10Tec iGrid ActiveX 4.0
What's New in the Release

Table of Contents

Keywords used to classify changes

- [New] – a totally new feature;
- [Change] – changes in a member functionality or interactive behavior;
- [Improvement] – a feature works better than in a previous version;
- [Removed] – a feature was totally removed;
- [Renaming] – a name of the member was changed so it is enough to rename it in your code.

New Tree and Grouping Functionality

1. [New] This version of iGrid allows you to create and maintain hierarchical structures of rows. This means that each row may have its own level of hierarchy, and the hierarchic relations between rows are determined based on their level values. These level values can be accessed for each row through a new **RowLevel** property of the Byte data type:

   ```
   Get/Let RowLevel(ByVal vRow As Variant) As Byte
   ```

   This property can store levels from 0 up to 255. By default all rows in iGrid has the 0 hierarchy level, and they are considered the top-level hierarchy rows. If you would like to create child rows for those top-level rows, you create rows with the row level equals to 1, etc. Nested rows are displayed with an indent called level indent against to their parent rows.

   These row level data are used by iGrid when we work with group rows created for rows with normal cells or we create a tree-like structure of normal rows. Group rows can work with normal rows this manner without any additional efforts from your side, but to make iGrid a tree-like grid, you need to explicitly specify what column of iGrid will display the tree structure. It is done with a new property called **TreeCol**:

```
Property Get/Let TreeCol As Long
```

It accepts the numeric index of the column to display the tree structure. By default it is 0, and no tree is visible.

When iGrid displays a tree in one of its columns (the **TreeCol** property isn't zero), the users can sort the grid interactively by clicking the column headers or the grid can be sorted programmatically using its methods. In all these cases iGrid keeps the tree structure you created (the parent-child relationships between nodes), and you have the ability to sort your trees using any criteria iGrid provides you with.

2. [New] When working with hierarchical data, it is useful to have plus/minus buttons in the rows of the grid to collapse and expand the nested rows. You specify whether a row has a tree button using the following new property:

```
Property Get/Let RowTreeButton(ByVal vRow As Variant) As ERowTreeButton
```

It accepts one of the following values from a new enumeration called **ERowTreeButton** introduced specially for that:

```
Enum ERowTreeButton
   igRowTreeButtonAbsent = 0
   igRowTreeButtonHidden = 1
   igRowTreeButtonVisible = 2
End Enum
```

The items of this enumeration are used to specify whether a row has the tree plus/minus button and whether it is visible. The **igRowTreeButtonAbsent** item means that a row does not have the tree button at all. The **igRowTreeButtonVisible** item is used to display the tree button (the corresponding indent at the left in the cell in the tree column is used for that). The **igRowTreeButtonHidden** item implies that there is a tree button, but it is hidden; i.e. the cell has only the level indent like in the case of **igRowTreeButtonVisible.** The **igRowTreeButtonHidden** item is used to display the hierarchy level in the cells when the rows do not have child rows and the tree buttons are not needed, but those rows still need level indents for the proper look of the tree.

When you display the tree button in a row, you can already use it to collapse and expand the child rows without additional coding as this feature is built-in functionality in iGrid. iGrid uses the row level data defined with the **RowLevel** property to properly collapse/expand child rows.

The default value of the **RowTreeButton** property is **igRowTreeButtonAbsent**.

3. [New] The iGrid rows have more new properties you can use when working with the tree/grouping functionality. These are:

- **Get/Let RowExpanded(ByVal vRow As Variant) As Boolean**. Specifies whether the row is collapsed (False) or expanded (True). You can expand/collapse a row by assigning True or False to this property. The default value is True.
- **Get/Let RowVisibleAsChild(ByVal vRow As Variant) As Boolean**. Indicates whether the row should be visible depending on the expanded/collapsed state of its parent. In fact, this property is changed by iGrid automatically when you collapse and expand rows, and in the most cases you should not change it by yourself. Note that this property is similar to the **RowVisible** property, but it is not the same. A row may be visible in general in the grid, and **RowVisible** is True in this case, but if this row belongs to a collapsed parent row, its **RowVisibleAsChild** property is changed from True to False, and thus **RowVisibleAsChild** indicates the real visibility state.
- **Get RowChildCount(ByVal vRow As Variant) As Long**. Allows you to know how many child rows (only on the next hierarchy level) the specified row has.

