10Tec iGrid ActiveX 6.0
What's New in the Release

Contents

Tags used to classify changes:

- [New] – a new feature, member or member parameter;
- [Change] – a change in a member functionality or interactive behavior;
- [Fixed] – a fixed bug or solved problem;
- [Removed] – a member was completely removed;
- [Enhancement] – some functionality was enhanced;
- [Optimization] – a feature has speed improvements and/or uses fewer resources;
- [Renaming] – a member was renamed;
- [Code-Upgrade] – indicates a change existing code may need when upgrading from the previous version.

Revamped context menu system

This release of iGrid introduces new tools related to the context menu system. The following operations can be easily implemented now in iGrid:

- You can add custom menu items to the built-in cell and column header context menus, or even replace them with your own menus.
- You can implement custom context menus for the header extra area and no-cell area.
- The native OS context menu displayed while editing text cells can be replaced with your own custom context menu.

Note that these tasks can be implemented using only iGrid's members, i.e. without any external components or VB/VBA intrinsic language tools. This is especially useful for MS Office VBA developers as this environment does not provide developers with built-in objects for building system-like context menus.

1. [New] Custom context menu items are defined using the new **ContextMenuCustomItems** property. It is indexed by the items of the new **EContextMenuSource** enumeration:

```
Enum EContextMenuSource
   igContextMenuCell = 1
   igContextMenuColHeader = 2
   igContextMenuNoCellArea = 3
   igContextMenuHeaderExtraArea = 4
   igContextMenuTextEdit = 5
End Enum
```

Every item of this enumeration defines the area a context menu belongs to, and the **ContextMenuCustomItems** property returns the collection of custom items for the context menu in the specified area. For instance, the following expression can be used to access the collection of custom context menu items for cells:

```
iGrid1.ContextMenuCustomItems(igContextMenuCell)
```

A collection of custom menu items is an object of the new **ContextMenuItemsObject** class. It provides you with basic features you can use to work with collections, such as the **Count** property and the **Add**, **Remove** and **Clear** methods. Custom menu items are indexed using numeric indices starting from 1. You can read/set the string caption, the enabled and checked statuses, and the optional tag to store any extra information for every menu item using the indexed **ItemCaption(), ItemEnabled(),ItemChecked()** and **ItemTag()** properties respectively of the **ContextMenuItemsObject**.

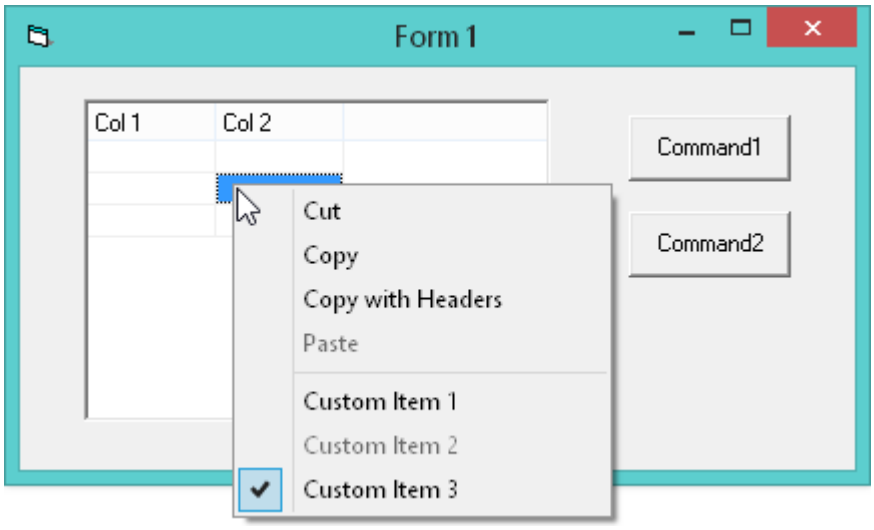
The **Add** method of the **ContextMenuItemsObject** is used to create menu items:

```
Sub Add( _
   ByVal sCaption As String, _
   Optional ByVal bEnabled As Boolean = True, _
   Optional ByVal bChecked As Boolean = False, _
   Optional ByVal vTag As Variant)
```

The **sCaption** parameter specifies the caption of the new menu item; a hyphen (-) is used to create a menu separator. The optional **bEnabled** and **bChecked** parameters are used to make the new menu item enabled/disabled and to add a check mark to it respectively. The **vTag** parameter is filled with an optional value of any type you want to associate with the new item.

Below is an example adding 3 custom items to the default cell context menu:

```
With iGrid1.ContextMenuCustomItems(igContextMenuCell)
   .Add "Custom Item 1"
   .Add "Custom Item 2", False
   .Add "Custom Item 3", , True
End With
```



Pay attention to the fact that iGrid automatically inserts a separator item before the first custom item if the built-in context menu already contains some default items.

2. [New] The new **ContextMenuPopup** event is raised before a context menu is displayed on the screen:

```
Event ContextMenuPopup( _
   ByVal eContextMenu As EContextMenuSource,
   ByVal lRowIfAny As Long, ByVal lColIfAny As Long, _
   ByRef bCancel As Boolean, ByRef bHideDefaultItems As Boolean)
```

The **eContextMenu** parameter indicates the iGrid area the context menu is displayed for.

The row and column index of the object related to the context menu are passed in the **lRowIfAny** and **lColIfAny** parameters. They can contain 0 if the corresponding parameter isn't applicable to the context menu. For instance, the row index makes sense only for real cells, while for column headers only **lColIfAny** will contain the column index and **lRowIfAny** will always equal 0.

