight** property on the Display property page.

When the program is run, Country field values that match an item in the **Value** column appear as the corresponding **DisplayValue** picture.

	Company	Country
▶	Maple Leaf Systems	
	Her Majesty's Software	
	Software Mart	
	Far East Distributors	
	Outback Software, Inc.	
	Northwest Purchasing Agents, Inc.	

As with textual translations, the underlying database is not affected; only the presentation of the data value is different. The same effect can be achieved in code as follows:

```
Dim Item As New ValueItem

With TDBGrid1.Columns("Country").ValueItems
```

```
Item.Value = "CAN"  
Item.DisplayValue = LoadPicture("canada.bmp")  
.Add Item  
Item.Value = "UK"  
Item.DisplayValue = LoadPicture("uk.bmp")  
.Add Item  
Item.Value = "USA"  
Item.DisplayValue = LoadPicture("usa.bmp")  
.Add Item  
Item.Value = "JPN"  
Item.DisplayValue = LoadPicture("japan.bmp")  
.Add Item  
Item.Value = "AUS"  
Item.DisplayValue = LoadPicture("australia.bmp")  
.Add Item  
.Translate = True  
End With
```


Displaying both text and pictures in a cell

Once you have configured the **ValueItems** collection to perform text-to-picture translations for a column, you can cause both the **Value** string and the **DisplayValue** bitmap to appear within the same cell by selecting the **AnnotatePicture** check box on the Values property page. Or, in code:

```
With TDBGrid1.Columns("Country").ValueItems
    .AnnotatePicture = True
End With
```

The placement of the bitmap within the cell is determined by the column's **Alignment** property. Left alignment places the bitmap on the left, and the text is formatted in the remaining space to the right of the bitmap.

Company	Country
Maple Leaf Systems	 CAN
Her Majesty's Software	 UK
Software Mart	 USA
Far East Distributors	 JPN
Outback Software, Inc.	 AUS
Northwest Purchasing Agents, Inc.	 USA

Right alignment places the bitmap on the right, and the text is formatted in the remaining space to the left of the bitmap.

Company	Country
Maple Leaf Systems	CAN 
Her Majesty's Software	UK 
Software Mart	USA 
Far East Distributors	JPN 
Outback Software, Inc.	AUS 
Northwest Purchasing Agents, Inc.	USA 

Center alignment places the bitmap in the center at the top of the cell, and the text is formatted in the remaining space below the bitmap.

Company	Country
Maple Leaf Systems	 CAN
Her Majesty's Software	 UK
Software Mart	 USA
Far East Distributors	 JPN
Outback Software, Inc.	 AUS
Northwest Purchasing Agents, Inc.	 USA

In all cases, the text is centered in the space allotted for it. When editing, the editor uses all space available in the text portion of the cell. When the **Presentation** property of the **ValueItems** collection is set to one of the combo box options, the bitmap will not change until editing is completed.

Note that in the preceding examples, the text is displayed as it is stored in the database without formatting. But what if you want to display both a picture *and* formatted text? Since the **ValueItem** object can only accommodate one translation, you cannot accomplish this with **ValueItems** alone. However, you can use the **FormatText** event to translate the text, then use the **ValueItems** collection to associate the translated text with a picture:

```
TDBGrid1.Columns("Country").NumberFormat = "FormatText Event"
```

In this example, the **NumberFormat** property is set to a special value that causes the **FormatText** event to fire:

```
Private Sub TDBGrid1_FormatText(ByVal ColIndex As Integer, _
    Value As Variant)

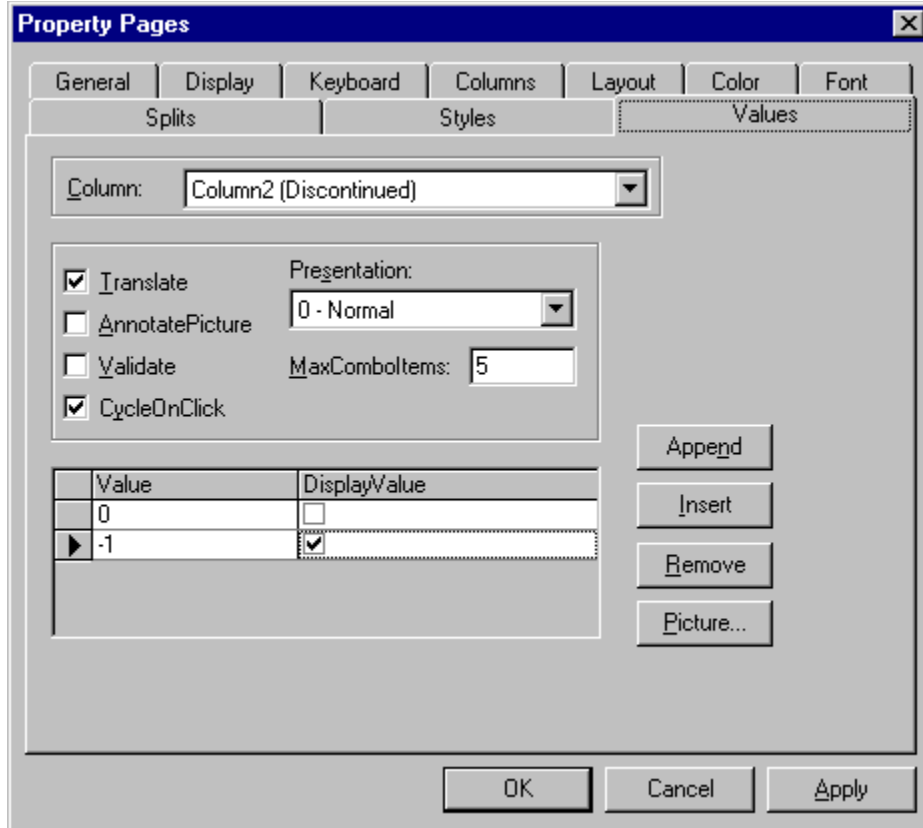
    Select Case Value
        Case "CAN"
            Value = "Canada"
        Case "UK"
            Value = "United Kingdom"
        Case "USA"
            Value = "United States"
        Case "JPN"
            Value = "Japan"
        Case "AUS"
            Value = "Australia"
    End Select
End Sub
```

The end result is that the underlying data is displayed as both descriptive text *and* a picture.

Company	Country
Maple Leaf Systems	 Canada
Her Majesty's Software	 United Kingdom
Software Mart	 United States
Far East Distributors	 Japan
Outback Software, Inc.	 Australia
Northwest Purchasing Agents, Inc.	 United States

Simulating check boxes with in-cell bitmaps

Another useful technique is to employ text-to-picture translations for boolean columns. In this scenario, there are only two allowable values, 0 and -1, and the pictures entered into the **DisplayValue** column represent the unchecked and checked states.



Note that both the **Translate** and **CycleOnClick** check boxes are selected. The former enables automatic data translation; the latter permits the end-user to toggle the value of a cell by clicking it.

ProductName	QuantityPerUnit	Discontinued
Chai	10 boxes x 20 bags	<input type="checkbox"/>
Chang	24 - 12 oz bottles	<input checked="" type="checkbox"/>
Aniseed Syrup	12 - 550 ml bottles	<input type="checkbox"/>
Chef Anton's Cajun Seasoning	48 - 6 oz jars	<input type="checkbox"/>
Chef Anton's Gumbo Mix	36 boxes	<input checked="" type="checkbox"/>
Grandma's Boysenberry Spread	12 - 8 oz jars	<input type="checkbox"/>
Uncle Bob's Organic Dried Pears	12 - 1 lb pkgs.	<input type="checkbox"/>
Northwoods Cranberry Sauce	12 - 12 oz jars	<input type="checkbox"/>
Mishi Kobe Niku	18 - 500 g pkgs.	<input checked="" type="checkbox"/>

Displaying allowable values as radio buttons

If the number of allowable values for a column is relatively small, you may want to consider a radio button presentation. At design time, go to the Values property page and set the **Presentation** property to 1 - Radio Button. Or, in code:

```
With TDBGrid1.Columns("Country").ValueItems
    .Presentation = dbgRadioButton
End With
```

If necessary, adjust the **Width** property of the column and the **RowHeight** property of the grid to accommodate all of the items.

Company	Country
Maple Leaf Systems	<input checked="" type="radio"/> Canada
	<input type="radio"/> United Kingdom
	<input type="radio"/> United States
	<input type="radio"/> Japan
Her Majesty's Software	<input type="radio"/> Canada
	<input checked="" type="radio"/> United Kingdom
	<input type="radio"/> United States
	<input type="radio"/> Japan
Software Mart	<input type="radio"/> Canada
	<input type="radio"/> United Kingdom
	<input checked="" type="radio"/> United States
	<input type="radio"/> Japan
Far East Distributors	<input type="radio"/> Canada
	<input type="radio"/> United Kingdom
	<input type="radio"/> United States
	<input checked="" type="radio"/> Japan

For a given cell, if the underlying data does not match any of the available values, none of the radio buttons will be selected for that cell. However, you can provide a default ValueItem that will be displayed instead.

Property Pages

General | Display | Keyboard | Columns | Layout | Color | Font

Splits | Styles | Values

Column: Column1 (Country)

Translate Presentation: 1 - Radio Button

AnnotatePicture

Validate MaxCombosItems: 5

CycleOnClick

Value	DisplayValue
USA	United States
JPN	Japan
AUS	Australia
	Other

Append

Insert

Remove

Picture...

OK Cancel Apply

In this example, the last **ValueItem** has an empty **Value** property so that any cells where `Country = ""` will

be displayed as Other. Also note that the entire last row in the property page grid is selected. This was done to mark the last ValueItem as the default, which means that Country fields that do not match any of the items will also be displayed as Other.

Selecting a row in the Values property page is equivalent to setting the DefaultItem property of the ValueItems collection at run time:

```
With TDBGrid1.Columns("Country").ValueItems
    .DefaultItem = 5
End With
```

Context-sensitive Help with CellTips

In many Windows applications, when the user points to a toolbar button and leaves the mouse at rest for a short time, a ToolTip window appears with the name of the associated command. You can provide similar context-sensitive help for your users with the **CellTips** property of True DBGrid.

The **CellTips** property determines whether the grid displays a pop-up text window when the cursor is idle. By default, this property is set to 0 - None, and cell tips are not displayed.

If the **CellTips** property is set to either 1 - Anchored or 2 - Floating, the **FetchCellTips** event will be fired whenever the grid has focus and the cursor is idle over a grid cell, record selector, column header, split header, or grid caption. The event will not fire if the cursor is over the scroll bars.

The setting 1 - Anchored aligns the cell tip window with either the left or right edge of the cell. The left edge is favored, but the right edge will be used if necessary in order to display as much text as possible.

Element	Symbol	Atomic No.
Hydrogen	H	1
Helium	He	2
Lithium	Li	3
Beryllium	Be	4
Boron	B	5
Carbon	C	6
Nitrogen	N	7
Oxygen	O	8
Fluorine	F	9
Neon	Ne	10

A soft, silver-white, metallic chemical element, the lightest known metal. It is used in thermonuclear explosives, in metallurgy, etc.

The setting 2 - Floating displays the cell tip window below the cursor, if possible.

Element	Symbol	Atomic No.
Hydrogen	H	1
Helium	He	2
Lithium	Li	3
Beryllium	Be	4
Boron	B	5
Carbon	C	6
Nitrogen	N	7
Oxygen	O	8
Fluorine	F	9
Neon	Ne	10

A soft, silver-white, metallic chemical element, the lightest known metal. It is used in thermonuclear explosives, in metallurgy, etc.

If you do not provide a handler for the **FetchCellTips** event, and the cursor is over a grid cell, the default behavior is to display a text box containing the cell's contents (up to 256 characters). This enables the user to peruse the contents of a cell even if it is not big enough to be displayed in its entirety. You can also program the **FetchCellTips** event to override the default cell text display in order to provide users with context-sensitive help.

A common application of the **FetchCellTips** event is to display the contents of an invisible column that provides additional information about the row being pointed to, as in the following example:

```
' General Declarations
Dim DescCol As TrueDBGrid50.Column

Private Sub Form_Load()
    Set DescCol = TDBGrid1.Columns("Description")
End Sub

Private Sub TDBGrid1_FetchCellTips( _
    ByVal SplitIndex As Integer, _
    ByVal ColIndex As Integer, _
```

```
ByValRowIndex As Long, _  
CellTip As String, _  
ByVal FullyDisplayed As Boolean, _  
ByVal TipStyle As TrueDBGrid50.StyleDisp)  
  
CellTip = DescCol.CellText(TDBGrid1.RowBookmark(RowIndex))  
End Sub
```

You can use the **CellTipsDelay** property to control the amount of time that must elapse before the cell tip window is displayed. You can use the **CellTipsWidth** property to control the width of the cell tip window.

How to Use Splits

In True DBGrid, a *split* is similar to the split window features of products such as Microsoft Excel and Word. You can use splits to present your data in multiple vertical panes. These vertical panes, or splits, can display data in different colors and fonts. They can scroll as a unit or individually, and they can display different sets of columns or the same set. You can also use splits to prevent one or more columns from scrolling. Unlike other grid products, fixed (nonscrolling) columns in True DBGrid do not have to be at the left edge of the grid, but can be at the right edge or anywhere in the middle. You can even have multiple groups of fixed columns within a grid. Splits open up an endless variety of possibilities for presenting data to users of your applications.

Whenever you use True DBGrid, you are always using a split. A grid always contains at least one split, and the default values for the split properties are set so that you can ignore splits until you want to use them. Therefore, you can skip this chapter if you do not need to create and manipulate more than one split within a grid.

You create and manipulate splits by working with **Split** objects and the **Splits** collection. Since an individual column can be visible in one split but hidden in another, each **Split** object maintains its own **Columns** collection. This gives you complete control over the appearance of each split and the columns they contain.

{button ,JI(`,`Referencing_Splits_and_Their_Properties')} Referencing Splits and Their Properties
{button ,JI(`,`Creating_and_Removing_Splits')} Creating and Removing Splits
{button ,JI(`,`Working_with_Columns_in_Splits')} Working with Columns in Splits
{button ,JI(`,`Sizing_and_Scaling_Splits')} Sizing and Scaling Splits
{button ,JI(`,`Creating_and_Resizing_Splits_through_User_Interaction')} Creating and Resizing Splits through User Interaction
{button ,JI(`,`Vertical_Scrolling_and_Split_Groups')} Vertical Scrolling and Split Groups
{button ,JI(`,`Horizontal_Scrolling_and_Fixed_Columns')} Horizontal Scrolling and Fixed Columns
{button ,JI(`,`Navigation_across_Splits')} Navigation across Splits

Referencing Splits and their Properties

A **TDBGrid** object initially contains a single split. If additional splits are created, you can determine or set the current split (that is, the split that has received focus) using the grid's **Split** property:

```
' Read the zero-based index of the current split
Variable% = TDBGrid1.Split

' Set focus to the split with an index equal to Variable%
TDBGrid1.Split = Variable%
```

Each split in a grid is a different view of the same data source, and behaves just like an independent grid. If you create additional splits without customizing any of the split properties, all splits will be identical and each will behave very much like the original grid with one split.

Note that some properties, such as **RecordSelectors** and **MarqueeStyle**, are supported by both the **TDBGrid** and **Split** objects. Three rules of thumb apply to properties that are common to a grid and its splits:

1. When you set or get the property of a **Split** object, you are accessing a specific split, and other splits in the same grid are not affected.
2. When you get the property of a **TDBGrid** object, you are accessing the same property within the current split.
3. When you set the property of a **TDBGrid** object, you are setting the same property within **all** splits.

To understand how these rules work in code, consider a grid with two splits, and assume that the current split index is 1 (that is, the grid's **Split** property returns 1). If you want to determine which marquee style is in use, the following statements are equivalent:

```
marquee% = TDBGrid1.MarqueeStyle
marquee% = TDBGrid1.Splits(1).MarqueeStyle
marquee% = TDBGrid1.Splits(TDBGrid1.Split).MarqueeStyle
```

To change the marquee style to a solid cell border for all of the splits in the grid, you would use:

```
TDBGrid1.MarqueeStyle = dbgSolidCellBorder
```

Note that this statement is equivalent to:

```
TDBGrid1.Splits(0).MarqueeStyle = dbgSolidCellBorder
TDBGrid1.Splits(1).MarqueeStyle = dbgSolidCellBorder
```

Likewise, to set the marquee style of each split to a different value:

```
TDBGrid1.Splits(0).MarqueeStyle = dbgNoMarquee
TDBGrid1.Splits(1).MarqueeStyle = dbgFloatingEditor
```

These rules apply only to a **TDBGrid** object and its associated **Split** objects. No other object pairs possess similar relationships.

{button ,Jl('`,`Split_properties_common_to_TDBGrid')} Split properties common to TDBGrid

{button ,Jl('`,`Split-only_properties_not_supported_by_TDBGrid')} Split-only properties not supported by TDBGrid

Split properties common to TDBGrid

The following properties, which are supported by both **Split** and **TDBGrid** objects, adhere to the rules described in the preceding section:

<u>AllowColMove</u>	Enables interactive column movement
<u>AllowColSelect</u>	Enables interactive column selection
<u>AllowRowSelect</u>	Enables interactive row selection
<u>AllowRowSizing</u>	Enables interactive row resizing
<u>AlternatingRowStyle</u>	Controls whether even/odd styles are applied to a split
<u>BackColor</u>	Sets/returns the background color
<u>CaptionStyle</u>	Controls the caption style for a split
<u>Columns</u>	Returns a collection of column objects for a split
<u>CurrentCellVisible</u>	Sets/returns modification status of the current cell
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditorStyle</u>	Controls the editor style for a split
<u>EvenRowStyle</u>	Controls the row style for even numbered rows
<u>ExtendRightColumn</u>	Sets/returns extended right column for a split
<u>FetchRowStyle</u>	Controls whether the FetchRowStyle event will be fired
<u>FirstRow</u>	Bookmark of row occupying first display line
<u>Font</u>	Specifies the overall font for a split
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading font for a split
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a split
<u>HighlightRowStyle</u>	Controls the marquee style when set to Highlight Row
<u>HScrollHeight</u>	Returns the horizontal scroll bar height, if present
<u>InactiveBackColor</u>	Sets/returns the inactive heading background color
<u>InactiveForeColor</u>	Sets/returns the inactive heading foreground color
<u>InactiveStyle</u>	Controls the inactive heading style for a split
<u>LeftCol</u>	Returns the leftmost visible column
<u>MarqueeStyle</u>	Sets/returns marquee style for a split
<u>OddRowStyle</u>	Controls the row style for odd numbered rows
<u>RecordSelectors</u>	Shows/hides selection panel at left border
<u>ScrollBars</u>	Sets/returns scroll bar style for a split
<u>SelectedBackColor</u>	Sets/returns the selected row background color
<u>SelectedForeColor</u>	Sets/returns the selected row foreground color
<u>SelectedStyle</u>	Controls the selected row and column style for an object
<u>SelEndCol</u>	Sets/returns rightmost selected column
<u>SelStartCol</u>	Sets/returns leftmost selected column
<u>Style</u>	Controls the normal style for an object
<u>VScrollWidth</u>	Returns the vertical scroll bar width, if present

NOTE: The **Caption** property is not included in this list, even though it is supported by both objects. Since grids and splits maintain separate caption bars, setting the **Caption** property of the grid does not apply the same string to each split caption.

Split-only properties not supported by TDBGrid

The following properties are supported only by **Split** objects. Therefore, to apply a value to the entire grid, you need to explicitly set the value for each split individually.

<u>AllowFocus</u>	Allows cells within a split to receive focus
<u>AllowSizing</u>	Enables interactive resizing for a split
<u>Index</u>	Returns the ordinal index of a split
<u>Locked</u>	If true, data entry prohibited for a split
<u>ScrollGroup</u>	Used to synchronize vertical scrolling between splits
<u>Size</u>	Sets/returns split width according to SizeMode
<u>SizeMode</u>	Controls whether a split is scalable or fixed size

Creating and Removing Splits

At design time, you can create and remove splits using the grid's visual editing menu. Please refer to [Visual Editing Mode](#) for details.

At run time, you can create and remove splits using the **Splits** collection's **Add** and **Remove** methods. Each method takes a zero-based split index:

```
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(0) ' Create a split with index 0
TDBGrid1.Splits.Remove 1      ' Remove the split with index 1
```

The new Splits(0) object is "cloned" from the old Splits(0) object. Both splits will have the same property values after the **Add** method executes. The old Splits(0) becomes Splits(1), the old Splits(1) becomes Splits(2), and so on.

You can determine the number of splits in a grid using the **Splits** collection's **Count** property:

```
' Set variable equal to the number of splits in TDBGrid1
variable = TDBGrid1.Splits.Count
```

You can iterate through all splits using the **Count** property, for example:

```
For n = 0 To TDBGrid1.Splits.Count - 1
    Debug.Print TDBGrid1.Splits(n).Caption
Next n
```

Of course, a more efficient way to code this would be to use a **For Each...Next** loop:

```
Dim S As TrueDBGrid50.Split
For Each S In TDBGrid1.Splits
    Debug.Print S.Caption
Next
```

The **Count** property is primarily used to append a new split to the end of the **Splits** collection, as follows:

```
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(TDBGrid1.Splits.Count)
```

The new **Split** object will inherit all of its properties from the last object in the collection.

Working with Columns in Splits

Each split in a True DBGrid control maintains its own **Columns** collection. This provides tremendous flexibility for controlling the look and behavior of individual splits. The grid is connected to a single data source, so the splits just present different views of the same data. Therefore, the **Columns** collection in each split contains the same number of columns and the columns are bound to the same data fields.

Note that some **Column** object properties, such as **Caption** and **DataField**, have the same value in each split. These properties are said to be *global*. For example, given a grid with two splits, the following code will always print the same values for a given column index *n*:

```
Debug.Print TDBGrid1.Splits(0).Columns(n).Caption
Debug.Print TDBGrid1.Splits(1).Columns(n).Caption
Debug.Print TDBGrid1.Columns(n).Caption
```

More importantly, if you set any of the global properties of a column within a particular split, that property will be set to the same value for all splits. For example, the following code will append a column to all splits (not just the first one) and bind the columns to the same database field (LastName).

```
Dim Cols As TrueDBGrid50.Columns
Set Cols = TDBGrid1.Splits(0).Columns

' Append a column to the end of the Columns collection
Dim C As TrueDBGrid50.Column
Set C = Cols.Add(Cols.Count)

' Set the DataField property of the newly created column
C.DataField = "LastName"
```

However, the values of other **Column** object properties, such as **Visible** and **BackColor**, may vary from split to split. These properties are said to be split-specific. For example, a column created at run time is not visible by default. Thus, the LastName column created in the preceding example is invisible in all splits. The following code makes it visible in the second split:

```
TDBGrid1.Splits(1).Columns("LastName").Visible = True
```

Since **Visible** is a split-specific property, the LastName column remains invisible in other splits.

{button ,JI(`,`Global_properties_and_methods_of_Column_object')} Global properties and methods of Column object

{button ,JI(`,`Split-specific_properties_of_Column_object')} Split-specific properties of Column object

Global properties and methods of Column object

The following **Column** object properties are global; that is, they always have the same value in each split:

<u>Caption</u>	Sets/returns column heading text
<u>ColIndex</u>	Returns the ordinal position of a column
<u>DataChanged</u>	Sets/returns modification status of a column in the current row
<u>DataField</u>	Data table field name for a column
<u>DataWidth</u>	Maximum number of characters available for column input
<u>DefaultValue</u>	Default value for new column data
<u>DropDown</u>	Sets the name of a TDBDropDown control for a column
<u>EditMask</u>	Input mask string for a column
<u>EditMaskUpdate</u>	Controls whether masked data is used for updates
<u>NumberFormat</u>	Data formatting string for a column
<u>Text</u>	Sets/returns displayed column text for the current cell
<u>Top</u>	Returns top column border in container coordinates
<u>Value</u>	Sets/returns underlying data value for the current row
<u>ValueItems</u>	Contains a collection of ValueItems for a column

The following **Column** object methods are also global:

<u>CellText</u>	Returns displayed text for any visible row
<u>CellValue</u>	Returns underlying value for any visible row

Split-specific properties of Column object

The following **Column** object properties are split-specific; that is, they may have different values across splits:

<u>Alignment</u>	Specifies horizontal text alignment
<u>AllowFocus</u>	Controls whether a column can receive focus
<u>AllowSizing</u>	Enables interactive resizing for a column
<u>BackColor</u>	Sets/returns the background color
<u>Button</u>	Controls whether a button appears within the current cell
<u>ButtonPicture</u>	Sets/returns the bitmap used for the in-cell button
<u>CellTop</u>	Returns top column border, adjusted for multiple lines
<u>ColIndex</u>	Returns the ordinal position of a column
<u>DividerStyle</u>	Divider style for right column border
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditorStyle</u>	Controls the editor style for a column
<u>FetchStyle</u>	Controls whether the FetchCellStyle event fires for a column
<u>Font</u>	Specifies the overall font for a column
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadAlignment</u>	Specifies column heading alignment
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading font for a column
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a column
<u>Left</u>	Returns column left border in container coordinates
<u>Locked</u>	If true, data entry prohibited for a column
<u>Order</u>	Sets/returns the display position of a column
<u>Style</u>	Controls the normal style for a column
<u>Visible</u>	Shows/hides a column
<u>Width</u>	Sets/returns column width in container coordinates
<u>WrapText</u>	True if cell text is word wrapped

Sizing and Scaling Splits

True DBGrid gives you full control over the size and scaling of individual splits. You can configure a split to occupy an exact width, hold a fixed number of columns, or adjust its size proportionally in relation to other splits. If you are just starting out with True DBGrid, you can use splits in a variety of ways without having to master all of the details.

At run time, the actual size of a **Split** object depends upon its **Size** and **SizeMode** properties. The **SizeMode** property specifies the unit of measurement; the **Size** property specifies the number of units. True DBGrid supports three different sizing modes for splits, as determined by the setting of the **SizeMode** property:

- | | |
|-----------------------|--|
| 0 - Scalable | Size denotes relative width in relation to other splits |
| 1 - Exact | Size specifies a fixed width in container coordinates |
| 2 - Number of Columns | Size specifies a fixed number of columns |

In code, you can use the constants `dbgScalable`, `dbgExact`, and `dbgNumberOfColumns` to refer to these settings.

A scalable split uses the value of its **Size** property to determine the percentage of space the split will occupy. For any scalable split, the percentage is determined by dividing its **Size** value by the sum of the **Size** values of all other scalable splits. Thus, you can consider the **Size** property of a scalable split to be the numerator of a fraction, the denominator of which is the sum of the scalable split sizes. Scalable splits compete for the space remaining after nonscalable splits have been allocated. By default, all splits are scalable, so they compete for the entire grid display region. **SizeMode** is always 0 - Scalable when a grid contains only one split.

An exact split uses the value of its **Size** property as its fixed width in container coordinates. Exact splits will be truncated if they will not fit within the horizontal grid boundaries. This mode is not applicable when a grid contains only one split.

A fixed-column split uses the **Size** property to indicate the exact number of columns that should always be displayed within the split. These splits automatically reconfigure the entire grid if the size of the displayed columns changes (either by code or user interaction), or if columns in the split are scrolled horizontally so that the widths of the displayed columns are different. This mode is primarily used to create fixed columns that do not scroll horizontally. However, it can be used for a variety of other purposes as well. This mode is not applicable when a grid contains only one split.

Note that when there is only one split (the grid's default behavior), the split spans the entire width of the grid, the **SizeMode** property is always 0 - Scalable, and the **Size** property is always 1. Setting either of these properties has no effect when there is only one split. If there are multiple splits, and you then remove all but one, the **SizeMode** and **Size** properties of the remaining split automatically revert to 0 and 1, respectively.

By default, the **SizeMode** property for a newly created split is 0 - Scalable, and the **Size** property is set to 1. For example, if you create two additional splits with the following code:

```
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(0) ' Create a Split at the left
Set S = TDBGrid1.Splits.Add(0) ' Create another
```

the resulting grid display will look like this.

First	Last	First	Last	First	Last
▶ Isaac	Alb	▶ Isaac	Alb	▶ Isaac	
Johann Sebastian	Ba	Johann Sebastian	Ba	Johann Sebastian	
Samuel	Bar	Samuel	Bar	Samuel	
Bela	Bart	Bela	Bart	Bela	
Ludwig van	Be	Ludwig van	Be	Ludwig van	
Alban	Ber	Alban	Ber	Alban	
Luciano	Ber	Luciano	Ber	Luciano	
Hector	Ber	Hector	Ber	Hector	
Leonard	Ber	Leonard	Ber	Leonard	
Georges	Biz	Georges	Biz	Georges	
Ernest	Bl	Ernest	Bl	Ernest	

$\frac{1}{3}$
 $\frac{1}{3}$
 $\frac{1}{3}$

Notice that each split occupies 1/3 of the total grid space. This is because there are three scalable splits, and each split has a **Size** of 1. If you change the sizes of the splits to 2, 2, and 3, respectively:

```
TDBGrid1.Splits(0).Size = 2 ' Change relative size to 2
TDBGrid1.Splits(1).Size = 2 ' Change relative size to 2
TDBGrid1.Splits(2).Size = 3 ' Change relative size to 3
```

the resulting grid display will look like this.

First	Last	First	Last	First	Last
▶ Isaac	Albeniz	▶ Isaac	Bach	▶ Isaac	Barber
Johann Sebastian	Bartok	Johann Sebastian	Beethoven	Johann Sebastian	Berg
Samuel	Ber	Samuel	Ber	Samuel	Ber
Bela	Ber	Bela	Ber	Bela	Ber
Ludwig van	Ber	Ludwig van	Ber	Ludwig van	Ber
Alban	Ber	Alban	Ber	Alban	Ber
Luciano	Ber	Luciano	Ber	Luciano	Ber
Hector	Ber	Hector	Ber	Hector	Ber
Leonard	Ber	Leonard	Ber	Leonard	Ber
Georges	Bizet	Georges	Bizet	Georges	Bizet
Ernest	Bl	Ernest	Bl	Ernest	Bl

$\frac{2}{7}$
 $\frac{2}{7}$
 $\frac{3}{7}$

Notice the sum of the split sizes (2+2+3) is 7, so the size of each split is a fraction with the numerator being the value of its **Size** property and a denominator of 7.

When a split's **SizeMode** is set to 1 - Exact, that split receives space before the other splits. This behavior is somewhat more complex, but understanding how scalable splits work is helpful. For example, assume that splits are set in the following way:

```
Split0.SizeMode = dbgScalable
Split0.Size = 1

Split1.SizeMode = dbgExact
Split1.Size = 2500

Split2.SizeMode = dbgScalable
Split2.Size = 2
```

After configuring the splits in this way, the resulting grid display will look like this.

First	First	Last	First	Last
▶ Isaac	▶ Isaac	Albeniz	▶ Isaac	Alben
Johann Seba	Johann Sebastian	Bach	Johann Sebastian	Bach
Samuel	Samuel	Barber	Samuel	Barbe
Bela	Bela	Bartok	Bela	Bartol
Ludwig van	Ludwig van	Beethove	Ludwig van	Beeth
Alban	Alban	Berg	Alban	Berg
Luciano	Luciano	Berio	Luciano	Berio
Hector	Hector	Berlioz	Hector	Berlio:
Leonard	Leonard	Bernsteir	Leonard	Berns
Georges	Georges	Bizet	Georges	Bizet
Ernest	Ernest	Bloch	Ernest	Bloch

1/3
2500 twips
2/3

The fixed-size split in the middle (Split1) is configured to exactly 2500 twips, and the remaining splits compete for the space remaining in the grid. Since the remaining splits are both scalable splits, they divide the remaining space among themselves according to the percentages calculated using their **Size** property values. So, the leftmost split occupies 1/3 of the **remaining** space, and the rightmost split occupies 2/3.

Splits with **SizeMode** set to 2 - Number of Columns behave almost identically to exact splits, except their size is determined by the width of an integral number of columns. The width, however, is dynamic, so resizing the columns or scrolling so that different columns are in view will cause the entire grid to reconfigure itself.

Avoid creating a grid with no scalable splits. Although True DBGrid handles this situation, it is difficult to work with a grid configured in this way. For example, if no splits are scalable, all splits will have an exact size, which may not fill the entire horizontal width of the grid. If the total width of the splits is too short, True DBGrid displays a "null-zone" where there are no splits. If the total width of the splits is wider than the grid, then True DBGrid will show only the separator lines for the splits that cannot be shown.

Creating and Resizing Splits through User Interaction

You can always create and resize splits in code. However, you can also let your users create and resize splits interactively by setting the **AllowSizing** property of a split to True. By default, the **AllowSizing** property is False, and users are prevented from creating and resizing splits.

A typical grid with **AllowSizing** set to False is shown in the following figure. Notice that there is no split box at the left edge of the horizontal scroll bar.

	First	Last	Country	Birth ▲
▶	Isaac	Albeniz	Spain	05/29/1860
	Bela	Bartok	Hungary	03/25/1881
	Alban	Berg	Austria	02/09/1885
	Ernest	Bloch	Switzerland	07/24/1880
	Benjamin	Britten	England	11/22/1913
	Max	Bruch	Germany	01/06/1838
	Claude	Debussy	France	08/22/1862
	Edward	Elgar	England	06/02/1857
	Manuel de	Falla	Spain	11/23/1876
	Gabriel	Faure	France	05/12/1845 ▼

If you set the split's **AllowSizing** property to True:

```
TDBGrid1.Splits(0).AllowSizing = True
```

a split box will appear at the left edge of the horizontal scroll bar, and the user will be able to create new splits at run time.

	First	Last	Country	Birth ▲
▶	Isaac	Albeniz	Spain	05/29/1860
	Bela	Bartok	Hungary	03/25/1881
	Alban	Berg	Austria	02/09/1885
	Ernest	Bloch	Switzerland	07/24/1880
	Benjamin	Britten	England	11/22/1913
	Max	Bruch	Germany	01/06/1838
	Claude	Debussy	France	08/22/1862
	Edward	Elgar	England	06/02/1857
	Manuel de	Falla	Spain	11/23/1876
	Gabriel	Faure	France	05/12/1845 ▼

When the user points to the split box, the pointer will turn into a double vertical bar with a down arrow, which the user can drag to the right to create a new split, as shown in the next figure.

	First	Last	First	Last	Cou ▲
▶	Isaac	Albeniz	▶ Isaac	Albeniz	Spai
	Bela	Bartok		Bela	Hun
	Alban	Berg		Alban	Aust
	Ernest	Bloch		Ernest	Swit
	Benjamin	Britten		Benjamin	Engl
	Max	Bruch		Max	Gerr
	Claude	Debussy		Claude	Fran
	Edward	Elgar		Edward	Engl
	Manuel de	Falla		Manuel de	Spai
	Gabriel	Faure		Gabriel	Fran ▼

The new split will inherit its properties from the original split. The **SizeMode** properties of both splits will be automatically set to 0 - Scalable, regardless of the **SizeMode** of the original split. The **Size** properties of both splits will be set to the correct ratio of the splits' sizes. The values of the **Size** properties may end up

being rather large. This is because True DBGrid needs to choose the least common denominator for the total split size, and the user may drag the pointer to an arbitrary position.

Note that both splits' **AllowSizing** properties are now True, and the divider between them is a double line, which indicates that the splits' sizes are now adjustable. If the user points to the split box between the two splits, the pointer will turn into a double vertical bar with horizontal arrows. The user can drag this pointer to the left or right to adjust the relative sizes of the splits.

If you set **AllowSizing** to False for either split, the user will no longer be able to adjust the split sizes. Suppose that you disable sizing for the first split:

```
TDBGrid1.Splits(0).AllowSizing = False
```

The split box at the left edge of the horizontal scroll bar in the first split will disappear and the divider between the two splits will turn into a solid line. This means that the user will no longer be able to create a new split from the first split, or adjust the sizes of either split. However, since the split box at the left edge of the second split still exists, the user can now create new splits by pointing to this split box and dragging the pointer to the right.

	First	Last		First	Last	Cou
▶	Isaac	Albeniz	▶	Isaac	Albeniz	Spai
	Bela	Bartok		Bela	Bartok	Hun
	Alban	Berg		Alban	Berg	Aust
	Ernest	Bloch		Ernest	Bloch	Swit
	Benjamin	Britten		Benjamin	Britten	Engl
	Max	Bruch		Max	Bruch	Gerr
	Claude	Debussy		Claude	Debussy	Fran
	Edward	Elgar		Edward	Elgar	Engl
	Manuel de	Falla		Manuel de	Falla	Spai
	Gabriel	Faure		Gabriel	Faure	Fran

To summarize:

- You can always create or resize splits in code, but the **AllowSizing** property controls whether users can create or resize splits interactively at run time.
- The user can resize the relative sizes of two splits only if both splits' **AllowSizing** properties are True. When the user completes a resize operation, the total size of the two splits remains unchanged, but the **SizeMode** properties of both splits will automatically be set to 0 - Scalable regardless of their previous settings. The **Size** properties of the two splits will be set to reflect the ratio of their new sizes.
- The user can create a new split by dragging the split box to the right, as long as both of the following conditions are met. First, the **AllowSizing** property of the split to the right of the split box must be True. Second, the **AllowSizing** property of the split to the left of the split box must be False, or the split box must belong to the leftmost split. The total size of the new split and the parent split will be equal to the original size of the parent split. The **SizeMode** properties of the two splits will be automatically set to 0 - Scalable, and the **Size** properties of the two splits will be set to reflect the correct ratio of their new sizes.

Vertical Scrolling and Split Groups

By default, the grid has only one split, with split index 0, and its **ScrollBars** property is set to 4 - Automatic. That is, the horizontal or vertical scroll bar will appear as necessary depending upon the column widths and the number of data rows available. The default split's **ScrollGroup** property is 1. Splits having the same **ScrollGroup** property setting will scroll vertically together. When a new split is created, it will inherit both the **ScrollBars** and **ScrollGroup** properties from the parent split. If all of the splits belonging to the same **ScrollGroup** have their **ScrollBars** properties set to 4 - Automatic, then True DBGrid will display the vertical scroll bar only at the rightmost split of the scroll group. Manipulating this scroll bar will cause all splits in the same scroll group to scroll simultaneously.

For example, if you create two additional splits with the following code:

```
Dim S As TrueDBGrid50.Split
Set S = TDBGrid1.Splits.Add(0) ' Create a Split at the left
Set S = TDBGrid1.Splits.Add(0) ' Create another
```

the resulting grid display will look like this.

First	Last	First	Last	First	Last
Isaac	Alb	Isaac	Alb	Isaac	
Johann Sebastian	Bar	Johann Sebastian	Bar	Johann Sebastian	
Samuel	Bar	Samuel	Bar	Samuel	
Bela	Bar	Bela	Bar	Bela	
Ludwig van	Bel	Ludwig van	Bel	Ludwig van	
Alban	Bel	Alban	Bel	Alban	
Luciano	Bel	Luciano	Bel	Luciano	
Hector	Bel	Hector	Bel	Hector	
Leonard	Bel	Leonard	Bel	Leonard	
Georges	Biz	Georges	Biz	Georges	
Ernest	Blc	Ernest	Blc	Ernest	

All three splits will have the same **ScrollBars** and **ScrollGroup** properties of 4 and 1, respectively. However, only one vertical scroll bar will be displayed, within the rightmost split. When the user operates this scroll bar, all three splits will scroll simultaneously.

You can change the **ScrollGroup** property of splits to create split groups that scroll independently. In the preceding example, setting the **ScrollGroup** property of the middle split to 2 creates a new scroll group:

```
TDBGrid1.Splits(1).ScrollGroup = 2 ' Create a new scroll group
```

After this statement executes, scrolling the middle split will not disturb the others.

First	Last	First	Last	First	Last
Bela	Bar	Isaac		Bela	
Ludwig van	Bel	Johann Sebastian		Ludwig van	
Alban	Bel	Samuel		Alban	
Luciano	Bel	Bela		Luciano	
Hector	Bel	Ludwig van		Hector	
Leonard	Bel	Alban		Leonard	
Georges	Biz	Luciano		Georges	
Ernest	Blc	Hector		Ernest	
Alexander	Bo	Leonard		Alexander	
Johannes	Br	Georges		Johannes	
Benjamin	Rib	Ernest		Benjamin	

Note that the middle split now contains a vertical scroll bar. This scroll bar operates only on the middle split, since it is the only one with its **ScrollGroup** property equal to 2. The vertical scroll bar in the rightmost split now controls the leftmost and rightmost splits only. It no longer affects the middle split.

You can create as many split groups as necessary by setting the **ScrollGroup** properties of the splits to different values. You can also explicitly control whether scroll bars will be displayed in a split by setting its **ScrollBars** property.

A common application of this feature is to create two independent split groups so that users can compare field values from different records by scrolling each split to view a different set of rows.

Horizontal Scrolling and Fixed Columns

Horizontal scrolling is independent for each split. Often, you need to prevent one or more columns from scrolling horizontally so that they will always be in view. True DBGrid provides you with an easy way to keep any number of columns from scrolling at any location within the grid (even in the middle!) by setting a few split properties.

As an example, assume that you have a grid with three splits. The following code will "fix" columns 0 and 1 in the middle split:

```
' Hide all columns in Splits(1) except for columns 0 and 1
Dim Cols As TrueDBGrid50.Columns
Dim C As TrueDBGrid50.Column
Set Cols = TDBGrid1.Splits(1).Columns
For Each C In Cols
    C.Visible = False
Next C
Cols(0).Visible = True
Cols(1).Visible = True

' Configure Splits(1) to display exactly two columns, and
' disable resizing
With TDBGrid1.Splits(1)
    .SizeMode = dbgNumberOfColumns
    .Size = 2
    .AllowSizing = False
End With
```

Usually, if you keep columns 0 and 1 from scrolling in one split, you will want to make them invisible in the other splits:

```
' Make columns 0 and 1 invisible in splits 0 and 2
Dim Cols As Columns
Set Cols = TDBGrid1.Splits(0).Columns
Cols(0).Visible = False
Cols(1).Visible = False
Set Cols = TDBGrid1.Splits(2).Columns
Cols(0).Visible = False
Cols(1).Visible = False
```

Navigation across Splits

Navigation across splits is controlled by the grid's **TabAcrossSplits** property and each split's **AllowFocus** property. Navigation across splits is best discussed with grid navigation as a whole. For more information, please refer to [Run Time Interaction](#).

How to Use Styles

True DBGrid uses a style model similar to that of Microsoft Excel to simplify the task of customizing a grid's appearance. A **Style** object is a named combination of font, color, and formatting information comprising the following properties:

<u>Alignment</u>	Specifies the horizontal text alignment
<u>BackColor</u>	Controls the background color
<u>Font</u>	Specifies the typeface, size, and other text characteristics
<u>ForeColor</u>	Controls the foreground color
<u>Locked</u>	Disallows in-cell editing when true
<u>Name</u>	Returns the programmer-specified style name
<u>Parent</u>	Returns the object from which a style inherits
<u>WrapText</u>	Enables wordwrapping for cell text

Once you have defined a style, you can apply it to a **TDBGrid**, **Column**, or **Split** object to control the appearance of all cells within that object. If you subsequently change a particular characteristic of a style, such as its background color, then any object to which that style has been applied will automatically change to reflect the new characteristic.

At design time, the Styles property page is used to create, modify, and delete style objects. You can then refer to these objects in code at run time and apply them to individual grid elements or the entire grid. You can also create new styles in code and use them just as if you had created them at design time.

Initially, all grids contain seven built-in styles as follows:

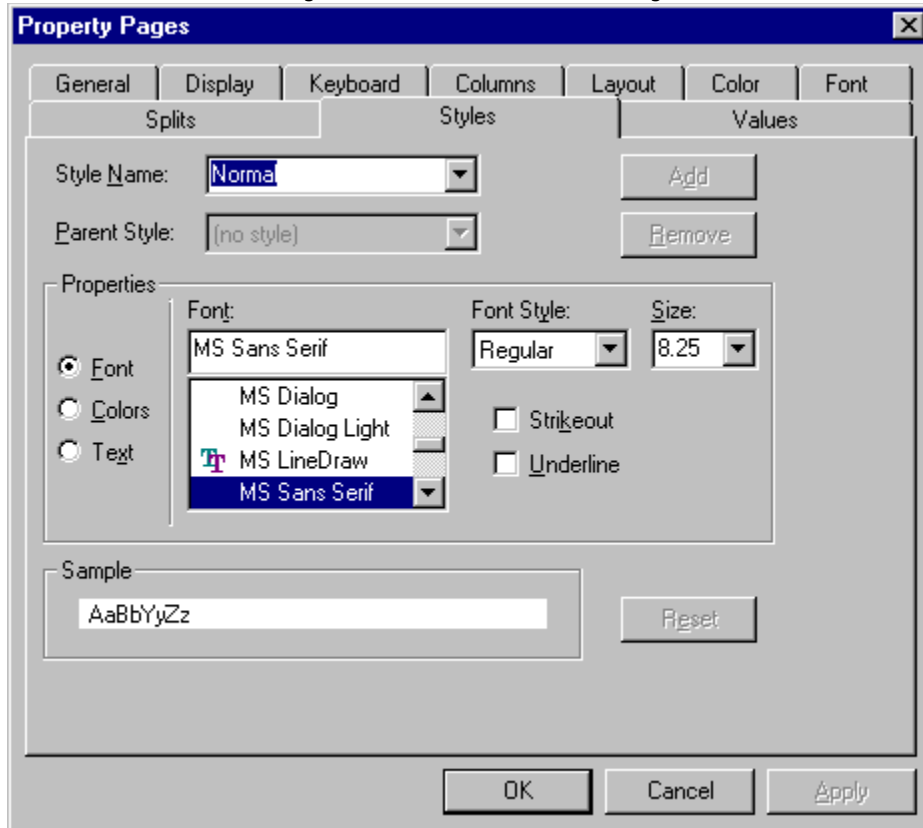
Normal	The root style
Heading	Normal + <u>BackColor</u> : System Button Face + <u>ForeColor</u> : System Button Text
Selected	Normal + <u>BackColor</u> : System Highlight + <u>ForeColor</u> : System Highlight Text
Caption	Heading + <u>Alignment</u> : Center
HighlightRow	Normal + <u>BackColor</u> : System Window Text + <u>ForeColor</u> : System Window Background
EvenRow	Normal + <u>BackColor</u> : Light Cyan
OddRow	Normal

You can redefine any style, whether it is a built-in style or one that you created. In this version of True DBGrid, styles are local to a grid and cannot be copied from one grid to another.

```
{button ,JI('',`Defining_Styles_at_Design_Time')} Defining Styles at Design Time
{button ,JI('',`Defining_Styles_at_Run_Time')} Defining Styles at Run Time
{button ,JI('',`Applying_Styles_at_Run_Time')} Applying Styles at Run Time
{button ,JI('',`Additional_Style_Properties')} Additional Style Properties
{button ,JI('',`Introduction_to_Cell_Styles')} Introduction to Cell Styles
{button ,JI('',`Specifying_Cell_Status_Values')} Specifying Cell Status Values
{button ,JI('',`Applying_Cell_Styles_by_Status')} Applying Cell Styles by Status
{button ,JI('',`Applying_Cell_Styles_by_Contents')} Applying Cell Styles by Contents
{button ,JI('',`Applying_Cell_Styles_by_Custom_Criteria')} Applying Cell Styles by Custom Criteria
```

Defining Styles at Design Time

Use the Styles property page to create, modify, and delete styles at design time. The Styles property page can be accessed by choosing the **Properties...** menu item from the grid control's context menu, which can be activated with the right mouse button. The following illustration shows the built-in Normal style.