- **Get RowParent(ByVal vRow As Variant) As Long**. Returns the index of the parent row for the specified row or 0 if it does not exist (for the rows with the zero hierarchy level).
- **Get RowLastNested(ByVal vRow As Variant) As Long.** Returns the index of the very last row nested under the specified row. Note that it can be not only a row from the next hierarchy level (a child row), but a row with a deeper level. If there are no nested rows, the index of the specified row is returned.

4. [New] The following new properties related to the general tree/grouping functionality were implemented:

   - **Get/Let LevelIndent As Long**. Specifies the size of the indent for each hierarchy level (in pixels). Default value: 16.
   - **Get/Let TreeButtonSize As Long**. Specifies the size of the tree buttons in the iGrid cells (in pixels). Note: isn't used by the grid when the drawing is performed using visual styles. Default value: 9.
   - **Get/Let TreeButtonBackColor, TreeButtonBorderColor, TreeButtonForeColor As OLE_COLOR**. Specify the background color, the color of the border and the color of the plus/minus signs in the tree buttons when they are drawn not using the current visual style. Default values: **vbButtonFace**, **vbButtonShadow**, **vbButtonText**.

5. [Change] The **AddRow** method became a function and has 5 new parameters: **vRowParent**, **bExpanded, btLevel, eTreeButton** and **bVisibleAsChild.** They contain the parameters of the new row related to the hierarchy structure. The method definition now looks like the following:

```
Public Function AddRow( _
      Optional ByVal sKey As String, _
      Optional ByVal vRowBefore As Variant, _
      Optional ByVal bVisible As Boolean = True, _
      Optional ByVal lHeight As Long = -1, _
      Optional ByVal bGroupRow As Boolean = False, _
      Optional ByVal vRowParent As Variant, _
      Optional ByVal bExpanded As Boolean = True, _
      Optional ByVal btLevel As Byte = 0, _
      Optional ByVal eTreeButton As ERowTreeButton = igRowTreeButtonAbsent, _
      Optional ByVal bVisibleAsChild As Boolean = True, _
      Optional ByVal vTag As Variant, _
      Optional ByVal lCount As Long = 1 _
   ) As Long
```

   The most significant parameter of these new 5 parameters is **vRowParent**. iGrid can automate some works from your side when you create tree data structures, and **vRowParent** in this method is a special key value which is used for that automation. If you specify a parent row in the **vRowParent** parameter, iGrid does the following things automatically to simplify your efforts in creating a tree structure:

   - The position of the new row in the grid is determined automatically. It will be the last child row of the specified row. The **vRowBefore** parameter is ignored if specified.
   - The level of the new row (the **RowLevel** property) will be the level of the parent row plus 1.
   - The **VisibleAsChild** state is set according to the **VisibleAsChild** and **RowExpanded** state of the specified parent row.
   - The tree button in the specified parent row is displayed automatically (the parent row's **RowTreeButton** property is set to **igRowTreeButtonVisible**). The **RowTreeButton** property of the new row is set to **igRowTreeButtonHidden** so the new row automatically has the required level indent to get the consistent look in the tree. These 2 automatic actions related to the tree buttons

are performed only if you set up a tree in your grid (the **TreeCol** property isn't zero) and a new Boolean **AddRowAutoTreeButton** property is set to True (the default value).

If you add a new child row this way, the position of the new row is determined automatically, and you may need this information to access the new row. This position is returned by the **AddRow** function, and can be used like in the following example:

```
Dim lNewRowPos As Long
lNewRowPos = iGrid1.AddRow(vRowParent:="Item-2.1")
iGrid1.CellValue(lNewRowPos, 1) = "Item-2.1.1"
```

Or, in a much shorter form:

```
iGrid1.CellValue(.AddRow(vRowParent:="Item-2.1"), 1) = "Item-2.1.1"
```

6. [Change] The **RemoveRow** method now removes the specified row automatically with all child rows if they exist. In this case the **lCount** parameter is ignored.

7. [Change] The functionality of the **MoveRow** method was enhanced a lot, and in this version it has the following signature:

```
Sub MoveRow( _
   ByVal vRow As Variant, ByVal vRowNewPos As Variant, _
   Optional ByVal lCount As Long = 1, _
   Optional ByVal eMode As EMoveRowMode = igMoveRowBefore)
```

First, now the method supports the **lCount** parameter which allows you to specify a block of rows to move.