The **bCancel** parameter passed by reference can be used to prohibit the displaying of the corresponding context menu at all.

If the context menu is allowed and it contains default items (for instance, these are the copy/paste commands in iGrid's cell context menu), they can be hidden using the **bHideDefaultItems** parameter. If you defined custom items for this menu, only your custom items will be displayed in this case.

Event handlers of this event can be used to redefine the list of custom context menu items dynamically depending on some conditions when a context menu should be displayed (for instance, you can include information regarding the column under the mouse pointer when the column header context menu is displayed). This event is also a good place if you want to display a context menu implemented with an external component.

Note that iGrid still has the **BuiltinContextMenus** property you can use to turn off the built-in context menu for cells in browse mode and the iGrid context menu for column headers in one property assignment. If we compare this property with the **ContextMenuPopup** event, the setting

```
iGrid1.BuiltinContextMenus = False
```

is equivalent to the following event handler sub:

```
Private Sub iGrid1_ContextMenuPopup(…)
   If (eContextMenu = igContextMenuCell) Or _
      (eContextMenu = igContextMenuColHeader) Then
      bHideDefaultItems = True
   End If
End Sub
```

Even if you set **BuiltinContextMenus** to False, iGrid still raises the **ContextMenuPopup** event and may display your custom context menu items defined with the **ContextMenuCustomItems** property.

3. [New] To process clicks on the custom items of iGrid's context menus, use the new **ContextMenuCustomItemClick** event. It provides you with the following information:

```
Event ContextMenuCustomItemClick( _
   ByVal eContextMenu As EContextMenuSource, _
   ByVal lRowIfAny As Long, ByVal lColIfAny As Long, _
   ByVal lCustomIndex As Long)
```

The **eContextMenu** parameter indicates the context menu type (the cell context menu, the column header context menu, etc.). The **lRowIfAny** and **lColIfAny** parameters contain the row and column index of the object the context menu is displayed for (the same values like in the **ContextMenuPopup** event).

The **lCustomIndex** parameter contains the index of the clicked menu item in the collection of the corresponding custom context menu items. For instance, if the user clicks 'Custom Item 1' in the example above, **lCustomIndex** will contain 2 (the first item is a separator).

4. [Enhancement] This release adds the ability to display the context menu for cells using the keyboard (in the previous builds, it can be done only with the right mouse button click). The cell context menu is displayed when the user presses the special Menu key on the keyboard or presses SHIFT+F10, which is the standard key combination for this purpose in Microsoft Windows.

5. [Removed][Code-Upgrade] This version of iGrid allows you to prohibit a context menu of every particular type using the **bCancel** parameter of the **ContextMenuPopup** event. The **bDoDefault** parameter of the **HeaderRightClick** event used for the same purpose in the previous version was removed to avoid duplication of functionality.

6. [Removed][Code-Upgrade] The **TextEditCustomContextMenu** property and the related **TextEditShowCustomContextMenu** event were removed. The functionality provided by those members can be implemented using the new **ContextMenuPopup** event. Here is the typical equivalent code for VB6 used to display a custom context menu when the user is editing a text cell:

```
Private Sub iGrid1_ContextMenuPopup(…)
   If eContextMenu = igContextMenuTextEdit Then
      bHideDefaultItems = True
      PopupMenu mnuCustomTextEditMenu
   End If
End Sub
```

However this approach with VB's **PopupMenu** statement does not work good if the user uses the keyboard to invoke the context menu (SHIFT+F10 or the Menu key). The **PopupMenu** statement above will display the context menu at the current position of the mouse pointer, which can be far enough from the edited cell. We recommend using the new built-in collection of custom context menu items **ContextMenuCustomItems(igContextMenuTextEdit)** instead as iGrid will properly position them in the context menu directly in the edited cell even if the context menu is invoked from the keyboard.

Pay attention to the fact that you can't add custom items to the context menu displayed when a text cell is being edited as this menu is provided by the OS services and can't be modified. You can only fully replace this menu with your custom context menu, and you need to set the **bHideDefaultItems** parameters of the **ContextMenuPopup** event to True for that (otherwise the default system context menu will pop up).

Combo cells: text editing and other improvements

1. [New] The new cell type **igCellTextCombo** allows you to create combo cells with the ability to input any strings not stored in the combo list. Actually cells of this type combine the features of the text (**igCellText**) and combo (**igCellCombo**) cells. For instance, you can start editing of such a cell by pressing an alpha-numeric key or F2, and then open the attached combo list using F4 and select the required string from it. Or you can open the combo list by pressing the cell's combo button with the mouse, and the text editor will be activated automatically allowing you to enter any text as well.

   An **igCellTextCombo** cell automatically supports the so-called autoexpand feature available for combo box fields in Microsoft Access: iGrid searches the values in the combo list to find those that match the entered characters, and automatically places the first underlying value that matches the characters entered so far into the cell. This feature can be turned on/off for a combo list using the new Boolean **ComboObject.AutoExpand** property.

In contrast to the standard Windows combo box control and the corresponding **ComboBox** control in Visual Basic, iGrid does not highlight the first combo list item that starts with the cell text when the combo list is opened – it does that only when an item equals the cell text. This allows you to save the entered cell text "as is" even if you opened a combo list while editing the cell.