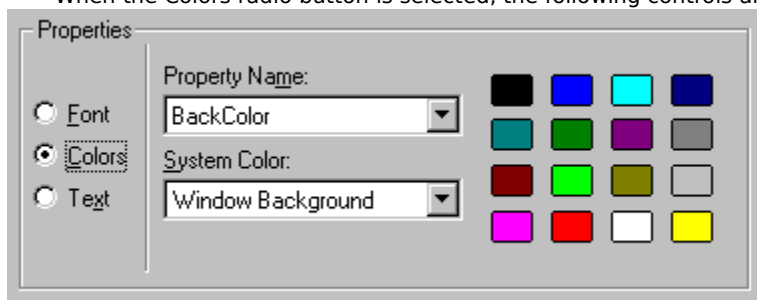


Initially, the Styles property page contains the following controls:

- | | |
|---------------------|--|
| Style Name | This combo box specifies which style is being edited. You can either select an existing style from the drop-down list or type in the name of a new or existing style. This control corresponds to the style's Name property. |
| Parent Style | This combo box specifies the name of the style from which the selected style inherits. For styles with no parent, such as the built-in Normal style, this combo box displays (no style) as shown in the preceding illustration. This control corresponds to the style's Parent property. |
| Add | This button creates a new style with the name specified in the Style Name combo box. Style names must be unique, so this button will be disabled if the text in the Style Name combo box matches the name of an existing style. This button corresponds to the Add method of the Styles collection. |
| Remove | This button deletes the chosen style. The built-in styles cannot be deleted, so this button will be disabled when one of the built-in styles is chosen. This button will also be disabled when you have entered text into the Style Name combo box that does not match the name of an existing style. This button corresponds to the Remove method of the Styles collection. |

Reset	This button resets the chosen style so that it inherits all of its font, color, and formatting attributes from its parent, if any. For styles with no parent, the Reset button causes the selected style to revert to the default settings held by the Normal style when the grid was first created. This button corresponds to the Reset method.
Font, Colors, Text	These radio buttons govern which controls appear in the Properties frame. Since not all controls will fit on the Styles property page at one time, these radio buttons are provided so that you can easily switch between control groups. Whenever the Styles property page is first displayed, the font controls are shown.
Font	This combo box specifies the typeface name of the chosen style's font. This control corresponds to the Name property of the Font object associated with the style.
Font Style	This combo box specifies the attributes of the chosen style's font. This control corresponds to the Bold and Italic properties of the Font object associated with the style.
Size	This combo box specifies the point size of the chosen style's font. This control corresponds to the Size property of the Font object associated with the style.
Strikeout	This check box specifies whether the chosen style's font has the strikeout attribute enabled. This control corresponds to the Strikethrough property of the Font object associated with the style.
Underline	This check box specifies whether the chosen style's font has the <u>underline</u> attribute enabled. This control corresponds to the Underline property of the Font object associated with the style.
Sample	This static area displays sample text that shows how a grid cell will appear when the chosen style is applied. Whenever you change a font, color, or alignment setting, the Sample area is updated so that you can see the results of the change before committing it with either the OK or Apply button.

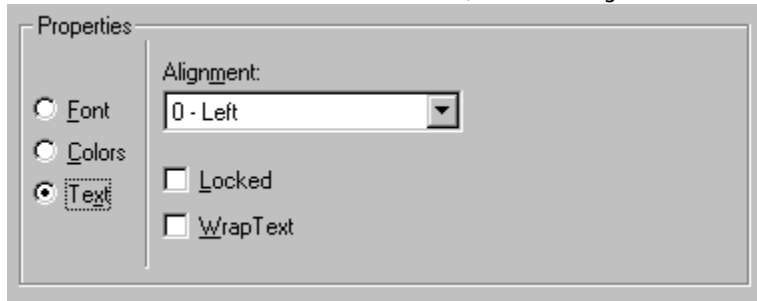
When the Colors radio button is selected, the following controls are displayed in the Properties frame.



Property Name	This combo box specifies the name of the style property being modified. It always contains two items corresponding to the style's BackColor and ForeColor properties.
System Color	This combo box allows you to specify a system color value (instead of a physical color) for the property shown in the Property Name combo box. Whenever you select an item in this combo box, any color button selection is removed.
Color Buttons	These 16 buttons allow you to specify a physical color value (instead of a system color) for the property shown in the Property Name combo box.

Whenever you click one of these buttons, its border is highlighted and any system color selection is cleared.

When the Text radio button is selected, the following controls are displayed in the Properties frame.



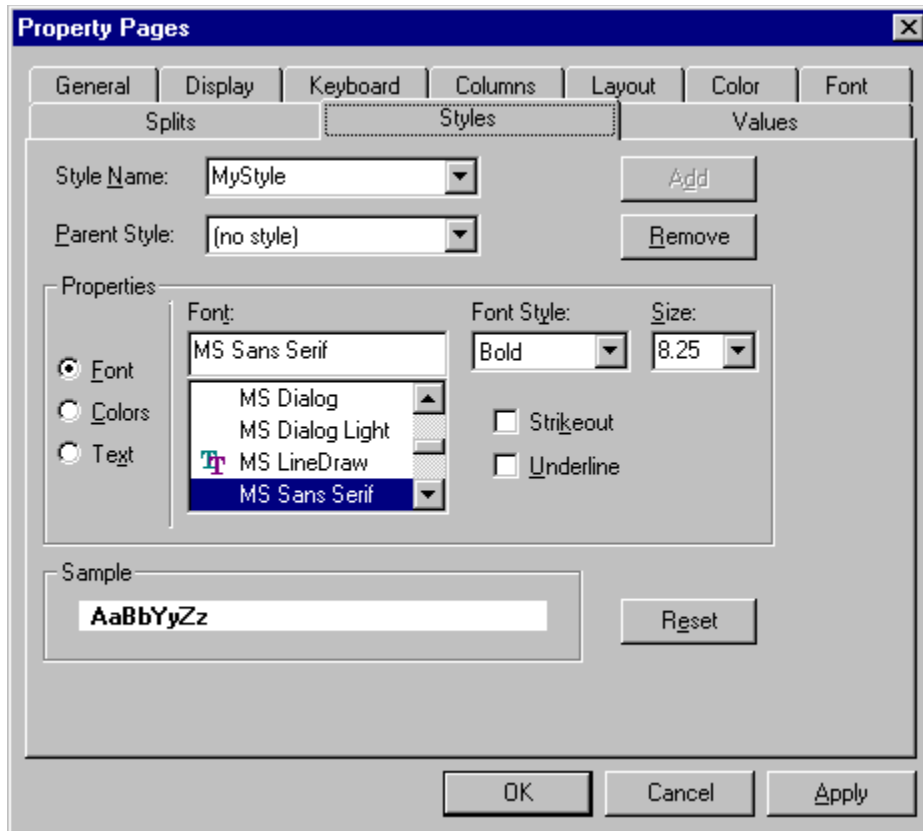
- | | |
|------------------|--|
| Alignment | This combo box specifies the horizontal text alignment (left, center, right, or general) for the chosen style. This control corresponds to the style's <u>Alignment</u> property. |
| Locked | This check box specifies whether the chosen style inhibits in-cell editing. If checked, editing is disallowed; if unchecked, editing is permitted. This control corresponds to the style's <u>Locked</u> property. |
| WrapText | This check box specifies whether the chosen style causes cell text to be word wrapped. If checked, a line break occurs before words that would otherwise be partially displayed; if unchecked, no line break occurs and text is clipped at the cell's right edge. This control corresponds to the style's <u>WrapText</u> property. |

As an example, suppose that you want to create a style named MyStyle with the following attributes:

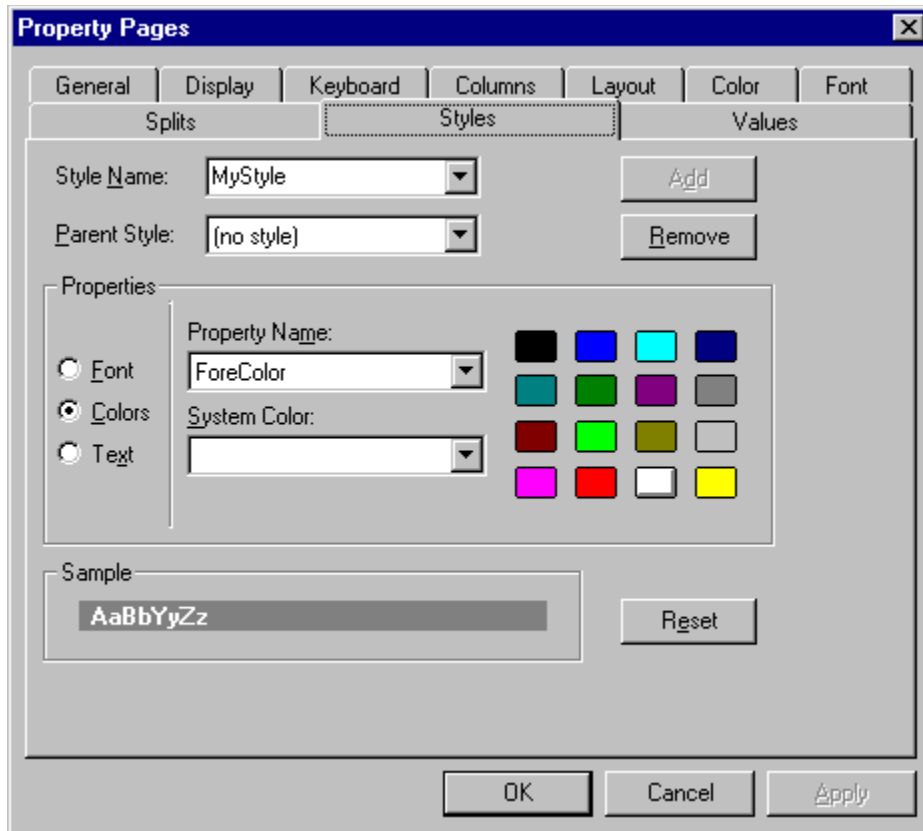
- | | |
|------------------|----------------------------------|
| Font | MS Sans Serif, Bold, 8.25 points |
| BackColor | Dark Gray |
| ForeColor | White |

First, select the Normal style from the StyleName combo box, then type MyStyle into the text portion of the same combo box. Note that the Add button is now enabled. Click the Add button to create the style. Now, the Add button is disabled again, but the Remove and Reset buttons are enabled since the newly created style is now current.

Next, select the Bold item from the Font Style combo box. Note that the Sample area now displays its text in boldface.



Now, select the Colors radio button to bring the color selection controls into view. The **BackColor** property is initially set to the system Window Background color as determined by your Control Panel settings. Click the dark gray button to change the **BackColor** property of MyStyle. Note that this clears the System Color combo box, highlights the border of the dark gray button, and updates the Sample area. To change the **ForeColor** property, select the ForeColor item from the Property Name combo box and click the white button. Finally, to save your changes to MyStyle, click the **Apply** button. The Styles property page should now look like this.



You can also click the **OK** button to save your changes and close the property page dialog. Note that the **Cancel** button will not undo the actions taken by the Parent Style combo box or the Add, Remove, and Reset buttons. Once a style has been created, it can only be deleted with the Remove button. Similarly, once a style has been deleted, the **Cancel** button will not bring it back.

Defining Styles at Run Time

The previous section demonstrated how to create a style at design time using the Styles property page. This section describes how to achieve the same result in code at run time. Let's begin with a discussion of how to work with collections in general and styles in particular.

Each **TDBGrid** control maintains a collection of styles that can be applied to its own elements. The **Styles** property allows you to access this collection at run time. For example, the following loop prints the names of all built-in and user-defined styles to the Visual Basic debug window:

```
Dim S As TrueDBGrid50.Style
For Each S In TDBGrid1.Styles
    Debug.Print S.Name
Next S
```

If you have not defined any styles of your own, then this loop prints the names of the seven built-in styles:

```
Normal
Heading
Selected
Caption
HighlightRow
EvenRow
OddRow
```

Using the `For Each...Next` statement is the preferred way to iterate over a collection. However, you can also reference an individual element of the **Styles** collection by using its zero-based index. The following example is equivalent to the previous one, although it is less economical:

```
For n = 0 To TDBGrid1.Styles.Count - 1
    Debug.Print TDBGrid1.Styles(n).Name
Next n
```

Typically, you will refer to a style by name. The following example changes the alignment and bold attribute of the built-in Normal style:

```
Dim S As TrueDBGrid50.Style
Set S = TDBGrid1.Styles("Normal")
S.Alignment = dbgCenter
S.Font.Bold = True
```

Note the use of the `Dim` and `Set` statements in assigning the Normal style object to the variable `s`. This example could have been written as follows:

```
TDBGrid1.Styles("Normal").Alignment = dbgCenter
TDBGrid1.Styles("Normal").Font.Bold = True
```

However, this style of coding is less efficient since the Normal style must be retrieved twice.

You can also use the `With...End With` statement to set multiple properties of a style object without explicitly assigning it to a variable:

```
With TDBGrid1.Styles("Normal")
    .Alignment = dbgCenter
    .Font.Bold = True
End With
```

To create a new style called `MyStyle`, use the **Add** method of the **Styles** collection, then use the `With...End With` statement to initialize its properties:

```
Dim MyStyle As TrueDBGrid50.Style
Set MyStyle = TDBGrid1.Styles.Add("MyStyle")
```

```
With MyStyle
    .BackColor = &H808080
    .ForeColor = &HFFFFFF
    .Font.Bold = True
End With
```

The resulting style is identical to the one created in the previous section using the Styles property page.

Applying Styles at Run Time

To apply a style to the cells in a **TDBGrid**, **Column**, or **Split** object, you simply set its **Style** property. The following example assumes that a style named MyStyle was created at design time using the Styles property page:

```
Dim MyStyle As TrueDBGrid50.Style
Set MyStyle = TDBGrid1.Styles("MyStyle")
TDBGrid1.Style = MyStyle
```

Unless you are going to modify MyStyle, there is really no need to assign it to a variable. Therefore, the preceding example can be shortened to:

```
TDBGrid1.Style = TDBGrid1.Styles("MyStyle")
```

However, even this notation can be simplified. The **Style** property is declared as a variant and accepts either a **Style** object (as in the previous examples) or a string. When set to a string, True DBGrid automatically finds the matching entry in its **Styles** collection, which allows you to apply a style with a simple assignment statement:

```
TDBGrid1.Style = "MyStyle"
```

The **Style** property always returns the name of the underlying style object. The following statement prints the name of the style associated with the first column of the grid:

```
Debug.Print TDBGrid1.Columns(0).Style
```

Now, let's consider some examples.

For a newly created grid, the built-in Normal style controls the font and color characteristics of all cells in all splits and columns. Similarly, the built-in Heading style controls the display of the column headings, if enabled. The default values of the Normal style (or any style created in code) are as follows, with the possible exception of the **Font** property, which is derived from the ambient font of the grid's container, such as a Visual Basic form:

Alignment	0 - Left
BackColor	System Window Background
Font	MS Sans Serif, Regular, 8.25 points
ForeColor	System Window Text
Locked	False
WrapText	False

At run time, with both caption and column headings enabled, the grid will look something like this.

True DBGrid Style Example			
	Title	Author	Year Published
▶	101+ FoxPro and dBase IV user-defined functions	Steele, Philip	1992
	4th dimension: a complete guide to database design	Knight, Timothy Orr	1994
	A guide to developing client/server SQL applications	Khoshafian, Setrag.	1993
	A guide to SQL	Pratt, Philip J	1992
	A guide to the SQL standard: a user's guide to SQL	Date, C. J.	1995
	A Hitchhiker's Guide to VBSQL: The Developer's Guide	Vaughn, William R.	1992
	A practical guide to Oracle database administration	Ault, Michael R	1994
	A primer on SQL	Ageloff, Roy	1993
	A visual introduction to SQL	Trimble, J. Harvey	1992
	A visual introduction to SQL	Chappell, David	1992

Now, let's create a new style named MyStyle in code, and apply it to the grid:

```

Dim MyStyle As TrueDBGrid50.Style
Set MyStyle = TDBGrid1.Styles.Add("MyStyle")

With MyStyle
    .BackColor = &H808080
    .ForeColor = &HFFFFFF
    .Font.Bold = True
End With

TDBGrid1.Style = MyStyle

```

Alternatively, you can create MyStyle at design time using the Styles property page. See [Defining Styles at Design Time](#) for details.

MyStyle now has the following characteristics:

Alignment	0 - Left
BackColor	Dark Gray
Font	MS Sans Serif, Bold, 8.25 points
ForeColor	White
Locked	False
WrapText	False

After the last line of code in the preceding example is executed, the grid automatically updates its display as follows.

True DBGrid Style Example		
Title	Author	Year Published
101+ FoxPro and dBase IV user-defin...	Steele, Philip	1992
4th dimension; a complete guide to dat	Knight, Timothy I	1994
A guide to developing client/server SQ	Khoshafian, Setr	1993
A guide to SQL	Pratt, Philip J	1992
A guide to the SQL standard : a user's	Date, C. J.	1995
A Hitchhiker's Guide to VBSQL : The D	Vaughn, William	1992
A practical guide to Oracle database a	Ault, Michael R	1994
A primer on SQL	Ageloff, Roy	1993
A visual introduction to SQL	Trimble, J. Harv	1992
A visual introduction to SQL	Chappell, David	1992

Note that the column headings and grid caption do not have the bold attribute set, since they are governed by the built-in Heading and Caption styles.

If the last line of code in the preceding example is changed to read:

```
TDBGrid1.Columns(0).Style = MyStyle
```

then MyStyle is only applied to the first column of the grid.

True DBGrid Style Example			
	Title	Author	Year Published
▶	101+ FoxPro and dBase IV user-defin...	Steele, Philip	1992
	4th dimension; a complete guide to dat	Knight, Timothy Orr	1994
	A guide to developing client/server SQ	Khoshafian, Setrag.	1993
	A guide to SQL	Pratt, Philip J	1992
	A guide to the SQL standard : a user's	Date, C. J.	1995
	A Hitchhiker's Guide to VBSQL : The D	Vaughn, William R.	1992
	A practical guide to Oracle database a	Ault, Michael R	1994
	A primer on SQL	Ageloff, Roy	1993
	A visual introduction to SQL	Trimble, J. Harvey	1992
	A visual introduction to SQL	Chappell, David	1992

Similarly, you can apply a style to an individual split.

Additional Style Properties

In addition to the **Style** property, which determines how cell text is displayed, True DBGrid also provides the following properties for customizing other aspects of the grid's display:

<u>HeadingStyle</u>	Controls the format of column headings, if present. Applies to <u>TDBGrid</u> , <u>Split</u> , and <u>Column</u> objects. Default value: Heading.
<u>SelectedStyle</u>	Controls the format of selected rows and columns, if any. Applies to <u>TDBGrid</u> and <u>Split</u> objects. Default value: Selected.
<u>CaptionStyle</u>	Controls the format of the grid's caption, if set. Applies to <u>TDBGrid</u> object. Default value: Caption.
<u>EditorStyle</u>	Controls the format of the in-cell editor when the <u>MarqueeStyle</u> property is not set to 6 - Floating Editor. Applies to <u>TDBGrid</u> , <u>Split</u> , and <u>Column</u> objects. Default value: Normal.
<u>InactiveStyle</u>	Controls the format of column headers when the <u>Appearance</u> property is set to 0 - Flat and the grid or split does not have focus. Applies to <u>TDBGrid</u> and <u>Split</u> objects. Default value: Heading.
<u>HighlightRowStyle</u>	Controls the format of a highlighted row or cell marquee (for <u>MarqueeStyle</u> settings 2, 3, and 4). Applies to <u>TDBGrid</u> and <u>Split</u> objects. Default value: HighlightRow.
<u>EvenRowStyle</u>	Controls the format of even-numbered rows when the <u>AlternatingRowStyle</u> property is True. Applies to <u>TDBGrid</u> and <u>Split</u> objects. Default value: EvenRow.
<u>OddRowStyle</u>	Controls the format of odd-numbered rows when the <u>AlternatingRowStyle</u> property is True. Applies to <u>TDBGrid</u> and <u>Split</u> objects. Default value: OddRow.

These properties work like the **Style** property---you simply set them to a different style object (or style name) to change the formatting of the associated grid component. For example, using the definition of MyStyle from the preceding section, the line:

```
TDBGrid1.HeadingStyle = "MyStyle"
```

changes the appearance of the column headings as follows.

True DBGrid Style Example			
	Title	Author	Year
▶	101+ FoxPro and dBase IV user-defined functions	Steele, Philip	1992
	4th dimension; a complete guide to database design	Knight, Timothy Orr	1994
	A guide to developing client/server SQL applications	Khoshafian, Setrag.	1993
	A guide to SQL	Pratt, Philip J	1992
	A guide to the SQL standard : a user's guide to SQL	Date, C. J.	1995
	A Hitchhiker's Guide to VBSQL : The Developer's Guide	Vaughn, William R.	1992
	A practical guide to Oracle database administration	Ault, Michael R	1994
	A primer on SQL	Ageloff, Roy	1993
	A visual introduction to SQL	Trimble, J. Harvey	1992
	A visual introduction to SQL	Chappell, David	1992

These properties also allow you to manipulate the underlying styles directly. This is often the most direct way to customize a particular grid component. For example, to center all column headings without changing the justification of column data, you would code:

```
TDBGrid1.HeadingStyle.Alignment = dbgCenter
```

Note that this statement actually changes the **Alignment** property of the built-in Heading style. You can

achieve the same result without writing any code by using the Styles property page at design time.

When you manipulate a style, keep in mind that other styles may inherit from it. For example, the built-in Caption style inherits everything except its **Alignment** value from the built-in Heading style. Therefore, if you set the bold attribute of the Heading style's font:

```
TDBGrid1.HeadingStyle.Font.Bold = True
```

then the grid's caption will appear bold as well.

True DBGrid Style Example				
	Title	Author	Year	▲
▶	101+ FoxPro and dBase IV user-defined functions	Steele, Philip	1992	
	4th dimension: a complete guide to database design	Knight, Timothy Orr	1994	
	A guide to developing client/server SQL applications	Khoshafian, Setrag.	1993	
	A guide to SQL	Pratt, Philip J	1992	
	A guide to the SQL standard : a user's guide to SQL	Date, C. J.	1995	
	A Hitchhiker's Guide to VBSQL : The Developer's	Vaughn, William R.	1992	
	A practical guide to Oracle database administration	Ault, Michael R	1994	
	A primer on SQL	Ageloff, Roy	1993	
	A visual introduction to SQL	Trimble, J. Harvey	1992	
	A visual introduction to SQL	Chappell, David	1992	▼

Note that this is **not** the same as setting the bold attribute of the grid's **HeadFont** property:

```
TDBGrid1.HeadFont.Bold = True
```

Setting the **HeadFont** property overrides the HeadingStyle for the column headings but does not change the grid's caption. The same is true of the **HeadForeColor** and **HeadBackColor** properties.

Introduction to Cell Styles

True DBGrid gives you three ways to control the color and font characteristics of individual cells:

- By Status** Each grid cell has a cell status which identifies its disposition (any combination of current, modified, part of a selected row, or part of a highlighted row). Using the **AddCellStyle** method, you can set style attributes which apply to any possible combination of cell status values.
- By Contents** You can specify a pattern (called a regular expression) which is used to perform pattern matching on cell contents. When the contents match the pattern supplied in the **AddRegexCellStyle** method, True DBGrid will automatically apply pre-selected style attributes to the cell.
- By Custom Criteria** Using the **FetchCellStyle** (or **FetchRowStyle**) event, you can make decisions about cell colors and fonts each time a cell (or row) is displayed.

You can use **Style** objects defined at design time as arguments to the **AddCellStyle** and **AddRegexCellStyle** methods. Or, you can create a temporary style in code and use it to specialize one or more attributes.

The **FetchCellStyle** and **FetchRowStyle** events pass a temporary **Style** object as the final parameter. By setting its properties, you can control the appearance of the cell specified by the other event parameters.

In this version of True DBGrid, per-cell font and color control can only be achieved by writing code. However, by creating styles at design time, you can keep this code to a minimum.

Specifying Cell Status Values

True DBGrid recognizes 16 distinct cell status values which are used in code to indicate the disposition of a cell. A cell status value is a combination of four separate conditions:

Current Cell	The cell is the current cell as specified by the Bookmark , Col , and Split properties. At any given time, only one cell can have this status. When the floating editor MarqueeStyle property setting is in effect, this condition is ignored.
Marquee Row	The cell is part of a highlighted row marquee. When the MarqueeStyle property indicates that the entire current row is to be highlighted, all visible cells in the current row have this additional condition set.
Updated Cell	The cell contents have been modified by the user but not yet written to the database. This condition is also set when cell contents have been modified in code with the Text or Value properties.
Selected Row	The cell is part of a row selected by the user or in code. The SelBookmarks collection contains a bookmark for each selected row.

True DBGrid defines the following constants corresponding to these cell conditions:

<code>dbgCurrentCell</code>	1 - Applies to the current cell
<code>dbgMarqueeRow</code>	2 - Applies to cells in a highlighted row marquee
<code>dbgUpdatedCell</code>	4 - Applies to cells that have been modified
<code>dbgSelectedRow</code>	8 - Applies to cells in a selected row

You can add these constants together to specify multiple cell conditions. For example, a cell status value of 9 (`dbgCurrentCell + dbgSelectedRow`) denotes a current cell within a selected row.

True DBGrid also defines the following constants, which are **not** meant to be combined with those listed earlier:

<code>dbgAllCells</code>	-1 - Applies to all cells
<code>dbgNormalCell</code>	0 - Applies to cells without status conditions

Use `dbgAllCells` to refer to all cells regardless of status. Use `dbgNormalCell` to refer to only those cells without any of the four basic cell conditions described earlier.

Applying Cell Styles by Status

Each cell in the True DBGrid display has a status value which identifies its disposition (any combination of current, modified, part of a selected row, or part of a highlighted row). Using the **AddCellStyle** method, you can set style attributes which apply to any possible combination of cell status values. The **AddCellStyle** method is supported by the **TDBGrid**, **Split**, and **Column** objects, enabling you to control the range of cells for which certain conditions apply.

For each unique status combination, you can set the foreground color, background color, and font attributes to be used for cells of that status. When a cell's status changes, True DBGrid checks to see if any color or font overrides are defined for that cell, and applies those attributes to the cell when it is displayed. **Style** objects are used to specify the color and font for a cell, as in the following example:

```
Dim S As New TrueDBGrid50.Style
S.ForeColor = vbRed
S.Font.Bold = True
TDBGrid1.AddCellStyle dbgCurrentCell, S
```

Here, a new temporary style object is created to specify the color and font overrides (red text, bold) to be applied to the current cell throughout the entire grid. Since the style object's **BackColor** property is not set explicitly, the background color of the current cell is not changed.

You can also use styles defined at design time as arguments to the **AddCellStyle** method:

```
Dim S As TrueDBGrid50.Style
Set S = TDBGrid1.Styles("RedBold")
TDBGrid1.AddCellStyle dbgCurrentCell, S
```

The preceding example can be simplified since the **AddCellStyle** method accepts a style name as well as an actual style object:

```
TDBGrid1.AddCellStyle dbgCurrentCell, "RedBold"
```

All of the preceding examples cause the text of the current cell to appear in red and bold. However, it is important to note that the status `dbgCurrentCell` applies only to cells which have **only** this status. Thus, cells which are current but also updated (`dbgCurrentCell + dbgUpdatedCell`) will not be displayed in red and bold unless you also execute the following statement:

```
TDBGrid1.AddCellStyle dbgCurrentCell + dbgUpdatedCell, "RedBold"
```

NOTE: The current cell status is only honored when the **MarqueeStyle** property is **not** set to 6 - Floating Editor. The floating editor marquee always uses the system highlight colors as determined by your Control Panel settings.

Although this method of specifying cell conditions offers more control and flexibility, it also requires that additional code be written for some common cases.

Calls to **AddCellStyle** take effect immediately, and can be used for interactive effects as well as overall grid characteristics.

Applying Cell Styles by Contents

You can tell True DBGrid to automatically apply colors and fonts to particular cells, based upon their displayed contents. To do so, you provide a pattern, called a *regular expression*, which the grid tests against the displayed value of each cell. Using the **AddRegexCellStyle** method, you can associate a regular expression with a set of style attributes, then apply them to any possible combination of cell status values. The **AddRegexCellStyle** method is supported by the **TDBGrid**, **Split**, and **Column** objects, allowing you to control the range of cells for which certain conditions apply.

The **AddRegexCellStyle** method is similar to the **AddCellStyle** method, but it requires an additional argument for the regular expression string. As with **AddCellStyle**, you can use either temporary or named styles. The following example uses a temporary style to display all cells in the first column that contain the string "SQL" in bold:

```
Dim S As New TrueDBGrid50.Style
S.Font.Bold = True
TDBGrid1.Columns(0).AddRegexCellStyle dbgAllCells, S, "SQL"
```

This feature allows you to implement "visual queries" that attach distinctive font or color attributes to cells that match a certain pattern.

True DBGrid Style Example			
	Title	Author	Year Published
▶	101+ FoxPro and dBase IV user-defined functions	Steele, Philip	1992
	4th dimension: a complete guide to database design	Knight, Timothy Orr	1994
	A guide to developing client/server SQL	Khoshafian, Setrag.	1993
	A guide to SQL	Pratt, Philip J	1992
	A guide to the SQL standard : a user's manual	Date, C. J.	1995
	A Hitchhiker's Guide to VBSQL : The Database	Vaughn, William R.	1992
	A practical guide to Oracle database administration	Ault, Michael R	1994
	A primer on SQL	Ageloff, Roy	1993
	A visual introduction to SQL	Trimble, J. Harvey	1992
	A visual introduction to SQL	Chappell, David	1992

Applying Cell Styles by Custom Criteria

For cases where regular expressions are insufficient to express your formatting requirements, you can use the **FetchCellStyle** event to customize fonts and colors on a per-cell basis. This event will only be fired for columns that have the **FetchStyle** property set to True.

For example, you may want to provide color coding for values that fall within a certain range. The following code assumes that the **FetchStyle** property is True for a single column of numeric data, and handles the **FetchCellStyle** event to display values greater than 1000 in blue:

```
Private Sub TDBGrid1_FetchCellStyle( _
    ByVal Condition As Integer, _
    ByVal Split As Integer, _
    Bookmark As Variant, _
    ByVal Col As Integer, _
    ByVal CellStyle As TrueDBGrid50.StyleDisp)

    Dim N As Long
    N = Val(TDBGrid1.Columns(Col).CellText(Bookmark))

    If N > 1000 Then CellStyle.ForeColor = vbBlue
End Sub
```

The **Split**, **Bookmark**, and **Col** arguments identify which cell the grid is displaying. The **CellStyle** argument conveys formatting information from your application to the grid. Since the **CellStyle** argument is a **Style** object, you can also change a cell's font characteristics in the **FetchCellStyle** event:

```
If N > 1000 Then CellStyle.Font.Italic = True
```

The **FetchCellStyle** event can also be used to apply formatting to one cell based upon the values of other cells, or even other controls. For example, suppose that you want to:

- Make the cell text red in column 4 if column 1 minus column 2 is negative.
- Make the cell text bold in column 7 if it matches the contents of a text box.

In this case, you need to set the **FetchStyle** property to True for columns 4 and 7, and handle the **FetchCellStyle** event as follows:

```
Private Sub TDBGrid1_FetchCellStyle( _
    ByVal Condition As Integer, _
    ByVal Split As Integer, _
    Bookmark As Variant, _
    ByVal Col As Integer, _
    ByVal CellStyle As TrueDBGrid50.StyleDisp)

    Select Case Col
        Case 4
            Dim Col1 As Long, Col2 As Long
            Col1 = CLng(TDBGrid1.Columns(1).CellText(Bookmark))
            Col2 = CLng(TDBGrid1.Columns(2).CellText(Bookmark))
            If Col1 - Col2 < 0 Then CellStyle.ForeColor = vbRed
        Case 7
            Dim S As String
            S = TDBGrid1.Columns(7).CellText(Bookmark)
            If S = Text1.Text Then CellStyle.Font.Bold = True
        Case Else
            Debug.Print "FetchCellStyle not handled: " & Col
    End Select
End Sub
```

```
End Select
End Sub
```

For efficiency reasons, you should only set **FetchStyle** to True for columns that you plan to handle in the **FetchCellStyle** event.

NOTE: The preceding examples use the **CellText** method for simplicity. However, the **CellText** and **CellValue** methods always create and destroy an internal clone of the dataset each time they are called, which may make them too inefficient to use in the **FetchCellStyle** event. To improve the performance of the grid's display cycle, use a **Recordset** clone to derive the cell text, if available. Unbound applications can access the underlying data source directly, which is generally faster than calling **CellText** or **CellValue**.

If you need to customize fonts and colors on a per-row instead of a per-cell basis, use the **FetchRowStyle** event, which will only be fired once per row for grids that have the **FetchRowStyle** property set to True. The syntax for this event is as follows:

```
Private Sub TDBGrid1_FetchRowStyle( _
    ByVal Split As Integer, _
    Bookmark As Variant, _
    ByVal RowStyle As TrueDBGrid50.StyleDisp)
```

Although you can use the **FetchRowStyle** event to implement an alternating row color scheme, an easier and more efficient way to accomplish the same task would be to use the **AlternatingRowStyle** property, together with the built-in EvenRow and OddRow styles.

The **FetchRowStyle** event is ideally suited for coloring the entire row of a grid based on the value of one or more columns. The following example demonstrates how to do this using a **Recordset** clone:

```
Dim RS As Recordset

Private Sub Form_Load()
    Data1.Refresh
    Set RS = Data1.Recordset.Clone
    TDBGrid1.FetchRowStyle = True
End Sub

Private Sub TDBGrid1_FetchRowStyle( _
    ByVal Split As Integer, _
    Bookmark As Variant, _
    ByVal RowStyle As TrueDBGrid50.StyleDisp)

    RS.Bookmark = Bookmark

    If RS.Fields("Country") = "Germany" Then
        RowStyle.BackColor = vbCyan
    End If
End Sub
```

Cell Editing Techniques

This chapter explains how to customize the behavior of cell editing in True DBGrid. For text entry fields, you can write handlers for the grid's editing events, specify an input mask template, or display a drop-down text editor for long strings. To provide a list of choices for the user, you can use the **ValueItems** collection, the data-aware **TDBDropDown** control, or even an arbitrary intrinsic or third-party control.

{button ,JI(`,`How_Cell_Editing_Works')} How Cell Editing Works

{button ,JI(`,`Handling_Editing_Events')} Handling Editing Events

{button ,JI(`,`Working_with_Text')} Working with Text

{button ,JI(`,`Input_Masking')} Input Masking

{button ,JI(`,`In-cell_Button')} In-cell Button

{button ,JI(`,`Drop-down_Controls')} Drop-down Controls

How Cell Editing Works

True DBGrid provides many features for customizing and controlling in-cell editing. The grid's default editing behavior depends on the setting of the **MarqueeStyle** property. If the floating editor marquee style is used, the editing behavior differs from that of other marquee styles. The following sections summarize True DBGrid's editing behavior and state any exceptions that arise when using the floating editor.

For more information on the **MarqueeStyle** property, see [Highlighting the Current Row or Cell](#).

{button ,JI(`,`Initiating_cell_editing')}` Initiating cell editing

{button ,JI(`,`Color_and_wordwrap')}` Color and wordwrap

{button ,JI(`,`Determining_modification_status')}` Determining modification status

{button ,JI(`,`Determining_cell_contents')}` Determining cell contents

{button ,JI(`,`Terminating_cell_editing')}` Terminating cell editing

Initiating cell editing

A cell is either in display or edit mode. The **EditActive** property sets and returns the desired mode. You can place the current cell in edit mode by setting **EditActive** to True, or end editing by setting it to False. The user may enter edit mode by clicking once on the current cell or by pressing the F2 key. A blinking text cursor (caret) will appear in the cell---at the beginning of the text when the cell is clicked and at the end when the F2 key is used. The **BeforeColEdit** event will be triggered when the cell enters edit mode. The **EditActive** property is True when the cell is in edit mode.

Floating Editor Differences: A blinking caret already exists at the beginning of the cell highlight even when in display mode. To enter edit mode, the user can click on any character location within the cell text to specify the text insertion point. The **BeforeColEdit** event is not triggered and the **EditActive** property is False until the user has made changes to the cell text.

Color and wordwrap

In edit mode, the cell color is determined by the **EditForeColor** and **EditBackColor** properties. The text being edited will word wrap, regardless of the setting of the column's **WrapText** property. If the text is too big to fit into the cell, a built-in drop-down edit control will automatically appear. For more information, see [Working with Text](#).

Floating Editor Differences: In edit mode, the text highlight disappears, and the cell color is the same as the normal cell color. The text being edited is word wrapped only if the column's **WrapText** property is True. The built-in drop-down edit control is not available.

Determining modification status

While editing is in progress, you can inspect the **DataChanged** property of the grid to determine whether the user has made any changes to the current row. Similarly, you can inspect the **DataChanged** property of an individual column to determine whether the user has made any changes to a specific cell within the current row.

You can set the grid's **DataChanged** property to False to exit editing, discard all changes to the current row, and refresh the current row display from the data source.

The **DataChanged** property of a **Column** object is read-only.

The icon in the record selector column of the current row reflects the status of the grid's **DataChanged** property. If **DataChanged** is False, a triangle-shaped arrow will be shown in the record selector column. If **DataChanged** is True, a pencil icon will appear instead.

Determining cell contents

While editing is in progress, the column's **Text** and **Value** properties contain the text the user currently sees in the modified row. Whenever the user presses a key, the **Change** event fires to notify your application that the user has just modified the current cell. However, the **Change** event doesn't mean the user is finished with the process, only that a single change has been made and the grid is still in edit mode.

The **Change** event does not fire when the grid is not in edit mode, such as when the contents of a cell are changed through code or when the user clicks a cell to cycle through a **ValueItems** collection.

Terminating cell editing

The user completes the editing process by performing any of the following:

- Pressing the ENTER key.
- Pressing the F2 key.
- Pressing the ESC key.
- Moving to another cell with the arrow keys, the TAB key, or the mouse.
- Setting focus to another control on the form.

Handling Editing Events

The following sections describe how you can alter the default editing behavior of True DBGrid by responding to its events.

{button ,JI(`',`Standard_keystroke_events')} Standard keystroke events

{button ,JI(`',`Column_editing_events')} Column editing events

{button ,JI(`',`Changing_cell_contents_with_a_single_keystroke')} Changing cell contents with a single keystroke

Standard keystroke events

True DBGrid supports the standard keystroke events common to many ActiveX controls:

<u>KeyDown</u>	Fired when the user presses a key.
<u>KeyPress</u>	Fired when the user presses an ANSI key.
<u>KeyUp</u>	Fired when the user releases a key.

The **KeyDown** and **KeyUp** events trap all keys, including function keys, ALT and SHIFT keys, and numeric keypad keys. The **KeyPress** event only traps letters and numbers, punctuation marks and symbols, and editing keys such as TAB, ENTER, and BACKSPACE.

You can use these events to restrict and modify user input as you would for any other intrinsic or ActiveX control. For example, the following **KeyPress** event handler converts the user's keystrokes to upper case letters, and prevents the user from entering non-alphanumeric characters:

```
Private Sub TDBGrid1_KeyPress(KeyAscii As Integer)
    ' Convert key to upper case
    KeyAscii = Asc(UCase(Chr$(KeyAscii)))

    ' Don't disable the Esc or Backspace keys
    If (KeyAscii = 27) Or (KeyAscii = 8) Then Exit Sub

    ' Cancel user key input if it is not a letter or a digit
    If (KeyAscii < 65 Or KeyAscii > 90) _
        And (KeyAscii < 48 Or KeyAscii > 57) Then
        KeyAscii = 0
    End If
End Sub
```

Column editing events

True DBGrid gives you full control over the cell editing process with the following events, listed in the order in which they occur during a successful editing attempt:

<u>BeforeColEdit</u>	Fired upon an attempt to edit column data.
<u>ColEdit</u>	Fired when the current cell enters edit mode.
<u>AfterColEdit</u>	Fired after column data is edited.

You can use the **BeforeColEdit** event to control the editability of cells on a per-cell basis, or to translate the initial keystroke into a default value.

The **ColEdit** event signals that the current cell has entered edit mode; the **AfterColEdit** event signals that edit mode was terminated. You can use these two events to provide additional feedback while editing is in progress:

```
Private Sub TDBGrid1_ColEdit(ByVal ColIndex As Integer)
    Select Case TDBGrid1.Columns(ColIndex).Caption
        Case "Code"
            Labell.Caption = "Enter 4-digit company code"
        Case "Description"
            Labell.Caption = "Enter full company name"
    End Select
End Sub

Private Sub TDBGrid1_AfterColEdit(ByVal ColIndex As Integer)
    Labell.Caption = "" ' Clear editing instructions
End Sub
```

Changing cell contents with a single keystroke

The **BeforeColEdit** event is an extremely versatile way to customize the behavior of True DBGrid editing. **BeforeColEdit** is fired before any other editing events occur, which gives you the opportunity to do virtually anything you want to before editing begins. For example, you can cancel the edit request and override the built-in text editor with your own drop-down list box.

A True DBGrid control can enter edit mode in one of four ways:

1. If the user clicks on the current cell with the mouse, editing begins with the current cell contents.
2. If the user presses the F2 key, editing also begins using the current cell contents.
3. If the user begins typing, the typed character replaces the contents of the cell and editing begins.
4. You can set the **EditActive** property in your code to force editing to begin.

The **BeforeColEdit** event fires in cases 1, 2, and 3, but *not* in case 4, since True DBGrid assumes you will never want to cancel a request made from code.

You may want to differentiate a user's edit request based upon whether they used the mouse or the keyboard to start editing. To facilitate this, one of the parameters to **BeforeColEdit** is **KeyAscii**, which will be zero if the user clicked on the cell with the mouse, and will be an ASCII character if the user typed a character to begin editing.

When **BeforeColEdit** is fired, the ASCII character hasn't yet been placed into the current cell, so if you cancel editing in **BeforeColEdit**, the ASCII key is discarded. This leads to an interesting technique.

Assume you have a boolean field called Done, and you have set its **NumberFormat** property to specify Yes/No as the display format. Further assume that, when the user presses Y or N, you want to change the cell contents immediately instead of entering edit mode. Here's how you could accomplish this in **BeforeColEdit**:

```
Private Sub TDBGrid1_BeforeColEdit(ByVal ColIndex As Integer, _
    ByVal KeyAscii As Integer, Cancel As Integer)

    With TDBGrid1.Columns(ColIndex)
        ' If this isn't the "Done" column, or if the user
        ' clicked with the mouse, then simply continue.

        If .DataField <> "Done" Or KeyAscii = 0 Then Exit Sub

        ' Cancel normal editing and set the field to the
        ' proper result based upon KeyAscii. Beep if an
        ' invalid character was typed.

        Cancel = True
        Select Case UCase$(Chr$(KeyAscii))
            Case "Y"
                .Value = -1
            Case "N"
                .Value = 0
            Case Else
                Beep
        End Select
    End With
End Sub
```

Note that the event handler terminates when **KeyAscii** is zero, so mouse editing is still permitted.

Working with Text

This section briefly describes the properties related to text editing.

{button ,JI(`,`Limiting_the_size_of_data_entry_fields')}} Limiting the size of data entry fields

{button ,JI(`,`Providing_a_drop-down_edit_control_for_long_fields')}} Providing a drop-down edit control for long fields

{button ,JI(`,`Selecting_and_replacing_text')}} Selecting and replacing text

Limiting the size of data entry fields

You can use the **DataWidth** property of a **Column** object to restrict the number of characters the user can enter. Setting this property to zero imposes no limits.