Second, the new **eMode** parameter can be used to specify the insert position for the **vRow** row – before (**igMoveRowBefore**) or after (**igMoveRowAfter**) the specified **vRowNewPos** row. The **eMode** parameter can accept one of the following values from a new **EMoveRowMode** enumeration

```
Enum EMoveRowMode
   igMoveRowBefore = 0
   igMoveRowAfter = 1
   igMoveRowAsChild = 2
End Enum
```

, but in fact only the first two items are used when you work with non-tree grids.

And third, the method can be used to rearrange nodes in a tree inside iGrid. In this case, if you move a node specified with the **vRow** parameter, all the nested rows are moved automatically (and the **lCount** parameter is ignored if it is specified). The **igMoveRowAsChild** value of the **eMode** parameter can be used with trees to change the parent node for tree items.

Note that when you move tree nodes, the visibility of the tree buttons for the corresponding parent nodes is changed automatically according to the new state of the tree - to the **igRowTreeButtonHidden** or **igRowTreeButtonVisible** value (see the **RowTreeButton** property).

8. [New] A set of new methods was implemented to automate your work if you wish to collapse or expand all the rows in the grid or all the nested rows on all levels of a specified row. These are:

```
Public Sub CollapseAllRows()
Public Sub ExpandAllRows()
Public Sub CollapseAllChildRows(ByVal vRow As Variant)
Public Sub ExpandAllChildRows(ByVal vRow As Variant)
```

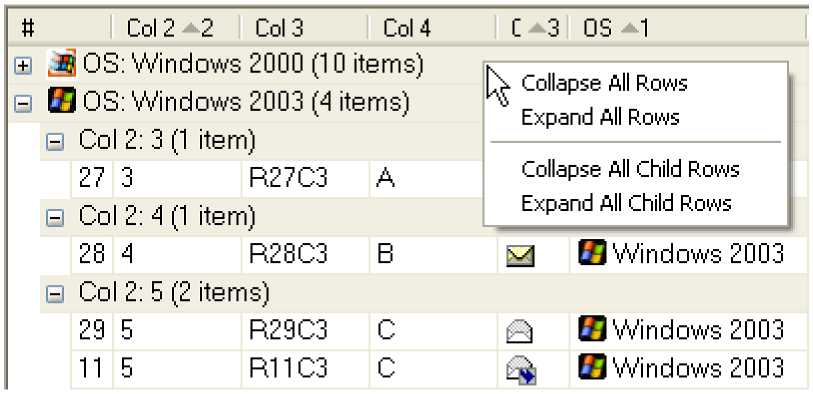
In the case of the last two methods the specified row itself is also collapsed or expanded respectively.

9. [New] You can use a new set of the interactive commands to collapse and expand rows when you work with trees or group rows in iGrid.

First of all, it is double-click you can do with the mouse. When you do that, the row changes its state from collapsed to expanded and vice versa.

From the keyboard, you can use the following 3 new keyboard commands to control the expanded state of a row. To expand a row, use the numpad '+' key. To collapse it, use the numpad '-' key. To expand the current row and all the nested rows, use '*' on the numpad keyboard.

There is also an alternative way to collapse and expand group rows or tree nodes using the mouse. Now iGrid displays a default context menu when you right-click its group row or a row when you have a tree set up:



#	Col 2 ▲2	Col 3	Col 4	C ▲3	OS ▲1
⊞ OS: Windows 2000 (10 items)					
⊟ OS: Windows 2003 (4 items)					
⊟ Col 2: 3 (1 item)					
27	3	R27C3	A		
⊟ Col 2: 4 (1 item)					
28	4	R28C3	B		Windows 2003
⊟ Col 2: 5 (2 items)					
29	5	R29C3	C		Windows 2003
11	5	R11C3	C		Windows 2003

These commands are equivalent to the 4 new methods **CollapseAllRows**, **ExpandAllRows**, **CollapseAllChildRows**, **ExpandAllChildRows** described above. See also a new **UIStrings** property you can use to change the default captions of this context menu's items. If you wish to prohibit the displaying of this context menu, use the **bDoDefault** parameter of the **MouseDown** event like in the following example:

```
Private Sub iGrid1_MouseUp(Button As Integer, Shift As Integer, _
      ByVal x As Single, ByVal y As Single, _
      ByVal lRow As Long, ByVal lCol As Long, _
      bDoDefault As Boolean)

   If Button = vbRightButton Then
      bDoDefault = False
   End If

End Sub
```

Note that you should do that only if the right mouse button was clicked not to prevent iGrid from functioning properly when other mouse buttons are clicked.