2. [New][Enhancement][Removed][Code-Upgrade] The **igComboBtnFlat** and **igCheckBoxFlat** flags in the **ECellTypeFlags** enumeration were used to set the flat look for cell check boxes and combo buttons in the previous versions. However, any grid uses either 3D or flat cell controls, and now you can easily specify that for the whole grid with one new Boolean property named **FlatCellControls** (the default value is False). As a result, the **igComboBtnFlat** and **igCheckBoxFlat** flags are no longer needed, and they were removed from the **ECellTypeFlags** enumeration.
3. [Optimization] Long combo lists (1000+ items) are opened much faster.
4. [Optimization][Fixed] The internal infrastructure of combo lists was redesigned to use fewer GDI resources and to eliminate flickering of the grid control in Microsoft Access when combo lists are opened and closed.
5. [Fixed] Some problems with drawing combo buttons using the hot-track effect were fixed.
6. [Fixed] Combo box cells displayed improper values after inserting new items into the related combo lists or removing items from them.

Loading cell values from/into arrays

This version of iGrid allows you to copy cell values from and into an array using the two new methods, **LoadFromArray** and **LoadIntoArray**. Both 1-dimensional and 2-dimensional arrays can be used for this.

The main advantage of these methods is the performance they provide in comparison to the equivalent traditional loops with **CellValue** calls. For instance, the **LoadFromArray** method executes about 10 times faster than the traditional **CellValue**-based loop for big amounts of data, and subsequent calls of this method for the same grid can give 25x-40x performance gain depending on the data. The main reason why these methods execute so fast is that they access cell values in the internal data storage directly, and the iGrid events related to interactive editing when cell values are changed (such as **BeforeCommitEdit**) are not raised in this case.

The arrays you can pass to these methods are also compatible with Microsoft Office VBA methods and properties, such as the **Range.Value** property in Microsoft Excel that can be used to retrieve or set a range of cells using an array.

1. [New] The new **LoadFromArray** method allows you to copy values from an array into iGrid cells:

```
Sub LoadFromArray( _
   ByVal vStartRow As Variant, _
   ByVal vStartCol As Variant, _
   ByRef vArray As Variant, _
   Optional ByVal bColMajorOrder As Boolean = False)
```

   The **vStartRow**/**vStartCol** parameters specify the row and column in the grid to start population at.

   The **vArray** parameter contains the array used as the data source. If it is a non-initialized array or it has more than 2 dimensions, iGrid raises its internal "Invalid procedure call or argument" error.

   The **bColMajorOrder** parameter is used differently depending on the number of dimensions in the specified array. If the array has 1 dimension and **bColMajorOrder** equals False (the default mode), iGrid places the values from the array into the **vStartCol** column starting from the **vStartRow**/**vStartCol** cell. If **bColMajorOrder** equals True, the array values are used to populate the **vStartRow** row starting from

the **vStartRow**/**vStartCol** cell. For 2-dimensional arrays, if **bColMajorOrder** equals False, the first dimension of the array is used as row index and the second dimension defines columns; otherwise the first dimension is used as column index and the second dimension is used to access rows.

2. [New] The new **LoadIntoArray** method is used to retrieve the values of cells in a rectangular cell range in the form of an array:

```
Sub LoadIntoArray( _
   ByVal vStartRow As Variant, _
   ByVal vStartCol As Variant, _
   ByRef vArray As Variant, _
   Optional ByVal bColMajorOrder As Boolean = False)
```

   The **vStartRow**/**vStartCol** parameters specify the top-left cell of the cell range to copy.

   The **vArray** parameter is used to pass a reference to the destination array the cells will be copied to. Pay attention to the fact that it must be an existing 1- or 2-dimensional array, and its size defines the total number of grid rows and columns that will be copied.

   If a 2-dimensional array is passed to the method and **bColMajorOrder** equals False (the default mode), the first dimension defines the number of rows to copy, and the second dimension defines the number of columns. If **bColMajorOrder** equals True, the first dimension is used as column index and the second dimension is used as row index.

   For a 1-dimensional array, iGrid copies the cells from the **vStartCol** column starting from the **vStartRow** into the specified array if **bColMajorOrder** equals False. If **bColMajorOrder** equals True, the cell values from the row **vStartRow** starting from the column **vStartCol** are copied into the array.

Two notes regarding these methods:

- The array is passed to both methods as a Variant variable, which allows you to pass an array of any data type for processing. In the case of the **LoadFromArray** method, any data type can be copied into iGrid cell values as they are Variants. However, you can get the VB "Type mismatch" error during the call of the **LoadIntoArray** method if the data type of an iGrid cell value isn't compatible with the base data type of the array.
- Both methods automatically process the row text column if the specified array has the corresponding column. For instance, if the grid has 5 main columns (its **ColCount** property equals 5) and you specify a 6-column array when calling **LoadFromArray**, the 6th column of the array will be uploaded into the row text column.

Changes related to the clipboard operations and the DELETE key

All changes described in this section are consequences from the following two key aspects of iGrid:

A. The Cut and Paste operations change the cell value, so they are considered an act of editing cell value – though the phase of interactive editing is absent in this case.

B. iGrid is a tool whose main purpose is to edit and validate entered data, and it should allow the user to enter only correct data – which includes the Cut and Paste operations as well.

The DELETE key functionality is equivalent to the Cut operation, except the fact that the selected cells are not copied into the clipboard. The changes to the DELETE key functionality are also described in this section by this reason.