Providing a drop-down edit control for long fields

Whenever the user attempts to edit cell text that is too big to fit within the cell, the grid will automatically activate a multiple-line drop-down text editor. While editing, text in the drop-down edit control will be wordwrapped regardless of the setting of the column's **WrapText** property. You can turn off the drop-down text editor and force editing to occur within cell boundaries by setting the grid's **EditDropDown** property to False (the default is True). The drop-down text editor is **not** available if the grid's **MarqueeStyle** property is set to 6 - Floating Editor. The following code uses the grid's built-in column button feature to activate the drop-down edit control to modify the cell data in the Comments column:

```
Private Sub Form_Load()  
    With TDBGrid1  
        .MarqueeStyle = dbgSolidCellBorder  
        .Columns("Comments").Button = True  
        .EditDropDown = True ' Redundant since default = True  
    End With  
End Sub  
  
Private Sub TDBGrid1_ButtonClick(ByVal ColIndex As Integer)  
    TDBGrid1.EditActive = True ' Place the cell into edit mode  
End Sub
```

If the current cell is in the Comments column, you can initiate editing either by clicking on the current cell or by clicking the built-in button.

Selecting and replacing text

True DBGrid supports the standard text selection properties found in many ActiveX controls:

SelLength Sets/returns the length of the selected text.

SelStart Sets/returns the start position of the selected text.

SelText Sets/returns the selected text.

NOTE: These properties are only effective when the grid is in edit mode, that is, when its **EditActive** property is True.

Input Masking

You can use the **NumberFormat** property to control the display format of column data. If your users need to edit a formatted column, it is desirable to maintain a consistent format during the editing process. True DBGrid provides an **EditMask** property that optionally works in concert with the **NumberFormat** property to ensure consistent data entry.

{button ,JI(`,`Specifying_an_input_mask_for_a_column')}} Specifying an input mask for a column
{button ,JI(`,`Specifying_a_date_mask_for_a_column')}} Specifying a date mask for a column
{button ,JI(`,`Using_an_input_mask_for_formatting')}} Using an input mask for formatting
{button ,JI(`,`Controlling_how_masked_input_is_updated')}} Controlling how masked input is updated

Specifying an input mask for a column

The **EditMask** property of the **Column** object is used to specify an input mask template for end-user data entry. You can construct your own input mask string using template characters similar to those recognized by the Visual Basic **Format\$** function. The input mask string is composed of special characters that represent either an input character that the user must enter, or a literal character that will be skipped over on input. Valid template characters are as follows:

#	Digit placeholder
@	Character placeholder
>	All characters following will be in uppercase
<	All characters following will be in lowercase
~	Turns off the previous "<" or ">"
?	Digit or character
\	Next character is treated as a literal
&	Any character

Any other character will be treated as a literal. For example, to specify a phone number template, you could use:

```
TDBGrid1.Columns("Phone").EditMask = "(###) ###-####"
```

In this example, the parentheses, space, and hyphen are all literals, while the pound signs signify digit placeholders.

After the user finishes editing a cell with this input mask, True DBGrid caches the modified cell text, but any literal characters in the input mask template will be stripped from the modified cell text beforehand. In the preceding example, only the 10 digits denoted by the # placeholders will be cached; the punctuation marks and the space will be omitted.

Specifying a date mask for a column

The **EditMask** property also supports a built-in DateMask option for formatting date fields. When the DateMask option is selected, the following input mask template will be used for editing:

mm/dd/yyyy

where

mm	Month placeholder (01-12)
dd	Date placeholder (01-31)
yyyy	Year placeholder

The user can enter either 1, 2, 3, or 4 digits in the year portion of the date, and the grid will save the year as entered by the user. If the user enters 1 or 2 digits for the year portion, the grid will make no interpretation for the year; that is, the grid will not assume whether it is the century 1900 or 2000, but will store the 1-digit or 2-digit year as entered. Before the date is updated to the database, you can interpret the year yourself in code, or let the underlying database system handle the interpretation and storage.

Note that if you select the DateMask option for the **EditMask** property, the date separators are part of the date format; they are not considered as literal characters and will always be cached by the grid. This is because most databases and formatters require the separator characters to be present in order to interpret the date correctly.

Using an input mask for formatting

Whereas the **EditMask** property is used to specify an input mask for *data entry*, the **NumberFormat** property is used to specify the *display format* of data in a grid cell. If the **NumberFormat** property of the column is not specified, the grid simply displays the cached text (stripped of literals) as is; if the **NumberFormat** property is specified, the grid sends the cached text to the display formatter.

Since it is common for the input and display formats to be the same, the **NumberFormat** property has an Edit Mask option. If you select this option, then the **EditMask** property setting will be used for both data input and display. However, the input and display formats need not be the same, so you are free to select a **NumberFormat** option that differs from the **EditMask** property.

Controlling how masked input is updated

Normally, after the user finishes editing a cell in a column which has its **EditMask** property set, True DBGrid caches the modified cell text, but any literal characters in the input mask template will be stripped from the modified cell text beforehand. However, you can override this behavior with the **EditMaskUpdate** property.

By default, the **EditMaskUpdate** property is False. This means that when the modified cell text is updated to the database, the grid sends the cached text (stripped of literals), not the formatted text displayed in the cell. You can override this default behavior by setting the **EditMaskUpdate** property to True, which causes the cached text to be formatted according to the **EditMask** property before being updated to the database.

Therefore, it is important to set **EditMaskUpdate** properly to ensure that the correct data is sent to the database for update.

In-cell Button

True DBGrid optionally displays a button at the right edge of the current cell, which you can use to indicate that a list of choices is available, perform a command associated with the contents of the cell, or display an arbitrary control or form for editing.

{button ,JI(`,`Enabling_the_in-cell_button')} Enabling the in-cell button

{button ,JI(`,`Detecting_in-cell_button_clicks')} Detecting in-cell button clicks

{button ,JI(`,`Customizing_the_in-cell_button_bitmap')} Customizing the in-cell button bitmap

Enabling the in-cell button

To enable the in-cell button for a **Column** object, select the Button check box on the Layout property page or set the column's **Button** property to True in code:

```
TDBGrid1.Columns(1).Button = True
```

The **Button** property is also enabled when the column's **DropDown** property is set to the name of a **TDBDropDown** control, or when the **Presentation** property of the associated **ValueItems** collection is set to one of the combo box options.


Detecting in-cell button clicks

The **ButtonClick** event is provided so that your code can respond when the user clicks the in-cell button. Its syntax is as follows:

```
Private Sub TDBGrid1_ButtonClick(ByVal ColIndex As Integer)
```

An example of the **ButtonClick** event was presented earlier in the section [Working with Text](#).

Customizing the in-cell button bitmap

By default, True DBGrid uses a down arrow  for the in-cell button. However, you can change the button bitmap at design time by clicking the **Picture** button on the Layout property page. Or, you can assign a bitmap to the **ButtonPicture** property in code at run time:

```
TDBGrid1.Columns(1).ButtonPicture = LoadPicture("arrow.bmp")
```

Note that the grid automatically draws the edges corresponding to the button's up/down states as appropriate, so you need only provide the interior image of the button. A light gray background is recommended.

Drop-down Controls

True DBGrid offers a wide variety of built-in controls and programming constructs that enable you to implement virtually any kind of drop-down cell editing interface. You can use the **ValueItems** collection to provide a simple pick list, or the **TDBDropDown** control to implement a data-aware multicolumn combo box. You can even use arbitrary Visual Basic or third-party controls to perform specialized editing functions.

{button ,JI(`,`Using_the_built-in_combo_box')} Using the built-in combo box

{button ,JI(`,`Detecting_built-in_combo_box_selections')} Detecting built-in combo box selections

{button ,JI(`,`Using_the_TDBDropDown_control')} Using the TDBDropDown control

{button ,JI(`,`Using_an_arbitrary_drop-down_control')} Using an arbitrary drop-down control

{button ,JI(`,`Using_the_built-in_column_button')} Using the built-in column button

Using the built-in combo box

The **Column** object's **ValueItems** collection optionally provides a built-in combo box interface that works in concert with its automatic data translation features. By default, the **Presentation** property is set to 0 - Normal, and the usual cell editing behavior is in effect for textual data. However, if you set the **Presentation** property to either 2 - Combo Box or 3 - Sorted Combo Box, then cells in the affected column display the in-cell button upon receiving focus. When the user clicks the in-cell button, a drop-down combo box appears.

Company	Country
Maple Leaf Systems	Canada
Her Majesty's Software	United Kingdom
Software Mart	United States
▶ Far East Distributors	Japan
Outback Software, Inc.	Canada
Northwest Purchasing Agents, Inc.	United Kingdom
	United States
	Japan
	Australia

The drop-down combo box contains one item for each member of the **ValueItems** collection. If the collection's **Translate** property is True, then the **DisplayValue** text is used for the combo box items; if it is False, then the **Value** text is used.

True DBGrid automatically sizes the drop-down combo box to fit the width of the column in which it is displayed. The height of the combo box is determined by both the number of items in the collection and the **MaxComboItems** property. If the number of items is less than or equal to **MaxComboItems**, which has a default value of 5, then all value items will be shown. If the number of items exceeds **MaxComboItems**, only **MaxComboItems** will be shown, but a scroll bar will appear at the right edge of the combo box to allow users to bring the other items into view.

Detecting built-in combo box selections

The **ComboSelect** event is fired when the user selects an item from the built-in combo box. This event is useful for determining the contents of the cell before the user exits edit mode.

Since the items displayed in the built-in combo box are often the only allowable values for the underlying data source, you may need to prevent your users from typing in the cell after making a selection. By setting the **EditActive** property to False within the **ComboSelect** event handler, you can force the grid to exit editing mode without allowing the user a chance to alter the selection (provided that the **MarqueeStyle** property is **not** set to 6 - Floating Editor).

```
Private Sub TDBGrid1_ComboSelect(ByVal ColIndex As Integer)
    TDBGrid1.EditActive = False
End Sub
```

In this case, it is also a good idea to explicitly disallow keyboard input with the **BeforeColEdit** event:

```
Private Sub TDBGrid1_BeforeColEdit(ByVal ColIndex As Integer, _
    ByVal KeyAscii As Integer, Cancel As Integer)

    If KeyAscii <> 0 Then ' Editing initiated via the keyboard
        If TDBGrid1.Columns(ColIndex).Caption = "Columns" Then
            Cancel = True
        End If
    End If
End Sub
```

Using the TDBDropDown control

The built-in drop-down combo box described in the preceding example is most useful when the allowable values are both known in advance and relatively few in number. A large **ValueItems** collection can be unwieldy to maintain at design time, and requires substantial coding to set up at run time. Moreover, you cannot bind the built-in combo box to a data control and have it populated automatically.

Using the techniques outlined later in this chapter, you could set up a secondary **TDBGrid** control to be used as a drop-down. However, if you need to display a list of values from another data source, the **TDBDropDown** control offers a more elegant solution, as it was designed explicitly for that purpose and can be set up entirely at design time.

Title	PubID	Year Published	ISBN
Men Are from Mars, Women Are from Venus : An Interactive	676	1995	0-0601950-6-1
Introduction to Computer Science and Programming Harpercollins	168	1994	0-0646714-5-3
Working With dBASE IV and dBASE III Plus			
Looking at dBASE IV			

PubID	Company Name
168	HARPERPERENNIAL LIBRARY
169	BROWN & BENCHMARK PUB
170	INTERPHARM PR
171	DUXBURY PR
172	SOFTWARE MASTERS
173	NORTH AMER PUB CO

To use the drop-down control, set the **DropDown** property of a grid column to the name of a **TDBDropDown** control at either design time or run time. At run time, when the user clicks the in-cell button for that column, the **TDBDropDown** control will appear below the grid's current cell. If the user selects an item from the drop-down control, the grid's current cell is updated.

Since the **TDBDropDown** control is a subset of **TDBGrid**, it shares many of the same properties, methods, and events. However, the following two properties are specific to the **TDBDropDown** control:

DataField Not to be confused with the **DataField** property of a **Column** object, this property specifies the grid column to be updated when a drop-down selection is made.

ListField This property specifies the name of the drop-down column to be used for incremental search.

When a **TDBDropDown** control becomes visible, its **DropDownOpen** event fires. Similarly, when the user makes a selection or the control loses focus, its **DropDownClose** event fires.

The **TDBDropDown** control supports all of the **DataMode** settings of the **TDBGrid** control. However, in order to enable incremental search for unbound, application, and storage modes, you need to implement the **UnboundFindData** event.

For more information on the differences between **TDBDropDown** and **TDBGrid**, see [TDBDropDown at Design Time](#).

Using an arbitrary drop-down control

Normally, True DBGrid's default editing behavior is sufficient for most applications. In some cases, however, you may want to customize this behavior. One valuable technique is to use a drop-down list or combo box, or even another True DBGrid control, to allow selection from a list of possible values. This is easy to do with True DBGrid using virtually any Visual Basic or third-party control. The general approach follows, and a working example is given in [Tutorial 9](#).

In general, displaying a drop-down list or combo instead of the standard True DBGrid editor involves the following steps:

1. True DBGrid fires the **BeforeColEdit** event each time the user wants to edit a cell. To override the default editing process, cancel True DBGrid's default editor by setting the **Cancel** argument to True. Put code in **BeforeColEdit** to display the editing control you wish to show instead. Typically, you place the substitute editing control or drop-down on the same form as the grid, but make it invisible until you need it.
2. When **BeforeColEdit** is triggered, there are five properties and one method you can use to determine the exact coordinates of the cell which is to be edited. The properties are **Left** (applies to grid and column), **Top** (grid and column), **CellTop** (column only, used with multiple line displays), **Width** (column only), and **RowHeight** (grid only). The method is **RowTop** (grid only). You can use these properties and method to position the custom editing control or drop-down relative to a grid cell. For example, you can place a ListBox control at the right edge of a cell and align its top border with that of the cell using the following code:

```
Dim Col As TrueDBGrid50.Column
Set Col = TDBGrid1.Columns(ColIndex)

With TDBGrid1
    List1.Left = .Left + Col.Left + Col.Width
    List1.Top = .Top + .RowTop(.Row)
End With
```

3. You need to put code in the drop-down or combo which completes the editing process by assigning the selected value to the **Text** or **Value** property of the column being edited.

This method does not work, however, when the grid's **MarqueeStyle** property is set to the default value of 6 - Floating Editor. When the floating editor marquee is used, the **BeforeColEdit** event does not fire until the cell has been changed by the user. However, you can use the built-in column button feature to activate the drop-down as described in the next section.

For more information, see [Highlighting the Current Row or Cell](#). An example of dropping down a Visual Basic ListBox control from a grid cell is given in [Tutorial 9](#).

Using the built-in column button

An alternative way to drop down a control from a cell is to use True DBGrid's built-in column button feature. If you set a column's **Button** property to True, a button will be displayed at the right edge of the current cell when it is in that column. Clicking the button fires the grid's **ButtonClick** event. You can then drop down a control from the cell using code inside the **ButtonClick** event. You can also use this event to trigger any action or calculation inside the cell.

For more information, see [In-Cell Button](#).

Property Reference

{button ,JI('',`Quick_Reference_for_All_Properties')}} Quick Reference for All Properties
{button ,JI('',`TDBGrid_Control_Properties')}} TDBGrid Control Properties
{button ,JI('',`TDBDropDown_Control_Properties')}} TDBDropDown Control Properties
{button ,JI('',`Column_Object_Properties')}} Column Object Properties
{button ,JI('',`Columns_Collection_Properties')}} Columns Collection Properties
{button ,JI('',`RowBuffer_Object_Properties')}} RowBuffer Object Properties
{button ,JI('',`Layouts_Collection_Properties')}} Layouts Collection Properties
{button ,JI('',`SelBookmarks_Collection_Properties')}} SelBookmarks Collection Properties
{button ,JI('',`Split_Object_Properties')}} Split Object Properties
{button ,JI('',`Splits_Collection_Properties')}} Splits Collection Properties
{button ,JI('',`Style_Object_Properties')}} Style Object Properties
{button ,JI('',`Styles_Collection_Properties')}} Styles Collection Properties
{button ,JI('',`ValueItem_Object_Properties')}} ValueItem Object Properties
{button ,JI('',`ValueItems_Collection_Properties')}} ValueItems Collection Properties

Quick Reference for All Properties

<u>AddNewMode</u>	Returns the disposition of the grid's AddNew row
<u>Alignment</u>	Specifies horizontal text alignment
<u>AllowAddNew</u>	Enables interactive record addition
<u>AllowArrows</u>	Enables use of arrow keys for grid navigation
<u>AllowColMove</u>	Enables interactive column movement
<u>AllowColSelect</u>	Enables interactive column selection
<u>AllowDelete</u>	Enables interactive record deletion
<u>AllowFocus</u>	Allows cells within a split to receive focus
<u>AllowRowSelect</u>	Enables interactive row selection
<u>AllowRowSizing</u>	Enables interactive row resizing
<u>AllowSizing</u>	Enables interactive resizing for a column or split
<u>AllowUpdate</u>	Enables interactive record updating
<u>AlternatingRowStyle</u>	Controls whether even/odd row styles are applied to an object
<u>AnnotatePicture</u>	True to show both underlying data and display value graphics
<u>Appearance</u>	Controls 3-D display of headings, caption, record selectors
<u>ApproxCount</u>	Sets/returns the approximate number of rows
<u>Array</u>	Specifies an XArray object as the data source
<u>BackColor</u>	Sets/returns the background color
<u>BOF</u>	Returns beginning-of-file status
<u>Bookmark (RowBuffer)</u>	Sets/returns bookmark of specified row in unbound events
<u>Bookmark (TDBGrid)</u>	Sets/returns bookmark of current grid row
<u>BorderStyle</u>	Sets/returns style for grid border
<u>Button</u>	Controls whether a button appears within the current cell
<u>ButtonPicture</u>	Sets/returns the bitmap used for the in-cell button
<u>Caption</u>	Sets/returns grid caption or column heading text
<u>CaptionStyle</u>	Controls the caption style for an object
<u>CellTips</u>	Enables pop-up cell tip window when the cursor is idle
<u>CellTipsDelay</u>	Sets/returns idle time for cell tip window
<u>CellTipsWidth</u>	Sets/returns width of cell tip window
<u>CellTop</u>	Returns top column border, adjusted for multiple lines
<u>Col</u>	Sets/returns current column number
<u>ColIndex</u>	Returns the ordinal position of a column
<u>ColumnCount</u>	Returns the number of columns in a RowBuffer
<u>ColumnHeaders</u>	Turns column headings on or off
<u>ColumnIndex</u>	Returns the column index of a column in the RowBuffer
<u>ColumnName</u>	Returns the field name of a column
<u>Columns</u>	Contains a collection of True DBGrid columns
<u>Count</u>	Returns the number of items in a collection
<u>CurrentCellModified</u>	Sets/returns modification status of the current cell
<u>CurrentCellVisible</u>	Sets/returns the visibility of the current cell
<u>CycleOnClick</u>	True to cycle through value items on click
<u>DataChanged</u>	Sets/returns modification status of the current row or cell
<u>DataField (Column)</u>	Data table field name for a column
<u>DataField (DropDown)</u>	Grid column to be updated with drop-down selection

<u>DataMode</u>	Specifies bound or unbound mode
<u>DataSource</u>	Specifies source of grid data
<u>DataWidth</u>	Maximum number of characters available for column input
<u>DefaultItem</u>	Index of default value item, or -1 if no default
<u>DefaultValue</u>	Default value for new column data
<u>DefColWidth</u>	Specifies column width for auto-created columns
<u>DisplayValue</u>	Sets/returns displayed text or graphics for a value item
<u>DividerStyle</u>	Divider style for right column border
<u>EditActive</u>	Returns status or enters/exits the cell editor
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditDropDown</u>	Controls whether a drop-down window is used for editing
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditorStyle</u>	Controls the editor style for an object
<u>EmptyRows</u>	Enables empty rows in an underpopulated grid
<u>Enabled</u>	Enables or disables user interaction
<u>EOF</u>	Returns end-of-file status
<u>ErrorText</u>	Returns the error message associated with the Error event
<u>EvenRowStyle</u>	Controls the row style for even-numbered rows
<u>ExposeCellMode</u>	Controls behavior of clicked rightmost visible cell
<u>ExtendRightColumn</u>	True if rightmost column extends to edge of split
<u>FetchRowStyle</u>	Controls whether the FetchRowStyle event will be fired
<u>FetchStyle</u>	Controls whether the FetchCellStyle event fires for a column
<u>FirstRow</u>	Bookmark of row occupying first display line
<u>Font</u>	Specifies the typeface, size, and other text characteristics
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadAlignment</u>	Specifies column heading alignment
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading and caption font for an object
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for an object
<u>HeadLines</u>	Number of lines allocated for heading text
<u>HighlightRowStyle</u>	Controls the marquee style when set to Highlight Row
<u>HScrollHeight</u>	Returns the horizontal scroll bar height, if present
<u>hWnd</u>	Returns the window handle of the grid
<u>hWndEditor</u>	Returns the window handle of the grid's editor
<u>InactiveBackColor</u>	Sets/returns the inactive heading background color
<u>InactiveForeColor</u>	Sets/returns the inactive heading foreground color
<u>InactiveStyle</u>	Controls the inactive heading style for an object
<u>Index</u>	Returns the ordinal index of a split
<u>IntegralHeight</u>	Controls whether partial rows are displayed
<u>LayoutFileName</u>	Sets/returns the name of a file containing grid layouts
<u>LayoutName</u>	Sets/returns the name of the current grid layout
<u>Layouts</u>	Returns a collection of layout names
<u>Left</u>	Returns column left border in container coordinates
<u>LeftCol</u>	Sets/returns the leftmost visible column
<u>ListField</u>	Sets/returns the name of the incremental search column

<u>Locked</u>	If true, data entry prohibited for an object
<u>MarqueeStyle</u>	Sets/returns marquee style for a split
<u>MarqueeUnique</u>	Restricts display of marquee to current split
<u>MaxComboItems</u>	Maximum number of items shown in a drop-down combo
<u>MultipleLines</u>	Controls whether individual records span multiple lines
<u>Name</u>	Returns the programmer-specified style name
<u>NumberFormat</u>	Data formatting string for a column
<u>OddRowStyle</u>	Controls the row style for odd-numbered rows
<u>Order</u>	Sets/returns the display position of a column
<u>Parent</u>	Sets/returns the object from which a style inherits
<u>Presentation</u>	Specifies how value items are displayed
<u>RecordSelectors</u>	Shows/hides selection panel at left border
<u>Row</u>	Specifies display line of current data row
<u>RowCount</u>	Returns the number of rows in a RowBuffer
<u>RowDividerStyle</u>	Selects style of row divider lines
<u>RowHeight</u>	Specifies height of all grid rows
<u>ScrollBars</u>	Sets/returns scroll bar style for an object
<u>ScrollGroup</u>	Used to synchronize vertical scrolling between splits
<u>SelBookmarks</u>	Contains a collection of selected row bookmarks
<u>SelectedBackColor</u>	Sets/returns the selected row and column background color
<u>SelectedForeColor</u>	Sets/returns the selected row and column foreground color
<u>SelectedItem</u>	Sets/returns bookmark of currently selected item
<u>SelectedStyle</u>	Controls the selected row and column style for an object
<u>SelEndCol</u>	Sets/returns rightmost selected column
<u>SelLength</u>	Sets/returns length of selected text
<u>SelStart</u>	Sets/returns start of selected text
<u>SelStartCol</u>	Sets/returns leftmost selected column
<u>SelText</u>	Sets/returns the selected text
<u>Size</u>	Sets/returns split width according to SizeMode
<u>SizeMode</u>	Controls whether a split is scalable or fixed size
<u>Split</u>	Sets/returns current split number
<u>Splits</u>	Contains a collection of True DBGrid splits
<u>Style</u>	Controls the normal style for an object
<u>Styles</u>	Contains a collection of True DBGrid styles
<u>TabAcrossSplits</u>	Allows tab and arrow keys to cross split boundaries
<u>TabAction</u>	Defines the behavior of the tab key
<u>Text</u>	Sets/returns displayed cell text for the current row
<u>Top</u>	Returns top column border in container coordinates
<u>Translate</u>	True to translate data values to display values
<u>Validate</u>	True to auto-validate input values
<u>Value (Column)</u>	Sets/returns underlying data value for the current row
<u>Value (RowBuffer)</u>	Sets/returns data value in unbound events
<u>Value (Style)</u>	Returns the programmer-specified style name
<u>Value (ValueItem)</u>	Sets/returns untranslated data value
<u>ValueItems</u>	Contains a collection of ValueItems for a column
<u>Visible</u>	Shows/hides a column

<u>VisibleCols</u>	Returns number of visible columns
<u>VisibleRows</u>	Returns number of visible display rows
<u>VScrollWidth</u>	Returns the vertical scroll bar width, if present
<u>Width</u>	Sets/returns column width in container coordinates
<u>WrapCellPointer</u>	Defines behavior of tab and arrow keys at row boundaries
<u>WrapText</u>	True if cell text is word wrapped

TDBGrid Control Properties

<u>AddNewMode</u>	Returns the disposition of the grid's AddNew row
<u>AllowAddNew</u>	Enables interactive record addition
<u>AllowArrows</u>	Enables use of arrow keys for grid navigation
<u>AllowColMove</u>	Enables interactive column movement
<u>AllowColSelect</u>	Enables interactive column selection
<u>AllowDelete</u>	Enables interactive record deletion
<u>AllowRowSelect</u>	Enables interactive row selection
<u>AllowRowSizing</u>	Enables interactive row resizing
<u>AllowUpdate</u>	Enables interactive record updating
<u>AlternatingRowStyle</u>	Controls whether even/odd row styles are applied to a grid
<u>Appearance</u>	Controls 3-D display of headings, caption, record selectors
<u>ApproxCount</u>	Sets/returns the approximate number of rows
<u>Array</u>	Specifies an XArray object as the data source
<u>BackColor</u>	Sets/returns the background color
<u>BOF</u>	Returns beginning-of-file status
<u>Bookmark</u>	Sets/returns bookmark of current row
<u>BorderStyle</u>	Sets/returns style for grid border
<u>Caption</u>	Sets/returns grid caption text
<u>CaptionStyle</u>	Controls the caption style for a grid
<u>CellTips</u>	Enables pop-up cell tip window when the cursor is idle
<u>CellTipsDelay</u>	Sets/returns idle time for cell tip window
<u>CellTipsWidth</u>	Sets/returns width of cell tip window
<u>Col</u>	Sets/returns current column number
<u>ColumnHeaders</u>	Turns column headings on or off
<u>Columns</u>	Contains a collection of True DBGrid columns
<u>CurrentCellModified</u>	Sets/returns modification status of the current cell
<u>CurrentCellVisible</u>	Sets/returns the visibility of the current cell
<u>DataChanged</u>	Sets/returns modification status of the current row
<u>DataMode</u>	Specifies bound or unbound mode
<u>DataSource</u>	Specifies source of grid data
<u>DefColWidth</u>	Specifies column width for auto-created columns
<u>EditActive</u>	Returns status or enters/exits the cell editor
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditDropDown</u>	Controls whether a drop-down window is used for editing
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditorStyle</u>	Controls the editor style for a grid
<u>EmptyRows</u>	Enables empty rows in an underpopulated grid
<u>Enabled</u>	Enables or disables user interaction
<u>EOF</u>	Returns end-of-file status
<u>ErrorText</u>	Returns the error message associated with the Error event
<u>EvenRowStyle</u>	Controls the row style for even-numbered rows
<u>ExposeCellMode</u>	Controls behavior of clicked rightmost visible cell
<u>ExtendRightColumn</u>	Returns current split extend column setting, sets all splits
<u>FetchRowStyle</u>	Controls whether the FetchRowStyle event will be fired

<u>FirstRow</u>	Bookmark of row occupying first display line
<u>Font</u>	Specifies the overall font for a grid
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading and caption font for a grid
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a grid
<u>HeadLines</u>	Number of lines allocated for heading text
<u>HighlightRowStyle</u>	Controls the marquee style when set to Highlight Row
<u>HScrollHeight</u>	Returns the horizontal scroll bar height, if present
<u>hWnd</u>	Returns the window handle of the grid
<u>hWndEditor</u>	Returns the window handle of the grid's editor
<u>InactiveBackColor</u>	Sets/returns the inactive heading background color
<u>InactiveForeColor</u>	Sets/returns the inactive heading foreground color
<u>InactiveStyle</u>	Controls the inactive heading style for a grid
<u>LayoutFileName</u>	Sets/returns the name of a file containing grid layouts
<u>LayoutName</u>	Sets/returns the name of the current grid layout
<u>Layouts</u>	Returns a collection of layout names
<u>LeftCol</u>	Sets/returns the leftmost visible column
<u>MarqueeStyle</u>	Returns current split marquee style, sets all splits
<u>MarqueeUnique</u>	Restricts display of marquee to current split
<u>MultipleLines</u>	Controls whether individual records span multiple lines
<u>OddRowStyle</u>	Controls the row style for odd-numbered rows
<u>RecordSelectors</u>	Shows/hides selection panel at left border
<u>Row</u>	Specifies display line of current data row
<u>RowDividerStyle</u>	Selects style of row divider lines
<u>RowHeight</u>	Specifies height of all grid rows
<u>ScrollBars</u>	Sets/returns scroll bar style for the grid
<u>SelBookmarks</u>	Contains a collection of selected row bookmarks
<u>SelectedBackColor</u>	Sets/returns the selected row background color
<u>SelectedForeColor</u>	Sets/returns the selected row foreground color
<u>SelectedStyle</u>	Controls the selected row and column style for a grid
<u>SelEndCol</u>	Sets/returns rightmost selected column
<u>SelLength</u>	Sets/returns length of selected text
<u>SelStart</u>	Sets/returns start of selected text
<u>SelStartCol</u>	Sets/returns leftmost selected column
<u>SelText</u>	Sets/returns the selected text
<u>Split</u>	Sets/returns current split number
<u>Splits</u>	Contains a collection of True DBGrid splits
<u>Style</u>	Controls the normal style for a grid
<u>Styles</u>	Contains a collection of True DBGrid styles
<u>TabAcrossSplits</u>	Allows tab and arrow keys to cross split boundaries
<u>TabAction</u>	Defines the behavior of the tab key
<u>Text</u>	Sets/returns displayed cell text for the current row
<u>VisibleCols</u>	Returns number of visible columns
<u>VisibleRows</u>	Returns number of visible display rows

VScrollWidth

Returns the vertical scroll bar width, if present

WrapCellPointer

Defines behavior of tab and arrow keys at row boundaries

TBDDropDown Control Properties

<u>AllowColMove</u>	Enables interactive column movement
<u>AllowColSelect</u>	Enables interactive column selection
<u>AllowRowSizing</u>	Enables interactive row resizing
<u>AlternatingRowStyle</u>	Controls whether even/odd row styles are applied
<u>Appearance</u>	Controls 3-D display of column headings
<u>ApproxCount</u>	Sets/returns the approximate number of rows
<u>Array</u>	Specifies an XArray object as the data source
<u>BackColor</u>	Sets/returns the background color
<u>Bookmark</u>	Sets/returns bookmark of current row
<u>BorderStyle</u>	Sets/returns style for drop-down border
<u>Col</u>	Sets/returns current column number
<u>ColumnHeaders</u>	Turns column headings on or off
<u>Columns</u>	Contains a collection of drop-down columns
<u>CurrentCellVisible</u>	Sets/returns the visibility of the current cell
<u>DataField</u>	Grid column to be updated with drop-down selection
<u>DataMode</u>	Specifies bound or unbound mode
<u>DataSource</u>	Specifies source of drop-down data
<u>DefColWidth</u>	Specifies column width for auto-created columns
<u>EmptyRows</u>	Enables empty rows in an underpopulated drop-down
<u>Enabled</u>	Enables or disables user interaction
<u>ErrorText</u>	Returns the error message associated with the Error event
<u>EvenRowStyle</u>	Controls the row style for even-numbered rows
<u>ExtendRightColumn</u>	Sets/returns extended right column for a drop-down
<u>FetchRowStyle</u>	Controls whether the FetchRowStyle event will be fired
<u>FirstRow</u>	Bookmark of row occupying first display line
<u>Font</u>	Specifies the overall font for a drop-down
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading font for a drop-down
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a drop-down
<u>HeadLines</u>	Number of lines allocated for heading text
<u>HighlightRowStyle</u>	Controls the marquee style when set to Highlight Row
<u>hWnd</u>	Returns the window handle of the drop-down
<u>IntegralHeight</u>	Controls whether partial rows are displayed
<u>LeftCol</u>	Sets/returns the leftmost visible column
<u>ListField</u>	Sets/returns the name of the incremental search column
<u>OddRowStyle</u>	Controls the row style for odd-numbered rows
<u>Row</u>	Specifies display line of current data row
<u>RowDividerStyle</u>	Selects style of row divider lines
<u>RowHeight</u>	Specifies height of all drop-down rows
<u>ScrollBars</u>	Sets/returns scroll bar style for the drop-down
<u>SelectedItem</u>	Sets/returns bookmark of currently selected item
<u>SelEndCol</u>	Sets/returns rightmost selected column

SelStartCol

Sets/returns leftmost selected column

Style

Controls the normal style for a drop-down

Styles

Contains a collection of drop-down styles

Text

Sets/returns displayed cell text for the current row

VisibleCols

Returns number of visible columns

VisibleRows

Returns number of visible display rows

Column Object Properties

<u>Alignment</u>	Specifies horizontal text alignment
<u>AllowFocus</u>	Controls whether a column can receive focus
<u>AllowSizing</u>	Enables interactive resizing for a column
<u>BackColor</u>	Sets/returns the background color
<u>Button</u>	Controls whether a button appears within the current cell
<u>ButtonPicture</u>	Sets/returns the bitmap used for the in-cell button
<u>Caption</u>	Sets/returns column heading text
<u>CellTop</u>	Returns top column border, adjusted for multiple lines
<u>ColIndex</u>	Returns the ordinal position of a column
<u>DataChanged</u>	Sets/returns modification status of a column in current row
<u>DataField</u>	Data table field name for a column
<u>DataWidth</u>	Maximum number of characters available for column input
<u>DefaultValue</u>	Default value for new column data
<u>DividerStyle</u>	Divider style for right column border
<u>DropDown</u>	Sets the TDBDropDown control for a column
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditMask</u>	Input mask string for a column
<u>EditMaskUpdate</u>	Controls whether masked data is used for updates
<u>EditorStyle</u>	Controls the editor style for a column
<u>FetchStyle</u>	Controls whether the FetchCellStyle event fires for a column
<u>Font</u>	Specifies the overall font for a column
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadAlignment</u>	Specifies column heading alignment
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading font for a column
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a column
<u>Left</u>	Returns column left border in container coordinates
<u>Locked</u>	If true, data entry prohibited for a column
<u>NumberFormat</u>	Data formatting string for a column
<u>Order</u>	Sets/returns the display position of a column
<u>Style</u>	Controls the normal style for a column
<u>Text</u>	Sets/returns displayed cell text for the current row
<u>Top</u>	Returns top column border in container coordinates
<u>Value</u>	Sets/returns underlying data value for the current row
<u>ValueItems</u>	Contains a collection of ValueItems for a column
<u>Visible</u>	Shows/hides a column
<u>Width</u>	Sets/returns column width in container coordinates
<u>WrapText</u>	True if cell text is word wrapped

Columns Collection Properties

Count

Returns the number of columns in the collection

Layouts Collection Properties

Count

Returns the number of layouts in the collection

RowBuffer Object Properties

<u>Bookmark</u>	Sets/returns bookmark of specified row
<u>ColumnCount</u>	Returns the number of columns
<u>ColumnIndex</u>	Returns the column index of a column
<u>ColumnName</u>	Returns the field name of a column
<u>RowCount</u>	Returns the number of rows
<u>Value</u>	Sets/returns data value of specified row/column

SelBookmarks Collection Properties

Count

Returns the number of selected rows

Split Object Properties

<u>AllowColMove</u>	Enables interactive column movement
<u>AllowColSelect</u>	Enables interactive column selection
<u>AllowFocus</u>	Allows cells within a split to receive focus
<u>AllowRowSelect</u>	Enables interactive row selection
<u>AllowRowSizing</u>	Enables interactive row resizing
<u>AllowSizing</u>	Enables interactive resizing for a split
<u>AlternatingRowStyle</u>	Controls whether even/odd row styles are applied to a split
<u>BackColor</u>	Sets/returns the background color
<u>Caption</u>	Sets/returns split caption text
<u>CaptionStyle</u>	Controls the caption style for a split
<u>Columns</u>	Returns a collection of column objects for a split
<u>CurrentCellVisible</u>	Sets/returns modification status of the current cell
<u>EditBackColor</u>	Sets/returns the editor background color
<u>EditForeColor</u>	Sets/returns the editor foreground color
<u>EditorStyle</u>	Controls the editor style for a split
<u>EvenRowStyle</u>	Controls the row style for even-numbered rows
<u>ExtendRightColumn</u>	Sets/returns extended right column for a split
<u>FetchRowStyle</u>	Controls whether the FetchRowStyle event will be fired
<u>FirstRow</u>	Bookmark of row occupying first display line
<u>Font</u>	Specifies the overall font for a split
<u>ForeColor</u>	Sets/returns the foreground color
<u>HeadBackColor</u>	Sets/returns the heading background color
<u>HeadFont</u>	Specifies the heading font for a split
<u>HeadForeColor</u>	Sets/returns the heading foreground color
<u>HeadingStyle</u>	Controls the heading style for a split
<u>HighlightRowStyle</u>	Controls the marquee style when set to Highlight Row
<u>HScrollHeight</u>	Returns the horizontal scroll bar height, if present
<u>InactiveBackColor</u>	Sets/returns the inactive heading background color
<u>InactiveForeColor</u>	Sets/returns the inactive heading foreground color
<u>InactiveStyle</u>	Controls the inactive heading style for a split
<u>Index</u>	Returns the ordinal index of a split
<u>LeftCol</u>	Returns the leftmost visible column
<u>Locked</u>	If true, data entry prohibited for a split
<u>MarqueeStyle</u>	Sets/returns marquee style for a split
<u>OddRowStyle</u>	Controls the row style for odd-numbered rows
<u>RecordSelectors</u>	Shows/hides selection panel at left border
<u>ScrollBars</u>	Sets/returns scroll bar style for a split
<u>ScrollGroup</u>	Used to synchronize vertical scrolling between splits
<u>SelectedBackColor</u>	Sets/returns the selected row background color
<u>SelectedForeColor</u>	Sets/returns the selected row foreground color
<u>SelectedStyle</u>	Controls the selected row and column style for an object
<u>SelEndCol</u>	Sets/returns rightmost selected column
<u>SelStartCol</u>	Sets/returns leftmost selected column
<u>Size</u>	Sets/returns split width according to SizeMode

SizeMode

Controls whether a split is scalable or fixed size

Style

Controls the normal style for an object

VScrollWidth

Returns the vertical scroll bar width, if present

Splits Collection Properties

Count

Returns the total number of splits

Style Object Properties

<u>Alignment</u>	Specifies the horizontal text alignment
<u>BackColor</u>	Controls the background color
<u>Font</u>	Specifies the typeface, size, and other text characteristics
<u>ForeColor</u>	Controls the foreground color
<u>Locked</u>	Disallows in-cell editing when true
<u>Name</u>	Returns the programmer-specified style name
<u>Parent</u>	Sets/returns the object from which a style inherits
<u>Value</u>	Returns the programmer-specified style name
<u>WrapText</u>	Word wraps cell text when true

Styles Collection Properties

Count

Returns the number of styles in the collection

ValueItem Object Properties

DisplayValue

Sets/returns displayed text or graphics

Value

Sets/returns untranslated data value

ValueItems Collection Properties

<u>AnnotatePicture</u>	True to show both underlying data and display value graphics
<u>Count</u>	Returns the number of value items in the collection
<u>CycleOnClick</u>	True to cycle through value items on click
<u>DefaultItem</u>	Index of default value item, or -1 if no default
<u>MaxComboItems</u>	Maximum number of items shown in a drop-down combo
<u>Presentation</u>	Specifies how value items are displayed
<u>Translate</u>	True to translate data values to display values
<u>Validate</u>	True to auto-validate input values

AddNewMode Property

Syntax TDBGrid.**AddNewMode**
Read-only at run time. Not available at design time.
Property applies to **TDBGrid** control.

Values	Description	Run Time
0 - No AddNew pending		dbgNoAddNew
1 - Current cell in AddNew row		dbgAddNewCurrent
2 - AddNew pending		dbgAddNewPending

Description The **AddNewMode** property returns a value that describes the location of the current cell with respect to the grid's AddNew row.

If the **AllowAddNew** property is True, the last row displayed in the grid is left blank to permit users to enter new records. If the **AllowAddNew** property is False, the blank row is not displayed, and **AddNewMode** always returns 0.

When **AllowAddNew** is True, the **AddNewMode** property returns one of the following values:

dbgNoAddNew	The current cell is not in the last row, and no AddNew operation is pending.
dbgAddNewCurrent	The current cell is in the last row, but no AddNew operation is pending.
dbgAddNewPending	The current cell is in the next to last row as a result of a pending AddNew operation initiated by the user through the grid's user interface, or by code as a result of setting the Value or Text properties of a column.

Note This property is valid in both bound and unbound modes.

Alignment Property

Syntax object.**Alignment** = value

Read/Write at run time and design time.

Property applies to **Column** and **Style** objects.

Values	Design Time	Run Time
	0 - Left (default)	dbgLeft
	1 - Right	dbgRight
	2 - Center	dbgCenter
	3 - General	dbgGeneral

Description The **Alignment** property returns or sets a value that determines the horizontal alignment of the values in a grid column or style object.

The General setting means that text will be left-aligned and numbers will be right-aligned. This setting is only useful in bound mode, where the grid can query the data source to determine the data types of individual columns.

AllowAddNew Property

- Syntax** TDBGrid.**AllowAddNew** = boolean
- Read/Write at run time and design time.
Property applies to **TDBGrid** control.
- Description** If True, the user can add records to the data source underlying the **TDBGrid** control.
If False (the default), the user cannot add records to the data source underlying the **TDBGrid** control.
- If the **AllowAddNew** property is True, the last row displayed in the grid is left blank to permit users to enter new records. If the **AllowAddNew** property is False, the blank row (usually referred to as the *AddNew row*) is not displayed.
- The underlying data source may not permit insertions even if the **AllowAddNew** property is True for the **TDBGrid** control. In this case, a trappable error occurs when the user tries to add a record.
- Note** If **AllowAddNew** is True, you should also set **AllowUpdate** to True so that users will be able to type in the cells of the AddNew row.

AllowArrows Property

Syntax TDBGrid.**AllowArrows** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description If True (the default), the user can use the arrow keys to move from cell to cell within the same row.

If False, the left and right arrow keys will move focus from control to control and cannot be used to move between cells.

The user cannot use the arrow keys to move out of the **TDBGrid** control when this property is set to True. If the **WrapCellPointer** property is also set to True, then the arrow keys will wrap around rows and the user can navigate the entire grid using the arrow keys.

AllowColMove Property

Syntax

object.**AllowColMove** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description

If True, the user can move selected columns.

If False (the default), the user cannot move selected columns.

Use the **AllowColMove** property to control whether the user can move selected columns by dragging the column divider highlight bar to the desired location. Any change in column order causes a **ColMove** event.

Note

The **AllowColSelect** property must also be True in order for the user to move selected columns.

AllowColSelect Property

Syntax object.**AllowColSelect** = boolean
Read/Write at run time and design time.
Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description If True (the default), the user can select columns.
If False, the user cannot select columns.
Use the **AllowColSelect** property to control whether the user can select columns by clicking or dragging within the column header area. Setting this property to False suppresses highlighting when the user clicks a column header, which is useful for applications that respond to the **HeadClick** event.

Note Both the **AllowColSelect** and **AllowColMove** properties must be True in order for the user to move selected columns.

AllowDelete Property

Syntax TDBGrid.**AllowDelete** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description If True, the user can delete records from the data source underlying the **TDBGrid** control.

If False (the default), the user cannot delete records from the data source underlying the **TDBGrid** control.

Use the **AllowDelete** property to prevent the user from deleting records from the data source through interaction with the **TDBGrid** control.

The underlying data source may not permit deletions even if the **AllowDelete** property is True for the **TDBGrid** control. In this case, a trappable error occurs when the user tries to delete a record.

AllowFocus Property

Syntax

object.**AllowFocus** = boolean

Read/Write at run time and design time.

Property applies to **Split** and **Column** objects.

Description

If True (the default), the user will be able to interactively select the object, giving it focus.

If False, the user will not be able to interactively select the object. When clicked, the object will not receive focus and the control (or grid column) that previously had focus will retain it.

For both split and column objects, setting **AllowFocus** to True enables cells within the object to receive focus. If set to False, there is no way to change the focus to a cell within the object.

However, if an object already has the focus, setting **AllowFocus** to False will not give focus to another split, column, or control.

If a cell in a column which does not allow focus is clicked, and the cell is in a row other than the current row, then the row is changed, but the column with the focus retains it.

For splits, you can use this property in combination with the **AllowSizing** property to completely prohibit the user from making any changes to a split (by setting both properties to False).

Unselectable splits are passed over when **TabAcrossSplits** is set to True.

For columns, **AllowFocus** is a *split-specific* property, which means that the following statements are equivalent:

```
TDBGrid1.Columns(1).AllowFocus = True
```

```
TDBGrid1.Splits(TDBGrid1.Split).Columns(1).AllowFocus = True
```

In other words, both of these statements affect column 1 in the current split only.