10. [Change] The **DblClick** event was changed because of the tree functionality which is built-in now. In the previous versions, the only possible action can be performed after you double-click an iGrid cell – editing. Now iGrid became smarter and it automatically determines whether you wish to collapse/expand the current row when you double-click it. It happens if at least one of the following conditions is true:

    (1) iGrid isn't editable;

    (2) iGrid is in row mode;

    (3) the user clicked a group row;

    (4) the user clicked a cell in the tree column.

iGrid suggests the corresponding action through a new **eAction** parameter of the **DblClick** event which has the following signature now:

```
Event DblClick(ByVal Button As Integer, ByVal Shift As Integer, ByVal x As
Single, ByVal y As Single, ByVal lRow As Long, ByVal lCol As Long, ByRef
eAction As EDblClickAction)
```

In the previous version the **bRequestEdit** Boolean argument was on the place of **eAction**, but now you can use **eAction** to know what action iGrid is about to do and to correct it if required – **eAction** is passed by reference. The **eAction** parameter is of a new **EDblClickAction** enumeration which lists all the possible actions:

```
Enum EDblClickAction
   igDblClickActionNone = 0
   igDblClickActionEdit = 1
   igDblClickActionCollapseExpand = 2
End Enum
```

Don't forget that even if you set **eAction** to **igDblClickActionEdit**, editing is possible only if you double-clicked a real cell (**DblClick** is raised also when you click an empty area outside of real cells) and the grid is editable (the **Editable** property).

Note that in this version of iGrid **DblClick** also has 4 other new parameters which provide you with the general information about this event like the other mouse events – **Button**, **Shift**, **x** and **y**.

11. [New] When you collapse or expand one row interactively by clicking its tree button, with double-click or a keyboard command, two new events are triggered:

```
Event BeforeRowCollapseExpand(ByVal lRow As Long, ByVal bNowExpanded As
Boolean, ByRef bDoDefault As Boolean)
Event AfterRowCollapseExpand(ByVal lRow As Long, ByVal bNowExpanded As
Boolean)
```

The **BeforeRowCollapseExpand** event tells you the current expanded state of the row in the **bNowExpanded** parameter and also allows you to prohibit collapse or expansion through the **bDoDefault** Boolean parameter passed by reference. If the collapse/expansion has been done, the **AfterRowCollapseExpand** event is raised after that and its **bNowExpanded** parameter contains the new expanded state of the row.

Note that these events are not raised if you collapse or expand one row in code (for instance, by changing the **RowExpanded** property). They are also not raised when you collapse/expand a set of rows using such methods as **CollapseAllRows** or do that in the iGrid interface with the cell context menu using such commands as "Collapse All Rows".

12. [New] This version of iGrid introduces a new method called **SortRowChildNodes** you can use to sort the child nodes of the specified tree node or group row:

```
Sub SortRowChildNodes(ByVal vRow As Variant)
```

Note that only the child nodes of the first nested level are sorted, all rows nested deeper are not sorted and remain on their places inside the sorted nodes.

To sort the root nodes, specify zero as the value of the **vRow** parameter.

The sort criteria used to sort the nodes is defined in the **SortObject** property of iGrid like for other methods used to sort the grid contents. This approach allows you to sort the child nodes not only by the column which the tree structure is displayed in, but also by other columns of the grid (multi-column sorting is also allowed).

13. [New] One new iGrid error related to the tree functionality was added. It is "Invalid hierarchy structure" (with the code &H8004020C). This error can be generated by a method that works with hierarchical data (for instance, **SortRowChildNodes**) if it finds a bug in the data structure while doing its work.

New Features Specific for Grouping Only

1. [Renaming] The **ShellSortObject** type was renamed to **SortGroupObject** because now it is also used to define group criteria.
2. [New] Now iGrid has a new **GroupObject** property used to define a group criteria in code. It has the same type as the **SortObject**, and you use the same rules and approaches when working with them. After you have defined a group criteria in the **GroupObject**, you call a new **Group** method to perform actual grouping. As an example, below is a typical code that groups iGrid by the cell values of the first column:

   ```
   iGrid1.GroupObject.Clear
   iGrid1.GroupObject.AddItem 1
   iGrid1.Group
   ```

   Note that if you have sorted iGrid before grouping (i.e. the **SortObject** has some columns with the corresponding sorting settings), iGrid keeps this sort criteria while grouping. In fact, iGrid combines the current sort criteria and the specified group criteria, and the grouped columns become the first columns iGrid is sorted by.
3. [New] As you can see, **GroupObject** has the same type as **SortObject**, and thus any available sort criteria (the **ESortTypes** enumeration) can be used as a group criteria. For instance, you can group iGrid by cell icons using the following code:

   ```
   iGrid1.GroupObject.Clear
   iGrid1.GroupObject.AddItem 1, igSortAsc, igSortByIcon
   iGrid1.Group
   ```

   Remember the following things when you group iGrid by cells with icons. If you use the **igSortByIcon** or **igSortByExtraIcon** criteria for that, only the icons are placed into the group rows (as the **CellIcon** or **CellExtraIcon** properties of the corresponding row text cells respectively). But if you group the grid by another criteria, and the cells from the group column contains icons (not extra icons), they are also placed into the group rows in addition to the text values that represent the groups. The icon of the first cell from a group is used, but if you are using the same icon with the same cell values (this is a normal situation in the vast majority of cases), this feature allows you to have consistent look in the group rows and their cells with no additional efforts from your side.
4. [New] If you group iGrid not by icons, by default iGrid adds the text of the corresponding column header to each group value. The format "<column name>: <group value>" is used for that. A new Boolean property called **PrefixGroupValues** allows you to turn this feature on/off. Its default value is True, but if you wish to place only group values in the group rows, set this property to False before grouping.
5. [New] iGrid raises a new **AfterAutoGroupRowCreated** event each time when a new group row has been created in an operation that creates the group rows automatically (for instance, when you call the **Group** method). The event has the following syntax:

   ```
   Event AfterAutoGroupRowCreated(ByVal lRow As Long, _
      ByVal lItemCount As Long, ByVal vAggrFuncValues As Variant)
   ```

   The **lRow** parameter is the index of the group row that has been just created. The **lItemCount** parameter contains the number of the rows with normal cells that belongs to the created group row. Note that if a group has sub-groups (i.e. other group rows with deeper levels), they are not counted in this value. The **vAggrFuncValues** parameter is used to pass the result values of the aggregate functions defined for iGrid with a new **AggrFuncs** object property described later; it has the Variant **Empty** value if no aggregate functions were defined.

In this event you can access the row text cell of the newly created group row or other cells in the grid to change their properties. For instance, you can modify the color of the group rows created automatically if you need say to highlight each level of hierarchy using a special color for each level. By default iGrid does not display the item count in the automatically created group rows, but the following approach with changing cells in this event can be used to display the item count values in the group rows:

```
Private Sub iGrid1_AfterAutoGroupRowCreated(ByVal lRow As Long, _
   ByVal lItemCount As Long, ByVal vAggrFuncValues As Variant)

   iGrid1.CellValue(lRow, iGrid1.RowTextCol) = _
      iGrid1.CellValue(lRow, iGrid1.RowTextCol) & _
      " (" & lItemCount & IIf(lItemCount = 1, " item)", " items)")

End Sub
```

iGrid creates the automatic group rows by enumerating its rows from bottom to top, and this allows to calculate the item count or access other items of the group that has been just created when this event is raised. However, note that due to this fact, if you access a group row using the **lRow** parameter in this event, later the index of this row can be changed as other group rows can be inserted above.

6. [New] The **AggrFuncs** object property of a new **AggrFuncsObject** type allows you to specify and compute aggregate functions for iGrid columns. The following 4 aggregate functions are available: SUM, MIN, MAX and AVG.

   You define what functions for what columns should be calculated with the **AggFuncs** object. It has a similar set of members as the **SortGroupObject:**

   - the **AddItem**, **RemoveItem** and **Clear** methods used to operate on the specified aggregate functions;
   - the **AggrCol** and **AggrFunc** indexed properties you can access each defined aggregate function and change its properties;
   - and the read-write **ItemCount** property you can use to know and change the number of the defined aggregate functions.

   An aggregate function is specified using one of the corresponding members of a new **EAggrFuncs** enumeration:

```
Enum EAggrFuncs
   igAggrFuncSum = 0
   igAggrFuncMin = 1
   igAggrFuncMax = 2
   igAggrFuncAvg = 3
End Enum
```

   As an example, here is a code snippet you can use to define the SUM aggregate function for the 3rd and 4th columns in a grid iGrid1:

```
iGrid1.AggrFuncs.AddItem 3, igAggrFuncSum
iGrid1.AggrFuncs.AddItem 4, igAggrFuncSum
```