1. [Enhancement] In the previous builds, the cell contents were cleared unconditionally and the **RequestEdit** and **AfterCommitEdit** events were raised when the user pressed the DELETE key. As of this

build you have the ability to control whether the cell contents should be cleared in the **BeforeCommitEdit** event, which is an integral part of the editing event infrastructure and is raised for this key too. The **AfterCommitEdit** and **CancelEdit** events are also raised now for the DELETE key depending on the result of the action.

To accept the new empty value, use the **eResult** parameter of the **BeforeCommitEdit** event. Set this parameter to **igEditResCommit** to accept or to **igEditResCancel** to reject. If you set **eResult** to **igEditResCommit**, the **AfterCommitEdit** event is raised after clearing the cell contents. In the case of **igEditResCancel** the **CancelEdit** event is raised. Note that in the general case **eResult** can be set to **igEditResProceed,** but this value does not have any effect as interactive editing is absent. To indicate that, the **bCanProceedEditing** parameter of the **BeforeCommitEdit** event is set to False.

All enhancements described above were also implemented for the Cut and Paste commands.

2. [Enhancement] The built-in iGrid logic converts string values entered while interactive editing into the value of the type of the current cell value. This logic is also applied to the Cut and Paste operations now. This enhancement allows you to control what values are pasted into cells and does not allow the user to place improper data in your grids. For instance, now you cannot paste string values that cannot be converted to integers into cells with integer values. The **BeforeCommitEdit** event is also a part of this functionality, so the logic coded in the event handler is also used in these operations.
3. [Enhancement] This build of iGrid allows you to clear the contents of a numeric cell while editing and save this value. In the previous builds the built-in Input Validation message box with the message “Type mismatch” was displayed in this case; now the zero value is placed into the cell. This functionality is also used for the DELETE key and for the Cut command.
4. [Change] In the previous builds, you could copy and paste cells within one grid with all related formatting and other cell properties due to the so-called internal clipboard. This could corrupt data and formatting in the target cells, and in this build only the cell texts are copied and pasted. The cell texts pasted from the clipboard are processed as if they were entered by the user interactively, and the traditional built-in logic with type coercion and new cell value validation in the **BeforeCommitEdit** event is used during this process.
5. [Fixed] Previously the Cut operation and the DELETE key copied the default cell value from the column default cell into the selected cell. Now only the cell value is cleared.
6. [Fixed] Previously pressing the DELETE key copied all the column default cell formatting settings (background color, font, etc.) to the selected cell. Now only the cell value is cleared.
7. [Fixed] The DELETE key did not work in multi-selection mode when iGrid had some selected cells but did not have the current cell.
8. [Fixed] The **sText** parameter of the **RequestEdit** event was not set for the Cut and Paste operations. Assigning values to the **lMaxLength** and **eTextEditOpt** arguments of this event also did not have any effect in these cases.
9. [Fixed] The DELETE key and the Cut and Paste operations did not work for a combo box cell properly: the cell was not updated after these operations, the corresponding combo list item for the new cell value was not found.

Changes in font properties

1. [New][Change][Fixed][Optimization][Code-Upgrade] iGrid 5.0.86 introduced one enhancement for easy adjustment of the default column font: iGrid stored a copy of its **Font** object in the **ColDefaultCell.oFont**

property for a new column when you created it. This feature allowed you to quickly make a column's font bold or italic using a statement like this:

```
iGrid1.ColDefaultCell(1).oFont.Bold = True
```

However, this enhancement caused improper behavior of iGrid if the whole grid font was changed through the **iGrid.Font** property later: cells whose font had not been set did not use the new grid font. For instance, the row text column created during the initialization of iGrid did not reflect changes of the **iGrid.Font** property at all and used the original default system font (MS Sans Serif, 8). Another bad side effect: if you added some columns to the grid, then changed the grid font and created new columns, new cells in the columns created before the font change would use the old grid font.

This enhancement was removed in this release, and now the **ColDefaultCell.oFont** property is not initialized by default again. As a result, you need to modify existing code to upgrade to this build of iGrid: create a new instance of the **StdFont** class and assign it to **ColDefaultCell.oFont** before accessing its sub-properties. For instance, the above code snippet should be converted to

```
Set iGrid1.ColDefaultCell(1).oFont = New StdFont
iGrid1.ColDefaultCell(1).oFont.Bold = True
```

To simplify upgrade, a new **FontClone** function was implemented. It returns a copy of the current grid font in which you can optionally set such properties as **Bold** or **Italic**. The full function declaration is the following:

```
Public Function FontClone( _
   Optional ByVal bBold As Variant, _
   Optional ByVal bItalic As Variant, _
   Optional ByVal bUnderline As Variant, _
   Optional ByVal bStrikethrough As Variant, _
   Optional ByVal sName As Variant, _
   Optional ByVal cSize As Variant _
) As StdFont
```

As you can see, you can set 6 main properties of the **Font** object using the optional parameters. For instance, if you need to use the bold grid font in the second column of your grid, it is enough to execute the following statement before you populate the grid:

```
Set iGrid1.ColDefaultCell(2).oFont = iGrid1.FontClone(bBold:=True)
```

Pay attention to the fact that all parameters have the Variant data type and theoretically can accept values of any type (this is done to have the ability to determine whether a parameter is missing in a call of this method). However we recommend that you pass only appropriate values to avoid the "Invalid property value" error when calling this function. The Hungarian notation used for the parameter names tells you what value types should be used for every parameter: these are Boolean values for the first four parameters, strings for font names, and a numeric value of the Currency data type to specify the font size (the **Size** property of the **StdFont** object uses the same data type).