Note

At design time, the **AllowFocus** property appears in both the Splits property page (for **Split** objects) and the Layout property page (for **Column** objects).

AllowRowSelect Property

- Syntax** object.**AllowRowSelect** = boolean
- Read/Write at run time and design time.
Property applies to **TDBGrid** control and **Split** object.
- Description** If True (the default), the user can select rows.
If False, the user cannot select rows.
- Use the **AllowRowSelect** property to control whether the user can select rows by clicking the record selector buttons. By setting this property to False, you can disable record selection without hiding the record selectors altogether, since you may want to use the record selectors to provide visual feedback when the current row is modified.
- Note** The user cannot select rows if the **RecordSelectors** property is set to False for all splits, even if **AllowRowSelect** is True.

AllowRowSizing Property

Syntax object.**AllowRowSizing** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description If True (the default), the user can resize rows.

If False, the user cannot resize rows.

If the **AllowRowSizing** property is True, the mouse pointer turns into a double-headed arrow when positioned over the row divider between any pair of record selectors, and the user can resize the rows by dragging. Any change in row size causes a **RowResize** event.

All rows of the **TDBGrid** control are always the same height, which is determined by the **RowHeight** property.

Note The user cannot resize rows if the **RecordSelectors** property is set to False for all splits.

AllowSizing Property

Syntax

object.**AllowSizing** = boolean

Read/Write at run time and design time.

Property applies to **Column** and **Split** objects.

Description

If True, the user can resize the column or split.

If False, the user cannot resize the column or split.

For columns, **AllowSizing** defaults to True. For splits, **AllowSizing** defaults to False.

If **AllowSizing** is True for a column, the mouse pointer turns into a double-headed arrow when positioned over that column's divider within the column heading area, and the user can resize the column by dragging. Any change in column size causes a **ColResize** event.

For the leftmost split with **AllowSizing** set to True, the mouse pointer turns into a pair of vertical lines with a downward arrow when positioned over that split's size box (at the lower left corner), and the user can create a new split by dragging. The creation of a new split causes a **SplitChange** event.

If **AllowSizing** is True for any other split, the mouse pointer turns into a pair of vertical lines with a double-headed arrow when positioned over that split's size box, and the user can resize the split by dragging. No event is fired in this case (except for the standard mouse events).

AllowUpdate Property

Syntax TDBGrid.**AllowUpdate** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description If True (the default), the user can modify data in the **TDBGrid** control.

If False, the user cannot modify data in the **TDBGrid** control.

When the **AllowUpdate** property is False, the user can still scroll through the **TDBGrid** control and select data, but cannot change any of the values; any attempt to change the data in the grid is ignored.

The underlying data source may not permit updates even if the **AllowUpdate** property is True for the **TDBGrid** control. In this case, a trappable error occurs when the user tries to change the record.

You can also use the **Column** object's **Locked** property to make individual columns of the **TDBGrid** control read-only, even if the **AllowUpdate** property is True. However, if **AllowUpdate** is False, then this setting takes precedence over the column settings (without changing the column settings).

AlternatingRowStyle Property

Syntax object.**AlternatingRowStyle** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property determines whether a grid or split displays odd-numbered rows in one style and even-numbered rows in another.

If True, the **OddRowStyle** and **EvenRowStyle** properties control the appearance of rows within the specified object.

If False (the default), the **Style** property controls the display of rows within the specified object.

At design time, you can change the colors and fonts used to render odd (even) rows by modifying the built-in OddRow (EvenRow) style using the Styles property page.

At run time, you can change the colors and fonts used to render odd (even) rows by modifying the **Style** object returned by the **OddRowStyle** (**EvenRowStyle**) property.

AnnotatePicture Property

Syntax

valueitems.**AnnotatePicture** = boolean

Read/Write at run time and design time.
Property applies to **ValueItems** collection.

Description

This property determines whether the column associated with a **ValueItems** collection can display both text and graphics in a single cell.

If True, both text and graphics are displayed in a cell when all of the following conditions are met:

- The **Presentation** property of the **ValueItems** collection is set to any value except 1 - Radio Button.
- The **Translate** property of the **ValueItems** collection is set to True.
- The underlying data value for a cell matches the **Value** property of a **ValueItem** member.
- The corresponding **DisplayValue** contains a bitmap, not text.

If False (the default), matching cells are rendered as the **Value** or **DisplayValue** setting, depending upon the value of the **Translate** property.

When both text and graphics are displayed, the placement of the bitmap within the cell is determined by the column's **Alignment** property. Left alignment places the bitmap on the left, and the text is formatted in the remaining space to the right of the bitmap. Right alignment places the bitmap on the right, and the text is formatted in the remaining space to the left of the bitmap. Center alignment places the bitmap in the center at the top of the cell, and the text is formatted in the remaining space below the bitmap. In all cases, the text is centered in the space allotted for it.

When editing, the editor uses all space available in the text portion of the cell. If the **Presentation** property is set to one of the combo box options, the bitmap will not change until editing is completed.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

Appearance Property

Syntax object.**Appearance** = value

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Values	Design Time	Run Time
0 - Flat		dbgFlat
1 - 3D (default)		dbg3D

Description When this property is set to 1 - 3D, the control paints its caption, headings, and record selectors with three-dimensional effects.

When this property is set to 0 - Flat, no visual effects are used.

The **Appearance** property is independent of the **BorderStyle** and **BackColor** properties and only affects the control's caption, headings, and record selectors. This behavior differs from that of many common ActiveX controls.

This property only affects the way in which the caption, headings, and record selectors are drawn; it does not affect their visibility. Use the **Caption**, **ColumnHeaders**, and **RecordSelectors** properties to control the visibility of these components.

Note This property is supported in both 16- and 32-bit versions of True DBGrid. Many 16-bit versions of common controls do not provide an **Appearance** property.

ApproxCount Property

Syntax	<code>object.ApproxCount = long</code> Read/Write at run time. Not available at design time. Property applies to TDBGrid and TDBDropDown controls.
Description	This property sets or returns the approximate row count used by the grid to calibrate the vertical scroll bar. Typically, the ApproxCount property is used in unbound mode to improve the accuracy of the vertical scroll bar. This is particularly useful for situations where the number of rows is known in advance, such as when an unbound grid is used in conjunction with an array.
Note	For a bound grid, setting the ApproxCount property has no effect. However, getting the ApproxCount property will query the underlying data source.

Array Property

Syntax object.**Array** = XArray

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description For a grid with its **DataMode** property set to 4 - Storage, the **Array** property specifies an APEX **XArray** object that acts as a data source. This property has no effect in other data modes.

Note At the time of this writing, APEX does not provide a 16-bit version of **XArray**, hence the **Array** property is not supported in 16-bit versions of True DBGrid.

BackColor Property

Syntax	<p>object.BackColor = color</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid and TDBDropDown controls; Column, Split, and Style objects.</p>
Description	<p>This property controls the background color of an object. Colors may be specified as RGB values or system default colors.</p> <p>By default, the background color of a grid, column, or split is determined by its Style property setting. Setting the BackColor property overrides the style setting without changing the definition of the style itself.</p> <p>If the BackColor property of a grid, column, or split is changed to the same value as the BackColor property of its corresponding style, then the object will inherit its background color from the style, and subsequent changes to the style's BackColor property will affect the object as well.</p> <p>For Style objects, the value of the BackColor property is inherited from the parent style (if any) unless explicitly overridden, in which case the aforementioned inheritance rules also apply.</p>

BOF Property

Syntax TDBGrid.**BOF**

Read-only at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description The **BOF** property operates like its **Recordset** counterpart. It returns True if the current record position is before the first record, False if the current record position is on or after the first record.

If the data source contains no records, then **BOF** will always return True.

Bookmark Property (RowBuffer)

Syntax rowbuffer.**Bookmark** (Row) = variant

Read/Write at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns or sets a bookmark for the specified row within a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control.

The Row argument is a long integer specifying the row where the bookmark is placed. The range of this argument can be from 0 to **RowCount** - 1.

In unbound mode, a bookmark contains a user-defined value that uniquely identifies each row of data.

In the **UnboundReadData** and **UnboundAddData** events, your code must provide bookmarks for the rows being fetched or added. In the **UnboundWriteData** and **UnboundDeleteRow** events, the grid passes one of these bookmarks as a parameter so your code can take the appropriate action.

Bookmark Property (TDBGrid)

Syntax object.**Bookmark** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property returns or sets the bookmark identifying the current row in a **TDBGrid** or **TDBDropDown** control.

Use the value returned by the **Bookmark** property to save a reference to the current row that remains valid even after another row becomes current.

When you set the **Bookmark** property to a valid value in code, the row associated with that value becomes the current row, and the grid adjusts its display to bring the new current row into view if necessary.

The **Bookmark** property is defined as a Variant to accommodate user-defined bookmarks in unbound mode.

Note In unbound mode, setting the **Bookmark** property to itself will force the current row to be updated via the **UnboundWriteData** event.

BorderStyle Property

Syntax	object. BorderStyle = value	
	Read/Write at run time and design time.	
	Property applies to TDBGrid and TDBDropDown controls.	
Values	Design Time	Run Time
	0 - None	dbgNoBorder
	1 - Fixed Single (default)	dbgFixedSingle
Description	This property determines whether a TDBGrid or TDBDropDown control has a border.	

Button Property

Syntax column.**Button** = boolean

Read/Write at run time and design time.
Property applies to **Column** object.

Description If True, a button will be displayed in the upper right corner of the current cell at run time.
If False (the default), no button will be displayed.

Typically, you enable the column button when you want to drop down a Visual Basic control (such as the built-in combo box, a bound list box, or even another True DBGrid control) for editing or data entry. When the button in the current cell is clicked, the **ButtonClick** event will be fired. You can then write code to drop down the desired control from the cell.

Note When you set the **Presentation** property of a column's **ValueItems** collection to either of the combo box options (sorted or unsorted), then the **Button** property for that column will be set to True. Similarly, if you set the **Presentation** property to the normal (text) or radio button option, then the **Button** property for that column will be set to False.

When you set the **DropDown** property of a column to the name of a **TDBDropDown** control, then the **Button** property for that column will be set to True. Similarly, if you set the **DropDown** property of a column to an empty string, then the **Button** property for that column will be set to False.

ButtonPicture Property

Syntax column.**ButtonPicture** = variant

Read/Write at run time and design time.

Property applies to **Column** object.

Description This property sets or returns the bitmap used to draw the in-cell button for the current cell in the specified column. The in-cell button is enabled when any of the following are true:

- The column's **Button** property is set to True.
- The column's **DropDown** property is set to True.
- The **Presentation** property of the column's **ValueItems** collection is set to 2 - Combo Box or 3 - Sorted Combo Box.

By default, True DBGrid uses a down arrow  for the in-cell button. However, you can change the button bitmap at design time by clicking the **Picture** button on the Layout property page. Or, you can assign a bitmap to the **ButtonPicture** property in code at run time:

```
TDBGrid1.Columns(1).ButtonPicture = LoadPicture("arrow.bmp")
```

Note The grid automatically draws the edges corresponding to the button's up/down states as appropriate, so you need only provide the interior image (a light gray background is recommended).

Caption Property

Syntax object.**Caption** = string

Read/Write at run time and design time.

Property applies to **TDBGrid** control, **Column** and **Split** objects.

Description For a **TDBGrid** control, this property determines the text displayed in the caption bar at the top of the grid.

For a **Column** or **Split** object, this property determines the text displayed in the object's heading area.

Setting the **Caption** property to an empty string for a **TDBGrid** control hides its caption bar.

Setting the **Caption** property to an empty string for a **Split** object hides the heading area, even if other splits have non-empty captions.

Setting the **Caption** property to an empty string for a **Column** object clears the text in the column's heading area but does not hide the heading. Column captions are only displayed if the **TDBGrid** control's **ColumnHeaders** property is set to True and the **HeadLines** property is not set to 0.

The **Caption** property is limited to 255 characters. Attempting to set the caption to more than 255 characters results in a trappable error.

CaptionStyle Property

Syntax object.**CaptionStyle** = variant

Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control and **Split** object.

Description This property sets or returns the **Style** object that controls the appearance of a **TDBGrid** control's caption bar or a **Split** object's heading area. By default, this is the built-in Caption style.

The value of the **Caption** property is not affected by changes to the **CaptionStyle** property.

CellTips Property

Syntax TDBGrid.**CellTips** = value
Read/Write at run time and design time.
Property applies to **TDBGrid** control.

Values	Design Time	Run Time
0 - None (default)		dbgNoCellTips
1 - Anchored		dbgAnchored
2 - Floating		dbgFloating

Description The **CellTips** property determines whether the grid displays a pop-up text window when the cursor is idle. By default, this property is set to 0 - None, and cell tips are not displayed.

If the **CellTips** property is set to either 1 - Anchored or 2 - Floating, the **FetchCellTips** event will be fired whenever the grid has focus and the cursor is idle over a grid cell, record selector, column header, split header, or grid caption. The event will not fire if the cursor is over the scroll bars.

The setting 1 - Anchored aligns the cell tip window with either the left or right edge of the cell. The left edge is favored, but the right edge will be used if necessary in order to display as much text as possible. The setting 2 - Floating displays the cell tip window below the cursor, if possible.

If you do not provide a handler for the **FetchCellTips** event, and the cursor is over a grid cell, the default behavior is to display a text box containing the cell's contents (up to 256 characters). This enables the user to peruse the contents of a cell even if it is not big enough to be displayed in its entirety. You can also program the **FetchCellTips** event to override the default cell text display in order to provide users with context-sensitive help.

Note Use the **CellTipsDelay** property to control the amount of idle time that must elapse before the cell tip window is displayed.

Use the **CellTipsWidth** property to control the width of the cell tip window.

CellTipsDelay Property

Syntax TDBGrid.**CellTipsDelay** = long

Read/Write at run time and design time.
Property applies to **TDBGrid** control.

Description The **CellTipsDelay** property controls the amount of idle time, in milliseconds, that must elapse before the cell tip window is displayed. By default, this property is set to 1000 (one second).
Setting this property to zero does not disable cell tips, but restores the default value of 1000.
To disable cell tips, set the **CellTips** property to 0 - None.

CellTipsWidth Property

Syntax TDBGrid.**CellTipsWidth** = single

Read/Write at run time and design time.
Property applies to **TDBGrid** control.

Description The **CellTipsWidth** property returns or sets the width of the cell tip window in terms of the coordinate system of the grid's container.

By default, this property is set to zero, which causes the cell tip window to grow or shrink to accommodate the cell tip text. You can override this behavior and give the cell tip window a fixed width by specifying a non-zero value for this property.

CellTop Property

Syntax

column.**CellTop**

Read-only at run time. Not available at design time.
Property applies to **Column** object.

Description

The **CellTop** property returns the vertical offset of the top of any cell in the specified column relative to the top of the containing row in terms of the coordinate system of the grid's container.

If the grid's **MultipleLines** property is False (the default value), a single record cannot span multiple lines in the grid, and the **CellTop** property returns zero for all columns.

If the grid's **MultipleLines** property is True, a single record may span multiple lines in the grid. For columns on the first line, the **CellTop** property returns zero. For columns on the second line, the **CellTop** property returns the cell height (the grid's **RowHeight** property). For columns on the third line, the **CellTop** property returns twice the cell height, and so on.

For example, the following code places a text box on top of the grid cell in the first column of the fourth displayed row:

```
With TDBGrid1
    Text1.Top = .Top + .RowTop(3) + .Columns(0).CellTop
End With
```

Note

To overlay the text box exactly on a grid cell, you may need to account for an extra pixel in the width and height, depending upon the settings of the **DividerStyle** and **RowDividerStyle** properties. In Visual Basic, if the **ScaleMode** property of the parent form is set to pixels, then you can simply add 1. If the **ScaleMode** is set to twips, then you can add the **TwipsPerPixelX** and **TwipsPerPixelY** properties of the **Screen** object.

Col Property

Syntax *object.Col* = integer

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property specifies the zero-based index of the current cell's column position. It may be set at run time to highlight a different cell within the current row. If the column is visible, the caret or marquee will be moved to the selected column. If the column is not visible, the grid will scroll to make it visible as a result of setting this property.

Setting the **Col** property at run time does not affect selected columns. Use the **SelEndCol** and **SelStartCol** properties to specify a selected region.

ColIndex Property

Syntax column.**ColIndex**

Read-only at run time. Not available at design time.
Property applies to **Column** object.

Description This property returns the zero-based index of a column within the **Columns** collection.

ColumnCount Property

Syntax rowbuffer.**ColumnCount**

Read-only at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns the number of columns in a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control.

ColumnHeaders Property

Syntax object.**ColumnHeaders** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description If True (the default), the control's column headers are displayed.

If False, the control's column headers are not displayed.

Use the **Caption** property to set the heading text of an individual **Column** object.

ColumnIndex Property

Syntax rowbuffer.**ColumnIndex** (Row, Col)

Read-only at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns the index of a column in a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control. The index corresponds to the **ColIndex** property of the grid column.

The **Col** argument is an integer specifying the desired column. The range of this argument can be from 0 to **ColumnCount** - 1.

The **Row** argument is an integer specifying the desired row. The range of this argument can be from 0 to **RowCount** - 1.

The **ColumnIndex** property allows you to determine the column index associated with a **RowBuffer** column. It is provided for identification of the column in order for the user to fill in the **Value** property array with appropriate column data.

ColumnName Property

Syntax rowbuffer.**ColumnName** (Col)

Read-only at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns the name of a column in a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control. The name corresponds to the **DataField** property of the grid column.

The Col argument is an integer specifying the desired column. The range of this argument can be from 0 to **ColumnCount** - 1.

The **ColumnName** property allows you to determine the field name associated with a **RowBuffer** column. It is provided for situations where the field names and/or column order are not known in advance.

Columns Property

Syntax object.**Columns**

Read-only at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property returns a collection of **Column** objects for a grid, drop-down control, or split.

Count Property

Syntax

collection.**Count**

Read-only at run time. Not available at design time.

Property applies to **Columns**, **Layouts**, **SelBookmarks**, **Splits**, **Styles**, and **ValueItems** collections.

Description

This property returns the number of items in a collection.

Collections are zero-based, which means that the items in a collection are indexed from 0 to **Count** - 1.

CurrentCellModified Property

Syntax TDBGrid.**CurrentCellModified** = boolean

Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description This property returns True if editing is in progress and the current cell (indicated by the **Bookmark** and **Col** properties) has been modified by the user. It returns False if the cell has not been modified or if editing is not in progress.

You can use this property to cancel any changes the user has made to the current text. For example, to program a function key to discard the user's changes (like the ESC key), trap the key code in the grid's **KeyDown** event and set **CurrentCellModified** to False. This will revert the current cell to its original contents.

CurrentCellVisible Property

Syntax object.**CurrentCellVisible** = boolean

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property returns True if the current cell (indicated by the **Bookmark** and **Col** properties) is visible within the displayed area of a grid or split. It returns False if the cell is not visible.

For a **TDBGrid** or **TDBDropDown** control, setting the **CurrentCellVisible** property to True causes the grid to scroll so that the current cell is brought into view. If a grid contains multiple splits, then the current cell becomes visible in each split.

For a **Split** object, setting the **CurrentCellVisible** property to True makes the current cell visible in that split only.

In all cases, setting this property to False is meaningless and is ignored.

CycleOnClick Property

- Syntax** valueitems.**CycleOnClick** = boolean
- Read/Write at run time and design time.
Property applies to **ValueItems** collection.
- Description** This property determines whether the user can cycle through the **ValueItem** objects contained in a column's **ValueItems** collection by clicking on the current cell.
- If True, the user can click on the current cell to display the next available item. If the last value item is displayed, then clicking displays the first item in the list.
- If False (the default), then the mouse operates as usual within the associated column.
- Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.
- Note** The **CycleOnClick** property has no effect when the **MarqueeStyle** property is set to 6 - Floating Editor.

DataChanged Property

Syntax	<p>object.DataChanged = boolean</p> <p>Read/Write at run time (TDBGrid). Read-only at run time (Column).</p> <p>Not available at design time.</p> <p>Property applies to TDBGrid control and Column object.</p>
Description	<p>For a TDBGrid control, the DataChanged property indicates the modification status of the current row. If True, then one or more columns in the current row have been modified. If False, then no changes have been made.</p> <p>When the DataChanged property of a TDBGrid control is True, you can use the DataChanged property of individual Column objects to determine the exact nature of the changes.</p> <p>For a TDBGrid control, setting this property to True has no effect. Setting this property to False exits editing, discards all changes to the current row, and refreshes the current row from the data source. Setting this property within the BeforeColUpdate event is disallowed, however.</p> <p>For a Column object, this property is read-only and cannot be set.</p>
Note	<p>The pencil in the RecordSelector column reflects the state of the grid's DataChanged property.</p>

DataField Property (Column)

Syntax

column.**DataField** = string

Read/Write at run time and design time.

Property applies to **Column** object.

Description

The **DataField** property returns or sets the name of the field in the database table to which a grid column is bound.

When the **DataMode** property of the grid is set to 0 - Bound, the **DataField** property is used to bind a column to a particular field in the database table. If the specified field does not exist in the database table, binding does not occur, and the column will be blank at run time.

To specify an unbound column in a bound grid, the **DataField** property must be empty in order for the column data to be requested in the **UnboundColumnFetch** event.

DataField Property (TDBDropDown)

Syntax TDBDropDown.**DataField** = string

Read/Write at run time and design time.
Property applies to **TDBDropDown** control.

Description The **DataField** property returns or sets the name of the grid column that will be updated when the user selects an item from a **TDBDropDown** control. The **DataField** property need not be the same as the **ListField** property used for incremental search.

If the **DataField** property is not specified, the **ListField** property specifies the column to be used for both incremental search and the selection value. If neither property is specified, then the first column in the **TDBDropDown** control will be used.

Note Do not confuse the **DataField** property of the **TDBDropDown** control with the **DataField** property of the **Column** object or intrinsic Visual Basic controls.

To associate a **TDBDropDown** control with a **Column** object that belongs to a **TDBGrid** control, set the column's **DropDown** property to the name of the drop-down control at either design time or run time.

DataMode Property

Syntax object.**DataMode** = value

Read-only at run time. Read/Write at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Values

Design Time

Run Time

0 - Bound	dbgBound
1 - Unbound	dbgUnbound
2 - Unbound Extended	dbgUnboundEx
3 - Application	dbgUnboundAp
4 - Storage	dbgUnboundSt

Description

When set to 0 - Bound, the control displays data available from its bound **DataSource**.

When set to 1 - Unbound, the control uses the original DBGrid unbound events to retrieve and update displayed data. When this mode is used, the grid fires the **UnboundReadData** event to fetch data. This setting was retained for backward compatibility with DBGrid and earlier versions of True DBGrid. If you are writing a new application, please use mode 2, 3, or 4 instead.

When set to 2 - Unbound Extended, the control uses the **UnboundReadDataEx** event to fetch data. The **UnboundReadDataEx** event is more efficient and easier to use than the **UnboundReadData** event of mode 1. This is the recommended setting for using the grid unbound with a database API that supports multiple-row fetches.

When set to 3 - Application, the control uses the **ClassicRead** event to fetch data one cell at a time. This mode is much easier to use than mode 2, particularly if data is being retrieved from a Visual Basic array. However, it can be less efficient than mode 2 if there are many columns because the grid needs to fire more events in order to retrieve data.

When set to 4 - Storage, the control uses an **XArray** object as a data source, and no unbound data retrieval or update events are fired. At run time, you create and populate an **XArray** object just as you would a standard Visual Basic array, then bind it to a **TDBGrid** or **TDBDropDown** control using the **Array** property. This is by far the simplest way to use True DBGrid in unbound mode.

Note

At the time of this writing, APEX does not provide a 16-bit version of **XArray**, hence mode 4 is not supported in 16-bit versions of True DBGrid.

DataSource Property

Syntax object.**DataSource**

Read/Write at design time. Not available at run time.
Property applies to **TDBGrid** and **TDBDropDown** controls.

Description The **DataSource** property specifies the data control used to bind a **TDBGrid** or **TDBDropDown** control to a database.

Note To bind a **TDBGrid** or **TDBDropDown** control to an APEX **XArray** object, use the **Array** property.

DataWidth Property

Syntax

column.**DataWidth** = long

Read/Write at run time and design time.

Property applies to **Column** object.

Description

This property holds the database width, in bytes, for a grid column. It is set to the appropriate field width (not display width) automatically when the layout of a bound grid is initialized at run time.

This property does not affect the physical size of a column, but imposes a limit on the number of characters that may be entered when editing a cell. If set to 0, no such limits are imposed. Setting this property does **not** cause truncation of existing data.

For bound grids, if the data source does not supply a width value, then no limits are imposed. For both bound and unbound grids, you can set the value of this property in code to restrict the amount of data the user can enter.

DefaultItem Property

Syntax valueitems.**DefaultItem** = integer

Read/Write at run time and design time.
Property applies to **ValueItems** collection.

Description This property returns or sets the zero-based index of the default item for a **ValueItems** collection associated with a column. The default value for this property is -1, which is used to indicate that there is no default item.

Use the **DefaultItem** property to provide an alternate display for data values not present in the **ValueItems** collection.

When the **DefaultItem** property is set to a valid collection index (an integer between 0 and **Count** - 1, inclusive), then the corresponding **ValueItem** is displayed when the grid encounters a value which is not present in the **ValueItems** collection.

When the **DefaultItem** property is set to -1, then the grid will not substitute a **ValueItem** when it encounters a value which is not present in the **ValueItems** collection.

A trappable error will occur if you attempt to set this property to an invalid value.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

Note At design time, the **DefaultItem** property is specified in the Values property page by clicking the record selector of the desired grid row. If no row is selected, then the **DefaultItem** property will be set to its default value of -1. To deselect a selected row, click its record selector again.

DefaultValue Property

Syntax	<p>column.DefaultValue = variant</p> <p>Read/Write at run time and design time.</p> <p>Property applies to Column object.</p>
Description	<p>This property returns or sets the default value of an unbound grid column in a bound grid. Unbound columns are typically used to display calculated fields or local data not maintained by the primary data source.</p> <p>This property applies only to unbound columns. The value specified will be preloaded into the last argument passed to the <u>UnboundColumnFetch</u> event.</p> <p>The DefaultValue property can also be used to identify specific columns in the <u>UnboundColumnFetch</u> event when columns are added, moved, or removed at run time.</p> <p>For bound columns or columns of an unbound grid, the grid does not use this property itself, but provides it as a placeholder for you to associate default values with the columns. In the <u>UnboundAddData</u> event, you can use this property to retrieve default values for columns that were not supplied by the end-user. Such columns will contain a Null variant in the corresponding RowBuffer.Value property array.</p> <p>This property can also be used as a tag for a column (whether it is bound or unbound). Arbitrary values can be stored and retrieved later</p>
Note	<p>Do not confuse unbound columns with unbound mode. The DefaultValue property has no effect on data displayed in an unbound grid.</p>

DefColWidth Property

Syntax object.**DefColWidth** = single

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property returns or sets the default width for all grid columns in terms of the coordinate system of the grid's container.

For bound grids, the **DefColWidth** property is respected under the following circumstances:

- When you execute the **Retrieve Fields** command at design time.
- When the grid's layout is initialized at run time.
- When a new column is created at run time.

For unbound grids, the **DefColWidth** property is only respected when a new column is created at run time.

If you set the **DefColWidth** property to 0, the grid automatically sizes all columns based on either the width of the column heading or the display width of the underlying field, whichever is larger. For example, to set the default column width to the width of the first column:

```
TDBGrid1.DefColWidth = TDBGrid1.Columns(0).Width
```

Note Setting the **DefColWidth** property at run time does not affect existing columns, only those that are subsequently created in code.

In bound mode, some data sources do not provide text field widths when requested by the grid. Therefore, if **DefColWidth** is 0, the actual column widths may not be what you expect since the grid must supply a default width.

DisplayValue Property

Syntax valueitem.**DisplayValue** = variant

Read/Write at run time and design time.
Property applies to **ValueItem** object.

Description This property returns or sets the translated data value for a member of a **ValueItems** collection.

The **DisplayValue** property may be set to a string that specifies the mapping between the underlying data value and its displayed representation. It may also be set to a picture, such as that returned by the **LoadPicture** function in Visual Basic.

If the **DisplayValue** property is not explicitly set, it returns the same result as the **Value** property.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

DividerStyle Property

Syntax

column.**DividerStyle** = value

Read/Write at run time and design time.
Property applies to **Column** object.

Values

Design Time

Run Time

0 - No dividers	dbgNoDividers
1 - Black line	dbgBlackLine
2 - Dark gray line (default)	dbgDarkGrayLine
3 - Raised	dbgRaised
4 - Inset	dbgInset
5 - ForeColor	dbgUseForeColor
6 - Light gray line	dbgLightGrayLine

Description

This property determines the style of the border drawn on the right edge of a grid column.

The **DividerStyle** property does not control whether a column can be resized by dragging its border. Use the **AllowSizing** property to control this behavior.

DropDown Property

Syntax column.**DropDown** = string

Read/Write at run time and design time.

Property applies to **Column** object.

Description This property associates the name of a **TDBDropDown** control with a column in a **TDBGrid** control. When the user clicks the column's in-cell button, the associated **TDBDropDown** control is displayed below the current cell.

Use the **DropDown** property and a **TDBDropDown** control to implement a multicolumn drop-down list box that works seamlessly with a **TDBGrid** control. The **ListField** property of the drop-down control determines which column is used for incremental search. The **DataField** property of the drop-down control determines which grid column is updated when the user selects an item.

Note When you set the **DropDown** property of a column to the name of a **TDBDropDown** control, then the **Button** property for that column will be set to True. Similarly, if you set the **DropDown** property of a column to an empty string, then the **Button** property for that column will be set to False.

EditActive Property

Syntax	<p>TDBGrid.EditActive = boolean</p> <p>Read/Write at run time. Not available at design time.</p> <p>Property applies to TDBGrid control.</p>
Description	<p>If this property returns True, then the current cell is currently being edited by the user. If False, then no editing is in progress.</p> <p>If the grid is not already in edit mode, setting EditActive to True will initiate editing on the current cell. The caret will be positioned at the end of the cell and the ColEdit event will be triggered.</p> <p>If the grid is already in edit mode, setting EditActive to False will exit edit mode. If the cell has been modified, this will trigger the following events: BeforeColUpdate, AfterColUpdate, and AfterColEdit.</p>
Note	<p>To cancel editing completely, set the CurrentCellModified property to False, then set EditActive to False.</p> <p>The EditActive property does not correspond to the pencil in the RecordSelector column. The pencil reflects the state of the grid's DataChanged property.</p>

EditBackColor Property

Syntax	<p>object.EditBackColor = color</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid control, Column and Split objects.</p>
Description	<p>This property controls the background color of an object's editing window when editing is in progress. Colors may be specified as RGB values or system default colors.</p> <p>By default, the editor background color of a grid, column, or split is determined by its EditorStyle property setting. Setting the EditBackColor property overrides the style setting without changing the definition of the style itself.</p> <p>If the EditBackColor property of a grid, column, or split is changed to the same value as the BackColor property of its corresponding editor style, then the object will inherit its editor background color from the style, and subsequent changes to the style's BackColor property will affect the object as well.</p>
Note	<p>The EditBackColor property only applies when the floating editor marquee (MarqueeStyle = 6) is not in effect.</p>

EditDropDown Property

Syntax	<p>TDBGrid.EditDropDown = boolean</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid control.</p>
Description	<p>This property controls whether editing will take place in a popup window or within cell boundaries.</p> <p>If True (the default), an edit window will pop up when the user attempts to edit a cell whose contents cannot be displayed completely within the confines of the current cell's boundaries. Unlike the built-in combo box, the drop-down edit window will only extend to the bottom of the grid.</p> <p>If False, editing will be confined to the current cell's boundaries.</p> <p>The drop-down edit window behaves just like a standard multiple-line TextBox control in Visual Basic. The <u>SelLength</u>, <u>SelStart</u>, and <u>SelText</u> properties are still available, and the arrow keys can be used to navigate within the edit window.</p>
Note	<p>If the user tries to edit the last row in the grid, the drop-down edit window will not be displayed and the user will have to edit within the current cell's boundaries.</p> <p>The <u>EditDropDown</u> property only applies when the floating editor marquee (<u>MarqueeStyle</u> = 6) is not in effect.</p>

EditForeColor Property

Syntax object.**EditForeColor** = color

Read/Write at run time and design time.

Property applies to **TDBGrid** control, **Column** and **Split** objects.

Description This property controls the foreground color of an object's editing window when editing is in progress. Colors may be specified as RGB values or system default colors.

By default, the editor foreground color of a grid, column, or split is determined by its **EditorStyle** property setting. Setting the **EditForeColor** property overrides the style setting without changing the definition of the style itself.

If the **EditForeColor** property of a grid, column, or split is changed to the same value as the **ForeColor** property of its corresponding editor style, then the object will inherit its editor foreground color from the style, and subsequent changes to the style's **ForeColor** property will affect the object as well.

Note The **EditForeColor** property only applies when the floating editor marquee (**MarqueeStyle** = 6) is not in effect.

EditMask Property

Syntax column.**EditMask** = string

Read/Write at run time and design time.

Property applies to **Column** object.

Description The EditMask property is used to specify an input mask template for end-user data entry. You can construct your own input mask string using template characters similar to those recognized by the Visual Basic **Format\$** function. The input mask string is composed of special characters that represent either an input character that the user must enter, or a literal character that will be skipped over on input. Valid template characters are as follows:

#	Digit placeholder
@	Character placeholder
>	All characters following will be in uppercase
<	All characters following will be in lowercase
~	Turns off the previous "<" or ">"
?	Digit or character
\	Next character is treated as a literal
&	Any character

Any other character will be treated as a literal.

After the user finishes editing a cell with this input mask, True DBGrid caches the modified cell text, but any literal characters in the input mask template will be stripped from the modified cell text beforehand.

The **EditMask** property also supports a built-in DateMask option for formatting date fields. When the DateMask option is selected, the following input mask template will be used for editing:

mm/dd/yyyy

mm Month placeholder (01-12)

dd Date placeholder (01-31)

yyyy Year placeholder

The user can enter either 1, 2, 3, or 4 digits in the year portion of the date, and the grid will save the year as entered by the user. If the user enters 1 or 2 digits for the year portion, the grid will make no interpretation for the year; that is, the grid will not assume whether it is the century 1900 or 2000, but will store the 1-digit or 2-digit year as entered. Before the date is updated to the database, you can interpret the year yourself in code, or let the underlying database system handle the interpretation and storage.

Note that if you select the DateMask option for the **EditMask** property, the date separators are part of the date format; they are not considered as literal characters and will always be cached by the grid. This is because most databases and formatters require the separator characters to be present in order to interpret the date correctly.

EditMaskUpdate Property

Syntax column.**EditMaskUpdate** = boolean

Read/Write at run time and design time.

Property applies to **Column** object.

Description Normally, after the user finishes editing a cell in a column which has its **EditMask** property set, True DBGrid caches the modified cell text, but any literal characters in the input mask template will be stripped from the modified cell text beforehand. However, you can override this behavior with the **EditMaskUpdate** property.

By default, the **EditMaskUpdate** property is False. This means that when the modified cell text is updated to the database, the grid sends the cached text (stripped of literals), not the formatted text displayed in the cell. You can override this default behavior by setting the **EditMaskUpdate** property to True, which causes the cached text to be formatted according to the **EditMask** property before being updated to the database.

EditorStyle Property

- Syntax** object.**EditorStyle** = variant
- Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control, **Column** and **Split** objects.
- Description** This property returns the **Style** object that controls the appearance of the cell editor within a grid, column, or split.
- Note** The **EditorStyle** property only applies when the floating editor marquee (**MarqueeStyle** = 6) is not in effect.

EmptyRows Property

- Syntax** object.**EmptyRows** = boolean
- Read/Write at run time and design time.
Property applies to **TDBGrid** and **TDBDropDown** controls.
- Description** The **EmptyRows** property returns or sets a value that determines how the grid displays rows below the last data row.
- If all of the records in the data source do not fill up the entire grid, setting **EmptyRows** to True will fill the unpopulated grid area with empty data rows. If **EmptyRows** is False (the default), then the unpopulated grid area will be blank and will be filled with the system 3D Objects color (or the system Button Face color) as determined by your Control Panel settings.
- Note** The **RowDividerStyle** property applies to data rows and empty rows alike. You cannot suppress row dividers for just the empty rows when **EmptyRows** is True.

Enabled Property

Syntax

object.**Enabled** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description

This property returns or sets a value that determines whether a control can respond to user-generated events.

If True (the default), the user can give focus to the control and manipulate it with the keyboard or mouse.

If False, the user cannot manipulate the control directly.

EOF Property

Syntax TDBGrid.**EOF**

Read-only at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description The **EOF** property operates like its **Recordset** counterpart. It returns True if the current record position is after the last record, False if the current record position is on or before the last record.
If the data source contains no records, then **EOF** will always return True.

ErrorText Property

Syntax object.**ErrorText**

Read-only at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property returns the error message string from the underlying data source.

When a database error occurs as a result of user interaction with the grid, such as when the user enters text into a numeric field and then attempts to update the current record by moving to another row, the grid's **Error** event will fire. However, the error code passed to the event handler in the **DataError** parameter may not identify the specific error that occurred, or may even differ across 16- and 32-bit operating environments. For these reasons, the **ErrorText** property is provided so that your application can parse the actual error message to determine the nature of the error.

Note The **ErrorText** property is only valid within a **TDBGrid** or **TDBDropDown** control's **Error** event handler. A trappable error will occur if you attempt to access it in any other context.

EvenRowStyle Property

Syntax object.**EvenRowStyle** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property sets or returns the **Style** object that controls the appearance of an even-numbered row in a grid or split where the **AlternatingRowStyle** property is set to True. By default, this is the built-in EvenRow style.

To change the appearance of odd-numbered rows, set the **OddRowStyle** property.

ExposeCellMode Property

Syntax TDBGrid.**ExposeCellMode** = value
Read/Write at run time and design time.
Property applies to **TDBGrid** control.

Values	Design Time	Run Time
0 - Scroll on Select (default)		dbgScrollOnSelect
1 - Scroll on Edit		dbgScrollOnEdit
2 - Never Scroll		dbgNeverScroll

Description This property controls how the rightmost column reacts when clicked by the user.

If set to 0 - Scroll on Select (the default), the grid will scroll to the left to display the rightmost column in its entirety. This can be somewhat disconcerting to users who commonly click on the grid to begin editing, as the grid will always shift to the left when the user clicks on the rightmost column.

If set to 1 - Scroll on Edit, the grid will not move when the rightmost column is clicked initially. However, if the user attempts to edit the cell, then the grid will scroll to the left to display the rightmost column in its entirety. This is exactly how Microsoft Excel works and is probably the most intuitive setting.

If set to 2 - Never Scroll, the grid will always leave the rightmost column clipped when clicked, even if the user subsequently attempts to edit the cell. Note that editing may be difficult if only a small portion of the column is visible. The chief reason to use this setting is if you know there will always be enough space available for editing (or if you disallow editing) and you never want the grid to shift accidentally.

Note The **ExposeCellMode** property only governs mouse clicks, not keyboard navigation.

ExtendRightColumn Property

Syntax	<p>object.ExtendRightColumn = boolean</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid and TDBDropDown controls, Split object.</p>
Description	<p>This property allows the rightmost column of a grid or split to extend to the object's right edge, provided that the object can accommodate all of the visible columns.</p> <p>If True, the last column will extend to the end of the grid or split.</p> <p>If False (the default), the area between the last column and the end of the grid or split will be filled using the system 3D Objects color (or the system Button Face color) as determined by your Control Panel settings.</p> <p>If a grid contains multiple splits, then setting its ExtendRightColumn property has the same effect as setting the ExtendRightColumn property of each split individually.</p>
Note	<p>This property now works even when the horizontal scroll bar is present. Prior to version 5.0, if a grid or split could not accommodate all of the visible columns, then setting this property to True had no effect.</p>

FetchRowStyle Property

Syntax object.**FetchRowStyle** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description If True, the **FetchRowStyle** event will be fired whenever the grid is about to display a row of data.

If False (the default), the **FetchRowStyle** event is not fired.

Set this value to True when you need to perform complex per-row formatting operations that can only be done using the **FetchRowStyle** event. For example, if you want to apply fonts and/or colors to all rows that satisfy certain criteria, then you need to set the **FetchRowStyle** property to True and write code for the **FetchRowStyle** event.

Note To display every other row in a different color or font, you can simply set the **AlternatingRowStyle** property to True.

FetchStyle Property

Syntax	<p>column.FetchStyle = boolean</p> <p>Read/Write at run time and design time.</p> <p>Property applies to Column object.</p>
Description	<p>If True, the FetchCellStyle event will be fired as needed to determine the font and color characteristics of each cell in the associated column.</p> <p>If False (the default), the FetchCellStyle event will not be fired.</p> <p>Set this value to True when you need to perform complex per-cell formatting operations that can only be done using the FetchCellStyle event. For example, if you want to apply fonts and/or colors to cells within a certain range, or cells that satisfy a complex expression, then you need to set FetchStyle to True for the desired column(s) and write code for the FetchCellStyle event.</p>
Note	<p>If you want to apply the same formatting to all cells within a row, then you should set the FetchRowStyle property instead of FetchStyle, and write code for the FetchRowStyle event instead of FetchCellStyle. This is much more efficient because events are fired on a per-row rather than on a per-cell basis.</p>

FirstRow Property

Syntax object.**FirstRow** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property returns or sets a value containing the bookmark for the first visible row in a grid or split.

For a **TDBGrid** or **TDBDropDown** control, setting the **FirstRow** property causes the grid to scroll so that the specified row becomes the topmost row. If a grid contains multiple splits, then the topmost row changes in each split, even if the splits have different **ScrollGroup** property settings.

For a **Split** object, setting the **FirstRow** property causes the specified row to become the topmost row for that split only.

Font Property

Syntax	<p>object.Font = font</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid and TDBDropDown controls; Column, Split, and Style objects.</p>
Description	<p>This property returns or sets the font object associated with a grid, column, split, or style.</p> <p>By default, the font of a grid, column, or split is determined by its Style property setting. Setting the Font property directly overrides the style setting without changing the definition of the style itself.</p> <p>If the Font property of an object is changed to the same value as the Font property of its corresponding style, then the object will inherit its font from the style, and subsequent changes to the style's Font property will affect the object as well.</p> <p>For Style objects, the value of the Font property is inherited from the parent style (if any) unless explicitly overridden, in which case the aforementioned inheritance rules also apply.</p>
Note	<p>For a TDBGrid control, TDBDropDown control, or Split object, if a change to the Font property results in a change to the average character width, then all affected columns are resized proportionally to reflect the new character width.</p> <p>However, for a Column object, changing the Font property does not resize the column, even if the average character width has changed.</p>

ForeColor Property

Syntax

object.**ForeColor** = color

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls; **Column**, **Split**, and **Style** objects.

Description

This property controls the foreground color of an object. Colors may be specified as RGB values or system default colors.

By default, the foreground color of a grid, column, or split is determined by its **Style** property setting. Setting the **ForeColor** property overrides the style setting without changing the definition of the style itself.

If the **ForeColor** property of an object is changed to the same value as the **ForeColor** property of its corresponding style, then the object will inherit its foreground color from the style, and subsequent changes to the style's **ForeColor** property will affect the object as well.

For **Style** objects, the value of the **ForeColor** property is inherited from the parent style (if any) unless explicitly overridden, in which case the aforementioned inheritance rules also apply.

HeadAlignment Property

Syntax column.**HeadAlignment** = value
Read/Write at run time and design time.
Property applies to **Column** object.

Values	Design Time	Run Time
	0 - Left (default)	dbgLeft
	1 - Right	dbgRight
	2 - Center	dbgCenter
	3 - General	dbgGeneral

Description The **HeadAlignment** property returns or sets a value that determines the alignment of the headings for an individual column.
The General setting means that the column's **Alignment** property will be used to format both the column headings and the cell text.

HeadBackColor Property

Syntax

object.**HeadBackColor** = color

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Description

This property controls the background color of an object's column headings. Colors may be specified as RGB values or system default colors.

By default, the heading background color of a grid, column, or split is determined by its **HeadingStyle** property setting. Setting the **HeadBackColor** property overrides the style setting without changing the definition of the style itself.

If the **HeadBackColor** property of a grid, column, or split is changed to the same value as the **BackColor** property of its corresponding heading style, then the object will inherit its heading background color from the style, and subsequent changes to the style's **BackColor** property will affect the object as well.

HeadFont Property

Syntax

object.**HeadFont** = font

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Description

This property returns or sets the font object associated with the column headings of a grid, column, or split.

By default, the heading font of a grid, column, or split is determined by its **HeadingStyle** property setting. Setting the **HeadFont** property directly overrides the heading style setting without changing the definition of the style itself.

If the **HeadFont** property of an object is changed to the same value as the **Font** property of its corresponding heading style, then the object will inherit its heading font from the style, and subsequent changes to the style's **Font** property will affect the object as well.

HeadForeColor Property

Syntax	<p>object.HeadForeColor = color</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid and TDBDropDown controls, Column and Split objects.</p>
Description	<p>This property controls the foreground color of an object's column headings. Colors may be specified as RGB values or system default colors.</p> <p>By default, the heading foreground color of a grid, column, or split is determined by its HeadingStyle property setting. Setting the HeadForeColor property overrides the style setting without changing the definition of the style itself.</p> <p>If the HeadForeColor property of a grid, column, or split is changed to the same value as the property of its corresponding heading style, then the object will inherit its heading foreground color from the style, and subsequent changes to the style's ForeColor property will affect the object as well.</p>

HeadingStyle Property

Syntax object.**HeadingStyle** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Description This property returns the **Style** object that controls the appearance of column headings within a grid, column, or split.

HeadLines Property

- Syntax** object.**HeadLines** = single
- Read/Write at run time and design time.
Property applies to **TDBGrid** and **TDBDropDown** controls.
- Description** This property returns or sets a value indicating the number of lines of text displayed in a **TDBGrid** or **TDBDropDown** control's column headers.
- The **HeadLines** property accepts a floating point number from 0 to 10. The default value is 1, which causes the grid to display the caption for each **Column** object within its header area.
- A setting of 0 removes the headings and has the same effect as setting the **ColumnHeaders** property to False.
- Note** By default, a **Column** object's caption contains the name of its underlying field as specified by the **DataField** property. You can use the **Caption** property to override the text displayed within column headers.

HighlightRowStyle Property

Syntax	<p>object.HighlightRowStyle = variant</p> <p>Read/Write at run time. Not available at design time.</p> <p>Property applies to TDBGrid and TDBDropDown controls, Split object.</p>
Description	<p>This property sets or returns the Style object that controls the appearance of a highlighted row or cell marquee. By default, this is the built-in HighlightRow style.</p> <p>The HighlightRowStyle value is only used when one of the following MarqueeStyle settings is in effect: 2 - Highlight Cell, 3 - Highlight Row, or 4 - HighlightRow, RaiseCell.</p> <p>The value of the MarqueeStyle property is not affected by changes to the HighlightRowStyle property.</p>
Note	<p>Prior to version 5.0, MarqueeStyle settings 2, 3, and 4 were rendered by inverting the normal cell colors, and could only be customized by repeated application of the AddCellStyle method. The HighlightRowStyle property was introduced to enable design-time customization of marquee colors.</p>

HScrollHeight Property

Syntax object.**HScrollHeight**

Read-only at run time. Not available at design time.

Property applies to **TDBGrid** control and **Split** object.

Description The **HScrollHeight** property returns the height of a split's horizontal scroll bar in container units, which depend on the **ScaleMode** of the container. If no horizontal scroll bar exists, then the returned value is zero. If object is a **TDBGrid** control, then its current split is used.

You can use the **HScrollHeight** and **VScrollWidth** properties to check if the scroll bars are present and to obtain the scroll bar size when designing the grid layout and sizing the grid and its columns.

hWnd Property

Syntax object.**hWnd**

Read-only at run time. Not available at design time.
Property applies to **TDBGrid** and **TDBDropDown** controls.

Description The **hWnd** property returns the unique window handle assigned to a **TDBGrid** or **TDBDropDown** control by the Microsoft Windows operating environment. Experienced users can pass the value of this property to Windows API calls that require a valid window handle.

Note Since the value of this property can change while a program is running, never store the **hWnd** value in a variable.

hWndEditor Property

Syntax TDBGrid.**hWndEditor**

Read-only at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description The **hWndEditor** property returns the unique window handle assigned to a **TDBGrid** control's editing window by the Microsoft Windows operating environment. Experienced users can pass the value of this property to Windows API calls that require a valid window handle.

When editing is not in progress, this property returns 0.

Note Since the value of this property can change while a program is running, never store the **hWndEditor** value in a variable.

Do not use the **hWndEditor** property to test whether editing is in progress. The **EditActive** property is provided for this purpose.

InactiveBackColor Property

- Syntax** object.**InactiveBackColor** = color
- Read/Write at run time and design time.
Property applies to **TDBGrid** control and **Split** object.
- Description** This property controls the background color of an object's column headings when another object has focus. Colors may be specified as RGB values or system default colors.
- By default, the inactive background color of a grid or split is determined by its **InactiveStyle** property setting. Setting the **InactiveBackColor** property overrides the style setting without changing the definition of the style itself.
- If the **InactiveBackColor** property of a grid or split is changed to the same value as the **BackColor** property of its corresponding inactive style, then the object will inherit its inactive background color from the style, and subsequent changes to the style's **BackColor** property will affect the object as well.
- Note** The inactive colors are only used when the grid's **Appearance** property is set to 0 - Flat. If the **Appearance** property is set to the default value of 1 - 3D, then the headings do not change when a grid or split receives or loses focus.

InactiveForeColor Property

Syntax	<p>object.InactiveForeColor = color</p> <p>Read/Write at run time and design time.</p> <p>Property applies to TDBGrid control and Split object.</p>
Description	<p>This property controls the foreground color of an object's column headings when another object has focus. Colors may be specified as RGB values or system default colors.</p> <p>By default, the inactive foreground color of a grid or split is determined by its InactiveStyle property setting. Setting the InactiveForeColor property overrides the style setting without changing the definition of the style itself.</p> <p>If the InactiveForeColor property of a grid or split is changed to the same value as the ForeColor property of its corresponding inactive style, then the object will inherit its inactive foreground color from the style, and subsequent changes to the style's ForeColor property will affect the object as well.</p>
Note	<p>The inactive colors are only used when the grid's Appearance property is set to 0 - Flat. If the Appearance property is set to the default value of 1- 3D, then the headings do not change when a grid or split receives or loses focus.</p>

InactiveStyle Property

- Syntax** object.**InactiveStyle** = variant
- Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control and **Split** object.
- Description** This property returns the **Style** object that controls the appearance of column headings within a grid or split when another object has focus.
- Note** The inactive style is only used when the grid's **Appearance** property is set to 0 - Flat. If the **Appearance** property is set to the default value of 1 - 3D, then the headings do not change when a grid or split receives or loses focus.

Index Property

Syntax

split.**Index**

Read-only at run time. Not available at design time.
Property applies to **Split** object.

Description

This property returns the zero-based index of a split within the **Splits** collection.

IntegralHeight Property

Syntax TDBDropDown.**IntegralHeight** = boolean

Read/Write at run time and design time.

Property applies to **TDBDropDown** control.

Description This property determines whether partial rows are displayed in a **TDBDropDown** control.

If True, partial rows are not displayed, and the height of the control will be reduced to eliminate the last partial row if necessary.

If False, partial rows are displayed, and the control retains its design-time height.

LayoutFileName Property

Syntax TDBGrid.**LayoutFileName** = string

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description This property sets or returns the string containing the filename used to save and retrieve grid layouts. Setting this property alone has no effect on the grid layout; the property value is used by the **LoadLayout** method of the grid and the **Add** and **Remove** methods of the **Layouts** collection.

At design time, if you first set the **LayoutFileName** property to a valid filename in which grid layouts are stored, you can then choose the **LayoutName** property from a drop-down list of layout names stored in the specified layout file.

LayoutName Property

Syntax TDBGrid.**LayoutName** = string

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description This property sets or returns the string (maximum length of 30 characters) containing the current layout name. Setting this property alone has no effect on the grid layout; the property value is used by the **LoadLayout** method of the grid.

At design time, if you first set the **LayoutFileName** property to a valid filename in which grid layouts are stored, you can then choose the **LayoutName** property from a drop-down list of layout names stored in the specified layout file.

Layouts Property

Syntax TDBGrid.**Layouts**

Read-only at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description This property returns a collection of layout names corresponding to the current setting of the **LayoutFileName** property.

Left Property

Syntax

column.**Left**

Read-only at run time. Not available at design time.
Property applies to **Column** object.

Description

This property returns the position of a column's left edge in terms of the coordinate system of the grid's container.

Use the **Left** property in conjunction with **Width**, **RowHeight**, and **RowTop** to determine the size and placement of controls displayed on top of a grid cell.

LeftCol Property

Syntax

object.**LeftCol** = integer

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description

This property returns or sets the zero-based index of the leftmost column in a grid or split.

For a **TDBGrid** or **TDBDropDown** control, setting the **LeftCol** property causes the grid to scroll so that the specified column becomes the leftmost column. If a grid contains multiple splits, then the leftmost column changes in each split.

For a **Split** object, setting the **LeftCol** property causes the specified column to become the leftmost column for that split only.

Use the **LeftCol** property in code to scroll a grid or split horizontally. Use the **FirstRow** property to determine the bookmark of the first visible row in a grid or split.

ListField Property

- Syntax** TDBDropDown.**ListField** = string
- Read/Write at run time and design time.
Property applies to **TDBDropDown** control.
- Description** The **ListField** property returns or sets the name of the column used for incremental search within a **TDBDropDown** control. The **ListField** property need not be the same as the **DataField** property used to specify the name of the grid column that will be updated when the user selects an item from a **TDBDropDown** control.
- If the **ListField** property is not specified, the **DataField** property specifies the column to be used for both incremental search and the selection value. If neither property is specified, then the first column in the **TDBDropDown** control will be used.
- Note** To associate a **TDBDropDown** control with a **Column** object that belongs to a **TDBGrid** control, set the column's **DropDown** property to the name of the drop-down control at either design time or run time.

Locked Property

Syntax	<p>object.Locked = boolean</p> <p>Read/Write at run time and design time.</p> <p>Property applies to Column, Split, and Style objects.</p>
Description	<p>This property returns or sets a value indicating whether an object can be edited.</p> <p>If True, the user cannot modify data in the column or split.</p> <p>If False (the default), the user can modify data in the column or split.</p> <p>If the TDBGrid control's AllowUpdate property is set to False, then editing is disabled for the entire grid regardless of the Locked property setting for individual columns and splits. If AllowUpdate is True, then the Locked property can be used to control the editability of individual columns and splits.</p> <p>For Split objects, setting the Locked property to True disables editing for all columns within that split regardless of their Locked property setting. If Locked is False for a split, then the Locked settings of individual columns within that split are respected.</p> <p>For Style objects, the Locked property controls the editability of the object to which the style is applied. It does not control the editability of the style object itself.</p> <p>By default, the locked state of a column or split is determined by its Style property setting. Setting the Locked property overrides the style setting without changing the definition of the style itself.</p> <p>If the Locked property of a column or split is changed to the same value as its corresponding style, then the object will inherit its locked state from the style, and subsequent changes to the style's Locked property will affect the object as well.</p> <p>For Style objects, the value of the Locked property is inherited from the parent style (if any) unless explicitly overridden, in which case the aforementioned inheritance rules also apply.</p>
Note	<p>The default value of the Locked property for a column is not derived from the DataUpdatable property for the underlying field. If both properties are False for a column, then an error will occur when the grid attempts to write changed data to the database.</p>

MarqueeStyle Property

Syntax object.**MarqueeStyle** = value

Read/Write at run time and design time.

Property applies to **TDBGrid** control and **Split** object.

Values

Design Time

Run Time

0 - Dotted Cell Border	dbgDottedCellBorder
1 - Solid Cell Border	dbgSolidCellBorder
2 - Highlight Cell	dbgHighlightCell
3 - Highlight Row	dbgHighlightRow
4 - Highlight Row, Raise Cell	dbgHighlightRowRaiseCell
5 - No Marquee	dbgNoMarquee
6 - Floating Editor (default)	dbgFloatingEditor

Description

This property determines how the current row and cell are highlighted within a grid or split. There are seven possible values for this property:

0 - Dotted Cell Border	The current cell within the current row will be highlighted by drawing a dotted border around the cell. In Microsoft Windows terminology, this is usually called a focus rectangle.
1 - Solid Cell Border	The current cell within the current row will be highlighted by drawing a solid box around the current cell. This is more visible than the dotted cell border, especially when 3-D divider properties are used for the grid.
2 - Highlight Cell	The entire current cell will be drawn using the attributes of the HighlightRowStyle property. This provides a very distinctive block-style highlight for the current cell.
3 - Highlight Row	The entire row containing the current cell will be drawn using the attributes of the HighlightRowStyle property. In this mode, it is not possible to visually determine which cell is the current cell, only the current row. When the grid or split is not editable, this setting is often preferred, since cell position is then irrelevant.
4 - Highlight Row, Raise Cell	The entire row will be highlighted as in setting 3, but the current cell within the row will be "raised" so that it appears distinctive. This setting doesn't appear clearly with all background color and divider settings. The best effect is achieved by using 3-D dividers and a light gray background.
5 - No Marquee	The marquee will not be shown. This setting is useful for cases where the current row is irrelevant, or where you don't want to draw the user's attention to the grid until necessary.
6 - Floating Editor	The current cell will be highlighted by a floating text editor window with a blinking caret (as in Microsoft Access). This is the default setting.

If a grid contains multiple splits, then setting its **MarqueeStyle** property has the same effect as setting the **MarqueeStyle** property of each split individually.

Note

If the floating editor marquee setting is in effect and the current cell contains radio buttons or graphics, then a dotted focus rectangle will be displayed.

Prior to version 5.0, **MarqueeStyle** settings 2, 3, and 4 were rendered by inverting the normal cell color(s), and could only be customized by repeated application of the **AddCellStyle**

method. The **HighlightRowStyle** property was introduced to enable design-time customization of marquee colors.

MarqueeUnique Property

Syntax TDBGrid.**MarqueeUnique** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description This property controls the display of the current cell marquee when there is more than one split. The current cell marquee is only displayed when the **MarqueeStyle** property for a grid or split has a value of 0, 1, 2, or 4.

If True (the default), then the current cell marquee is only displayed within the current split.

If False, then all splits with a **MarqueeStyle** setting of 0, 1, 2, or 4 will display a marquee at the current cell, provided that the current cell is visible.

In most cases, a single current cell marquee is preferable, and you will not need to change this property.

If this property is set to False, you may then see several different current cell marquees. The actual current cell is determined by the setting of the **Split** property.

Note Although the floating editor **MarqueeStyle** (6) is technically a current cell marquee, only one floating editor will be displayed, even if **MarqueeUnique** is set to False.

MaxComboItems Property

Syntax valueitems.**MaxComboItems** = integer

Read/Write at run time and design time.
Property applies to **ValueItems** collection.

Description This property controls the maximum number of items to be displayed in the built-in combo box. The default value for this property is 5.

When the **Presentation** property of a column's **ValueItems** collection is set to either of the combo box options (sorted or unsorted), the **MaxComboItems** property determines the combo box height in terms of the number of value items displayed.

If the total number of value items is less than or equal to **MaxComboItems**, then all value items will be shown. If the total number of value items exceeds **MaxComboItems**, only **MaxComboItems** will be shown, but a scroll bar will appear at the right edge of the drop-down combo to allow users to bring the other value items into view.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

MultipleLines Property

Syntax TDBGrid.**MultipleLines** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description This property determines whether a single row can span multiple lines. In this context, the terms *line* and *row* are defined as follows:

- A *line* is a single *physical row* of cells displayed in a grid. Do not confuse this with a line of text inside a grid cell; depending upon the settings of the **RowHeight** and **WrapText** properties, data in a grid cell may be displayed in multiple lines of text.
- A *row* displays a single record and may contain multiple lines or multiple *physical rows*.

The default value of **MultipleLines** is False, which means that a single record or row cannot span multiple lines. If necessary, the user can operate the horizontal scroll bar to view all of the columns within a row. This is how the grid normally displays data.

However, if the **MultipleLines** property is True, then a single record or row may span multiple lines. This feature enables the user to view simultaneously all of the columns (fields) of a record within the width of the grid without scrolling horizontally.

When **MultipleLines** is set to True, the horizontal scroll bar will be hidden if present, regardless of the setting of the **ScrollBars** property. The grid will automatically span or wrap the columns to multiple lines so that all columns will be visible within the width of the grid. If the resulting column layout is not to your liking, you can adjust it at either design time or run time by changing the widths and orders of the columns.

Note If the **ScrollBars** property is set to 4 - Automatic, the design time layout may not match the run time layout, owing to the absence of the scroll bar at design time. To ensure that the layout does not change at run time, use a different setting for the **ScrollBars** property.

Name Property

Syntax style.**Name**

Read-only at run time. Read/Write at design time.
Property applies to **Style** object.

Description This property returns the name of a style object.

The **Name** property is set at design time in the Styles property page when a style is first created. Styles cannot be renamed, even at design time.

When a **TDBGrid** control is first created, its **Styles** collection contains seven built-in styles named Normal, Heading, Selected, Caption, HighlightRow, EvenRow, and OddRow.

Note For an independent style object, the **Name** property always returns an empty string. An independent style object is not a member of the Styles collection, but is a standalone object created in code with a `Dim` or `Set` statement using the `New` keyword.

NumberFormat Property

Syntax

column.**NumberFormat** = string

Read/Write at run time and design time.

Property applies to **Column** object.

Description

This property returns or sets a value indicating the format string for a grid column. By default, the **NumberFormat** property contains an empty string, and column data is unformatted.

For numeric data, the following predefined format names can be used:

General Number	Display number as is, with no thousand separators.
Currency	Display number with thousand separator, if appropriate; display two digits to the right of the decimal separator. Note that output is based on system locale settings.
Fixed	Display at least one digit to the left and two digits to the right of the decimal separator.
Standard	Display number with thousands separator, at least one digit to the left and two digits to the right of the decimal separator.
Percent	Display number multiplied by 100 with a percent sign (%) appended to the right; always display two digits to the right of the decimal separator.
Scientific	Use standard scientific notation.
Yes/No	Display No if number is 0; otherwise, display Yes.
True/False	Display False if number is 0; otherwise, display True.
On/Off	Display Off if number is 0; otherwise, display On.

For date and time data, the following predefined format names can be used:

General Date	Display a date and/or time. For real numbers, display a date and time (for example, 4/3/93 05:34 PM); if there is no fractional part, display only a date (for example, 4/3/93); if there is no integer part, display only a time (for example, 05:34 PM). Date display is determined by your system settings.
Long Date	Display a date according to your system's long date format.
Medium Date	Display a date using the medium date format appropriate for the language version of Visual Basic.
Short Date	Display a date using your system's short date format.
Long Time	Display a time using your system's long time format: includes hours, minutes, seconds.
Medium Time	Display a time in 12-hour format using hours and minutes and the AM/PM designator.
Short Time	Display a time using the 24-hour format (for example, 17:45).

For arbitrary data, the following predefined format names can be used:

Edit Mask	Use the column's EditMask property to format the data for display as well as editing.
FormatText Event	Fire the FormatText event for the associated column. This option allows you to write your own formatting code for situations where

Visual Basic's intrinsic formatting is unavailable or does not suit your needs.

The **NumberFormat** property also accepts user-defined format strings. See the Microsoft Visual Basic documentation (Format function) for details.

If the **NumberFormat** property is set to an invalid string, cell data are displayed as #ERR#.

Note

The **NumberFormat** property works only in container environments that support Visual Basic formatting through OLE. If a container does not provide this support, the **NumberFormat** property can still be set without causing an error, but cell data will not be formatted.

However, the FormatText Event option can be used in any container environment, even if Visual Basic formatting is unavailable.

OddRowStyle Property

Syntax object.**OddRowStyle** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property sets or returns the **Style** object that controls the appearance of an odd-numbered row in a grid or split where the **AlternatingRowStyle** property is set to True. By default, this is the built-in OddRow style.

To change the appearance of even-numbered rows, set the **EvenRowStyle** property.

Order Property

Syntax column.**Order** = integer

Read/Write at run time. Not available at design time.
Property applies to **Column** object.

Description This property sets or returns the zero-based display position of a column within the **Columns** collection.

Use the **Order** property to determine the location of a column relative to other columns within the same split, subject to end-user move operations. If **AllowColMove** is never set to True, then this property returns the same value as **ColIndex**.

You can also set the **Order** property in code to move a single unselected column or all selected columns. For example, consider a grid with four columns. To move the last column (index 3) all the way to the left, you would code:

```
TDBGrid1.Columns(3).Order = 0
```

To reverse this action, you would set the order to the number of columns:

```
TDBGrid1.Columns(3).Order = TDBGrid1.Columns.Count
```

Note that you still use index 3 to refer to the original last column even after it has been moved. This allows code that references columns by numeric index instead of by name to remain consistent, which is especially critical for unbound mode applications.

Note If one or more columns are selected, then setting the **Order** property of an unselected column has no effect. However, setting the **Order** property of a selected column moves all columns in the selected range.

Parent Property

Syntax style.**Parent**

Read/Write at run time and design time.
Property applies to **Style** object.

Description This property sets or returns the parent style of a named style object. If a style has no parent, then this property returns a null variant.

The **Parent** property is used at run time to change the parent style from which the style in question inherits. Typically, this is done when creating a new style in code, as in the following example:

```
Dim BoldHeading As TrueDBGrid50.Style
Set BoldHeading = TDBGrid1.Styles.Add("BoldHeading")
BoldHeading.Parent = "Heading"
BoldHeading.Font.Bold = True
```

This code first creates a new style, BoldHeading, then sets its parent to the built-in Heading style. This causes the new style to inherit all attributes from the built-in style. The bold attribute of the new style's font is then overridden.

Note For an independent style object, a trappable error will occur if you attempt to set the **Parent** property. An independent style object is not a member of the **Styles** collection, but is a standalone object created in code with a `Dim` or `Set` statement using the `New` keyword.

Presentation Property

Syntax valueitems.**Presentation** = value
Read/Write at run time and design time.
Property applies to **ValueItems** collection.

Values	Design Time	Run Time
0 - Normal (default)		dbgNormal
1 - Radio Button		dbgRadioButton
2 - Combo Box		dbgComboBox
3 - Sorted Combo Box		dbgSortedComboBox

Description This property determines how the members of a **ValueItems** collection are displayed within the associated column.

If set to 0 - Normal (the default), value items are displayed as text or graphics depending upon the setting of the **DisplayValue** and **Translate** properties.

If set to 1 - Radio Button, value items are displayed as a group of radio buttons within the cell.

If set to 2 - Combo Box, value items are displayed in a drop-down combo box within the current cell.

If set to 3 - Sorted Combo Box, value items are displayed in sorted order in a drop-down combo box within the current cell.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

RecordSelectors Property

Syntax object.**RecordSelectors** = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control and **Split** object.

Description This property returns or sets a value indicating if record selectors are displayed in a grid or split.

If True (the default), record selectors are displayed at the left edge of the grid or split.

If False, record selectors are not displayed.

If a grid contains multiple splits, then setting its **RecordSelectors** property has the same effect as setting the **RecordSelectors** property of each split individually.

Note When the user selects a row by clicking its record selector, the bookmark of the selected row is added to the **SelBookmarks** collection.

Row Property

Syntax object.**Row** = integer

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property specifies the zero-based index of the current row relative to the first displayed row. It may be set at run time to highlight a different cell within the current column.

The **Row** property accepts values ranging from 0 to **VisibleRows** - 1. An error occurs if you attempt to set it to an invalid row index.

If the current row is not visible, then this property returns -1.

For a **TDBGrid** control, changing the **Row** property at run time does not affect selected rows.

Use the collection returned by the **SelBookmarks** property to select or deselect individual rows.

For a **TDBDropDown** control, changing the **Row** property at run time also changes the value of the **SelectedItem** property.

RowCount Property

Syntax rowbuffer.RowCount = long

Read/Write at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns or sets the number of rows in a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control.

In the **UnboundReadData** event, this property indicates how many rows the grid is requesting. After filling those rows by setting the **Value** and **Bookmark** properties, your event procedure should set the **RowCount** property to the number of rows actually fetched.

In the **UnboundAddData** and **UnboundWriteData** events, this property is always set to 1, since only a single row can be added or updated at a time. However, you can set this property to 0 to indicate that the add or update operation failed.

Note When a **RowBuffer** object is passed to an unbound event procedure, the initial value of the **RowCount** property also specifies the maximum value. An error will occur if you attempt to exceed the maximum value.

RowDividerStyle Property

Syntax object.**RowDividerStyle** = value

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Values	Design Time	Run Time
	0 - No dividers	dbgNoDividers
	1 - Black line	dbgBlackLine
	2 - Dark gray line (default)	dbgDarkGrayLine
	3 - Raised	dbgRaised
	4 - Inset	dbgInset
	5 - ForeColor	dbgUseForeColor
	6 - Light gray line	dbgLightGrayLine

Description This property determines the style of the border drawn between grid rows.

The **RowDividerStyle** property does not control whether rows can be resized by dragging their border. Use the **AllowRowSizing** property to control this behavior.

RowHeight Property

Syntax

object.**RowHeight** = single

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description

This property returns or sets the height of all grid rows in terms of the coordinate system of the grid's container.

The **RowHeight** property accepts a floating point number from 0 to 10,000. The default value depends upon the character height of the current font.

A setting of 0 causes the grid to readjust its display so that each row occupies a single line of text in the current font. Therefore, the following statements will set the row height so that exactly two lines of text are shown in each row:

```
TDBGrid1.RowHeight = 0
```

```
TDBGrid1.RowHeight = TDBGrid1.RowHeight * 2
```

If the control's **AllowRowSizing** property is set to True, then the user can adjust the **RowHeight** property at run time by dragging the row divider between any pair of record selectors.

ScrollBars Property

Syntax object.**ScrollBars** = value

Read/Write at run time and design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Values	Design Time	Run Time
0 - None		dbgNone
1 - Horizontal		dbgHorizontal
2 - Vertical		dbgVertical
3 - Both		dbgBoth
4 - Automatic (default)		dbgAutomatic

Description This property returns or sets a value indicating whether a grid or split has horizontal or vertical scroll bars.

The default setting for this property causes horizontal and/or vertical scroll bars to be displayed only if the object's contents extend beyond its borders.

If a grid contains multiple splits, then setting its **ScrollBars** property has the same effect as setting the **ScrollBars** property of each split individually.

ScrollGroup Property

Syntax

split.**ScrollGroup** = integer

Read/Write at run time and design time.

Property applies to **Split** object.

Description

This property is used to synchronize vertical scrolling between splits. All splits with the same **ScrollGroup** setting will be synchronized when vertical scrolling occurs within any one of them. Splits belonging to different groups can scroll independently, allowing different splits to display different parts of the database.

If the **ScrollBars** property for a split is set to 4 - Automatic, then only the rightmost split of the group will have a vertical scroll bar. If there is only one split, then setting this property has no effect.

Setting the **FirstRow** property for one split affects all other splits in the same group, keeping the group synchronized.

Newly created splits have a **ScrollGroup** value of 1.

SelBookmarks Property

Syntax	TDBGrid. SelBookmarks Read-only at run time. Not available at design time. Property applies to <u>TDBGrid</u> control.
Description	This property returns a collection of selected row bookmarks.

SelectedBackColor Property

Syntax object.**SelectedBackColor** = color

Read/Write at run time and design time.

Property applies to **TDBGrid** control and **Split** object.

Description This property controls the background color of a selected row and column within a grid or split. Colors may be specified as RGB values or system default colors.

By default, the selected background color of a grid or split is determined by its **SelectedStyle** property setting. Setting the **SelectedBackColor** property overrides the style setting without changing the definition of the style itself.

If the **SelectedBackColor** property of a grid or split is changed to the same value as the **BackColor** property of its corresponding selected style, then the object will inherit its selected background color from the style, and subsequent changes to the style's **BackColor** property will affect the object as well.

SelectedForeColor Property

Syntax object.**SelectedForeColor** = color

Read/Write at run time and design time.

Property applies to **TDBGrid** control and **Split** object.

Description This property controls the foreground color of a selected row and column within a grid or split. Colors may be specified as RGB values or system default colors.

By default, the selected foreground color of a grid or split is determined by its **SelectedStyle** property setting. Setting the **SelectedForeColor** property overrides the style setting without changing the definition of the style itself.

If the **SelectedForeColor** property of a grid or split is changed to the same value as the **ForeColor** property of its corresponding selected style, then the object will inherit its selected foreground color from the style, and subsequent changes to the style's **ForeColor** property will affect the object as well.

SelectedItem Property

Syntax	TDBDropDown. SelectedItem = variant Read/Write at run time. Not available at design time. Property applies to TDBDropDown control.
Description	<p>This property returns or sets the bookmark identifying the selected item in a TDBDropDown control.</p> <p>Use the value returned by the SelectedItem property to determine the current row in a TDBDropDown control.</p> <p>When you set the SelectedItem property to a valid value in code, the row associated with that value becomes the current row, and the drop-down grid adjusts its display to bring the new current row into view if necessary.</p> <p>The SelectedItem property is defined as a Variant to accommodate user-defined bookmarks in unbound mode.</p>
Note	For the TDBDropDown control, the SelectedItem and Bookmark properties are synonymous.

SelectedStyle Property

Syntax object.**SelectedStyle** = variant

Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control and **Split** object.

Description This property returns or sets the **Style** object that controls the appearance of selected rows and columns within a grid or split.

SelEndCol Property

Syntax object.**SelEndCol** = integer

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property returns or sets the zero-based ordinal position of the rightmost selected column in a split. If no columns are selected, then this property returns -1.

If a grid contains multiple splits, then setting its **SelEndCol** property has the same effect as setting the **SelEndCol** property of the current split. The index of the current split is available through the **TDBGrid** control's **Split** property.

Setting this property to -1 deselects all columns and also sets the **SelStartCol** property to -1.

Note You can also use the **ClearSelCols** method to deselect all columns within a split.

SelLength Property

Syntax TDBGrid.**SelLength** = long

Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description This property returns or sets the number of characters selected within the grid's editing window.

When editing is not in progress, this property returns 0.

Setting **SelLength** to a value less than 0 causes a run time error.

Use the **SelLength** property in combination with the **SelStart** and **SelText** properties to set the insertion point, establish an insertion range, select substrings, or clear text. These properties are useful for implementing copy, cut, and paste operations that transfer string data to and from the clipboard.

SelStart Property

Syntax

TDBGrid.**SelStart** = long

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** control.

Description

This property returns or sets the starting point of the text selection within the grid's editing window. If no text is currently selected, then this property indicates the position of the insertion point.

When editing is not in progress, this property returns 0.

Setting **SelStart** to a value greater than **SelLength** sets it to **SelLength**. Changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.

Use the **SelStart** property in combination with the **SelLength** and **SelText** properties to set the insertion point, establish an insertion range, select substrings, or clear text. These properties are useful for implementing copy, cut, and paste operations that transfer string data to and from the clipboard.

SelStartCol Property

Syntax object.**SelStartCol** = integer

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Description This property returns or sets the zero-based ordinal position of the leftmost selected column in a split. If no columns are selected, then this property returns -1.

If a grid contains multiple splits, then setting its **SelStartCol** property has the same effect as setting the **SelStartCol** property of the current split. The index of the current split is available through the **TDBGrid** control's **Split** property.

Setting this property to -1 deselects all columns and also sets the **SelEndCol** property to -1.

Note You can also use the **ClearSelCols** method to deselect all columns within a split.

SelText Property

Syntax TDBGrid.**SelText** = string

Read/Write at run time. Not available at design time.
Property applies to **TDBGrid** control.

Description This property returns or sets the string containing the currently selected text within the grid's editing window. If no text is currently selected, then this property returns an empty string.

Setting **SelText** to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Use the **SelText** property in combination with the **SelStart** and **SelLength** properties to set the insertion point, establish an insertion range, select substrings, or clear text. These properties are useful for implementing copy, cut, and paste operations that transfer string data to and from the clipboard.

Size Property

Syntax	<p>split.Size = variant</p> <p>Read/Write at run time and design time.</p> <p>Property applies to Split object.</p>
Description	<p>This property returns or sets the size of a split. The meaning of the value returned by this property is determined by the split's SizeMode property setting.</p> <p>If SizeMode is set to the default value of 0 - Scalable, then the value returned by the Size property is an integer indicating the relative size of the split with respect to other scalable splits.</p> <p>If SizeMode is set to 1 - Exact, then the value returned by the Size property is a floating point number indicating the exact size of the split in terms of the coordinate system of the grid's container.</p> <p>If SizeMode is set to 2 - Number of Columns, then the value returned by the Size property is an integer indicating the number of columns displayed in the split.</p>
Note	<p>Note that when there is only one split (the grid's default behavior), the split spans the entire width of the grid, the SizeMode property is always 0 - dbgScalable, and the Size property is always 1. Setting either of these properties has no effect when there is only one split. If there are multiple splits, and you then remove all but one, the SizeMode and Size properties of the remaining split automatically revert to 0 and 1, respectively.</p>

SizeMode Property

Syntax split.**SizeMode** = value

Read/Write at run time and design time.
Property applies to **Split** object.

Values

<u>Design Time</u>	<u>Run Time</u>
0 - Scalable (default)	dbgScalable
1 - Exact	dbgExact
2 - Number of Columns	dbgNumberOfColumns

Description

This property determines how the **Size** property is used to determine the actual size of a split.

If set to 0 - Scalable (the default), then the value returned by the **Size** property is an integer indicating the relative size of the split with respect to other scalable splits. For example, if a grid contains 3 scalable splits with **Size** properties equal to 1, 2, and 3, then the size of each split would be 1/6, 1/3, and 1/2 of the total grid width, respectively.

If set to 1 - Exact, then the value returned by the **Size** property is a floating point number indicating the exact size of the split in terms of the coordinate system of the grid's container. This setting allows you to fix the size of the split so that it always has the same width, even if new splits are added or existing splits are removed.

If set to 2 - Number of Columns, then the value returned by the **Size** property is an integer indicating the number of columns displayed in the split, and the split will adjust its width to display the number of full columns specified by the **Size** property. For example, if **Size** is set to 2, and the user scrolls the split horizontally, then the width of the split will change so that 2 full columns are displayed, regardless of how wide the columns are.

Note

Consider a grid containing both scalable splits and splits with a fixed number of columns. If a split with a fixed number of columns is scrolled horizontally, the total width remaining for the scalable splits may change because grid columns are generally of different widths. However, the ratios of the sizes of the scalable splits remain the same as specified by their **Size** properties.

Note that when there is only one split (the grid's default behavior), the split spans the entire width of the grid, the **SizeMode** property is always 0 - dbgScalable, and the **Size** property is always 1. Setting either of these properties has no effect when there is only one split. If there are multiple splits, and you then remove all but one, the **SizeMode** and **Size** properties of the remaining split automatically revert to 0 and 1, respectively.

Split Property

Syntax	TDBGrid. Split = integer Read/Write at run time. Not available at design time. Property applies to <u>TDBGrid</u> control.
Description	This property specifies the zero-based index of the current split.

Splits Property

Syntax	TDBGrid. Splits Read-only at run time. Not available at design time. Property applies to TDBGrid control.
Description	This property returns a collection of <u>Split</u> objects.

Style Property

Syntax object.**Style** = variant

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Description This property returns or sets the **Style** object that controls the normal appearance of a cell within a grid, column, or split.

Styles Property

Syntax	object. Styles Read-only at run time. Not available at design time. Property applies to <u>TDBGrid</u> and <u>TDBDropDown</u> controls.
Description	This property returns a collection of <u>Style</u> objects.

TabAcrossSplits Property

Syntax TDBGrid.TabAcrossSplits = boolean

Read/Write at run time and design time.

Property applies to **TDBGrid** control.

Description This property controls the behavior of the tab and arrow keys at split borders.

If True, the tab and arrow keys will move the current cell across split boundaries. When at the last column of the rightmost split (or the first column of the leftmost split), they will either wrap to the next row, stop, or move to other controls depending on the values of the **WrapCellPointer** and **TabAction** properties.

If False (the default), the tab and arrow keys will not move the current cell across split boundaries. They will either wrap to the next row, stop, or move to other controls depending on the values of the **WrapCellPointer** and **TabAction** properties.

Note The **TabAcrossSplits** property does not determine if the tab and arrow keys will move from cell to cell, or from control to control, or wrap to the next row. Use the **AllowArrows**, **WrapCellPointer**, and **TabAction** properties to control this behavior. If the tab and arrow keys are able to move from cell to cell, this property determines whether they will move across split boundaries to adjacent splits.

TabAction Property

Syntax TDBGrid.**TabAction** = value

Read/Write at run time and design time.
Property applies to TDBGrid control.

Values

Design Time

Run Time

0 - Control Navigation (default)

dbgControlNavigation

1 - Column Navigation

dbgColumnNavigation

2 - Grid Navigation

dbgGridNavigation

Description

This property defines the behavior of the tab key.

If set to 0 - Control Navigation (the default), the tab key moves to the next or previous control on the form.

If set to 1 - Column Navigation, the tab key moves the current cell to the next or previous column. However, if this action would cause the current row to change, then the next or previous control on the form receives focus.

If set to 2 - Grid Navigation, the tab key moves the current cell to the next or previous column. The behavior of the tab key at row boundaries is determined by the **WrapCellPointer** property. When this setting is used, the tab key never results in movement to another control.

Note

The **TabAction** property does not determine if the tab key will cross split boundaries. Use the **TabAcrossSplits** property to control this behavior.

Text Property

Syntax object.**Text** = string

Read/Write at run time. Not available at design time.

Property applies to **TDBGrid** and **TDBDropDown** controls, **Column** object.

This is the default property of the **TDBGrid** and **TDBDropDown** controls.

Description When applied to a **Column** object, this property returns or sets the formatted data value in a column for the current row.

The value returned by the **Text** property is derived from the underlying data value by applying the formatting as specified by the **NumberFormat** property of the **Column** object.

When the **Text** property is set for a formatted column, the underlying data value cannot be derived, and the **Text** and **Value** properties will subsequently return the same result.

Use the **Value** property to access the underlying data value in a column for the current row.

When applied to a **TDBGrid** or **TDBDropDown** control, this property returns or sets the text of the current cell. If the current cell is at EOF or BOF, an empty string is returned.

Top Property

Syntax

column.**Top**

Read-only at run time. Not available at design time.
Property applies to **Column** object.

Description

This property returns the position of a column's top edge relative to the top of the grid in terms of the coordinate system of the grid's container.

If the column contains a header, the **Top** property returns the position of the header's top edge; if the column does not contain a header, the **Top** property returns the position of the top edge of the column's cell within the first displayed row.

If the grid's **MultipleLines** property is False (the default value), a single record cannot span multiple lines in the grid, and the **Top** property returns the same value for all columns.

If the grid's **MultipleLines** property is True, a single record may span multiple lines in the grid. For columns on the first line, the **Top** property returns the height of the grid's caption bar and split headings, if present. For columns on succeeding lines, the **Top** property returns this value plus an appropriate multiple of the column header height.

Columns on the same line will return the same **Top** property value, while columns occupying lower lines in a row will return larger **Top** property values since they are farther away from the top of the grid.

For example, the following code places a text box on top of the header for the first column:

```
Text1.Top = TDBGrid1.Top + TDBGrid1.Columns(0).Top
```

Use the **Top** property in conjunction with **Left**, **Width**, and **RowTop** to determine the exact location and size of a column heading.

Note

To overlay the text box exactly on a column header, you may need to account for an extra pixel in the width and height, depending upon the settings of the **DividerStyle** and **RowDividerStyle** properties. In Visual Basic, if the **ScaleMode** property of the parent form is set to pixels, then you can simply add 1. If the **ScaleMode** is set to twips, then you can add the **TwipsPerPixelX** and **TwipsPerPixelY** properties of the **Screen** object.

Translate Property

Syntax

valueitems.**Translate** = boolean

Read/Write at run time and design time.

Property applies to **ValueItems** collection.

Description

This property determines whether a column's underlying data values are automatically displayed in an alternate form as specified by the **ValueItem** objects contained in a column's **ValueItems** collection.

If True, data values that match the **Value** property of a **ValueItem** are displayed using the corresponding **DisplayValue** setting. The **DisplayValue** property may contain either text or graphics.

If False (the default), no translation is performed.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

Validate Property

Syntax

valueitems.**Validate** = boolean

Read/Write at run time and design time.

Property applies to **ValueItems** collection.

Description

This property determines whether values entered by the user must match one of the **ValueItem** objects contained in a column's **ValueItems** collection.

If True, the grid automatically validates the user's input when the current cell is changed. If the cell contents do not match the **DisplayValue** setting of one of the **ValueItem** objects, then focus remains on the current cell and its prior contents are restored.

If False (the default), the grid performs no validation. However, you can still use the **BeforeColUpdate** event to validate the user's changes.

The **BeforeColUpdate** event will not be executed for a column if both of the following are true:

1. The associated **ValueItems** collection contains at least one **ValueItem**.
2. The **Validate** property of the **ValueItems** collection is set to True.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

Value Property (Column)

Syntax

column.**Value** = variant

Read/Write at run time. Not available at design time.

Property applies to **Column** object.

Description

This property returns or sets the underlying data value in a column for the current row.

The **Value** property is useful for simulating data entry within a cell. When this property is set, the value displayed in the cell respects the setting of the column's **NumberFormat** property.

This property always returns a string variant, even if the data type of the underlying field is numeric.

Use the **Text** property to access the formatted data value in a column for the current row.

Value Property (RowBuffer)

Syntax rowbuffer.**Value** (Row, Col) = variant

Read/Write at run time. Not available at design time.
Property applies to **RowBuffer** object.

Description This property returns or sets the data value for the specified cell within a **RowBuffer** object passed to an unbound event procedure for a **TDBGrid** control.

The Row argument is a long integer specifying the row where the value is placed. The range of this argument can be from 0 to **RowCount** - 1.

The Col argument is an integer specifying the column where the value is placed. The range of this argument can be from 0 to **ColumnCount** - 1.

In the **UnboundReadData** event, your code must provide data values for the rows being fetched. In the **UnboundAddData** and **UnboundWriteData** events, the user's changes are passed to your event procedures via a **RowBuffer** object.

Value Property (Style)

Syntax style.**Value** = variant

Read/Write at run time (with restrictions). Not available at design time.
Property applies to **Style** object.

Description This property returns the name of a style object.

For style objects, the **Value** property is a synonym for the **Name** property, so the following statements are equivalent:

```
Debug.Print TDBGrid1.Styles(0)
Debug.Print TDBGrid1.Styles(0).Name
```

For style objects that are members of the **Styles** collection, this property is read-only, and a trappable error will occur if you attempt to set it in code. However, for independent style objects, and for the **CellStyle** argument of the **FetchCellStyle** event, the **Value** property may be set in code to initialize the style object:

```
CellStyle.Value = "MyStyle"
CellStyle = "MyStyle" ' Both statements are equivalent
```

Note For an independent style object, the **Value** property always returns an empty string. An independent style object is not a member of the Styles collection, but is a standalone object created in code with a `Dim` or `Set` statement using the `New` keyword.

Value Property (ValueItem)

Syntax valueitem.**Value** = variant

Read/Write at run time and design time.
Property applies to **ValueItem** object.

Description This property returns or sets the underlying data value for a member of a **ValueItems** collection.

The **DisplayValue** property returns the translated data value for a value item.

Use the **ValueItems** property to access the **ValueItems** collection for a **Column** object.

ValueItems Property

Syntax column.**ValueItems**

Read-only at run time. Not available at design time.
Property applies to **Column** object.

Description This property returns a collection of **ValueItem** objects for a column.

Visible Property

Syntax column.**Visible** = boolean

Read/Write at run time and design time.
Property applies to **Column** object.

Description This property returns a boolean indicating whether a column in a grid or split is currently visible or capable of being scrolled into view.

If True, then the column has not been hidden in code or by the user.

If False, then the column is hidden and cannot be scrolled into view.

For columns created at design time, the default value of this property is True. For columns created in code at run time, the default value of this property is False.

Note If a column has been scrolled out of view, its **Visible** property remains True.

VisibleCols Property

Syntax object.**VisibleCols**

Read-only at run time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description For **TDBGrid** controls, this property returns the number of visible columns in the current split. The value returned includes both fully and partially displayed columns. You can use the **Split** property of the **TDBGrid** control to determine the index of the current split.

For **TDBDropDown** controls, this property returns the number of visible columns in the entire control, since there can only be one split. The value returned includes both fully and partially displayed columns.

VisibleRows Property

Syntax object.**VisibleRows** = integer

Read-only at run time.

Property applies to **TDBGrid** and **TDBDropDown** controls.

Description This property returns the number of visible rows in the control. The value returned includes both fully and partially displayed rows.

VScrollWidth Property

Syntax object.**VScrollWidth**

Read-only at run time. Not available at design time.

Property applies to **TDBGrid** control and **Split** object.

Description The **VScrollWidth** property returns the width of a split's vertical scroll bar in container units, which depend on the **ScaleMode** of the container. If no vertical scroll bar exists, then the returned value is zero. If object is a **TDBGrid** control, then its current split is used.

You can use the **VScrollWidth** and **HScrollHeight** properties to check if the scroll bars are present and to obtain the scroll bar size when designing the grid layout and sizing the grid and its columns.

Width Property

- Syntax** column.**Width** = single
- Read/Write at run time and design time.
Property applies to **Column** object.
- Description** This property returns or sets the width of a column in terms of the coordinate system of the grid's container.
- Use the **Width** property in conjunction with **Left**, **RowHeight**, and **RowTop** to determine the size and placement of controls displayed on top of a grid cell.
- Note** The **DefColWidth** property controls the default width of new columns created at run time.

WrapCellPointer Property

Syntax TDBGrid.**WrapCellPointer** = boolean

Read/Write at run time and design time.
Property applies to **TDBGrid** control.

Description This property determines the behavior of the arrow keys if **AllowArrows** is True.

If True, the cell pointer will wrap from the last column to the first in the next row (or from the first column to the last in the previous row).

If False (the default), the cell pointer will not wrap to the next (or previous) row, but will stop at the last (or first) column of the current row.

If **TabAcrossSplits** is False, the cell pointer will wrap only within the current split. If **TabAcrossSplits** is True, the cell pointer will move from one split to the next before wrapping occurs.

If **TabAction** is set to 2 - Grid Navigation, the tab key will behave like the arrow keys, and will automatically wrap to the next or previous cell.

WrapText Property

Syntax object.**WrapText** = boolean

Read/Write at run time and design time.
Property applies to **Column** and **Style** objects.

Description This property returns or sets a value indicating whether an object wraps text at cell boundaries.

If True, a line break occurs before words that would otherwise be partially displayed.

If False (the default), no line break occurs and text is clipped at the cell's right edge.

Use this property in conjunction with the **RowHeight** property to produce multiline displays.