A lot of system resources are saved due to the fact that font objects of column default cells are not initialized by default, and the more columns and grids you have in your application, the better the effect. In the previous versions, iGrid created a copy of its font object for every column, but now these objects are created by the developer only when they are needed.

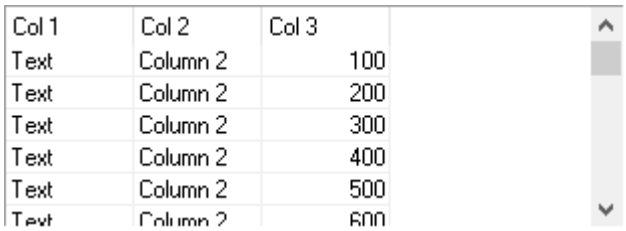
2. [Change][Fixed][Enhancement][Code-Upgrade] All problems with cloning the current grid font described above are also applicable to the **ComboObject.Font** property. In the current version the **Font** property of every new combo object is not initialized by default. This also gives you a new possibility: all combo objects with the **Font** property set to **Nothing** will reflect changes in the grid font immediately with no extra code.

A try to set the **ComboObject.Font** property to **Nothing** generated an error in the previous builds, and this problem was also fixed.

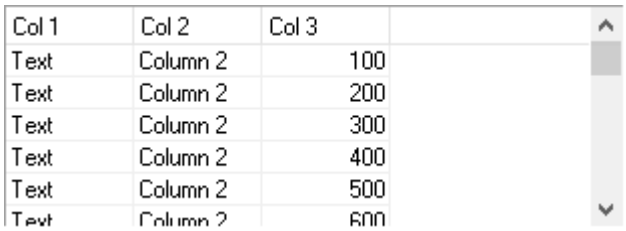
3. [Fixed] Setting the **iGrid.Header.Font** property to **Nothing** caused the Error 91 'Object variable or With block variable not set'. Now a more meaningful iGrid specialized error 514 'Invalid procedure call or argument' is raised in this case to indicate that clearing the header font object is not allowed.
4. [Fixed] Changes of the sub-properties of the **ColDefaultCell.oFont** property or setting it to **Nothing** did not have any effect while creating new cells in some scenarios.
5. [Fixed] The **CellFont** property that returns the effective font used to draw the cell text did not take into account the cell font settings made in the **CellDynamicFormatting** and **RowDynamicFormatting** events.

Windows 10 and new header features

1. [New][Enhancement] iGrid can draw an extra line over the standard header contents in the last line of pixels if the header is drawn using the OS visual style. This feature is mainly used to fix the problem with the iGrid header when it is drawn using visual styles in Windows 10. This OS uses a flat user interface, and the theme of this OS provides us with column headers without a separating line at the bottom:



As you can see, every column header and the first cell beneath it look like one big combined cell if the grid displays vertical and horizontal grid lines (the default setting for most grids). The new extra header bottom line fixes this problem:



The visibility of this line is controlled with the help of the new **BottomLineVisibility** property of the **Header** object property of iGrid. It is a value of the new **EHeaderBottomLineVisibility** enumeration:

```
Enum EHeaderBottomLineVisibility
   igHdrBottomLineAuto
   igHdrBottomLineOn
   igHdrBottomLineOff
End Enum
```

The default value is **igHdrBottomLineAuto**, which implies that the extra line is drawn if the grid is used in Windows 10 or later versions. The **igHdrBottomLineOn** value causes iGrid to draw the extra line regardless of any conditions; **igHdrBottomLineOff** turns the extra line totally off.

The color of the extra header line at the bottom can be set or retrieved using the new **Header.BottomLineColor** property. Its default value is RGB(229, 229, 229), which is the color of the column divider in the standard header drawn with visual styles in Windows 10.

2. [New] 5 new properties can be used to control the colors used to draw column headers when visual styles are turned off in the header. In other words, now you have full control over the colors used to draw the header – which was not possible in the previous versions.

   The new **ButtonEdgeColorOuterLeftTop**, **ButtonEdgeColorOuterRightBottom**, **ButtonEdgeColorInnerLeftTop**, **ButtonEdgeColorInnerRightBottom** properties of the **Header** object allow you to specify the colors used to draw the column header buttons. If the header has a flat look, only the right-bottom edge is drawn, and its color is retrieved from the **ButtonEdgeColorOuterRightBottom** property.

   These colors are also used to draw the extra header part to the right of the last column header. To have consistent look, the colors from the **ButtonEdgeColorOuterLeftTop** and **ButtonEdgeColorInnerRightBottom** properties are used to draw the sort icons when their style is set to 3D triangles.

   When a column header is pressed, its edge is drawn as a rectangle, and the color of this rectangle can be read/set using the new **ButtonEdgeColorPressed** property.

3. [Change][Code-Upgrade] The **Header.Buttons** property no longer affects the header look. In the previous versions the header became flat if this property was set to False and visual styles were not used in the header. Now the flat look is set independently using the existing **Header.Flat** property.

4. [New][Change][Enhancement] The **EHeaderAutoHeightFlags** enumeration contains the new **igHAHOnFontsChange** flag to specify that iGrid should auto-height its header when one of the fonts used in it (**Header.Font**, **Header.SortInfoFont**) is changed. This flag was added to the default value of the **Header.AutoHeightFlags** property.