By default, the word wrap behavior of a column is determined by its **Style** property setting. Setting the **WrapText** property overrides the style setting without changing the definition of the style itself.

If the **WrapText** property of a column is changed to the same value as its corresponding style, then the column will inherit its word wrap behavior from the style, and subsequent changes to the style's **WrapText** property will affect the column as well.

For **Style** objects, the value of the **WrapText** property is inherited from the parent style (if any) unless explicitly overridden, in which case the aforementioned inheritance rules also apply.

Method Reference

{button ,JI(``,`Quick_Reference_for_All_Methods')}} Quick Reference for All Methods
{button ,JI(``,`TDBGrid_Control_Methods')}} TDBGrid Control Methods
{button ,JI(``,`TDBDropDown_Control_Methods')}} TDBDropDown Control Methods
{button ,JI(``,`Column_Object_Methods')}} Column Object Methods
{button ,JI(``,`Layouts_Collection_Methods')}} Layouts Collection Methods
{button ,JI(``,`Columns_Collection_Methods')}} Columns Collection Methods
{button ,JI(``,`SelBookmarks_Collection_Methods')}} SelBookmarks Collection Methods
{button ,JI(``,`Split_Object_Methods')}} Split Object Methods
{button ,JI(``,`Splits_Collection_Methods')}} Splits Collection Methods
{button ,JI(``,`Style_Object_Methods')}} Style Object Methods
{button ,JI(``,`Styles_Collection_Methods')}} Styles Collection Methods
{button ,JI(``,`ValueItems_Collection_Methods')}} ValueItems Collection Methods

Quick Reference for All Methods

<u>AboutBox</u>	Displays the About box
<u>Add</u>	Adds a new item to a collection
<u>AddCellStyle</u>	Adds a cell condition to an object
<u>AddRegexCellStyle</u>	Adds a regular expression cell condition to an object
<u>CaptureImage</u>	Returns a captured image of a control's display
<u>CellText</u>	Returns displayed column text for any row
<u>CellValue</u>	Returns underlying column value for any row
<u>Clear</u>	Removes all value items from the collection
<u>ClearCellStyle</u>	Removes a cell condition from an object
<u>ClearFields</u>	Clears the current column/field layout
<u>ClearRegexCellStyle</u>	Removes a regular expression cell condition from an object
<u>ClearSelCols</u>	Deselects all selected columns in a split
<u>Close</u>	Disconnects a grid from its data source
<u>ColContaining</u>	Identifies a column under an X coordinate
<u>Delete</u>	Deletes the current row and moves to the next row
<u>GetBookmark</u>	Returns a bookmark relative to the current row
<u>HoldFields</u>	Uses the current column/field layout for all displays
<u>Item</u>	Returns a member of a collection given a name or index
<u>LoadLayout</u>	Loads a saved layout
<u>MoveFirst</u>	Moves to the first row
<u>MoveLast</u>	Moves to the last row
<u>MoveNext</u>	Moves to the next row
<u>MovePrevious</u>	Moves to the previous row
<u>MoveRelative</u>	Moves to the specified row and offset
<u>PostMsg</u>	Posts a message to a control to fire the PostEvent event
<u>ReBind</u>	Reinitializes a control from its data source
<u>Refresh</u>	Updates a control's screen display
<u>Remove</u>	Removes a member from a collection
<u>ReOpen</u>	Reconnects a grid to its data source
<u>Reset</u>	Resets style properties to inherited values
<u>RowBookmark</u>	Returns the bookmark corresponding to a display row
<u>RowContaining</u>	Identifies a row under a Y coordinate
<u>RowTop</u>	Returns the Y position of a row's top border
<u>Scroll</u>	Scrolls a control's data area
<u>SplitContaining</u>	Identifies a split under an X, Y coordinate pair
<u>Update</u>	Updates a grid's current row

TDBGrid Control Methods

<u>AboutBox</u>	Displays the About box
<u>AddCellStyle</u>	Adds a cell condition to a grid
<u>AddRegexCellStyle</u>	Adds a regular expression cell condition to a grid
<u>CaptureImage</u>	Returns a captured image of a grid's display
<u>ClearCellStyle</u>	Removes a cell condition from a grid
<u>ClearFields</u>	Clears the current column/field layout
<u>ClearRegexCellStyle</u>	Removes a regular expression cell condition from a grid
<u>ClearSelCols</u>	Deselects all selected columns in the current split
<u>Close</u>	Disconnects a grid from its data source
<u>ColContaining</u>	Identifies a column under an X coordinate
<u>Delete</u>	Deletes the current row and moves to the next row
<u>GetBookmark</u>	Returns a bookmark relative to the current row
<u>HoldFields</u>	Uses the current column/field layout for all displays
<u>LoadLayout</u>	Loads a saved layout
<u>MoveFirst</u>	Moves to the first row
<u>MoveLast</u>	Moves to the last row
<u>MoveNext</u>	Moves to the next row
<u>MovePrevious</u>	Moves to the previous row
<u>MoveRelative</u>	Moves to the specified row and offset
<u>PostMsg</u>	Posts a message to a grid to fire the PostEvent event
<u>ReBind</u>	Reinitializes a grid from its data source
<u>Refresh</u>	Updates a grid's screen display
<u>ReOpen</u>	Reconnects a grid to its data source
<u>RowBookmark</u>	Returns the bookmark corresponding to a display row
<u>RowContaining</u>	Identifies a row under a Y coordinate
<u>RowTop</u>	Returns the Y position of a row's top border
<u>Scroll</u>	Scrolls a grid's data area
<u>SplitContaining</u>	Identifies a split under an X, Y coordinate pair
<u>Update</u>	Updates a grid's current row

TDBDropDown Control Methods

<u>AboutBox</u>	Displays the About box
<u>AddCellStyle</u>	Adds a cell condition to a dropdown
<u>AddRegexCellStyle</u>	Adds a regular expression cell condition to a dropdown
<u>CaptureImage</u>	Returns a captured image of a dropdown's display
<u>ClearCellStyle</u>	Removes a cell condition from a dropdown
<u>ClearFields</u>	Clears the current column/field layout
<u>ClearRegexCellStyle</u>	Removes a regular expression cell condition from a dropdown
<u>ClearSelCols</u>	Deselects all selected columns
<u>ColContaining</u>	Identifies a column under an X coordinate
<u>GetBookmark</u>	Returns a bookmark relative to the current row
<u>HoldFields</u>	Uses the current column/field layout for all displays
<u>PostMsg</u>	Posts a message to the dropdown to fire the PostEvent event
<u>ReBind</u>	Reinitializes a dropdown from its data source
<u>Refresh</u>	Updates a dropdown's screen display
<u>RowBookmark</u>	Returns the bookmark corresponding to a display row
<u>RowContaining</u>	Identifies a row under a Y coordinate
<u>RowTop</u>	Returns the Y position of a row's top border
<u>Scroll</u>	Scrolls a dropdown's data area

Column Object Methods

<u>AddCellStyle</u>	Adds a cell condition to a column
<u>AddRegexCellStyle</u>	Adds a regular expression cell condition to a column
<u>CellText</u>	Returns displayed text for any row
<u>CellValue</u>	Returns underlying value for any row
<u>ClearCellStyle</u>	Removes a cell condition from a column
<u>ClearRegexCellStyle</u>	Removes a regular expression cell condition from a column

Columns Collection Methods

Add

Adds a new column to the collection

Item

Returns a single column object given a name or index

Remove

Removes an existing column from the collection

Layouts Collection Methods

Add

Adds a new layout to the collection

Item

Returns the name of a layout given an index

Remove

Removes an existing layout from the collection

SelBookmarks Collection Methods

Add

Adds a bookmark to the list of selected rows

Item

Returns an individual selected row bookmark

Remove

Removes a bookmark from the list of selected rows

Split Object Methods

<u>AddCellStyle</u>	Adds a cell condition to a split
<u>AddRegexCellStyle</u>	Adds a regular expression cell condition to a split
<u>ClearCellStyle</u>	Removes a cell condition from a split
<u>ClearRegexCellStyle</u>	Removes a regular expression cell condition from a split
<u>ClearSelCols</u>	Deselects all selected columns in a split

Splits Collection Methods

Add

Adds a new split at the given index

Item

Returns a single split object given an index

Remove

Removes the split with the given index

Style Object Methods

Reset

Resets style properties to inherited values

Styles Collection Methods

Add

Adds a new named style to the collection

Item

Returns a single style object given a name or index

Remove

Removes an existing style from the collection

ValueItems Collection Methods

<u>Add</u>	Appends a new value item to the collection
<u>Clear</u>	Removes all value items from the collection
<u>Item</u>	Returns a single value item given an index
<u>Remove</u>	Removes a value list item from the collection

AboutBox Method

Syntax object.**AboutBox**

Method applies to TDBGrid and TBDropDown controls.

Arguments None

Return Value None

Description This method displays the version number and copyright notice for the specified control.

Add Method

Syntax	object. Add (item) Method applies to Columns , Layouts , SelBookmarks , Splits , Styles , and ValueItems collections.
Arguments	<u>item</u> is an expression or object that specifies the member to add to the collection.
Return Value	A reference to the newly created object where appropriate, otherwise none.
Description	<p>For the Columns, Splits, and Styles collections, this method creates a new instance of the appropriate object, adds it to the collection, and returns it to the caller.</p> <p>For the Layouts, SelBookmarks, and ValueItems collections, this method adds the specified object to the collection without returning a value.</p> <p>The data type of the <u>item</u> argument depends on the collection. For the Columns and Splits collections, <u>item</u> is a zero-based integer denoting the index of the newly created column or split.</p> <p>For the SelBookmarks collection, <u>item</u> is a variant representing a bookmark that identifies a particular row. Depending upon the setting of the DataMode property, <u>item</u> may have been obtained from a bound data source, an unbound mode or application mode event, or an XArray row index.</p> <p>For the Styles collection, <u>item</u> is the unique name of the style to be created.</p> <p>For the ValueItems collection, <u>item</u> is an independent ValueItem object.</p> <p>For the Layouts collection, <u>item</u> is the name of a grid layout to be saved to the binary layout file specified by the LayoutFileName property. All of the grid's persistent properties are saved in their current state. If the named layout already exists in the file, the property settings are overwritten. If the named layout does not already exist, it is added to the file.</p>

AddCellStyle Method

Syntax object.**AddCellStyle** condition, style

Method applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Arguments condition is a combination of one or more CellStyleConstants.

style is a **Style** object that specifies font and color attributes.

Return Value None

Description This method allows you to control the font and color of cells within a grid, column, or split according to the status values (CellStyleConstants) specified by the condition argument:

- | | |
|--------------------|---|
| 1 - dbgCurrentCell | The cell is the current cell as specified by the Bookmark , Col , and Split properties. At any given time, only one cell can have this status. When the MarqueeStyle property is set to 6 - Floating Editor, this condition is ignored. |
| 2 - dbgMarqueeRow | The cell is part of a highlighted row marquee. When the MarqueeStyle property indicates that the entire current row is to be highlighted, all visible cells in the current row have this additional condition set. |
| 4 - dbgUpdatedCell | The cell contents have been modified by the user but not yet written to the database. This condition is also set when cell contents have been modified in code with the Text or Value properties. |
| 8 - dbgSelectedRow | The cell is part of a row selected by the user or in code. The SelBookmarks collection contains a bookmark for each selected row. |
| 0 - dbgNormalCell | The cell satisfies none of these conditions. |

You can add the first four values together to specify multiple cell conditions. For example, a cell status value of 9 (dbgCurrentCell + dbgSelectedRow) denotes a current cell within a selected row. You can also use a cell status value of 0 (dbgNormalCell) to refer to only those cells without any of the four basic cell conditions.

The style argument specifies the attributes that will override the default font and color characteristics for cells within an object. For example, the following code causes updated cells to be displayed in red:

```
Dim S As New TrueDBGrid50.Style
S.ForeColor = vbRed
TDBGrid1.AddCellStyle dbgUpdatedCell, S
```

Each time the **AddCellStyle** method is invoked, the specified cell condition is added to the list of existing conditions. Hence, by repeated use of this method it is possible to set up multiple conditions to affect the appearance of a grid, column, or split.

Note If a cell condition already exists for a particular condition value, the new style setting replaces the existing one.

AddRegexCellStyle Method

Syntax	object. AddRegexCellStyle condition, style, regex Method applies to TDBGrid and TDBDropDown controls, Column and Split objects.
Arguments	<u>condition</u> is a combination of one or more CellStyleConstants. <u>style</u> is a Style object that specifies font and color attributes. <u>regex</u> is a regular expression string.
Return Value	None
Description	This method allows you to control the font and color of cells within a grid, column, or split according to their contents. The status values(CellStyle Constants) specified by the <u>condition</u> argument determine which cells are affected: 1 - dbgCurrentCell The cell is the current cell as specified by the Bookmark , Col , and Split properties. At any given time, only one cell can have this status. When the floating editor MarqueeStyle property setting is in effect, this condition is ignored. 2 - dbgMarqueeRow The cell is part of a highlighted row marquee. When the MarqueeStyle property indicates that the entire current row is to be highlighted, all visible cells in the current row have this additional condition set. 4 - dbgUpdatedCell The cell contents have been modified by the user but not yet written to the database. This condition is also set when cell contents have been modified in code with the Text or Value properties. 8 - dbgSelectedRow The cell is part of a row selected by the user or in code. The SelBookmarks collection contains a bookmark for each selected row. 0 - dbgNormalCell The cell satisfies none of these conditions. -1 - dbgAllCells All cells satisfy this condition. You can add the first four values together to specify multiple cell conditions. For example, a cell status value of 9 (dbgCurrentCell + dbgSelectedRow) denotes a current cell within a selected row. You can also use a cell status value of 0 (dbgNormalCell) to refer to only those cells without any of the four basic cell conditions. To designate that a cell condition should apply to all cells regardless of status, use a cell status value of -1 (dbgAllCells). The <u>regex</u> argument is a regular expression string that describes the pattern matching to be performed on cell contents. The regular expressions supported by True DBGrid are a subset of standard Unix regular expression syntax and are not compatible with the Visual Basic Like operator. The following special characters are supported: p* Any pattern followed by an asterisk matches zero or more occurrences of that pattern. For example, ab*c matches ac, abc, and abbcy (partial match). p+ Any pattern followed by a plus sign matches one or more occurrences of that pattern. For example, ab+c matches abc and abbcy, but not ac. [list] A list of case-sensitive characters enclosed in brackets matches any one of those characters in the given position in the string. Character ranges can be used, as in [abcd], which is equivalent to [a-d]. Multiple ranges can also be used. For example, [A-Za-z0-9] matches

	any letter or digit. Bracketed patterns can also be combined with either the * or + operators. The pattern [A-Z]+ matches a sequence of one or more uppercase letters.
[^list]	If a list starts with a caret, it matches any character except those specified in the list.
. (period)	A period represents any single character.
^p	A caret at the beginning of a pattern forces a match to occur at the start of a cell. Otherwise, the pattern can match anywhere within a cell.
p\$	A dollar sign at the end of a pattern forces a match to occur at the end of a cell. Otherwise, the pattern can match anywhere within a cell.
\c	Any character preceded by a backslash represents itself, unless enclosed in brackets, in which case the backslash is interpreted literally.

Any other character represents itself and will be compared with respect to case.

The style argument specifies the attributes that will override the default font and color characteristics for cells within an object. For example, the following code causes normal cells containing the letters "SQL" to be displayed in bold:

```
Dim S As New TrueDBGrid50.Style
S.Font.Bold = True
TDBGrid1.AddRegexCellStyle dbgNormalCell, S, "SQL"
```

Each time the **AddRegexCellStyle** method is invoked, the specified cell condition is added to the list of existing conditions. Hence, by repeated use of this method it is possible to set up multiple conditions to affect the appearance of a grid, column, or split.

Note

If a cell condition already exists for a particular pair of condition and regex values, the new style setting replaces the existing one.

CaptureImage Method

Syntax object.**CaptureImage**

Method applies to **TDBGrid** and **TDBDropDown** controls.

Arguments None

Return Value A picture object containing a snapshot of the control's display.

Description This method returns an image of the grid in a format that you can assign to the **Picture** property of a Visual Basic form or control, or the **PaintPicture** method of the **Printer** object.

CellText Method

Syntax column.**CellText** (bookmark)

Method applies to **Column** object.

Arguments bookmark is a variant representing a grid row.

Return Value A string containing the displayed column text for the specified row.

Description The **CellText** method returns a formatted string representation of the data in a column for the row specified by the bookmark argument. Using the **CellText** method is similar to accessing the **Text** property, except that you can select a specific row from which to retrieve the value.

The value returned by the **CellText** method is derived from the underlying data value by applying the formatting as specified by the **NumberFormat** property of the **Column** object.

Using the **CellText** method to extract information from a cell doesn't affect the current selection.

Use the **CellValue** method to access the unformatted data value for the specified row.

CellValue Method

Syntax column.**CellValue** (bookmark)

Method applies to **Column** object.

Arguments bookmark is a variant representing a grid row.

Return Value A variant containing the underlying data value for the specified row.

Description The **CellValue** method returns the raw data value in a column for the row specified by the bookmark argument. Using the **CellValue** method is similar to accessing the **Value** property, except that you can select a specific row from which to retrieve the value.

Using the **CellValue** method to extract information from a cell doesn't affect the current selection.

Use the **CellText** method to access the formatted data value for the specified row.

Clear Method

Syntax valueitems.**Clear**

Method applies to ValueItems collection.

Arguments None

Return Value None

Description The Clear method removes all ValueItem objects from a ValueItems collection associated with a Column object.

ClearCellStyle Method

- Syntax** object.**ClearCellStyle** condition
- Method applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.
- Arguments** condition is a combination of one or more CellStyleConstants.
- Return Value** None
- Description** The **ClearCellStyle** method removes a cell condition established with a previous call to the **AddCellStyle** method for the object in question. If no such cell condition exists, then calling this method has no effect.
- If the condition argument is -1 (dbgAllCells), then all non-regex cell conditions are removed, regardless of status.

ClearFields Method

Syntax object.**ClearFields**

Method applies to **TDBGrid** and **TDBDropDown** controls.

Arguments None

Return Value None

Description The **ClearFields** method restores the default grid layout (with two blank columns) so that subsequent **ReBind** operations will automatically derive new column bindings from the (possibly changed) data source. You can cancel the grid's automatic layout behavior by invoking the **HoldFields** method.

ClearRegexCellStyle Method

Syntax object.**ClearRegexCellStyle** condition [, regex]

Method applies to **TDBGrid** and **TDBDropDown** controls, **Column** and **Split** objects.

Arguments condition is a combination of one or more CellStyleConstants.

regex is an optional regular expression string.

Return Value None

Description The **ClearRegexCellStyle** method removes a cell condition established with a previous call to the **AddRegexCellStyle** method for the object in question. If no such cell condition exists, then calling this method has no effect.

If regex is omitted, then all cell conditions for any regular expression matching the condition argument are removed. If condition is -1 (`dbgAllCells`), then all regex cell conditions are removed, regardless of status or expression.

If regex is supplied, then the single cell condition matching both arguments is removed. However, if condition is -1 (`dbgAllCells`), then all cell conditions for the specified regular expression are removed, regardless of status.

ClearSelCols Method

Syntax object.**ClearSelCols**

Method applies to **TDBGrid** and **TDBDropDown** controls, **Split** object.

Arguments None

Return Value None

Description The **ClearSelCols** method deselects all columns in a split. If no columns are selected, then this method does nothing.

If a grid contains multiple splits, then invoking its **ClearSelCols** method has the same effect as invoking the **ClearSelCols** method for the current split. The index of the current split is available through the **TDBGrid** control's **Split** property.

Use the **SelStartCol** and **SelEndCol** properties to determine the current column selection range for a split.

Close Method

Syntax TDBGrid.**Close** [repaint]

Method applies to **TDBGrid** control.

Arguments repaint is an optional boolean that determines whether the grid should clear its display.

Return Value None

Description The **Close** method disconnects the grid from the data source. The optional repaint argument instructs the grid whether or not to "clear" the grid of data. If repaint is True (the default if not specified), the grid is cleared of all data; if repaint is False, the data currently on display at the time of the close remains, but the grid's user interface is disabled so that no operations can be performed on the grid. All database related coding operations (such as **MoveNext**, **MovePrevious**) will return data access errors until the connection is re-established with any of the following methods: **ReBind**, **Refresh**, or **ReOpen**.

Passing a value of False to the **Close** method is useful when operations are performed that would otherwise "flicker" the display.

The data source connection is automatically reopened whenever the grid's **ReBind**, **Refresh**, or **ReOpen** methods are executed. The grid will be repopulated with data and the appropriate row will be made current. Care should be taken when using **ReBind** or **ReOpen** with unbound grids, as these operations assume the current row bookmark still exists.

In the case of a bound grid, **Close** allows database operations to be performed without interference from the grid. For example, closing the grid and closing the Recordset and Database objects associated with the Data control:

```
TDBGrid1.Close  
Data1.Recordset.Close  
Data1.Database.Close
```

would allow database operations which manipulate the database files (such as pack or copy) to be performed.

The **Close** method can also be used to temporarily disconnect a grid from its data source when many notification-generating operations need to be performed. For example, if your application contains a loop that deletes all of the selected records in a grid, you can first **Close** (disconnect) the grid as follows:

```
TDBGrid1.Close False
```

so that no notifications will be processed by the grid. Otherwise, the grid will show every row movement and deletion as the records are deleted. The False argument tells the grid not to repaint the screen, which has the effect of leaving the grid display unchanged. When the record delete operations are completed, perform a grid **ReOpen** (or **ReBind**) to reconnect and repaint the grid with the remaining (undeleted) records. This technique is also useful for RDC/RDO where clones are not available.

ColContaining Method

Syntax object.**ColContaining** (coordinate)

Method applies to **TDBGrid** and **TDBDropDown** controls.

Arguments coordinate is a single that defines a horizontal coordinate (X value) in twips.

Return Value An integer corresponding to the index of the column beneath the specified X coordinate.

Description The **ColContaining** method returns the **ColIndex** value of the grid column containing the specified coordinate. This value ranges from 0 to **Columns.Count** - 1.

This method is useful when working with mouse and drag events when you are trying to determine where the user clicked or dropped another control in terms of a grid column.

If coordinate is outside of the grid's data area, this method returns -1.

The **ColContaining** method returns the **ColIndex** of the column indicated, not the visible column position. For example, if coordinate falls within the first visible column, but two columns have been scrolled off the left side of the control, the **ColContaining** method returns 2, not 0 (assuming that the user did not move any columns previously).

Note You must convert the coordinate argument to twips, even if the container's **ScaleMode** setting specifies a different unit of measurement.

Delete Method

Syntax TDBGrid.**Delete**

Method applies to TDBGrid control.

Arguments None

Return Value None

Description The Delete method deletes the current record, then *automatically* moves to the next available record. If the last record is deleted, then EOF becomes the current position.

GetBookmark Method

Syntax	object. GetBookmark (offset) Method applies to TDBGrid and TDBDropDown controls.
Arguments	<u>offset</u> is a long integer that defines the target row relative to the current row.
Return Value	A variant containing a bookmark relative to the current row as specified by <u>offset</u> .
Description	<p>The GetBookmark method returns a bookmark for a row relative to the current row, which need not be visible.</p> <p>If <u>offset</u> is 0, this method returns the bookmark of the current row. The value returned will be the same as that returned by the Bookmark property.</p> <p>If <u>offset</u> is 1, this method returns the bookmark of the row following the current row. Similarly, if <u>offset</u> is -1, this method returns the bookmark of the row preceding the current row.</p> <p>If <u>offset</u> is an arbitrary integer N, this method returns the bookmark of the Nth row following the current row if N is positive, or preceding the current row if N is negative.</p> <p>If <u>offset</u> targets a row after the last available record or before the first available record, then this method returns Null.</p>
Note	Do not confuse the GetBookmark method with the RowBookmark method, which can only return bookmarks for visible rows.

HoldFields Method

Syntax object.**HoldFields**

Method applies to **TDBGrid** and **TDBDropDown** controls.

Arguments None

Return Value None

Description The **HoldFields** method sets the current column/field layout as the customized layout so that subsequent **ReBind** operations will use the current layout for display. You can resume the grid's automatic layout behavior by invoking the **ClearFields** method.

The **HoldFields** method is especially useful in the unbound modes when you have specified the column layout in code and would like to keep it intact after a **ReBind** operation.

Item Method

- Syntax** object.**Item** (index)
- Method applies to **Columns**, **SelBookmarks**, **Splits**, **Styles**, and **ValueItems** collections.
- Arguments** index is an expression that specifies the collection member to be accessed.
- Return Value** A reference to the specified object.
- Description** Use the **Item** method to access a specific member of a True DBGrid collection.
- All collections accept a zero-based index argument. The **Columns** and **Styles** collections also accept named arguments.
- Note** The **Item** method is not required. The following expressions are equivalent:
- ```
TDBGrid1.Columns(0)
TDBGrid1.Columns.Item(0)
```

## LoadLayout Method

**Syntax** TDBGrid.**LoadLayout**

Method applies to **TDBGrid** control.

**Arguments** None

**Return Value** None

**Description** The **LoadLayout** method loads a previously saved grid layout (as specified in the **LayoutName** property) from a binary layout storage file (as specified in the **LayoutFileName** property), and configures the grid accordingly. Before calling this method, you must set the **LayoutFileName** and **LayoutName** properties to valid values. You can use this method to easily change the grid layout at run time.

To save the current grid layout to a binary layout storage file, use the **Add** method of the **Layouts** collection. To remove a named layout from a binary layout storage file, use the **Remove** method of the **Layouts** collection.

## MoveFirst Method

**Syntax** TDBGrid.**MoveFirst**

Method applies to TDBGrid control.

**Arguments** None

**Return Value** None

**Description** The MoveFirst method operates like its **Recordset** counterpart; it moves the current record to the first record available.

## MoveLast Method

**Syntax** TDBGrid.**MoveLast**

Method applies to TDBGrid control.

**Arguments** None

**Return Value** None

**Description** The MoveLast method operates like its **Recordset** counterpart; it moves the current record to the last record available.

## MoveNext Method

**Syntax** TDBGrid.**MoveNext**

Method applies to TDBGrid control.

**Arguments** None

**Return Value** None

**Description** The MoveNext method operates like its **Recordset** counterpart; it moves the current record to the next record available.



## MovePrevious Method

|                     |                                                                                                                                              |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | TDBGrid. <b>MovePrevious</b><br>Method applies to <u>TDBGrid</u> control.                                                                    |
| <b>Arguments</b>    | None                                                                                                                                         |
| <b>Return Value</b> | None                                                                                                                                         |
| <b>Description</b>  | The <u>MovePrevious</u> method operates like its <b>Recordset</b> counterpart; it moves the current record to the previous record available. |

## MoveRelative Method

|                     |                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | TDBGrid. <b>MoveRelative</b> offset [, bookmark]<br>Method applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                           |
| <b>Arguments</b>    | <u>offset</u> is a long integer that specifies the number of records to move. A positive value indicates forward movement; a negative value indicates backward movement.<br><u>bookmark</u> is an optional variant that specifies the origin of the relative movement. If not specified, the current record is assumed. |
| <b>Return Value</b> | None                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b>  | The <b>MoveRelative</b> method operates like the <b>Move</b> method of the <b>Recordset</b> object; it moves the current record <u>offset</u> rows relative to the specified <u>bookmark</u> .                                                                                                                          |

## PostMsg Method

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | object. <b>PostMsg</b> MsgId<br>Method applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Arguments</b>    | <u>MsgId</u> is an integer that identifies the message being posted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Return Value</b> | None                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b>  | <p>The <b>PostMsg</b> method is used in conjunction with the <b>PostEvent</b> event to postpone operations which are illegal within the grid's events. If the <b>PostMsg</b> method is called, the grid will fire the <b>PostEvent</b> event with the <u>MsgId</u> of the corresponding <b>PostMsg</b> invocation after all pending operations are completed. You can then safely perform all desired operations in the <b>PostEvent</b> event.</p> <p>For example, it is not possible to perform the Data control's <b>Refresh</b> method within the grid's <b>AfterUpdate</b> event because database operations are still pending, and the refresh cannot be tolerated. Instead of performing the refresh in the <b>AfterUpdate</b> event, you can call the <b>PostMsg</b> method (with a <u>MsgId</u> value of 1, for instance). After all pending database operations are completed, the grid will fire the <b>PostEvent</b> event. You can then perform the refresh operation safely in this event, after confirming that the <u>MsgId</u> argument passed in is 1.</p> <p>You can use any non-zero integral value for <u>MsgId</u>, which will be passed to the <b>PostEvent</b> event for identification purposes.</p> <p>The special case where <u>MsgId</u> is zero is used to clear any pending <b>PostMsg</b> invocations which have not yet been processed. A <b>PostEvent</b> event will fire for this case.</p> |
| <b>Note</b>         | Take care to avoid recursive situations when using <b>PostMsg</b> and <b>PostEvent</b> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |

## ReBind Method

**Syntax**            object.**ReBind**

Method applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        None

**Return Value**     None

**Description**     This method re-establishes the connection with the bound data source, causing the **TDBGrid** control to perform the same operations that occur when you set the **DataSource** property at design time.

If you have not modified the grid columns at design time, then executing the **ReBind** method will reset the columns, headings, and other properties based on the current data source.

However, if you have altered the columns in any way at design time (even if you leave the **DataField** properties blank), then the grid will assume that you wish to maintain the modified grid layout and will not automatically reset the columns.

For an unbound grid with its **DataMode** property set to 2 - Unbound Extended or 3 - Application, this method discards all data and fires the **UnboundReadDataEx** or **ClassicRead** event to refill the grid with records from the unbound data source. After the grid has finished refilling its cache, it fires the **FirstRowChange** and **RowColChange** events.

For an unbound grid with its **DataMode** property set to 1 - Unbound, this method is similar to the **Refresh** method except that the grid attempts to restore the current and topmost rows.

**Note**                To force the grid to reset the column bindings even if the columns were modified at design time, invoke the **ClearFields** method immediately before **ReBind**. Conversely, to cancel the grid's automatic layout response and force the grid to use the current column/field layout, invoke the **HoldFields** method immediately before **ReBind**.

The **HoldFields** method is especially useful in the unbound modes when you have specified the column layout in code and would like to keep it intact after a **ReBind** operation.

## Refresh Method

**Syntax**            object.**Refresh**

Method applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        None

**Return Value**     None

**Description**      For a bound grid (one with its **DataMode** property set to 0) and an unbound grid with its **DataMode** property set to 2 - Unbound Extended or 3 - Application, this method simply forces the entire grid to be repainted. Database access may or may not occur in these cases.

For an unbound grid one with its **DataMode** property set to 1 - Unbound, this method discards all data and fires the **UnboundReadData** event to refill the grid with records from the unbound data source. After the grid has finished refilling its cache, it fires the **FirstRowChange** and **RowColChange** events.

**Note**                To force a bound grid to discard all of its data and refetch records from the database, use the Data control's **Refresh** method.

If the grid's data source has been disconnected with the **Close** method, calling **Refresh** will implicitly **ReOpen** the data source and set the current row to the first available row.

## Remove Method

**Syntax**            object.**Remove** (index)

Method applies to **Columns**, **Layouts**, **SelBookmarks**, **Splits**, **Styles**, and **ValueItems** collections.

**Arguments**        index is an expression that specifies the collection member to be removed.

**Return Value**     None

**Description**      Use the **Remove** method to delete a specific member of a True DBGrid collection.

All collections accept a zero-based index argument. The **Columns** and **Styles** collections also accept named arguments.

## ReOpen Method

**Syntax** TDBGrid.**ReOpen** [bookmark]

Method applies to **TDBGrid** control.

**Arguments** bookmark is an optional variant that specifies the row to position to immediately after the data source is reopened. If specified, it must be a valid bookmark.

**Return Value** None

**Description** The **ReOpen** method reconnects the grid to its data source. The grid is repopulated and the current row is positioned to the row identified by the optional bookmark argument. If bookmark is not specified, then the current row reflects the current row of the data source.

In unbound mode, if the current row needs to be changed after the **ReOpen** operation, you can significantly reduce the number of **UnboundReadData** (or **UnboundReadDataEx**) events that fire by specifying the optional bookmark argument.

**Note** If **ReOpen** is called while the grid is still connected to its data source, the grid implicitly performs a **Close** first:

```
TDBGrid1.Close False
```

The optional bookmark is honored as usual when the data source is reopened.

## Reset Method

**Syntax**            style.**Reset**

Method applies to **Style** object.

**Arguments**        None

**Return Value**     None

**Description**      This method causes a style to inherit all of its properties from its parent style.



## RowBookmark Method

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | object. <b>RowBookmark</b> (rownumber)<br>Method applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Arguments</b>    | <u>rownumber</u> is an integer denoting a displayed row.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Return Value</b> | A variant containing a bookmark corresponding to the display row specified by <u>rownumber</u> .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Description</b>  | <p>The <b>RowBookmark</b> method returns a bookmark for a visible row relative to the first displayed row.</p> <p>If <u>rownumber</u> is 0, this method returns the bookmark of the first displayed row. The value returned will be the same as that returned by the <b>FirstRow</b> property.</p> <p>Allowable values for the <u>rownumber</u> argument range from 0 to <b>VisibleRows</b> - 1.</p> <p>This method only returns the current row (as determined by the grid's <b>Bookmark</b> property) if the current row is visible and the <u>rownumber</u> argument is equal to the grid's <b>Row</b> property.</p> |
| <b>Note</b>         | Do not confuse the <b>RowBookmark</b> method with the <b>GetBookmark</b> method, which returns a bookmark relative to the current row, even if the row is not visible.                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## RowContaining Method

**Syntax** object.**RowContaining** (coordinate)

Method applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** coordinate is a single that defines a vertical coordinate (Y value) in twips.

**Return Value** An integer corresponding to the display row beneath the specified Y coordinate.

**Description** The **RowContaining** method returns the zero-based index of the display row containing the specified coordinate. This value ranges from 0 to **VisibleRows** - 1.

When handling mouse and drag events, this method is useful when you need to determine where the user clicked or dropped another control in terms of a grid row.

If coordinate is outside of the grid's data area, this method returns -1.

**Note** You must convert the coordinate argument to twips, even if the container's **ScaleMode** setting specifies a different unit of measurement.

## RowTop Method

**Syntax**            object.**RowTop** (rownumber)

Method applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        rownumber is an integer denoting a displayed row.

**Return Value**     A single corresponding to the Y position of the specified display row, based on the coordinate system of the grid's container.

**Description**      The **RowTop** method returns the Y coordinate of the top of a visible row relative to the top of the grid, as given by the grid's **Top** property.

Allowable values for the rownumber argument range from 0 to **VisibleRows** - 1.

Use the **RowTop** method in conjunction with **RowHeight**, **Left**, and **Width** to determine the size and placement of controls displayed on top of a grid cell.

## Scroll Method

**Syntax** object.**Scroll** coloffset, rowoffset

Method applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** coloffset is a long integer denoting the number of columns to scroll and the direction in which to scroll them.

rowoffset is a long integer denoting the number of rows to scroll and the direction in which to scroll them.

**Return Value** None

**Description** This method scrolls the grid horizontally and vertically in a single operation.

Positive offsets scroll right and down. Negative offsets scroll left and up. Column offsets that are out of range cause a trappable error. Row offsets that are out of range scroll to the beginning or end of the database.

The same effect can be achieved by setting the **LeftCol** and **FirstRow** properties, but these must be set independently.

## SplitContaining Method

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>       | TDBGrid. <b>SplitContaining</b> (x, y)<br>Method applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Arguments</b>    | <u>x</u> and <u>y</u> are singles that define a coordinate pair in twips.                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Return Value</b> | An integer corresponding to the index of the split beneath the specified coordinate pair.                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b>  | <p>The <b>SplitContaining</b> method returns the Index value of the split containing the specified coordinate pair. This value ranges from 0 to <b>Splits.Count</b> - 1.</p> <p>This method is useful when working with mouse and drag events when you are trying to determine where the user clicked or dropped another control in terms of a grid column.</p> <p>If either argument is outside of the grid's data area, this method returns -1.</p> |
| <b>Note</b>         | You must convert the <u>x</u> and <u>y</u> arguments to twips, even if the container's <b>ScaleMode</b> setting specifies a different unit of measurement.                                                                                                                                                                                                                                                                                            |

## Update Method

**Syntax** TDBGrid.**Update**

Method applies to TDBGrid control.

**Arguments** None

**Return Value** None

**Description** This methods forces data from the current row to be updated to the underlying database. It applies to both a bound grid and an unbound grid.



## Event Reference

---

{button ,JI(`,`Quick\_Reference\_for\_All\_Events')} Quick Reference for All Events

{button ,JI(`,`TDBGrid\_Control\_Events')} TDBGrid Control Events

{button ,JI(`,`TDBDropDown\_Control\_Events')} TDBDropDown Control Events



## Quick Reference for All Events

|                                          |                                                          |
|------------------------------------------|----------------------------------------------------------|
| <b><u>AfterColEdit</u></b>               | Fired after column data is edited                        |
| <b><u>AfterColUpdate</u></b>             | Occurs after data moves from cell to the grid buffer     |
| <b><u>AfterDelete</u></b>                | Occurs after record deletion from grid                   |
| <b><u>AfterInsert</u></b>                | Occurs after record insertion in grid                    |
| <b><u>AfterUpdate</u></b>                | Occurs after record changes are written to the database  |
| <b><u>BeforeColEdit</u></b>              | Fired upon an attempt to edit column data                |
| <b><u>BeforeColUpdate</u></b>            | Occurs before data moves from cell to the grid buffer    |
| <b><u>BeforeDelete</u></b>               | Occurs before record deletion from grid                  |
| <b><u>BeforeInsert</u></b>               | Occurs before record insertion in grid                   |
| <b><u>BeforeUpdate</u></b>               | Occurs before record changes are written to the database |
| <b><u>ButtonClick</u></b>                | Occurs when a column button has been clicked             |
| <b><u>Change</u></b>                     | Occurs when the grid contents have changed               |
| <b><u>ClassicAdd</u></b>                 | Fired when a new row is added to the unbound dataset     |
| <b><u>ClassicDelete</u></b>              | Fired when an unbound row needs to be deleted            |
| <b><u>ClassicRead</u></b>                | Fired when the control requires unbound data for display |
| <b><u>ClassicWrite</u></b>               | Fired when an unbound row needs to be modified           |
| <b><u>Click</u></b>                      | Fired when a mouse click occurs                          |
| <b><u>ColEdit</u></b>                    | Fired when column data is edited                         |
| <b><u>ColMove</u></b>                    | Occurs before repainting when columns are moved          |
| <b><u>ColResize</u></b>                  | Occurs before repainting when a column is resized        |
| <b><u>ComboSelect</u></b>                | Fired when the user selects a ValueItems combo entry     |
| <b><u>DblClick</u></b>                   | Fired when a mouse double click occurs                   |
| <b><u>DragCell</u></b>                   | Fired when a drag operation is initiated                 |
| <b><u>DropDownClose</u></b>              | Fired when the dropdown is closed                        |
| <b><u>DropDownOpen</u></b>               | Fired when the dropdown is displayed                     |
| <b><u>Error</u></b>                      | Occurs when an associated action fails                   |
| <b><u>FetchCellStyle</u></b>             | Fired when the FetchStyle property is True for a column  |
| <b><u>FetchCellTips</u></b>              | Fired when the CellTips property is set to True          |
| <b><u>FetchRowStyle</u></b>              | Fired when the FetchRowStyle property is set to True     |
| <b><u>FirstRowChange</u></b>             | Fired when the FirstRow property changes                 |
| <b><u>FormatText</u></b>                 | Fired when specified by the NumberFormat property        |
| <b><u>HeadClick</u></b>                  | Occurs when a column header has been clicked             |
| <b><u>KeyDown</u></b>                    | Occurs when a key is pressed                             |
| <b><u>KeyPress</u></b>                   | Occurs when an ANSI key is pressed and released          |
| <b><u>KeyUp</u></b>                      | Occurs when a key is released                            |
| <b><u>LeftColChange</u></b>              | Fired when the LeftCol property changes                  |
| <b><u>MouseDown</u></b>                  | Occurs when a mouse button is pressed                    |
| <b><u>MouseMove</u></b>                  | Occurs when the mouse moves                              |
| <b><u>MouseUp</u></b>                    | Occurs when a mouse button is released                   |
| <b><u>OnAddNew</u></b>                   | Fired when a user action causes an AddNew operation      |
| <b><u>Paint</u></b>                      | Fired when a control repaints itself                     |
| <b><u>PostEvent</u></b>                  | Occurs after a PostMsg method is called                  |
| <b><u>RowChange</u></b>                  | Occurs when a different row becomes current              |
| <b><u>RowColChange</u></b>               | Occurs when a different cell becomes current             |
| <b><u>RowResize</u></b>                  | Occurs when rows are resized                             |
| <b><u>Scroll</u></b>                     | Occurs when a control is scrolled using the scroll bars  |
| <b><u>SelChange</u></b>                  | Occurs when the current selected cell range changes      |
| <b><u>SplitChange</u></b>                | Occurs when a different split becomes current            |
| <b><u>UnboundAddData</u></b>             | Fired when a new row is added to the unbound dataset     |
| <b><u>UnboundColumnFetch</u></b>         | Fetches unbound column data when a control is bound      |
| <b><u>UnboundDeleteRow</u></b>           | Fired when an unbound row needs to be deleted            |
| <b><u>UnboundFindData</u></b>            | Fired when the dropdown needs to find a row              |
| <b><u>UnboundGetRelativeBookmark</u></b> | Fired when the control needs a relative bookmark         |

**UnboundReadData**  
**UnboundReadDataEx**  
**UnboundWriteData**  
**ValueItemError**

Fired when the control requires unbound data for display  
Fired when the control requires unbound data for display  
Fired when an unbound row needs to be modified  
Fired when invalid data is typed into a ValueItems column

## TDBGrid Control Events

|                                          |                                                           |
|------------------------------------------|-----------------------------------------------------------|
| <b><u>AfterColEdit</u></b>               | Fired after column data is edited                         |
| <b><u>AfterColUpdate</u></b>             | Occurs after data moves from cell to the grid buffer      |
| <b><u>AfterDelete</u></b>                | Occurs after record deletion from grid                    |
| <b><u>AfterInsert</u></b>                | Occurs after record insertion in grid                     |
| <b><u>AfterUpdate</u></b>                | Occurs after record changes are written to the database   |
| <b><u>BeforeColEdit</u></b>              | Fired upon an attempt to edit column data                 |
| <b><u>BeforeColUpdate</u></b>            | Occurs before data moves from cell to the grid buffer     |
| <b><u>BeforeDelete</u></b>               | Occurs before record deletion from grid                   |
| <b><u>BeforeInsert</u></b>               | Occurs before record insertion in grid                    |
| <b><u>BeforeUpdate</u></b>               | Occurs before record changes are written to the database  |
| <b><u>ButtonClick</u></b>                | Occurs when a column button has been clicked              |
| <b><u>Change</u></b>                     | Occurs when the grid contents have changed                |
| <b><u>ClassicAdd</u></b>                 | Fired when a new row is added to the unbound dataset      |
| <b><u>ClassicDelete</u></b>              | Fired when an unbound row needs to be deleted             |
| <b><u>ClassicRead</u></b>                | Fired when the grid requires unbound data for display     |
| <b><u>ClassicWrite</u></b>               | Fired when an unbound row needs to be modified            |
| <b><u>Click</u></b>                      | Fired when a mouse click occurs                           |
| <b><u>ColEdit</u></b>                    | Fired when column data is edited                          |
| <b><u>ColMove</u></b>                    | Occurs before grid repainting when columns are moved      |
| <b><u>ColResize</u></b>                  | Occurs before grid repainting when a column is resized    |
| <b><u>ComboSelect</u></b>                | Fired when the user selects a ValueItems combo entry      |
| <b><u>DblClick</u></b>                   | Fired when a mouse double click occurs                    |
| <b><u>DragCell</u></b>                   | Fired when a drag operation is initiated                  |
| <b><u>Error</u></b>                      | Occurs when an associated action fails                    |
| <b><u>FetchCellStyle</u></b>             | Fired when the FetchStyle property is True for a column   |
| <b><u>FetchCellTips</u></b>              | Fired when the CellTips property is set to True           |
| <b><u>FetchRowStyle</u></b>              | Fired when the FetchRowStyle property is set to True      |
| <b><u>FirstRowChange</u></b>             | Fired when the FirstRow property changes                  |
| <b><u>FormatText</u></b>                 | Fired when specified by the NumberFormat property         |
| <b><u>HeadClick</u></b>                  | Occurs when a column header has been clicked              |
| <b><u>KeyDown</u></b>                    | Occurs when a key is pressed                              |
| <b><u>KeyPress</u></b>                   | Occurs when an ANSI key is pressed and released           |
| <b><u>KeyUp</u></b>                      | Occurs when a key is released                             |
| <b><u>LeftColChange</u></b>              | Fired when the LeftCol property changes                   |
| <b><u>MouseDown</u></b>                  | Occurs when a mouse button is pressed                     |
| <b><u>MouseMove</u></b>                  | Occurs when the mouse moves                               |
| <b><u>MouseUp</u></b>                    | Occurs when a mouse button is released                    |
| <b><u>OnAddNew</u></b>                   | Fired when a user action causes an AddNew operation       |
| <b><u>Paint</u></b>                      | Fired when the grid repaints itself                       |
| <b><u>PostEvent</u></b>                  | Occurs after a PostMsg method is called                   |
| <b><u>RowColChange</u></b>               | Occurs when a different cell becomes current              |
| <b><u>RowResize</u></b>                  | Occurs when rows are resized                              |
| <b><u>Scroll</u></b>                     | Occurs when the grid is scrolled using the scroll bars    |
| <b><u>SelChange</u></b>                  | Occurs when the current selected cell range changes       |
| <b><u>SplitChange</u></b>                | Occurs when a different split becomes current             |
| <b><u>UnboundAddData</u></b>             | Fired when a new row is added to the unbound dataset      |
| <b><u>UnboundColumnFetch</u></b>         | Fetches unbound column data when the grid is bound        |
| <b><u>UnboundDeleteRow</u></b>           | Fired when an unbound row needs to be deleted             |
| <b><u>UnboundGetRelativeBookmark</u></b> | Fired when the grid needs a relative bookmark             |
| <b><u>UnboundReadData</u></b>            | Fired when the grid requires unbound data for display     |
| <b><u>UnboundReadDataEx</u></b>          | Fired when the grid requires unbound data for display     |
| <b><u>UnboundWriteData</u></b>           | Fired when an unbound row needs to be modified            |
| <b><u>ValueItemError</u></b>             | Fired when invalid data is typed into a ValueItems column |



## TBDDropDown Control Events

|                                          |                                                           |
|------------------------------------------|-----------------------------------------------------------|
| <b><u>ClassicRead</u></b>                | Fired when the dropdown needs unbound data for display    |
| <b><u>Click</u></b>                      | Fired when a mouse click occurs                           |
| <b><u>ColMove</u></b>                    | Occurs before repainting when columns are moved           |
| <b><u>ColResize</u></b>                  | Occurs before repainting when a column is resized         |
| <b><u>DbClick</u></b>                    | Fired when a mouse double click occurs                    |
| <b><u>DragCell</u></b>                   | Fired when a drag operation is initiated                  |
| <b><u>DropDownClose</u></b>              | Fired when the dropdown is closed                         |
| <b><u>DropDownOpen</u></b>               | Fired when the dropdown is displayed                      |
| <b><u>Error</u></b>                      | Occurs when an associated action fails                    |
| <b><u>FetchCellStyle</u></b>             | Fired when the FetchStyle property is True for a column   |
| <b><u>FirstRowChange</u></b>             | Fired when the FirstRow property changes                  |
| <b><u>FormatText</u></b>                 | Fired when specified by the NumberFormat property         |
| <b><u>HeadClick</u></b>                  | Occurs when a column header has been clicked              |
| <b><u>KeyDown</u></b>                    | Occurs when a key is pressed                              |
| <b><u>KeyPress</u></b>                   | Occurs when an ANSI key is pressed and released           |
| <b><u>KeyUp</u></b>                      | Occurs when a key is released                             |
| <b><u>LeftColChange</u></b>              | Fired when the LeftCol property changes                   |
| <b><u>MouseDown</u></b>                  | Occurs when a mouse button is pressed                     |
| <b><u>MouseMove</u></b>                  | Occurs when the mouse moves                               |
| <b><u>MouseUp</u></b>                    | Occurs when a mouse button is released                    |
| <b><u>Paint</u></b>                      | Fired when the dropdown repaints itself                   |
| <b><u>PostEvent</u></b>                  | Occurs after a PostMsg method is called                   |
| <b><u>RowChange</u></b>                  | Occurs when a different row becomes current               |
| <b><u>RowResize</u></b>                  | Occurs when rows are resized                              |
| <b><u>Scroll</u></b>                     | Occurs when the dropdown is scrolled with the scroll bars |
| <b><u>SelChange</u></b>                  | Occurs when the current selected cell range changes       |
| <b><u>UnboundColumnFetch</u></b>         | Fetches unbound column data when the control is bound     |
| <b><u>UnboundFindData</u></b>            | Fired when the dropdown needs to find a row               |
| <b><u>UnboundGetRelativeBookmark</u></b> | Fired when the dropdown needs a relative bookmark         |
| <b><u>UnboundReadData</u></b>            | Fired when the dropdown needs unbound data for display    |
| <b><u>UnboundReadDataEx</u></b>          | Fired when the dropdown needs unbound data for display    |
| <b><u>ValueItemError</u></b>             | Fired when invalid data is typed into a ValueItems column |

## AfterColEdit Event

**Syntax** TDBGrid\_ **AfterColEdit** (ByVal ColIndex As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column that was edited.

**Description** The **AfterColEdit** event occurs after editing is completed in a grid cell.

When the user completes editing within a grid cell, as when tabbing to another column in the same row, pressing the ENTER key, or clicking on another cell, the **BeforeColUpdate** and **AfterColUpdate** events are executed, and data from the cell is moved to the grid's copy buffer. The **AfterColEdit** event immediately follows the **AfterColUpdate** event.

When editing is completed in a grid cell, this event is always triggered, even if no changes were made to the cell or the **BeforeColUpdate** event was canceled.

The **AfterColEdit** event will not be fired if the **BeforeColEdit** event is canceled.

## AfterColUpdate Event

**Syntax** TDBGrid\_ **AfterColUpdate** (ByVal ColIndex As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column that was updated.

**Description** The **AfterColUpdate** event occurs after data is moved from a cell to the grid's internal copy buffer.

When the user completes editing within a grid cell, as when tabbing to another column in the same row, pressing the ENTER key, or clicking on another cell, the **BeforeColUpdate** event is executed, and unless canceled, data from the cell is moved to the grid's copy buffer. Once moved, the **AfterColUpdate** event is executed.

The **AfterColUpdate** event occurs after the **BeforeColUpdate** event, and only if the Cancel argument in the **BeforeColUpdate** event is not set to True.

Once the **AfterColUpdate** event procedure begins, the cell data has already been moved to the grid's copy buffer and can't be canceled, but other updates can occur before the data is committed to the **Recordset**.

## AfterDelete Event

**Syntax** TDBGrid\_ **AfterDelete** ( )

Event applies to **TDBGrid** control.

**Arguments** None

**Description** The **AfterDelete** event occurs after the user deletes a selected record from the grid.

When the user selects a record selector in the grid and presses DEL or CTRL+X, the **BeforeDelete** event is executed, and unless canceled, the row is deleted. Once the row is deleted, the **AfterDelete** event is executed.

While the **AfterDelete** event procedure is executing, the bookmark of the deleted row is available in the collection provided by the **SelBookmarks** property.

The **AfterDelete** event cannot be canceled. It is fired in both bound and unbound modes.



## AfterInsert Event

**Syntax** TDBGrid\_ **AfterInsert** ( )

Event applies to **TDBGrid** control.

**Arguments** None

**Description** The **AfterInsert** event occurs after the user inserts a new record into the grid. It can be used to update other tables or to perform post-update cleanup of other controls.

When the user selects the AddNew row (the last row in the grid) and enters a character in one of the cells, the **BeforeInsert** event is executed, and unless canceled, the row is scrolled up one line and its record selector changes to show that it has been modified. However, a new row has **not** yet been added to the database.

Once the user commits the new row by moving to another row within the grid, the **BeforeUpdate** event is triggered, followed by the **AfterUpdate** and **AfterInsert** events. If the **BeforeUpdate** event is canceled, then the **AfterUpdate** and **AfterInsert** events will not be fired.

When the **AfterInsert** event is triggered, the record has already been added to the database. The **Bookmark** property can be used to access the new record.

The **AfterInsert** event cannot be canceled. It is fired in both bound and unbound modes.

## AfterUpdate Event

**Syntax** TDBGrid\_AfterUpdate ( )

Event applies to TDBGrid control.

**Arguments** None

**Description** The AfterUpdate event occurs after changed data has been written to the database from the grid.

When the user moves to another row, or the Update method of the grid or **Recordset** object is executed, data is moved from the grid's copy buffer to the Data control's copy buffer and written to the database. Once the write operation is complete, the AfterUpdate event is triggered.

The Bookmark property can be used to access the updated record.

The AfterUpdate event cannot be canceled. It is fired in both bound and unbound modes.

## BeforeColEdit Event

- Syntax** TDBGrid\_ **BeforeColEdit** (ByVal ColIndex As Integer, ByVal KeyAscii As Integer, Cancel As Integer)
- Event applies to **TDBGrid** control.
- Arguments** ColIndex is an integer that identifies the column about to be edited.
- KeyAscii is an integer representing the ANSI key code of the character typed by the user to initiate editing, or 0 if the user initiated editing by clicking the mouse. KeyAscii is passed by value, not by reference; you cannot change its value to initiate editing with a different character.
- Cancel is an integer that may be set to True to prevent the user from editing the cell.
- Description** The **BeforeColEdit** event occurs just before the user enters edit mode by typing a character. If a floating editor marquee is not in use, this event also occurs when the user clicks the current cell or double clicks another cell.
- If your event procedure sets the Cancel argument to True, the cell will not enter edit mode. Otherwise, the **ColEdit** event is fired immediately, followed by the **Change** and **KeyUp** events for the KeyAscii argument, if non-zero.
- Use this event to control the editability of cells on a per-cell basis, or to translate the initial keystroke into a default value.
- Note** The KeyAscii argument can only be 0 if a floating editor marquee is not in use.

## BeforeColUpdate Event

**Syntax** TDBGrid\_ **BeforeColUpdate** (ByVal ColIndex As Integer, OldValue As Variant, Cancel As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column about to be updated.

OldValue is a variant containing the original data.

Cancel is an integer that may be set to True to prevent the update from occurring.

**Description** The **BeforeColUpdate** event occurs after editing is completed in a cell, but before data is moved from the cell to the grid's internal copy buffer.

The data specified by the OldValue argument moves from the cell to the grid's copy buffer when the user completes editing within a cell, as when tabbing to another column in the same row, pressing the ENTER key, or clicking on another cell. Before the data has been moved from the cell into the grid's copy buffer, the **BeforeColUpdate** event is triggered. This event gives your application an opportunity to check the individual grid cells before they are committed to the grid's copy buffer.

If your event procedure sets the Cancel argument to True, the previous value is restored in the cell, the grid retains focus, and the **AfterColUpdate** event is not triggered. You can also change the current cell text by setting OldValue to the value you want to display (other than the previous value).

To restore OldValue in the cell and permit the user to move focus off of the cell, set Cancel to False and set the cell to OldValue as follows:

```
Cancel = False
TDBGrid1.Columns(ColIndex).Value = OldValue
```

Setting the Cancel argument to True prevents the user from moving focus away from the control until the application determines that the data can be safely moved back to the grid's copy buffer.

## BeforeDelete Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>BeforeDelete</b> (Cancel As Integer)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Arguments</b>   | <u>Cancel</u> is an integer that may be set to True to prevent the deletion from occurring.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>Description</b> | <p>The <b>BeforeDelete</b> event occurs before a selected record is deleted from the grid.</p> <p>When the user selects a record selector in the grid and presses DEL or CTRL+X, the <b>BeforeDelete</b> event is triggered to give your application a chance to override the user's action.</p> <p>If your event procedure sets the <u>Cancel</u> argument to True, the row is not deleted. Otherwise, the grid deletes the row and triggers the <b>AfterDelete</b> event.</p> <p>The bookmark of the row selected for deletion is available in the collection provided by the <b>SelBookmarks</b> property.</p> |
| <b>Note</b>        | If more than one row is selected, the error message <i>Cannot delete multiple rows</i> is displayed, and the <b>BeforeDelete</b> event will not be fired.                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

## BeforeInsert Event

**Syntax** TDBGrid\_ **BeforeInsert** (Cancel As Integer)

Event applies to **TDBGrid** control.

**Arguments** Cancel is an integer that may be set to True to prevent the insertion from occurring.

**Description** The **BeforeInsert** event occurs before a new record is added from the grid.

When the user selects the AddNew row (the last row in the grid) and enters a character in one of the cells, the **BeforeInsert** event is triggered to give your application a chance to override the user's action.

If your event procedure sets the Cancel argument to True, no insertion takes place and the cell is cleared. Otherwise, the AddNew row is scrolled up one line and its record selector changes to show that it has been modified. However, a new row has **not** yet been added to the database.

Once the user commits the new row by moving to another row within the grid, the **BeforeUpdate** event is triggered, followed by the **AfterUpdate** and **AfterInsert** events. If the **BeforeUpdate** event is canceled, then the **AfterUpdate** and **AfterInsert** events will not be fired.

## BeforeUpdate Event

**Syntax** TDBGrid\_ **BeforeUpdate** (Cancel As Integer)

Event applies to **TDBGrid** control.

**Arguments** Cancel is an integer that may be set to True to prevent the update from occurring.

**Description** The **BeforeUpdate** event occurs before data is moved from the grid's internal copy buffer to the Data control's copy buffer and then written to the database.

When the user moves to another row, or the **Update** method of the grid or **Recordset** object is executed, data is moved from the grid's copy buffer to the Data control's copy buffer and then written to the database.

Just before the data is moved from the grid's copy buffer back into the Data control's copy buffer, the **BeforeUpdate** event is triggered. Unless the copy operation is canceled, the **AfterUpdate** event is triggered after the data has been moved back into the Data control's copy buffer and written to the database.

The **Bookmark** property can be used to access the updated record.

If your event procedure sets the Cancel argument to True, focus remains on the control, the **AfterUpdate** event is not triggered, and the record is not saved to the database.

You can use this event to validate data in a record before permitting the user to commit the change to the Data control's copy buffer. Setting the Cancel argument to True prevents the user from moving focus to another control until the application determines whether the data can be safely moved back to the Data control's copy buffer.

## ButtonClick Event

**Syntax** TDBGrid\_ **ButtonClick** (ByVal ColIndex As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column whose button was clicked.

**Description** This event is fired when the current cell's built-in button is clicked. The built-in button is enabled for a column when its **Button** property is set to True, its **DropDown** property is set to the name of a valid **TDBDropDown** control, or the **Presentation** property of its associated **ValueItems** collection is set to one of the combo box options.

Typically, you enable the column button when you want to drop down a Visual Basic control (such as the built-in combo box, a bound list box, or even another True DBGrid control) for editing or data entry. When the button in the current cell is clicked, the **ButtonClick** event will be fired. You can then write code to drop down the desired control from the cell.



## Change Event

**Syntax** TDBGrid\_Change ( )

Event applies to **TDBGrid** control.

**Arguments** None

**Description** The **Change** event occurs when the user changes the text within a grid cell.

This event is only fired when the current cell is being edited and the user enters or deletes a character, pastes text from the clipboard, or cuts text to the clipboard. It does not apply to database changes.

## ClassicAdd Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>ClassicAdd</b> (NewRowBookmark As Variant, ByVal Col As Integer, Value As Variant)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Arguments</b>   | <u>NewRowBookmark</u> is a variant that must be set to a unique bookmark for subsequent references to the newly added row.<br><u>Col</u> is an integer that identifies the column to receive the new value.<br><u>Value</u> is a variant used to transfer the new data from the grid to the unbound data source.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Description</b> | The <b>ClassicAdd</b> event is fired when the user adds a new row of data to an application mode grid (one with its <b>DataMode</b> property set to 3 - Application). This event alerts your application that it must add a new row of data to its unbound dataset.<br><br>This event adds data one cell at a time, so it may fire multiple times in order to add data for all the columns in a row. The <u>Col</u> argument contains the column index corresponding to the newly entered cell data. If the user has not added data to a column, the <b>ClassicAdd</b> event will not fire for that column index. If you want to add a default value to a column, you can do that in the grid's <b>BeforeUpdate</b> event.<br><br>The <u>Value</u> argument contains the cell data entered by the user.<br><br><u>NewRowBookmark</u> <b>must</b> be set to the bookmark of the newly added row, or else the user will not be able to move to another row without first canceling the new row with the ESC key. If the <u>NewRowBookmark</u> is set to different values when the event is called with different column indexes, the grid will use the last non Null value returned by the user.<br><br>This event will not be fired if the <b>AllowAddNew</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowAddNew</b> is never set to True. |
| <b>Note</b>        | If the add operation fails in the underlying data source, then you should set <u>NewRowBookmark</u> to Null.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

## ClassicDelete Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>ClassicDelete</b> (Bookmark As Variant)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Arguments</b>   | <u>Bookmark</u> is a variant that uniquely identifies the row to be deleted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>Description</b> | <p>The <b>ClassicDelete</b> event is fired when the user deletes an existing row within an unbound grid (one with its <b>DataMode</b> property set to 3). This event alerts your application that it must delete the row specified by the <u>Bookmark</u> argument from its unbound dataset.</p> <p>The <u>Bookmark</u> argument contains a bookmark supplied by your application in either the <b>ClassicRead</b> or <b>ClassicAdd</b> event.</p> <p>This event will not be fired if the <b>AllowDelete</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowDelete</b> is never set to True.</p> |
| <b>Note</b>        | If the delete operation fails in the underlying data source, then you should set <u>Bookmark</u> to Null to inform the grid of the failure. This will cause the grid's <b>Error</b> event to fire. The row specified by <u>Bookmark</u> will remain selected.                                                                                                                                                                                                                                                                                                                                                                                                 |

## ClassicRead Event

**Syntax**            object **ClassicRead** (Bookmark As Variant, ByVal Col As Integer, Value As Variant)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        Bookmark is a variant that identifies the row being requested.

Col is an integer that identifies the column being requested.

Value is a variant used to transfer unbound column data to the grid.

**Description**     The **ClassicRead** event is fired when an unbound grid (one with its **DataMode** property set to 3) needs to display the value of a cell as specified by the Bookmark and Col arguments.

To return an unbound value to the grid, simply set the Value argument to the desired result. If you do not assign a value, the cell will remain blank.

## ClassicWrite Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>ClassicWrite</b> (Bookmark As Variant, ByVal Col As Integer, Value As Variant)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Arguments</b>   | <u>Bookmark</u> is a variant that uniquely identifies the row to be updated.<br><u>Col</u> is an integer that identifies the column to be updated.<br><u>Value</u> is a variant used to transfer data from the grid to the unbound data source.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b> | <p>The <b>ClassicWrite</b> event is fired when the user modifies an existing row within an application mode grid (one with its <b>DataMode</b> property set to 3 - Application) and attempts to commit the changes by moving to a different row or by calling the <b>Update</b> method. This event alerts your application that it must update the cell specified by the <u>Bookmark</u> and <u>Col</u> arguments within its unbound dataset.</p> <p>The <u>Bookmark</u> argument contains a bookmark supplied by your application in either the <b>ClassicRead</b> or <b>ClassicAdd</b> event.</p> <p>The <u>Value</u> argument contains the cell data entered by the user.</p> <p>This event will not be fired if the <b>AllowUpdate</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowUpdate</b> is never set to True.</p> |
| <b>Note</b>        | <p>If the update operation fails in the underlying data source, then you should set the <u>Bookmark</u> argument to Null.</p> <p>You can force the <b>ClassicWrite</b> event to occur in code by calling the <b>Update</b> method. This method is particularly valuable when the unbound dataset contains a single row and <b>AllowAddNew</b> is False, since there is no way for the user to trigger the update by moving to another row in this case.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                 |

## Click Event

**Syntax** object\_Click ( )

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** None

**Description** The **Click** event occurs when the user presses then releases a mouse button over the grid. Clicking a grid also generates **MouseDown** and **MouseUp** events in addition to the **Click** event. The order of events for both the **TDBGrid** and **TDBDropDown** controls is **MouseDown**, **MouseUp**, and **Click**.

When the user clicks a noncurrent row, the **Click** event fires *before* the grid attempts to reposition to the row that was clicked. If the attempt succeeds, the grid then fires the **RowColChange** (or **RowChange**) event. For this reason, you should not use the **Click** event to perform operations that depend upon the current row.

**Note** You can use the **PostMsg** method and **PostEvent** event to defer processing until the row change has completed.

## ColEdit Event

**Syntax** TDBGrid\_**ColEdit** (ByVal ColIndex As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column being edited.

**Description** The **ColEdit** event occurs when a cell first enters edit mode by typing a character. If a floating editor marquee is not in use, this event also occurs when the user clicks the current cell or double clicks another cell.

The **ColEdit** event immediately follows the **BeforeColEdit** event only when the latter is not canceled.

When the user completes editing within a grid cell, as when tabbing to another column in the same row, pressing the ENTER key, or clicking on another cell, the **BeforeColUpdate** and **AfterColUpdate** events are executed if the data has been changed. The **AfterColEdit** event is then fired to indicate that editing is completed.

## ColMove Event

- Syntax**            object\_ **ColMove** (ByVal Position As Integer, Cancel As Integer)
- Event applies to **TDBGrid** and **TDBDropDown** controls.
- Arguments**        Position is an integer that specifies the target location of the selected columns.  
Cancel is an integer that may be set to True to prohibit movement.
- Description**      The **ColMove** event occurs when the user has finished moving the selected columns. Your event procedure can either accept the movement or cancel it altogether.
- The Position argument ranges from 0, which denotes the left edge, to the total number of columns, which denotes the right edge.
- To determine which columns are being moved, examine the **SelStartCol** and **SelEndCol** properties.
- If you set the Cancel argument to True, no movement occurs. Selected columns always remain selected, regardless of the Cancel setting.
- Note**                This event is only fired when the user moves the selected columns to a **new** location.



## ColResize Event

**Syntax**            object\_ **ColResize** (ByVal ColIndex As Integer, Cancel As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        ColIndex is an integer that identifies the column that was resized.

Cancel is an integer that may be set to True to undo resizing.

**Description**      The **ColResize** event occurs after the user has finished resizing a column, but before columns to the right are repositioned. Your event procedure can accept the change, alter the degree of change, or cancel the change completely.

If you set the Cancel argument to True, the previous column width is restored and no repainting occurs. To alter the degree of change, set the **Width** property of the **Column** object specified by the ColIndex argument to the desired value.

It is not necessary to execute the **Refresh** method within this event procedure. Doing so causes the grid to be repainted even if the Cancel argument is True.

## ComboSelect Event

**Syntax** TDBGrid\_ **ComboSelect** (ByVal ColIndex As Integer)

Event applies to **TDBGrid** control.

**Arguments** ColIndex is an integer that identifies the column being edited.

**Description** The **ComboSelect** event is triggered when the user selects (by clicking) a built-in combo box item. The built-in combo box is enabled for a column when the **Presentation** property of its associated **ValueItems** collection is set to one of the combo box options.

This event is useful for determining the contents of the cell before the user exits editing mode. By setting the **EditActive** property to False within this event procedure, you can force the grid to exit editing mode without allowing the user a chance to edit his selection.

## DbiClick Event

**Syntax**            object\_DbiClick ( )

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        None

**Description**      The **DbiClick** event occurs when the user presses then releases a mouse button twice in rapid succession over the grid.

Double clicking a grid also generates **MouseDown**, **MouseUp**, and **Click** events in addition to the **DbiClick** event. The order of events for the **TDBGrid** control is **MouseDown**, **MouseUp**, **Click**, **DbiClick**, and **MouseUp**.

**Note**                When the **MarqueeStyle** property of a **TDBGrid** control is set to the default value of 6 - Floating Editor, the **DbiClick** event will not fire when the user double-clicks a noncurrent cell within the grid. This is because the first click is used by the floating editor to begin editing, placing the cell into edit mode at the character on which the click occurred. Double-clicking the current cell of the grid fires the **DbiClick** event normally, however.

## DragCell Event

**Syntax** TDBGrid\_**DragCell** (ByVal SplitIndex As Integer, RowBookmark As Variant, ByVal ColIndex As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** SplitIndex is an integer that identifies the split containing the cell being dragged.

RowBookmark is a variant that identifies the row containing the cell being dragged.

ColIndex is an integer that identifies the column containing the cell being dragged.

**Description** The **DragCell** event is triggered when the user presses the left mouse button and starts dragging the mouse. This event is used to notify the programmer when the user wants to begin a drag-and-drop operation. Using the SplitIndex, RowBookmark, and Col arguments, you can determine the exact location of the mouse pointer at the start of the drag-and-drop operation.

You can initiate dragging in this event automatically by invoking the grid's **Drag** method.

## DropDownClose Event

**Syntax** TDBDropDown\_ **DropDownClose** ( )

Event applies to **TDBDropDown** control.

**Arguments** None

**Description** The **DropDownClose** event is triggered when a **TDBDropDown** control is closed, which occurs when:

- The user selects an item from the dropdown.
- The user clicks the current cell's built-in button.
- The user presses ALT+DOWN ARROW.
- The dropdown loses focus.

The built-in button is enabled for a column when its **DropDown** property is set to the name of a valid **TDBDropDown** control.

## DropDownOpen Event

**Syntax** TDBDropDown\_**DropDownOpen** ( )

Event applies to **TDBDropDown** control.

**Arguments** None

**Description** The **DropDownOpen** event is triggered when a **TDBDropDown** control is opened, which occurs when:

- The user clicks the current cell's built-in button.
- The user presses ALT+DOWN ARROW.

The built-in button is enabled for a column when its **DropDown** property is set to the name of a valid **TDBDropDown** control.

## Error Event

- Syntax**      object\_ **Error** (DataError As Integer, Response As Integer)
- Event applies to **TDBGrid** and **TDBDropDown** controls.
- Arguments**    DataError is an integer that identifies the error that occurred.
- Response is an integer that may be set to 0 to suppress error message display.
- Description**    The **Error** event occurs only as the result of a data access error that takes place when no Visual Basic code is being executed.
- Even if your application handles run time errors in code, errors can still occur when none of your code is executing, as when the user clicks a Data control button or changes the current record by interacting with a bound control. If a data access error results from such an action, the **Error** event is fired.
- If you set the Response argument to 0, no error message will be displayed.
- If the Response argument retains its default value of 1, or if you do not code an event procedure for the **Error** event, the message associated with the error will be displayed.
- Note**            Use the ErrorText property to retrieve the error string that will be displayed.

## FetchCellStyle Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <p>TDBGrid_ <b>FetchCellStyle</b> (ByVal Condition As Integer, ByVal Split As Integer, Bookmark As Variant, ByVal Col As Integer, ByVal CellStyle As TrueDBGrid50.StyleDisp)</p> <p>TDBDropDown_ <b>FetchCellStyle</b> (ByVal Condition As Integer, Bookmark As Variant, ByVal Col As Integer, ByVal CellStyle As TrueDBGrid50.StyleDisp)</p> <p>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.</p>                                                                                                                                                                                                      |
| <b>Arguments</b>   | <p><u>Condition</u> is the sum of one or more CellStyleConstants describing the disposition of the cell being displayed.</p> <p><u>Split</u> is an integer that identifies the split containing the cell being displayed. This argument is omitted for <b>TDBDropDown</b> controls.</p> <p><u>Bookmark</u> is a variant that identifies the row containing the cell being displayed.</p> <p><u>Col</u> is an integer that identifies the column containing the cell being displayed.</p> <p><u>CellStyle</u> is a <b>Style</b> object used to override the font and color characteristics of the cell being displayed.</p> |
| <b>Description</b> | <p>The <b>FetchCellStyle</b> event occurs when the grid is about to display cell data in a column whose <b>FetchStyle</b> property is set to True. By setting one or more properties of the <b>Style</b> object passed in the <u>CellStyle</u> parameter, your application can change the appearance of individual cells.</p>                                                                                                                                                                                                                                                                                              |



## FetchCellTips Event

**Syntax** TDBGrid\_ **FetchCellTips** (ByVal SplitIndex As Integer, ByVal ColIndex As Integer, ByVal RowIndex As Long, CellTip As String, ByVal FullyDisplayed As Boolean, ByVal TipStyle As TrueDBGrid50.StyleDisp)

Event applies to **TDBGrid** control.

**Arguments** SplitIndex is the zero-based index of the split the cursor is over.

ColIndex is an integer that identifies the column the cursor is over. This is either a zero-based column index or a (negative) CellTipConstants value.

RowIndex is an integer that identifies the row the cursor is over. This is either a zero-based row index or a (negative) CellTipConstants value.

CellTip contains the text to be displayed in the pop-up text box.

FullyDisplayed is a boolean that is True if CellTip will fit entirely within the boundaries of the cell.

TipStyle is a **Style** object used to override the font and color characteristics of the cell tip text.

**Description** If the **CellTips** property is not set to 0 - None, the **FetchCellTips** event will be fired whenever the grid has focus and the cursor is idle for a small amount of time (defined by the **CellTipsDelay** property) over the grid cells area, the record selectors, the column header, the split header, or the grid caption. This event will not fire if the cursor is over the scroll bars.

If the cursor is over a grid cell, CellTip contains the contents of the cell as text. By default, the grid will display up to 256 characters of the cell contents in a pop-up text box, enabling the user to peruse the contents of a cell even if it is not big enough to be displayed in its entirety. Instead of displaying the cell text, you can also modify CellTip to display your own message. However, if you set CellTip to Null or an empty string, the text box will not be displayed.

If the cursor is not over a grid column, ColIndex will be negative and equal to one of the following CellTipConstants, depending upon the cursor position:

dbgOnRecordSelector      Cursor is over a record selector

dbgOnEmptyColumn      Cursor is over the blank area to the right of the last column

If the cursor is not over a data row, RowIndex will be negative and equal to one of the following CellTipConstants, depending upon the cursor position:

dbgOnColumnHeader      Cursor is over a column header

dbgOnSplitHeader      Cursor is over a split header

dbgOnEmptyRow      Cursor is over the empty rows area (if **EmptyRows** is True) or the blank area (if **EmptyRows** is False)

dbgOnCaption      Cursor is over the grid caption

dbgOnAddNew      Cursor is over the AddNew row

If the cursor is over an empty row (that is, a row below the AddNew row), or is not over the grid cells area, the value of CellTip is Null, and the pop-up text box will not be displayed. If you modify CellTip so that it is no longer Null, the text box will display the changed value.

By setting the properties of the TipStyle object, you can control the background color, text color, and font of the pop-up text box. By default, the TipStyle object uses the system ToolTip colors and the font attributes of the current column.

You can program this event to provide context-sensitive help or tips to your users. For example, if the user points to column header, you can provide a more detailed description of the column. If the user points to a record selector, you can display instructions for selecting multiple records.

You can also provide content-sensitive help to your users using this event. By default, CellTip contains the text of the cell under the cursor. However, you can also determine other cell values if desired. Using the grid's Row property and RowBookmark method, you can retrieve the bookmark of the row under the cursor, then use the CellValue or CellText method to derive other cell values.

## FetchRowStyle Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <p>TDBGrid_ <b>FetchRowStyle</b> (ByVal Split As Integer, Bookmark As Variant, ByVal RowStyle As TrueDBGrid50.StyleDisp)</p> <p>TBDDropDown_ <b>FetchRowStyle</b> (Bookmark As Variant, ByVal RowStyle As TrueDBGrid50.StyleDisp)</p> <p>Event applies to <b>TDBGrid</b> and <b>TBDDropDown</b> controls.</p>                                                                                          |
| <b>Arguments</b>   | <p><u>Split</u> is the zero-based index of the split for which formatting information is being requested. This argument is omitted for <b>TBDDropDown</b> controls.</p> <p><u>Bookmark</u> is a variant that uniquely identifies the row to be displayed.</p> <hr/> <p><u>RowStyle</u> is a <b>Style</b> object used to convey font and color information to the grid.</p>                             |
| <b>Description</b> | <p>The <b>FetchRowStyle</b> event is fired whenever the grid is about to display a row of data, but only if the <b>FetchRowStyle</b> property is True for the grid or one of its splits.</p> <p>Use the <b>FetchRowStyle</b> event to control formatting on a per-row basis, as it is much more efficient than coding the <b>FetchCellStyle</b> event to apply the same formatting to all columns.</p> |
| <b>Note</b>        | <p>A common application of row-based formatting is to present rows in alternating colors to enhance their readability. Although you could use the <b>FetchRowStyle</b> event to achieve this effect, the <b>AlternatingRowStyle</b> property is easier to use, as it requires no coding.</p>                                                                                                           |

## FirstRowChange Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>FirstRowChange</b> (ByVal SplitIndex As Integer)<br>TDBDropDown_ <b>FirstRowChange</b> ( )<br>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Arguments</b>   | <u>SplitIndex</u> is the zero-based index of the split in which the row change occurred. This argument is omitted for <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Description</b> | <p>The <b>FirstRowChange</b> event occurs when the first displayed row of a control or split is changed. This event is triggered under several circumstances:</p> <ul style="list-style-type: none"><li>• When the grid is first displayed.</li><li>• When the user scrolls through the grid with the vertical scroll bar or navigation keys.</li><li>• When data in the grid is changed in a way that implicitly affects the first row, such as when the first displayed record is deleted.</li><li>• When the <b>FirstRow</b> property is changed in code to a different value.</li></ul> <p>When multiple cell change events are sent, the order will be <b>SplitChange</b>, <b>FirstRowChange</b>, <b>LeftColChange</b>, and <b>RowColChange</b>. None of these events will be sent until any modified data has been validated with the <b>BeforeColUpdate</b> event.</p> |

## FormatText Event

**Syntax**      object\_ **FormatText** (ColIndex As Integer, Value As Variant)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**    ColIndex is an integer that identifies the column being displayed.

Value is a variant containing the underlying data value.

**Description**    The **FormatText** event occurs when the grid is about to display cell data in a column whose **NumberFormat** property is set to the string FormatText Event.

The Value argument contains the underlying data value and also serves as a placeholder for the formatted data to be displayed.

This event allows you to provide your own text formatting for cases where Visual Basic's intrinsic formatting is either unavailable or does not suit your needs.

## HeadClick Event

- Syntax**            object\_HeadClick (ColIndex As Integer)  
Event applies to **TDBGrid** and **TDBDropDown** controls.
- Arguments**        ColIndex is an integer that identifies the column header that was clicked.
- Description**        The **HeadClick** event occurs when the user clicks on the header for a particular grid column.  
One possible action for this event is to re-sort the **Recordset** object based on the selected column.

## KeyDown, KeyUp Events

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <p>object_ <b>KeyDown</b> (KeyCode As Integer, Shift As Integer)</p> <p>object_ <b>KeyUp</b> (KeyCode As Integer, Shift As Integer)</p> <p>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Arguments</b>   | <p><b>KeyCode</b> is an integer or constant representing a Windows key code. For example, the Visual Basic object library provides the constants <code>vbKeyF1</code> (the F1 key) and <code>vbKeyHome</code> (the HOME key).</p> <p><b>Shift</b> is an integer that corresponds to the state of the SHIFT, CTRL, and ALT keys at the time of the event. The <b>Shift</b> argument is a bit field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both CTRL and ALT are pressed, the value of <b>Shift</b> is 6.</p> |
| <b>Description</b> | <p>The <b>KeyDown</b> (<b>KeyUp</b>) event occurs when the user presses (releases) a key. Although these events are fired for most keystrokes, they are most often used for:</p> <ul style="list-style-type: none"><li>• Extended character keys such as function keys.</li><li>• Navigation keys.</li><li>• Combinations of keys with standard keyboard modifiers.</li><li>• Distinguishing between the numeric keypad and regular number keys.</li></ul>                                                                                                                                                                                                                                                                                                                        |
| <b>Note</b>        | <p>The tab key does not generate <b>KeyDown</b> or <b>KeyUp</b> events.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## KeyPress Event

**Syntax**            object\_ **KeyPress** (KeyAscii As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        KeyAscii is an integer representing an ANSI key code. KeyAscii is passed by reference; changing it sends a different character to the grid. Changing KeyAscii to 0 cancels the keystroke so the grid receives no character.

**Description**      The **KeyPress** event occurs when the user presses and releases one of the following kinds of keys:

- A printable keyboard character.
- The CTRL key combined with an alphabetic or special character.
- The ENTER or BACKSPACE key.

Use the **KeyPress** event to test keystrokes for validity or to format characters as they are typed.



## LeftColChange Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>LeftColChange</b> (ByVal SplitIndex As Integer)<br>TDBDropDown_ <b>LeftColChange</b> ( )<br>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Arguments</b>   | <u>SplitIndex</u> is the zero-based index of the split in which the column change occurred. This argument is omitted for <b>TDBDropDown</b> controls.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b> | The <b>LeftColChange</b> event occurs when the first visible column of a grid or split is changed. This event is triggered under several circumstances: <ul style="list-style-type: none"><li>• When the grid is first displayed.</li><li>• When the user scrolls through the grid with the horizontal scroll bar or navigation keys.</li><li>• When the <b>LeftCol</b> property is changed in code to a different value.</li><li>• When the <b>Visible</b> property of the current left column is set to False.</li><li>• When the <b>Width</b> property of the current left column is set to 0.</li><li>• When the user resizes the current left column so that it is no longer visible.</li><li>• When the user moves the current left column or moves another column into its place.</li></ul> When multiple cell change events are sent, the order will be <b>SplitChange</b> , <b>FirstRowChange</b> , <b>LeftColChange</b> , and <b>RowColChange</b> . None of these events will be sent until any modified data has been validated with the <b>BeforeColUpdate</b> event. |

## MouseDown, MouseUp Events

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <p>object_ <b>MouseDown</b> (Button As Integer, Shift As Integer, X As Single, Y As Single)</p> <p>object_ <b>MouseUp</b> (Button As Integer, Shift As Integer, X As Single, Y As Single)</p> <p>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>Arguments</b>   | <p><b>Button</b> is an integer that identifies the button that was pressed or released to cause the event. The <b>Button</b> argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.</p> <p><b>Shift</b> is an integer that corresponds to the state of the SHIFT, CTRL, and ALT keys when the button specified in the <b>Button</b> argument is pressed or released. A bit is set if the key is down. The <b>Shift</b> argument is a bit field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both CTRL and ALT are pressed, the value of <b>Shift</b> is 6.</p> <p><b>X</b> and <b>Y</b> are single-precision numbers that specify the current location of the mouse pointer. They are always expressed in terms of the coordinate system of the grid's container.</p> |
| <b>Description</b> | <p>Use a <b>MouseDown</b> or <b>MouseUp</b> event procedure to specify actions that will occur when a given mouse button is pressed or released. Unlike the <b>Click</b> and <b>DbClick</b> events, <b>MouseDown</b> and <b>MouseUp</b> events enable you to distinguish between the left, right, and middle mouse buttons. You can also write code for mouse/keyboard combinations that use the SHIFT, CTRL, and ALT keyboard modifiers.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

## MouseMove Event

**Syntax**            object\_ **MouseMove** (Button As Integer, Shift As Integer, X As Single, Y As Single)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        Button is an integer that corresponds to the state of the mouse buttons in which a bit is set if the button is down. The Button argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The Button argument indicates the complete 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 pressed.

Shift is an integer that corresponds to the state of the SHIFT, CTRL, and ALT keys. A bit is set if the key is down. The Shift argument is a bit field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both CTRL and ALT are pressed, the value of Shift is 6.

X and Y are single-precision numbers that specify the current location of the mouse pointer. They are always expressed in terms of the coordinate system of the grid's container.

**Description**        The **MouseMove** event is generated continually as the mouse pointer moves across the grid. This event is primarily useful for implementing drag-and-drop behavior on a per-column basis.

## OnAddNew Event

**Syntax** TDBGrid\_ **OnAddNew** ( )  
Event applies to **TDBGrid** control.

**Arguments** None

**Description** The **OnAddNew** event occurs when an AddNew operation has been initiated by either of the following:

- The user modifies a cell within the AddNew row. Typically, this occurs as soon as the user types a character, but may also occur as a result of a built-in radio button or combo box selection.
- The **Value** or **Text** property of a column is set in code when the current cell is within the AddNew row.

This event is fired in both bound and unbound modes. However, it will only be fired if the grid's **AllowAddNew** property is True.

When the **OnAddNew** event is fired, the value of the **AddNewMode** property is 2 - AddNew Pending.

## Paint Event

**Syntax** object\_ **Paint** ( )

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** None

**Description** The **Paint** event is triggered whenever the grid repaints itself (that is, whenever it receives a `WM_PAINT` message). This occurs frequently in the Windows environment and is generally useful only for special circumstances. In this event, programmers familiar with the Windows API may use the grid's **hWnd** property to paint special effects such as lines, bitmaps, and icons in appropriate cells of the grid.

## PostEvent Event

**Syntax** object\_**PostEvent** (ByVal MsgId As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** **MsgId** is an integer that identifies the message posted by the **PostMsg** method.

**Description** The **PostEvent** event is used in conjunction with the **PostMsg** method to postpone operations that are illegal within the grid's events. If the **PostMsg** method is called, the grid will fire the **PostEvent** event with the **MsgId** of the corresponding **PostMsg** invocation after all pending operations are completed. You can then safely perform all desired operations in the **PostEvent** event.

For example, it is not possible to perform the Data control's **Refresh** method within the grid's **AfterUpdate** event because database operations are still pending, and the refresh cannot be tolerated. Instead of performing the refresh in the **AfterUpdate** event, you can call the **PostMsg** method (with a **MsgId** value of 1, for instance). After all pending database operations are completed, the grid will fire the **PostEvent** event. You can then perform the refresh operation safely in this event, after confirming that the **MsgId** argument passed in is 1.

The special case where **MsgId** is zero is used to clear any pending **PostMsg** invocations that have not yet been processed. A **PostEvent** event will fire for this case.

A following code illustrates a typical **PostEvent** handler:

```
Private Sub TDBGrid1_PostEvent(ByVal MsgId As Integer)
 Select Case MsgId
 Case 0
 Exit Sub
 Case 1
 Data1.Refresh
 Case 2
 ' Process other postponed operations
 End Select
End Sub
```

**Note** Take care to avoid recursive situations when using **PostMsg** and **PostEvent**.

## RowChange Event

**Syntax** TDBDropDown\_**RowChange** ( )

Event applies to **TDBDropDown** control.

**Arguments** None

**Description** The **RowChange** event occurs when the current row changes to a different row. This event is triggered under several circumstances:

- When a dropdown is first displayed.
- When the user moves the current row by clicking on another row or using the navigation keys.
- During incremental searching.

The current row position is provided by the **Bookmark** property.

## RowColChange Event

**Syntax** TDBGrid\_**RowColChange** (LastRow As Variant, ByVal LastCol As Integer)

Event applies to **TDBGrid** control.

**Arguments** LastRow is a variant bookmark that identifies the former current row.

LastCol is an integer that identifies the former current column.

**Description** The **RowColChange** event occurs when the current cell changes to a different cell. This event is triggered under several circumstances:

- When the grid is first displayed.
- When the user moves the current cell by clicking another cell or using the navigation keys.
- When data in the grid is changed in a way that implicitly affects the current row, such as when the current row is deleted when it is the last row in the grid.
- When the **Bookmark**, **Row**, **Col**, or **Split** properties are changed in code to a different value.

The current cell position is provided by the **Bookmark** and **Col** properties. The previous cell position is specified by the LastRow and LastCol arguments.

If the user edits data and then moves the current cell position to a new row, the update events for the original row are completed before another cell becomes the current cell.

If a cell change also results in a change to the current split, then the **SplitChange** event will always precede the **RowColChange** event.



## RowResize Event

**Syntax**            object\_**RowResize** (Cancel As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        Cancel is an integer that may be set to True to undo resizing.

**Description**      The **RowResize** event occurs when the user has finished resizing a grid row. Your event procedure can accept the change, alter the degree of change, or cancel the change completely.

The **TDBGrid** control's **RowHeight** property determines the height of all rows in the control.

If you set the Cancel argument to True, the previous row height is restored and no repainting occurs. To alter the degree of change, set the **RowHeight** property to the desired value.

It is not necessary to execute the **Refresh** method within this event procedure. Doing so causes the grid to be repainted even if the Cancel argument is True.

## Scroll Event

**Syntax** object\_**Scroll** (Cancel As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** Cancel is an integer that may be set to True to prevent the scroll operation from occurring.

**Description** The **Scroll** event occurs when the user scrolls the grid horizontally or vertically using the scroll bars.

This event is fired before the grid is repainted to display the results of the scroll operation. If you set the Cancel argument to True, the scroll operation fails and no repainting occurs.

It is not necessary to execute the **Refresh** method within this event procedure. Doing so causes the grid to be repainted even if the Cancel argument is True.

You can use this event to perform calculations or to manipulate controls that must be coordinated with ongoing changes in the grid's scroll bars.

**Note** Within this event procedure, the values of the **FirstRow** and **LeftCol** properties are **not** updated to reflect the pending scroll operation. You cannot determine the orientation or magnitude of the pending scroll operation by examining these properties.

Avoid using a **MsgBox** statement or function in this event.

This event **only** fires when the user operates the scroll bars; it will not fire in response to keyboard navigation, data control notifications, or code.

## SelChange Event

**Syntax** object\_**SelChange** (Cancel As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments** Cancel is an integer that may be set to True to undo the new selection.

**Description** The **SelChange** event occurs when the user selects a different range of rows or columns. This event is triggered under several circumstances:

- When the user selects a single row by clicking its record selector.
- When the user adds a row to the list of selected rows by clicking its record selector while holding down the CTRL key.
- When the user selects a single column by clicking its header.
- When the user changes the range of selected columns by dragging to an adjacent column within the header row.
- When the user extends the range of selected columns by holding down the SHIFT key and clicking an unselected column header.
- When the user clears the current row or column selection by clicking an individual cell, in which case this event precedes **RowColChange**.

The current range of selected columns is provided by the **SelStartCol** and **SelEndCol** properties. The bookmarks of the selected rows are available in the **SelBookmarks** collection. Within this event procedure, these properties and collections reflect the user's pending selection(s).

If your event procedure sets the Cancel argument to True, the previous row and column selections (if any) are restored, and the aforementioned properties revert to their previous values.

This event is only triggered by user interaction with the grid. It **cannot** be triggered by code.

**Note** When the user selects a column, any row selections are cleared. Similarly, when the user selects a row, any column selections are cleared.

## SplitChange Event

**Syntax** TDBGrid\_**SplitChange** ( )

Event applies to **TDBGrid** control.