5. [Enhancement] iGrid supports autoscrolling for interactive column reordering: the grid is automatically scrolled in the horizontal direction when the column header of the currently dragged column is moved outside of the grid control. The speed of scrolling depends on the distance from the cursor to the corresponding grid edge – the more the distance, the faster the speed.

   This automatic scrolling helps users to place columns in desired positions in wide grids much faster as column dragging operation is not limited by the viewport.

   The same progressive autoscrolling approach was implemented for the drag select operation when the user is selecting a rectangular block of cells holding down the mouse button in multiselection mode. The previous versions of iGrid allowed the user to scroll the grid only with a constant speed in this case.

New tools to control modifications in iGrid

The set of built-in tools the user can use to modify iGrid and the corresponding iGrid members were revised in this release. As a result, new members were introduced. The names of the new properties start with “Allow”, which helps the developer to find these properties related to user interaction easier.

1. [New] The new Boolean property **AllowSorting** can be used to disable interactive sorting of the grid. When the default value of this property, True, is changed to False, column sorting is no longer performed when the user clicks a column header. The sort commands are also excluded from the built-in column header context menu.

   The new similar Boolean **AllowGrouping** property is used to disable interactive grouping, which can be done from the built-in context menu for column headers.

   Note that you can disable both operations or only one of them. This allows your users group iGrid without sorting. In this new mode group rows are created without changing the order of rows, and iGrid

can have groups with repeated values. This can be useful in some situations – for instance, if you need to analyze a long list of records ordered chronologically if the row order change is not desirable.

The Group method also has the corresponding new parameter to implement grouping without sorting from code:

```
Sub Group(Optional ByVal bAllowSorting As Boolean = True)
```

Pay attention to the fact that you can also disable column sorting when the user clicks column headers if you set the **Header.Buttons** property to False. However in this case the column headers become non-clickable buttons and the built-in column header context menu with the sort/group commands is still available for the user.

2. [New][Enhancement][Code-upgrade] In the previous versions the user can edit the contents of group rows if the developer did not write an event handler of the **RequestEdit** event to prohibit this action. However, in the vast majority of cases group rows editing should be disabled by default, and this is true in the new iGrid. Now you need to allow editing in group rows explicitly using the new Boolean **AllowGroupRowEditing** property, which is set to False by default.
3. [Enhancement] If the sort type of a column is set to **igSortNone**, the sort and group commands in the built-in column header context menu become disabled for this column.

Other changes and enhancements

1. [New][Enhancement][Removed][Code-Upgrade] In the previous versions, the combination of the **Appearance** and **BorderStyle** properties defined the look of the iGrid border. There were 6 possible combinations of the values of these properties, but they gave only 4 different border types and in some cases the result was non-intuitive. These properties no longer exist, and their functionality was combined into one new property called **BorderType**.

   The new property accepts one of the values from the new **EBorderType** enumeration type:

```
Enum EBorderType
   igBorderNone = 0
   igBorderThinFlat = 1
   igBorderThin3D = 2
   igBorderThick3D = 3
End Enum
```

   The following table lists all combinations of the Appearance and **BorderStyle** property values and the corresponding value from the new **BorderType** property:

BorderStyle	Appearance	BorderType
igBorder3D	igAppearance3D	igBorderThick3D
igBorder3D	igAppearanceFlat	igBorderThinFlat
igBorderThin	igAppearance3D	igBorderThin3D
igBorderThin	igAppearanceFlat	igBorderThinFlat
igBorderNone	igAppearance3D	igBorderNone
igBorderNone	igAppearanceFlat	igBorderNone

2. [New] The new **RowVisibleIndex** property returns the visible order number of the specified row. If a row is hidden, this property returns 0 for it. The similar property for columns, **ColVisibleIndex**, was implemented.

3. [New] This release of iGrid provides you with the new method named **GetOptimalCellHeight**:

```
Function GetOptimalCellHeight( _
      Optional ByVal lTextLineCount As Long = 1, _
      Optional ByVal btImageList As Byte = 0, _
      Optional ByVal btExtraImageList As Byte = 0, _
      Optional ByVal eIconToText As ECellIconToText = igCellIconToTextLeft, _
      Optional ByVal bCheckVisible As Boolean = False, _
      Optional ByVal eCheckPos As ECellCheckPos = igCheckPosLeft, _
      Optional ByVal bComboButton As Boolean = False, _
      Optional ByVal btIndentTop As Byte = 0, _
      Optional ByVal btIndentBottom As Byte = 0, _
      Optional ByVal oFont As StdFont = Nothing, _
      Optional ByVal bHGridLineAllowed As Boolean = True _
   ) As Long
```

   This method is extremely useful if you need to set the default row height to the minimal value that is enough to display the contents of your future cells without clipping before you add new rows to the grid. Generally you call this method after you have made all grid settings using its properties (like **Font**, **GridLines**, **FocusRect**, etc.) but before you create rows:

```
iGrid1.DefaultRowHeight = iGrid1.GetOptimalCellHeight()
```

   The new method allows you to estimate the optimal height of the iGrid cell using its typical constituent items – such as text, icons, combo button. You specify these parts using the optional parameters of the **GetOptimalCellHeight** method.

   This method works similar to the **AutoHeightRow** method, but there is one important difference. **AutoHeightRow** works with real cells, whereas **GetOptimalCellHeight** calculates the best height using a virtual cell.

   Another important point related to this method is that it takes into account the current OS font setting (custom DPI scaling), which allows you to create scale-independent grids.

4. [New][Code-Upgrade] The **AfterCommitEdit** event was supplemented with the new **vOldValue** parameter that contains the previous cell value before it has been changed:

```
Event AfterCommitEdit(ByVal lRow As Long, ByVal lCol As Long, _
   ByVal vOldValue As Variant)
```

   Having this value, you can compare it with the new value of the cell returned by the **CellValue** property and perform or do not perform some operations (for instance, update the underlying database if a field in a table has been really changed).

5. [New] The new **AfterCellCheckChange** event is raised after the user has changed the state of cell check box. It works like the **AfterCommitEdit** event in the cell value change operation, but its purpose is to provide you with the similar functionality for cell check box controls that are not bound to cell values. The event has the following syntax:

```
Event AfterCellCheckChange(ByVal lRow As Long, ByVal lCol As Long, _
   ByVal eOldCheckState As ECellCheckState)
```

   The event has the **eOldCheckState** parameter for the commonality with the **AfterCommitEdit** event. It can be used to determine whether the check state has been really changed (the new value can be provided by the developer in the **CellCheckChange** event, and it may equal the current check state).

6. [New] iGrid raises the new **InputValidationError** event when it is about to display the built-in Input Validation message box, which is displayed if a new cell value cannot be coerced to the type of the current cell value. The event has the following syntax:

```
Event InputValidationError(ByVal lConvErr As Long, ByRef sErrDescr As String)
```

The **lConvErr** parameter contains the corresponding Visual Basic error code generated while applying one of the VB CInt(), CDbl(), CDate(), etc. functions to the new string representation of the cell value. The **sErrDescr** parameter contains the error description that will be displayed in the built-in Input Validation message box.

The second parameter is passed by reference, which gives you the ability to change it before it will be displayed to the user. You can use this feature to provide a more detailed description of the problem for the user, or to localize it.

7. [New][Change][Code-Upgrade] The two new events you can use to track row and column change were implemented:

```
Event CurRowChange(ByVal lNewRowIfAny As Long, ByVal lOldRowIfAny As Long)
Event CurColChange(ByVal lNewColIfAny As Long, ByVal lOldColIfAny As Long)
```

The parameters of the events allow you to know the new and previously selected row/column. The **CurCellChange** event was also supplemented with the new parameters containing the row and column index of the previously selected cell, and its existing **lRowIfAny** and **lColIfAny** parameters were renamed to indicate that these parameters contain the row and column index of the new current cell:

```
Event CurCellChange(ByVal lNewRowIfAny As Long, ByVal lNewColIfAny As Long, _
   ByVal lOldRowIfAny As Long, ByVal lOldColIfAny As Long)
```

8. [New] The optional **lMaximumHeight** parameter was added to the **AutoHeightRow** method. The optional **lMaximumWidth** parameter was added to the **AutoWidthCol** method.

9. [New] The new read-only Boolean **IsEditing** property indicates whether a cell is being edited at the moment. Note that this property can return True not only for text box cells, but for combo box cells as well (if a combo list is opened when you access this property).

10. [New] The read-only Boolean property **IsFocused** was added. It indicates whether iGrid has the input focus. The value of this property can be used, for instance, in cell custom drawing routines to know how to highlight selected cells depending on the grid focused state.

11. [New] The two new properties, **GridLineThicknessH** and **GridLineThicknessV**, can be used to set or retrieve the thickness of the horizontal and vertical grid lines respectively. These properties accept Long values not less than 1.

12. [New] The missing **ComboObject**.**ItemIcon** property to read/set the index of a combo list item icon was implemented.

13. [Change][Enhancement] When the **HighlightSelIcons** property is set to True and iGrid should highlight the icons in selected cells, the color from the **HighlightBackColor** property is used for the highlight effect instead of the black color in the previous versions.

    If the **Header.HotTrackFlags** property contains the **igHdrHotIcon** flag and iGrid should highlight the icon in the column header under the mouse pointer, the **Header.HotTrackForeColor** color is used for the highlight effect instead of the system window default background color (which is white in the vast majority of cases).

    Both improvements give you a more consistent look in the hot column header and selected cells.

14. [Enhancement][Code-Upgrade] The Long **lStartRow** parameter of the **FindSearchMatchRow** method was converted into the Variant **vStartRow** parameter, which allows you to pass the numerical index or string key of a row like you can do that in any other member expecting a reference to a row.

15. [New][Code-Upgrade] In the previous builds of iGrid the **AutoWidthCol** and **AutoHeightRow** methods processed cells only visible by the user at the moment (note that cells in collapsed group rows or cells in columns with width equal to 0 were excluded). Sometimes the developer needs to process all cells, or process all cells in visible rows/columns without excluding cells hidden interactively by collapsing group rows or sizing a column to the width of 0. To specify explicitly what cells should be processed, use the new **eCellVisibility** parameter of these methods:

```
Sub AutoWidthCol( _
      ByVal vCol As Variant, _
      Optional ByVal lMinimumWidth As Long = -1, _
      Optional ByVal lMaximumWidth As Long = -1, _
      Optional ByVal eCellVisibility As ECellVisibilityFilter = _
         igCellVisCurrentlyVisible)

Sub AutoHeightRow( _
      ByVal vRow As Variant, _
      Optional ByVal lMinimumHeight As Long = -1, _
      Optional ByVal lMaximumHeight As Long = -1, _
      Optional ByVal eCellVisibility As ECellVisibilityFilter = _
         igCellVisCurrentlyVisible)
```

The **eCellVisibility** parameter accepts one of the values of the new **ECellVisibilityFilter** enumeration:

```
Public Enum ECellVisibilityFilter
   igCellVisAllCells = 0
   igCellVisCurrentlyVisible = 1
   igCellVisVisibleProperty = 2
End Enum
```

The default value of the **eCellVisibility** parameter in these two methods corresponds the behavior in the previous versions of iGrid.