**Arguments** None

**Description** The **SplitChange** event occurs when the current cell changes to a different cell in another split. This event is triggered under several circumstances:

- When the grid is first displayed.
- When the user clicks a cell in another split, subject to the setting of the **AllowFocus** property.
- When the user presses a navigation key to cross a split boundary, subject to the setting of the **TabAcrossSplits** property.
- When the **Split** property is changed in code to a different value.
- When a new split is inserted before the current split via code or user interaction.
- When the current split is removed via code or user interaction.

If the user edits data and then moves the current cell position to a new row in another split, the update events for the original row are completed before the **SplitChange** event is executed.

If a split change also results in a change to the current row or column, then the **SplitChange** event will always precede the **RowColChange** event.

## UnboundAddData Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>UnboundAddData</b> (ByVal RowBuf As TrueDBGrid50.RowBuffer, NewRowBookmark As Variant)<br><br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| <b>Arguments</b>   | <b>RowBuf</b> is a <b>RowBuffer</b> object used to transfer new row data to the grid.<br><b>NewRowBookmark</b> is a variant that must be set to a unique bookmark for subsequent references to the newly added row.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Description</b> | <p>The <b>UnboundAddData</b> event is fired when the user adds a new row of data to an unbound grid (one with its <b>DataMode</b> property set to 1 - Unbound or 2 - Unbound Extended). This event alerts your application that it must add a new row of data to its unbound dataset.</p> <p>The <b>RowBuf</b> argument contains a single row of data to be written to the unbound dataset. Since only one row of data can be added at a time, the value of its <b>RowCount</b> property will always be 1. The number of columns in a <b>RowBuffer</b> is given by its <b>ColumnCount</b> property.</p> <p>The <b>Value</b> property of the <b>RowBuf</b> argument contains the cell data entered by the user. If the user did not modify a particular cell, then the corresponding entry in the <b>Value</b> property will contain a Null variant. In Visual Basic, the <b>IsNull</b> function can be used to test for this condition.</p> <p>Before returning from this event, <b>NewRowBookmark</b> <b>must</b> be set to the bookmark of the newly added row, or else the user will not be able to move to another row without first canceling the new row with the ESC key.</p> <p>This event will not be fired if the <b>AllowAddNew</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowAddNew</b> is never set to True.</p> |
| <b>Note</b>        | If the add operation fails in the underlying data source, then you should set the row buffer's <b>RowCount</b> property to 0 to inform the grid of the failure. The grid will not display an error message, but will leave the row in a modified state. At that point, the user can either correct the data or press the ESC key to cancel the operation.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

## UnboundColumnFetch Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>UnboundColumnFetch</b> (Bookmark As Variant, Col As Integer, Value As Variant)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Arguments</b>   | <b>Bookmark</b> is a variant that identifies the row being requested.<br><b>Col</b> is an integer that identifies the column being requested.<br><b>Value</b> is a variant used to transfer unbound column data to the grid.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p>The <b>UnboundColumnFetch</b> event is fired when a bound grid (one with its <b>DataMode</b> property set to 0 - Bound) needs to display the value of a cell in an unbound column as specified by the <b>Bookmark</b> and <b>Col</b> arguments. For a bound grid, any column with an empty <b>DataField</b> property and a non-empty <b>Caption</b> property is considered an unbound column.</p> <p>To return an unbound value to the grid, simply set the <b>Value</b> argument to the desired result. If you do not assign a value, the cell will remain blank.</p> <p>Use this event to implement calculated fields based on other columns or to display local data alongside remote bound data.</p> <p>Your application is responsible for storing data entered into an unbound column by the user. Use the <b>Column</b> object's <b>Text</b> property to retrieve unbound values within the <b>BeforeUpdate</b> and <b>BeforeInsert</b> events.</p> <p>If an unbound column is used to display a calculated result based on other columns, then you do not need to store the unbound values since they can always be calculated "on the fly" using either the <b>Column</b> object's <b>Text</b> property or data access objects.</p> |
| <b>Note</b>        | <p>Do not confuse unbound columns with unbound mode. The <b>UnboundColumnFetch</b> event is only fired when a bound grid contains one or more unbound columns.</p> <p>During the execution of this event, row movement is not permitted.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

## UnboundDeleteRow Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>UnboundDeleteRow</b> (Bookmark As Variant)<br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Arguments</b>   | <u>Bookmark</u> is a variant that uniquely identifies the row to be deleted.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Description</b> | <p>The <b>UnboundDeleteRow</b> event is fired when the user deletes an existing row within an unbound grid (one with its <b>DataMode</b> property set to 1 - Unbound or 2 - Unbound Extended). This event alerts your application that it must delete the row specified by the <u>Bookmark</u> argument from its unbound dataset.</p> <p>The <u>Bookmark</u> argument contains a bookmark supplied by your application in the <b>UnboundReadData</b>, <b>UnboundReadDataEx</b>, or <b>UnboundAddData</b> event.</p> <p>This event will not be fired if the <b>AllowDelete</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowDelete</b> is never set to True.</p> |
| <b>Note</b>        | If the delete operation fails in the underlying data source, then you should set <u>Bookmark</u> to Null to inform the grid of the failure. This will cause the grid's <b>Error</b> event to fire. The row specified by <u>Bookmark</u> will remain selected.                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

## UnboundFindData Event

**Syntax** TDBDropDown\_ **UnboundFindData** (StartLocation As Variant, ByVal ReadPriorRows As Boolean, ByVal IncludeCurrent As Boolean, ByVal Col As Integer, ByVal Value As Variant, ByVal SeekFlags As Integer, NewLocation As Variant)

Event applies to **TDBDropDown** control.

**Arguments** StartLocation is a bookmark that specifies the starting row for the search.

ReadPriorRows indicates the direction in which the dropdown is searching for data. If False, you should provide data in the forward direction starting with the row specified by StartLocation. If True, you should provide data in the backward direction, starting with the row specified by StartLocation.

IncludeCurrent indicates the inclusion of StartLocation in the search. If False, you should not use StartLocation when searching for data. If True, StartLocation should be included in the search.

Col is a column index that specifies the data column for the search.

Value is the value to be searched for.

SeekFlags is an UnboundFindConstants value that provides additional information about how the search should be performed.

The NewLocation argument is initially Null. However, before returning from this event, you should set it to a bookmark that uniquely identifies the row where the data was found. If you do not set the value of NewLocation, it is assumed that no values match and the dropdown will be positioned at the first row.

**Description** When a dropdown control is activated, it will attempt to position to the record that matches the current cell text of its parent grid. To do this, the dropdown will fire the **UnboundFindData** event, which allows you to set the current record position within the dropdown.

The SeekFlags argument specifies how to compare the Value argument to dropdown column data:

|                  |                                                                                                       |
|------------------|-------------------------------------------------------------------------------------------------------|
| dbgSeekLT        | Find first column data less than <u>Value</u>                                                         |
| dbgSeekLE        | Find first column data less than or equal to <u>Value</u>                                             |
| dbgSeekEQ        | Find first column data equal to <u>Value</u>                                                          |
| dbgSeekGE        | Find first column data greater than or equal to <u>Value</u>                                          |
| dbgSeekGT        | Find first column data greater than <u>Value</u>                                                      |
| dbgSeekPartialEQ | Find first column data that partially matches <u>Value</u> starting from the first character position |

**Note** This event will not fire when **DataMode** is set to 0 - Bound.



## UnboundGetRelativeBookmark Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | <p>object <b>UnboundGetRelativeBookmark</b> (StartLocation As Variant, ByVal Offset As Long, NewLocation As Variant, ApproximatePosition As Long)</p> <p>Event applies to <b>TDBGrid</b> and <b>TDBDropDown</b> controls.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <b>Arguments</b>   | <p><b>StartLocation</b> is a bookmark that, together with <b>Offset</b>, specifies the row to be returned in <b>NewLocation</b>. A <b>StartLocation</b> of Null indicates a request for a row from BOF or EOF.</p> <p><b>Offset</b> specifies the relative position (from <b>StartLocation</b>) of the row to be returned in <b>NewLocation</b>. A positive number indicates a forward relative position; a negative number indicates a backward relative position.</p> <p><b>NewLocation</b> is a variable that receives the bookmark of the row specified by <b>StartLocation</b> plus <b>Offset</b>. If the row specified is beyond the first or the last row (that is, beyond BOF or EOF), then <b>NewLocation</b> should be set to Null.</p> <p><b>ApproximatePosition</b> is a variable that optionally receives the ordinal position of <b>NewLocation</b>. Setting this variable will enhance the ability of the grid to display its vertical scroll bar accurately. If the exact ordinal position of <b>NewLocation</b> is not known, you can set it to a reasonable, approximate value, or just ignore this parameter.</p> |
| <b>Description</b> | <p>This event is mandatory when the <b>DataMode</b> property is set to 3 - Application. For <b>DataMode</b> 1 - Unbound, this event is optional. It is not used when the <b>DataMode</b> property is set to 2 - Unbound Extended.</p> <p>This event is used in conjunction with the <b>UnboundReadData</b> or <b>ClassicRead</b> events when the grid needs to obtain positional information about your underlying data. By coding this event for <b>DataMode</b> setting 1 - Unbound, you can dramatically improve the performance of the grid. However, you do not need to change existing applications; you can ignore this event and they will continue to function properly.</p> <p>Before returning from this event, you must set <b>NewLocation</b> to a valid bookmark. For example, if <b>Offset</b> is 1 (or -1), then you must return in <b>NewLocation</b> the bookmark of the row that follows (or precedes) <b>StartLocation</b>. However, if the requested row is beyond the first or last row, then you should return Null in <b>NewLocation</b> to inform the grid of BOF/EOF conditions.</p>                       |

## UnboundReadData Event

- Syntax**      object **UnboundReadData** (ByVal RowBuf As TrueDBGrid50.RowBuffer, StartLocation As Variant, ByVal ReadPriorRows As Boolean)
- Event applies to **TDBGrid** and **TDBDropDown** controls.
- Arguments**    **RowBuf** is a **RowBuffer** object used to transfer row data to the grid.
- StartLocation** is a variant bookmark that identifies the row to position to before fetching the next or previous page of records. If Null, the grid is requesting the first or last page of records as determined by the **ReadPriorRows** argument.
- ReadPriorRows** is a boolean that determines the direction in which rows are to be fetched. If True, the grid is requesting records that precede **StartLocation**. If False, the grid is requesting records that follow **StartLocation**.
- Description**    The **UnboundReadData** event is fired when an unbound grid (one with its **DataMode** property set to 1 - Unbound) requires data for display, such as when it is first loaded or the user scrolls the grid display.
- The **RowBuf** argument acts like a two-dimensional array of variants corresponding to the grid cells being fetched. By populating its **Value** property with the appropriate data, your event procedure transfers rows from the unbound dataset to the grid.
- Use the row buffer's **RowCount** property to determine how many rows of data the grid is requesting. Use its **ColumnCount** property to determine the number of columns to be populated.
- The **RowBuf** argument is also used to store a set of variant bookmarks that uniquely identify rows in the unbound dataset. The format of these bookmarks is determined solely by your application. For example, they may be primary key fields, row numbers, or array indexes, depending upon the nature of the unbound dataset.
- Your event procedure supplies bookmarks to the grid by populating the row buffer's **Bookmark** property for each row returned. Keep in mind that the bookmarks you provide in the **UnboundReadData** event may be subsequently passed to the **UnboundWriteData** and **UnboundDeleteRow** events depending on how the user interacts with the grid. In addition, bookmark-based **TDBGrid** properties and methods such as **Bookmark**, **FirstRow**, **GetBookmark**, and **RowBookmark** are also designed to work with these unbound bookmarks.
- It is not necessary to fill the row buffer completely, and it is in fact acceptable to return no rows at all. The row buffer's **RowCount** property can be set to indicate that fewer rows were returned than requested. The grid interprets this to mean that there are no more rows to retrieve in the indicated direction. Thus, it is only necessary to fill the row buffer completely if there are more valid rows to be retrieved.
- Note**            True DBGrid is very conservative in its assumptions about row counts and BOF/EOF conditions. As a result, it may seem that the **UnboundReadData** event fires "too often." This should not be interpreted as a sign of inefficiency, but rather as an assurance that an unbound grid performs accurately with large multiuser databases.

## UnboundReadDataEx Event

- Syntax** object **UnboundReadDataEx** (ByVal RowBuf As TrueDBGrid50.RowBuffer, StartLocation As Variant, ByVal Offset As Long, ApproximatePosition As Long)
- Event applies to **TDBGrid** and **TDBDropDown** controls.
- Arguments** **RowBuf** is a **RowBuffer** object used to transfer row data to the grid.
- StartLocation** is a bookmark that, together with **Offset**, specifies the starting row for data transfer. A **StartLocation** of Null indicates a request for data from BOF or EOF. For example, if **StartLocation** is Null and **Offset** is 2 (or -2), then you should retrieve data starting from the second (or second to last) row.
- Offset** specifies the relative position (from **StartLocation**) of the first row of data to be transferred. A positive number indicates a forward relative position; a negative number indicates a backward relative position.
- ApproximatePosition** is a variable that optionally receives the ordinal position of the first row of data to be transferred. Setting this variable will enhance the ability of the grid to display its vertical scroll bar accurately. If the exact ordinal position of the row is not known, you can set it to a reasonable, approximate value, or just ignore this parameter.
- Description** The **UnboundReadDataEx** event is fired when an unbound grid (one with its **DataMode** property set to 2 - Unbound Extended) requires a bookmark for a specific row or data for display, such as when it is first loaded or the user scrolls the grid display.
- Before returning from the **UnboundReadDataEx** event, you must fill the **Bookmark** array of **RowBuf** with unique row identifiers, and the **Value** array with the actual data. For example, if **Offset** is 1 (or -1), then you must fill in **RowBuf**, starting from the row that follows (or precedes) **StartLocation**.
- The **RowBuf** argument acts like a two-dimensional array of variants corresponding to the grid cells being fetched. By populating its **Value** property with the appropriate data, your event procedure transfers rows from the unbound dataset to the grid.
- Use the row buffer's **RowCount** property to determine how many rows of data the grid is requesting. Use its **ColumnCount** property to determine the number of columns to be populated, if any. If **ColumnCount** is zero, the grid is requesting the bookmark of a single row; if **ColumnCount** is nonzero, the grid is requesting **RowCount** rows of data and their corresponding bookmarks.
- The **ColumnIndex** property specifies the grid column index corresponding to a row buffer column index; you must fill in the **Value** property array with column data according to the column index specified by the **ColumnIndex** property array.
- The **RowBuf** argument is also used to store a set of variant bookmarks that uniquely identify rows in the unbound dataset. The format of these bookmarks is determined solely by your application. For example, they may be primary key fields, row numbers, or array indexes, depending upon the nature of the unbound dataset.
- Your event procedure supplies bookmarks to the grid by populating the row buffer's **Bookmark** property for each row returned. Keep in mind that the bookmarks you provide in the **UnboundReadDataEx** event may be subsequently passed to the **UnboundWriteData** and **UnboundDeleteRow** events depending on how the user interacts with the grid. In addition, bookmark-based **TDBGrid** properties and methods such as **Bookmark**, **FirstRow**, **GetBookmark**, and **RowBookmark** are also designed to work with these unbound bookmarks.
- It is not necessary to fill the row buffer completely, and it is in fact acceptable to return no rows at all. The row buffer's **RowCount** property can be set to indicate that fewer rows were returned than requested. The grid interprets this to mean that there are no more rows to retrieve in the indicated direction. Thus, it is only necessary to fill the row buffer completely if

there are more valid rows to be retrieved.

## UnboundWriteData Event

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Syntax</b>      | TDBGrid_ <b>UnboundWriteData</b> (ByVal RowBuf As TrueDBGrid50.RowBuffer, WriteLocation As Variant)<br><br>Event applies to <b>TDBGrid</b> control.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Arguments</b>   | <b>RowBuf</b> is a <b>RowBuffer</b> object used to transfer modified row data from the grid to the unbound data source.<br><br><b>WriteLocation</b> is a variant bookmark that uniquely identifies the row to be updated.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Description</b> | <p>The <b>UnboundWriteData</b> event is fired when the user modifies an existing row within an unbound grid (one with its <b>DataMode</b> property set to 1 - Unbound or 2 - Unbound Extended) and attempts to commit the changes by moving to a different row. This event alerts your application that it must update the row specified by the <b>WriteLocation</b> argument within its unbound dataset.</p> <p>The <b>RowBuf</b> argument contains a single row of data to be written to the unbound dataset. Since only one row of data can be updated at a time, the value of its <b>RowCount</b> property will always be 1. The number of columns in a <b>RowBuffer</b> is given by its <b>ColumnCount</b> property.</p> <p>The <b>WriteLocation</b> argument contains a bookmark supplied by your application in either the <b>UnboundReadData</b>, <b>UnboundReadDataEx</b>, or <b>UnboundAddData</b> event.</p> <p>The <b>Value</b> property of the <b>RowBuf</b> argument contains the cell data entered by the user. If the user did not modify a particular cell, then the corresponding entry in the <b>Value</b> property will contain a Null variant, <b>not</b> the current cell contents. In Visual Basic, the <b>IsNull</b> function can be used to test for this condition.</p> <p>This event will not be fired if the <b>AllowUpdate</b> property is set to False. Conversely, if you do not implement this event, then you must ensure that <b>AllowUpdate</b> is never set to True.</p> |
| <b>Note</b>        | <p>If the update operation fails in the underlying data source, then you should set the row buffer's <b>RowCount</b> property to 0 to inform the grid of the failure. The grid will not display an error message, but will leave the row in a modified state. At that point, the user can either correct the data or press the ESC key to cancel the operation.</p> <p>You can force the <b>UnboundWriteData</b> event to occur in code by calling the <b>Update</b> method. This method is particularly valuable when the unbound dataset contains a single row and <b>AllowAddNew</b> is False, since there is no way for the user to trigger the update by moving to another row in this case.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

## ValueItemError Event

**Syntax**            object\_ **ValueItemError** (ColIndex As Integer)

Event applies to **TDBGrid** and **TDBDropDown** controls.

**Arguments**        ColIndex is an integer that identifies the column being edited.

**Description**      The **ValueItemError** event is triggered when the user attempts to enter invalid data into a column that is using value lists. This event is only triggered when the associated **ValueItems** collection has its **Validate** property set to True.

This event is useful even if the user is permitted to enter values not present in the **ValueItems** collection, as you may want to add the new value to the collection in this event. It also allows you to control how you want to respond to incorrect input.



## Constant Reference

---

{button ,JI(`,`AddNewMode\_Constants')}} AddNewMode Constants  
{button ,JI(`,`Alignment\_Constants')}} Alignment Constants  
{button ,JI(`,`Appearance\_Constants')}} Appearance Constants  
{button ,JI(`,`BorderStyle\_Constants')}} BorderStyle Constants  
{button ,JI(`,`CellStyle\_Constants')}} CellStyle Constants  
{button ,JI(`,`CellTip\_Constants')}} CellTip Constants  
{button ,JI(`,`CellTipPresentation\_Constants')}} CellTipPresentation Constants  
{button ,JI(`,`DataMode\_Constants')}} DataMode Constants  
{button ,JI(`,`DividerStyle\_Constants')}} DividerStyle Constants  
{button ,JI(`,`Error\_Constants')}} Error Constants  
{button ,JI(`,`ExposeCellMode\_Constants')}} ExposeCellMode Constants  
{button ,JI(`,`MarqueeStyle\_Constants')}} MarqueeStyle Constants  
{button ,JI(`,`Presentation\_Constants')}} Presentation Constants  
{button ,JI(`,`ScrollBars\_Constants')}} ScrollBars Constants  
{button ,JI(`,`SplitSizeMode\_Constants')}} SplitSizeMode Constants  
{button ,JI(`,`TabAction\_Constants')}} TabAction Constants  
{button ,JI(`,`UnboundFind\_Constants')}} UnboundFind Constants



## AddNewMode Constants

**Applies To**     AddNewMode property

| <b>Values</b> | <b>Description</b>         | <b>Run Time</b>  |
|---------------|----------------------------|------------------|
| 0             | No AddNew pending          | dbgNoAddNew      |
| 1             | Current cell in AddNew row | dbgAddNewCurrent |
| 2             | AddNew pending             | dbgAddNewPending |

## Alignment Constants

**Applies To**      Alignment and HeadAlignment properties

| <b>Values</b> | <b><u>Design Time</u></b> | <b><u>Run Time</u></b> |
|---------------|---------------------------|------------------------|
|               | 0 - Left (default)        | dbgLeft                |
|               | 1 - Right                 | dbgRight               |
|               | 2 - Center                | dbgCenter              |
|               | 3 - General               | dbgGeneral             |

## Appearance Constants

**Applies To**     Appearance property

| <b>Values</b> | <b>Design Time</b> | <b>Run Time</b> |
|---------------|--------------------|-----------------|
|               | 0 - Flat           | dbgFlat         |
|               | 1 - 3D (default)   | dbg3D           |

## BorderStyle Constants

**Applies To**     BorderStyle property

| <b>Values</b>              | <b>Design Time</b> | <b>Run Time</b> |
|----------------------------|--------------------|-----------------|
| 0 - None                   |                    | dbgNoBorder     |
| 1 - Fixed Single (default) |                    | dbgFixedSingle  |

## CellStyle Constants

**Applies To** AddCellStyle, AddRegexCellStyle, ClearCellStyle, and ClearRegexCellStyle methods; FetchCellStyle event

| <b>Values</b> | <b>Description</b>              | <b>Run Time</b> |
|---------------|---------------------------------|-----------------|
| -1            | All Cells                       | dbgAllCells     |
| 0             | Cells without status conditions | dbgNormalCell   |
| 1             | Current cell                    | dbgCurrentCell  |
| 2             | Cells in a highlighted row      | dbgMarqueeRow   |
| 4             | Cells that have been modified   | dbgUpdatedCell  |
| 8             | Cells in a selected row         | dbgSelectedRow  |

## CellTip Constants

**Applies To**      FetchCellTips event

| <b>Values</b>           | <b>Description</b> | <b>Run Time</b>     |
|-------------------------|--------------------|---------------------|
| -1 - On Record Selector |                    | dbgOnRecordSelector |
| -2 - On Empty Column    |                    | dbgOnEmptyColumn    |
| -1 - On Column Header   |                    | dbgOnColumnHeader   |
| -2 - On Split Header    |                    | dbgOnSplitHeader    |
| -3 - On Empty Row       |                    | dbgOnEmptyRow       |
| -4 - On Caption         |                    | dbgOnCaption        |
| -5 - On AddNew Row      |                    | dbgOnAddNew         |

## CellTipPresentation Constants

**Applies To**     CellTips property

| <b>Values</b>      | <b>Design Time</b> | <b>Run Time</b> |
|--------------------|--------------------|-----------------|
| 0 - None (default) |                    | dbgNoCellTips   |
| 1 - Anchored       |                    | dbgAnchored     |
| 2 - Floating       |                    | dbgFloating     |

## DataMode Constants

**Applies To**      DataMode property

| <b>Values</b> | <b>Design Time</b>   | <b>Run Time</b> |
|---------------|----------------------|-----------------|
|               | 0 - Bound (default)  | dbgBound        |
|               | 1 - Unbound          | dbgUnbound      |
|               | 2 - Unbound Extended | dbgUnboundEx    |
|               | 3 - Application      | dbgUnboundAp    |
|               | 4 - Storage          | dbgUnboundSt    |



## DividerStyle Constants

**Applies To** DividerStyle and RowDividerStyle properties

| <b>Values</b>                | <b>Design Time</b> | <b>Run Time</b>  |
|------------------------------|--------------------|------------------|
| 0 - No dividers              |                    | dbgNoDividers    |
| 1 - Black line               |                    | dbgBlackLine     |
| 2 - Dark gray line (default) |                    | dbgDarkGrayLine  |
| 3 - Raised                   |                    | dbgRaised        |
| 4 - Inset                    |                    | dbgInset         |
| 5 - ForeColor                |                    | dbgUseForeColor  |
| 6 - Light gray line          |                    | dbgLightGrayLine |

## Error Constants

| <b>Applies To</b> | Trappable errors for <b>TDBGrid</b> and <b>TDBDropDown</b> controls |                 |
|-------------------|---------------------------------------------------------------------|-----------------|
| <b>Values</b>     | <b>Description</b>                                                  | <b>Run Time</b> |
|                   | 4097 - Cannot initialize data bindings                              | dbgBINDERROR    |
|                   | 4098 - Invalid setting for <i>name</i> property                     | dbgINVPROPVAL   |
|                   | 6145 - Invalid column index                                         | dbgCOLINDEX     |
|                   | 6146 - Control not properly initialized                             | dbgNOTINIT      |
|                   | 6147 - Column not found                                             | dbgCNOTFOUND    |
|                   | 6148 - Invalid row number                                           | dbgINVROWNUM    |
|                   | 6149 - Invalid bookmark                                             | dbgINVBOOKMARK  |
|                   | 6150 - Invalid selected row bookmark index                          | dbgBADSELRIDX   |
|                   | 6151 - Scroll arguments out of range                                | dbgSCROLLRANGE  |
|                   | 6152 - Invalid setting for ScrollBars property                      | dbgINVSBSTYLE   |
|                   | 6153 - Error occurred while trying to update record                 | dbgUPDERROR     |
|                   | 6154 - Error occurred while trying to add record                    | dbgADDEROR      |
|                   | 6155 - Error occurred while trying to delete record                 | dbgDELEERROR    |
|                   | 6156 - Data type mismatch during field update                       | dbgCOLDATA      |
|                   | 6157 - Data type incompatible with column data type                 | dbgINCOMPAT     |
|                   | 6158 - <i>name</i> is not a valid data field name                   | dbgFIELDERR     |
|                   | 6159 - Cannot delete multiple rows                                  | dbgDELMULTROWS  |
|                   | 6160 - Data access error                                            | dbgDATAACCESS   |
|                   | 6161 - Operation is invalid within the event <i>name</i>            | dbgBADEVENT     |
|                   | 6162 - Property is not available in this context                    | dbgNOPROPNOW    |
|                   | 6163 - No current record                                            | dbgNOCURREC     |
|                   | 6164 - Caption text is too long                                     | dbgCAPTOOLONG   |
|                   | 6244 - Invalid split index                                          | dbgSPLITINDEX   |
|                   | 6245 - Invalid value list index                                     | dbgVLINDEX      |
|                   | 6246 - Error accessing value item                                   | dbgVITEMERR     |
|                   | 6247 - Invalid style index                                          | dbgSTYLEINDEX   |
|                   | 6248 - Duplicate style name                                         | dbgDUPSTYLE     |
|                   | 6249 - Error accessing style                                        | dbgSTYLEERR     |
|                   | 6250 - Error updating style                                         | dbgUPDSTYLE     |
|                   | 6251 - Error removing style                                         | dbgREMSTYLE     |
|                   | 6252 - Error adding cell condition                                  | dbgADDCELLCOND  |
|                   | 6253 - Invalid style name                                           | dbgSTYLENAME    |
|                   | 6254 - Error applying style                                         | dbgAPPLYSTYLE   |
|                   | 6255 - Bitmap is too large                                          | dbgBMPTOOLARGE  |

## ExposeCellMode Constants

**Applies To**     ExposeCellMode property

| <b>Values</b>                  | <b>Design Time</b> | <b>Run Time</b>   |
|--------------------------------|--------------------|-------------------|
| 0 - Scroll on Select (default) |                    | dbgScrollOnSelect |
| 1 - Scroll on Edit             |                    | dbgScrollOnEdit   |
| 2 - Never Scroll               |                    | dbgNeverScroll    |

## MarqueeStyle Constants

**Applies To** MarqueeStyle property

| <b>Values</b>                 | <b>Design Time</b> | <b>Run Time</b>          |
|-------------------------------|--------------------|--------------------------|
| 0 - Dotted Cell Border        |                    | dbgDottedCellBorder      |
| 1 - Solid Cell Border         |                    | dbgSolidCellBorder       |
| 2 - Highlight Cell            |                    | dbgHighlightCell         |
| 3 - Highlight Row             |                    | dbgHighlightRow          |
| 4 - Highlight Row, Raise Cell |                    | dbgHighlightRowRaiseCell |
| 5 - No Marquee                |                    | dbgNoMarquee             |
| 6 - Floating Editor (default) |                    | dbgFloatingEditor        |

## Presentation Constants

**Applies To**      Presentation property

| <b>Values</b> | <b>Design Time</b>   | <b>Run Time</b>   |
|---------------|----------------------|-------------------|
|               | 0 - Normal (default) | dbgNormal         |
|               | 1 - Radio Button     | dbgRadioButton    |
|               | 2 - Combo Box        | dbgComboBox       |
|               | 3 - Sorted Combo Box | dbgSortedComboBox |

## ScrollBars Constants

**Applies To**     ScrollBars property

| <b>Values</b>           | <b>Design Time</b> | <b>Run Time</b> |
|-------------------------|--------------------|-----------------|
| 0 - None                |                    | dbgNone         |
| 1 - Horizontal          |                    | dbgHorizontal   |
| 2 - Vertical            |                    | dbgVertical     |
| 3 - Both                |                    | dbgBoth         |
| 4 - Automatic (default) |                    | dbgAutomatic    |

## SplitSizeMode Constants

**Applies To**     SizeMode property

| <b>Values</b>          | <b>Design Time</b> | <b>Run Time</b>    |
|------------------------|--------------------|--------------------|
| 0 - Scalable (default) |                    | dbgScalable        |
| 1 - Exact              |                    | dbgExact           |
| 2 - Number of Columns  |                    | dbgNumberOfColumns |

## TabAction Constants

**Applies To**     TabAction property

| <b>Values</b>                    | <b>Design Time</b> | <b>Run Time</b>      |
|----------------------------------|--------------------|----------------------|
| 0 - Control Navigation (default) |                    | dbgControlNavigation |
| 1 - Column Navigation            |                    | dbgColumnNavigation  |
| 2 - Grid Navigation              |                    | dbgGridNavigation    |



## UnboundFind Constants

**Applies To** UnboundFindData event

| <b>Values</b>              | <b>Description</b> | <b>Run Time</b>  |
|----------------------------|--------------------|------------------|
| -1 - Less Than             |                    | dbgSeekLT        |
| -2 - Less Than or Equal    |                    | dbgSeekLE        |
| -3 - Equal                 |                    | dbgSeekEQ        |
| -4 - Greater Than or Equal |                    | dbgSeekGE        |
| -5 - Greater Than          |                    | dbgSeekGT        |
| -6 - Partially Equal       |                    | dbgSeekPartialEQ |

## XArray Reference

---

{button ,JI(``, `XArray\_Object\_Properties`)} XArray Object Properties

{button ,JI(``, `XArray\_Object\_Methods`)} XArray Object Methods

## XArray Object Properties

|                          |                                                          |
|--------------------------|----------------------------------------------------------|
| <b><u>AutoReDim</u></b>  | Controls redimensioning when the last element is deleted |
| <b><u>Count</u></b>      | Returns the number of elements for a given dimension     |
| <b><u>LowerBound</u></b> | Returns the lower bound for a given dimension            |
| <b><u>UpperBound</u></b> | Returns the upper bound for a given dimension            |
| <b><u>Value</u></b>      | Sets/returns the value of an individual array element    |

## XArray Object Methods

|                         |                                                                  |
|-------------------------|------------------------------------------------------------------|
| <b><u>AboutBox</u></b>  | Displays a dialog box with information about the array object    |
| <b><u>Clear</u></b>     | Deallocates all data associated with an array object             |
| <b><u>Delete</u></b>    | Deletes an index from a given dimension                          |
| <b><u>DeleteDim</u></b> | Deletes a given dimension in its entirety                        |
| <b><u>Get</u></b>       | Returns the value of an array element                            |
| <b><u>Insert</u></b>    | Inserts a new index into a given dimension                       |
| <b><u>InsertDim</u></b> | Inserts a new dimension into an array object                     |
| <b><u>ReDim</u></b>     | Sets the dimensions of an array object while preserving its data |
| <b><u>Set</u></b>       | Assigns a value to an array element                              |

## **AboutBox Method (XArray)**

**Syntax** XArray.**AboutBox**

**Arguments** None

**Return Value** None

**Description** This method displays the copyright notice for XArray.

## AutoReDim Property (XArray)

**Syntax** XArray.**AutoReDim** = boolean

Read/Write at run time. Not available at design time.

**Arguments** None

**Description** This property sets or returns a boolean that controls whether a dimension is automatically removed when its last element is deleted.

If True (the default), deleting the last element of a dimension removes the dimension in its entirety while preserving the data in other dimensions.

If False, deleting the last element of a dimension does not remove the dimension, but causes it to have zero dimensions. This behavior is like that of standard Visual Basic arrays.

When you set the **Array** property of a **TDBGrid** or **TDBDropDown** control to an **XArray** object, the grid will automatically set the **AutoReDim** property of the **XArray** object to False.

**Example** Consider an **XArray** with two rows and three columns, initialized as follows:

```
Dim MyArray As New XArray
Dim i, j As Integer
MyArray.ReDim 0, 1, 0, 2
For i = 0 To 1
 For j = 0 To 2
 MyArray(i, j) = "Row " & i & ", Col " & j
 Next j
Next i
```

If **AutoReDim** is True, then deleting the last remaining row effectively turns a two-dimensional array into a one-dimensional array:

```
MyArray.Delete 1, 0 ' delete first row
MyArray.Delete 1, 0 ' delete last remaining row
Debug.Print MyArray.Count(1) ' prints 3
Debug.Print MyArray.Count(2) ' prints 0
Debug.Print MyArray(0) ' prints "Row 1, Col 0"
Debug.Print MyArray(1) ' prints "Row 1, Col 1"
Debug.Print MyArray(1, 99) ' prints "Row 1, Col 1"
```

Note that the second dimension index is ignored in the last line.

However, if **AutoReDim** is False, then deleting the last remaining row removes all dimensions:

```
MyArray.Delete 1, 0 ' delete first row
MyArray.Delete 1, 0 ' delete last remaining row
Debug.Print MyArray.Count(1) ' prints 0
Debug.Print MyArray.Count(2) ' prints 0
Debug.Print MyArray(0) ' Error: Subscript out of range
Debug.Print MyArray(1) ' Error: Subscript out of range
Debug.Print MyArray(1, 99) ' Error: Subscript out of range
```

## Clear Method (XArray)

**Syntax** XArray.**Clear**

**Arguments** None

**Return Value** None

**Description** This method deallocates all data associated with the **XArray** while preserving its dimensions. After the **Clear** method executes, all of the array elements contain empty variants, and both of the following Visual Basic expressions evaluate to True for any element:

```
IsEmpty(element)
```

```
VarType(element) = 0
```

## Count Property (XArray)

**Syntax** XArray.**Count** (nDim)

Read-only at run time. Not available at design time.

**Arguments** nDim is a one-based integer specifying an array dimension.

**Description** This property returns a long integer that specifies the number of elements contained in a given dimension of an **XArray**. The value returned is always equal to:

```
MyArray.UpperBound(nDim) - MyArray.LowerBound(nDim) + 1
```

If the specified dimension does not exist, the **Count** property returns 0.

**Example** The following example uses one-based row indexes (the first dimension) and zero-based column indexes (the second dimension):

```
MyArray.ReDim 1, 100, 0, 5
Dim N As Long
N = MyArray.Count(1) ' returns 100
N = MyArray.Count(2) ' returns 6
```



## Delete Method (XArray)

**Syntax** XArray.**Delete** nDim, index

**Arguments** nDim is a one-based integer specifying an array dimension.

index is a long integer specifying an element position within the dimension nDim.

**Return Value** None

**Description** This method deletes the element at position index from the dimension specified by nDim while preserving data and shifting the indexes of the remaining elements appropriately.

When the last element in a dimension is deleted, the **AutoReDim** property determines whether the dimension is removed completely.

**Example** The following example initializes an **XArray** with 10 rows and 6 columns, then removes the fifth column:

```
MyArray.ReDim 0, 9, 0, 5
Dim i, j As Integer
For i = 0 To 9
 For j = 0 To 5
 MyArray(i, j) = "Row " & i & ", Col " & j
 Next j
Next i
Debug.Print MyArray(1, 4) ' prints Row 1, Col 4
Debug.Print MyArray(1, 5) ' prints Row 1, Col 5

' remove the 4th index in the 2nd dimension
MyArray.Delete 2, 4
Debug.Print MyArray.Count(2) ' prints 5
Debug.Print MyArray.UpperBound(2) ' prints 4
Debug.Print MyArray(1, 4) ' prints Row 1, Col 5

' the next line now gives a "Subscript out of range" error
Debug.Print MyArray(1, 5)
```

## DeleteDim Method (XArray)

**Syntax** XArray.**DeleteDim** nDim

**Arguments** nDim is a one-based integer specifying an array dimension.

**Return Value** None

**Description** This method removes the dimension specified by nDim while preserving data and shifting the remaining dimensions as appropriate.

**Example** The following example initializes an XArray with 10 rows and 4 columns, then removes the first dimension. The resulting one-dimensional array contains the four elements from the former first row:

```
MyArray.ReDim 1, 10, 1, 4
MyArray(1, 1) = "earth"
MyArray(1, 2) = "fire"
MyArray(1, 3) = "water"
MyArray(1, 4) = "air"

' delete the first dimension
MyArray.DeleteDim 1
Debug.Print MyArray.Count(1) ' prints 4
Debug.Print MyArray.Count(2) ' prints 0
Debug.Print MyArray(1) ' prints earth
Debug.Print MyArray(4) ' prints air
```

## Get Method (XArray)

**Syntax** XArray.**Get** var, index1 [, index2, index3, ..., index10]

**Arguments** var is a variable that receives a variant value from the specified array element.

index1 through index10 are long integers specifying an index into the dimension corresponding to their order in the argument list.

**Return Value** None

**Description** The **Get** method provides an alternate way of retrieving a value from an **XArray** element and is useful for those VBA dialects that do not support **XArray**'s default **Value** property, such as early versions of VBScript.

The var argument receives the variant value of the specified **XArray** element. index1 is the index of the first dimension, index2 is the index of the second dimension, and so on. The number of indexes required is equal to the number of dimensions specified with the **ReDim** method.

**Example** The three statements following the variable declaration are all equivalent:

```
Dim var As Variant
MyArray.Get var, x, y
var = MyArray.Value(x, y)
var = MyArray(x, y)
```

## Insert Method (XArray)

**Syntax** XArray.**Insert** nDim, index

**Arguments** nDim is a one-based integer specifying an array dimension.

index is a long integer specifying an element position within the dimension nDim.

**Return Value** None

**Description** This method inserts a new element at position index into the dimension specified by nDim while preserving data and shifting the indexes of the existing elements appropriately.

If index is greater than the value of the **UpperBound** property for dimension nDim, then **UpperBound** is adjusted to the new index.

**Example** The following example initializes an **XArray** with 10 rows and 6 columns, then inserts a new column before the fifth column:

```
MyArray.ReDim 0, 9, 0, 5
Dim i, j As Integer
For i = 0 To 9
 For j = 0 To 5
 MyArray(i, j) = "Row " & i & ", Col " & j
 Next j
Next i
Debug.Print MyArray(1, 4) ' prints Row 1, Col 4
Debug.Print MyArray(1, 5) ' prints Row 1, Col 5

' insert a new 4th index in the 2nd dimension
MyArray.Insert 2, 4
Debug.Print MyArray.Count(2) ' prints 7
Debug.Print MyArray.UpperBound(2) ' prints 6
Debug.Print IsEmpty(MyArray(1, 4)) ' prints True
Debug.Print MyArray(1, 5) ' prints Row 1, Col 4
Debug.Print MyArray(1, 6) ' prints Row 1, Col 5
```

## InsertDim Method (XArray)

**Syntax** XArray.**InsertDim** nDim, lbound, ubound

**Arguments** nDim is a one-based integer specifying an array dimension.

lbound is a long integer specifying the lower bound of the new dimension.

ubound is a long integer specifying the upper bound of the new dimension.

**Return Value** None

**Description** This method inserts a new dimension at the position specified by nDim while preserving existing data and shifting dimensions as appropriate. The lbound and ubound arguments define the lower and upper bounds for accessing elements in the new dimension.

**Example** The following example initializes an **XArray** with 10 rows and 4 columns, then inserts a new second dimension with 7 elements:

```
MyArray.ReDim 1, 10, 1, 4
Dim i, j As Integer
For i = 1 To 10
 For j = 1 To 4
 MyArray(i, j) = 0
 Next j
Next i
MyArray(1, 4) = 100
MyArray(2, 3) = 200

' insert a new 2nd dimension with LowerBound=0, UpperBound=6
MyArray.InsertDim 2, 0, 6
Debug.Print MyArray.Count(1) ' prints 10
Debug.Print MyArray.Count(2) ' prints 7
Debug.Print MyArray.Count(3) ' prints 4

' set an element using all three dimensions
MyArray(2, 6, 3) = 300
Debug.Print MyArray(2, 6, 3) ' prints 300
Debug.Print MyArray(1, 0, 4) ' prints 100, formerly (1, 4)
Debug.Print MyArray(2, 0, 3) ' prints 200, formerly (2, 3)
Debug.Print MyArray(1, 0, 3) ' prints 0, formerly (1, 3)

' this prints True, since the element is uninitialized
Debug.Print IsEmpty(MyArray(1, 1, 4))

' this gives an error: Subscript out of range
Debug.Print MyArray(0, 1, 4)
```

## LowerBound Property (XArray)

**Syntax** XArray.**LowerBound** (nDim)

Read-only at run time. Not available at design time.

**Arguments** None

**Description** This property returns a long integer that specifies the lower bound index for a given dimension of an **XArray**.

If the specified dimension does not exist, the **LowerBound** property returns 0.

**Example** The following example uses one-based row indexes (the first dimension) and zero-based column indexes (the second dimension):

```
MyArray.ReDim 1, 100, 0, 5
Dim N As Long
N = MyArray.LowerBound(1) ' returns 1
N = MyArray.LowerBound(2) ' returns 0
```

## ReDim Method (XArray)

**Syntax** XArray.**ReDim** lb1, ub1 [, lb2, ub2, ..., lb10, ub10]

**Arguments** lb1 (ub1) through lb10 (ub10) are long integers specifying the lower (upper) bound of the dimension corresponding to their order in the argument list.

**Return Value** None

**Description** This method is used to set or reset the dimensions of an **XArray** object while preserving any existing data. For each array dimension, you must specify both a lower bound index (lb1, lb2, ...) and an upper bound index (ub1, ub2, ...).

A newly created **XArray** object does not have any default dimensions; therefore, you must use the **ReDim** method before you can assign or access array elements.

**Example** The following example creates and initializes a two-dimensional **XArray**. The first dimension has 100 elements, with indexes starting at 1 and ending at 100. The second dimension has 6 elements, with indexes starting at 0 and ending at 5.

```
Dim MyArray As New XArray
MyArray.ReDim 1, 100, 0, 5
```

**Note** Although **XArray** supports up to 10 dimensions, when used in conjunction with True DBGrid's storage mode (**DataMode** 4), only two-dimensional and one-dimensional arrays make sense.

## Set Method (XArray)

**Syntax** XArray.**Set** value, index1 [, index2, index3, ..., index10]

**Arguments** value is a variant to be assigned to an array element.

index1 through index10 are long integers specifying an index into the dimension corresponding to their order in the argument list.

**Return Value** None

**Description** The Set method provides an alternate way of assigning a value to an XArray element and is useful for those VBA dialects that do not support XArray's default Value property, such as early versions of VBScript.

The value argument is the variant value to be assigned to the XArray element. index1 is the index of the first dimension, index2 is the index of the second dimension, and so on. The number of indexes required is equal to the number of dimensions specified with the ReDim method.

**Example** The following three statements are equivalent:

```
MyArray.Set "Hello", x, y
MyArray.Value(x, y) = "Hello"
MyArray(x, y) = "Hello"
```



## UpperBound Property (XArray)

**Syntax** XArray.**UpperBound** (nDim)

Read-only at run time. Not available at design time.

**Arguments** None

**Description** This property returns a long integer that specifies the upper bound index for a given dimension of an **XArray**.

If the specified dimension does not exist, the **UpperBound** property returns 0.

**Example** The following example uses one-based row indexes (the first dimension) and zero-based column indexes (the second dimension):

```
MyArray.ReDim 1, 100, 0, 5
Dim N As Long
N = MyArray.UpperBound(1) ' returns 100
N = MyArray.UpperBound(2) ' returns 5
```

## Value Property (XArray)

**Syntax** XArray.**Value** (index1 [, index2, index3, ..., index10]) = variant

Read/Write at run time. Not available at design time.

**Arguments** index1 through index10 are long integers specifying an index into the dimension corresponding to their order in the argument list.

**Description** The Value property sets or returns the variant value of an individual XArray element. The number of indexes required is equal to the number of dimensions specified with the ReDim method. For instance, given an array with three dimensions, the following statement assigns a string value to an individual element:

```
MyArray.Value(x, y, z) = "Hello"
```

Similarly, the following statement retrieves it:

```
s$ = MyArray.Value(x, y, z)
```

Note that the Value property is the default property for XArray. Therefore, the preceding statements can be shortened to:

```
MyArray(x, y, z) = "Hello"
```

```
s$ = MyArray(x, y, z)
```

**Example** Since the elements of an XArray object are true variants, you can arbitrarily mix data types within a row (or column), as in the following example:

```
MyArray(1, 1) = "Hello" ' string
MyArray(1, 2) = 1.98 ' floating point
MyArray(1, 3) = 250000 ' long integer
MyArray(2, 2) = True ' boolean
MyArray(2, 3) = LoadPicture("xyz.bmp") ' bitmap
```

**Note** Although XArray can be used to store picture data, True DBGrid's storage mode (DataMode 4), only accepts string and numeric data, and will not render XArray picture data as an in-cell graphic.