The **FindSearchMatchRow** method had the **bVisibleRowsOnly** Boolean parameter to control whether to process currently invisible cells, but it did not allow the developer to include cells hidden interactively into the search. To provide this functionality, the **bVisibleRowsOnly** parameter was replaced with the **eCellVisibility** parameter of the **ECellVisibilityFilter** type:

```
Function FindSearchMatchRow( _
      ByVal vSearchCol As Variant, _
      ByVal sSearchString As String, _
      Optional ByVal vStartRow As Varinat, _
      Optional ByVal bLoop As Boolean = False, _
      Optional ByVal eMatchMode As ESearchMatchMode = _
         igSearchMatchStartsWith, _
      Optional ByVal bMatchCase As Boolean = False, _
      Optional ByVal eCellVisibility As ECellVisibilityFilter = _
         igCellVisCurrentlyVisible _
   ) As Long
```

Pay attention to the fact that **bVisibleRowsOnly** parameter was the 4th parameter in the previous version, but its new equivalent has been moved to the last position in this release. The fact is that VB/VBA allows us to pass a Boolean value as the value of a parameter of an enumeration type. As a result, this is not considered a compilation error, and existing code may work improperly with the new version of iGrid and the developer will not get any warning from the compiler regarding that.

16. [Enhancement][Removed][New][Code-Upgrade] When you sorted iGrid in cell mode, it scrolled its contents both in the vertical and horizontal direction if required to show the current cell in the viewport. End users did not like that the grid was scrolled in the horizontal direction so the column they sorted the grid by may have disappeared from the viewport if it did not contain the current cell.

In this version of iGrid the horizontal scrolling position is not changed after sorting – iGrid scrolls its contents only in the vertical direction to ensure that the row containing the current cell remains in the viewport. The **SortScrollToCurCell** property used to turn this behavior on/off in the previous versions was renamed to **SortScrollToCurRow** to correspond to the new behavior.

17. [New] The **CellEffectiveForeColor** and **CellEffectiveBackColor** properties were implemented. These read-only properties return the text color and background color used by iGrid when it draws the cell on the screen. Note that these values are not the current values of the **CellForeColor** and **CellBackColor** properties in the general case. The values returned by **CellEffectiveForeColor** and **CellEffectiveBackColor** are based on all factors related to cell drawing (the **CellForeColor** and **CellBackColor** properties, the **CellDynamicFormatting** and **RowDynamicFormatting** events, the **BackColorOddRows**/**BackColorEvenRows** properties, etc.)
18. [New] The new **InvertSelection** method for inverting selection was implemented.
19. [Enhancement] The titles of the built-in “Paste Operation” and “Input Validation” message boxes were capitalized according to the Microsoft Capitalization Guideline.

Fixed bugs

1. [Fixed] iGrid did not paint odd and even rows correctly with the colors from the **BackColorOddRows** and **BackColorEvenRows** properties after changing the **RowVisibleAsChild** property.
2. [Fixed] The **Sys(igSysRowsVisScrollCount)** call did not return the correct value after changing the **RowVisible** and **RowHeight** properties.
3. [Fixed] The **CancelEdit** event was not raised if the editing has been cancelled in the **BeforeCommitEdit** event.
4. [Fixed] Setting the sort info font property (**Header.SortInfoFont**) changed the font of column header text (**Header.Font**).
5. [Fixed] The grid did not update its vertical scroll bar correctly after changing the height of the header.
6. [Fixed] iGrid crashed if the user pressed the right mouse button in the no-cell area and then released it when the mouse pointer was over a cell.
7. [Fixed] Problems with drawing column headers while doing interactive column reordering were fixed.
8. [Fixed] Bugs in the code used to draw the extra header part when visual styles were off were fixed.
9. [Fixed] The built-in cell and column header tooltips appeared very small (13x5 pixels) and empty if the Large Fonts setting in the OS was turned on.
10. [Fixed] An unwanted horizontal gray line appeared in cell and column header tooltips in Windows 10.
11. [Fixed] The shadow around the cell and column header tooltips disappeared after the tooltip fade animation was complete and the app did not use ComCtl32.dll of the version 6.0 or higher. In the current version the shadow isn’t displayed during the animation as the system tooltips cannot use shadow in this situation at all.
12. [Fixed] The mouse pointer flickered in the column divider area for columns that could not be resized (their **ColAllowSizing** property was set to false).
13. [Fixed] Adding/removing rows dynamically before sorting in the **BeforeContentsSorted** event is processed properly now.
14. [Fixed] The **AutoHeightRow** method did not take into account combo buttons in cells.

15. [Fixed] The cell text editor did not use the dynamic color and font settings made in the **CellDynamicFormatting** and **RowDynamicFormatting** events.
16. [Fixed] The values of the **CellBackColor** and **CellForeColor** properties may have overwritten the cell color settings made dynamically in the **CellDynamicFormatting** and **RowDynamicFormatting** events.
17. [Fixed] The **igTextRTLReading** text flag had no effect when the cell was in edit mode.
18. [Fixed] Missing descriptions of iGrid members for Object browsers were added.