   When you call the **Group** method or the user is grouping the grid using the built-in context menu for the column headers, the results of the defined aggregate functions are passed in the **vAggrFuncValues** parameter of the **AfterAutoGroupRowCreated** event in the form of an array of **Variant** values. This array contains the values corresponding to the defined aggregate functions in the **AggrFuncs** object, and they are indexed starting from 1. In our example with the SUM function for the 3rd and 4th columns, you could display these total values in the group rows using the following event handler of the **AfterAutoGroupRowCreated** event:

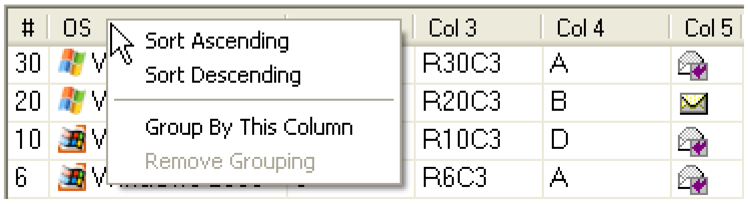
```
Private Sub iGrid1_AfterAutoGroupRowCreated(ByVal lRow As Long, _
      ByVal lItemCount As Long, ByVal vAggrFuncValues As Variant)

   iGrid1.CellValue(lRow, iGrid1.RowTextCol) = _
      iGrid1.CellValue(lRow, iGrid1.RowTextCol) & _
      "  SUM(Col3): " & vAggrFuncValues(1) & _
      "  SUM(Col4): " & vAggrFuncValues(2)
End Sub
```

Note that you can define several aggregate functions for the same column – for instance, calculate the MIN and MAX values for it.

7. [New] If iGrid is grouped by some columns, you can change the sort order of these grouped columns by clicking its headers.

8. [New] iGrid has a new context menu displayed when you right click a column header. This menu can be used to sort and group iGrid by the required columns. The main purpose of this menu is to give the user an interactive way to group iGrid without additional efforts from the programmer's side:



The contents of this menu are changed dynamically depending on the current grouping and the grouping state of the clicked column. The full list of the commands available in the header context menu is the following:

1) "Sort Ascending" – sorts the column which header has been clicked ascending. This sorting is added to the existing sort criteria (the same as if you clicked the column holding down the SHIFT, CTRL or ALT modifier key).

2) "Sort Descending" – the same as previous, but sorts the column descending.

3) "Group By This Column" – group the grid by this column. Appears only if the grid isn't grouped by the column.

4) "Don't Group By This Column" – removes this column from the current grouping. Appears only if the grid is grouped by the column.

5) "Remove Grouping". This command appears always, but it is enabled only if iGrid is grouped by at least one column. It allows you to remove all groups in iGrid (the same as if you issued the **Ungroup** method).

The iGrid header context menu is displayed only when you click a real column header, not the empty area at the right next to the last visible column header. You can select its items using the left or right mouse buttons.

See also a new **UIStrings** property you can use to change the default captions of this context menu's items.

9. [New][Change] To protect iGrid from displaying its default column header context menu, use a new Boolean parameter called **bDoDefault** of the **HeaderRightClick** event:

   ```
   Event HeaderRightClick(ByVal lCol As Long, ByVal Shift As Integer, ByVal x As
   Long, ByVal y As Long, ByRef bDoDefault As Boolean)
   ```

   This parameter is passed by reference, and you set it to False to prohibit the displaying of this context menu.

10. [New] You can remove group rows after you have grouped iGrid using a new **Ungroup** method. The same effect has the **Group** method if you call it on the empty **GroupObject** (with no columns).

11. [New] After you have grouped iGrid by some columns, iGrid contains the information about the current group criteria in an internal object similar to the **GroupObject** property and uses it in some cases to properly support the interactive grouping operations even if you cleared the **GroupObject** or redefined the group criteria in it (for instance, these internal settings are used when the user clicks the column header of a grouped column to change the sort order of the corresponding group rows). If you control your group rows mainly manually from code, you may need to clear this internal object, and a new **ResetEffectiveGroupObject** method does this work.

12. [New] A new Boolean property called **DefaultAutoGroupRowExpanded** can be used to specify the expanded state of the group rows created automatically while grouping the grid. By default it is True, and all the group rows are expanded. If you set this property to False, all the group rows will be created collapsed.

13. [New] A new pair of events, **BeforeContentsGrouped** and **AfterContentsGrouped**, was implemented. They are raised before a grouping operation and after it respectively. Note that clicking the column header of a grouped column in fact causes iGrid to regroup its contents, so these events are also raised in this case.

14. [New] iGrid allows you to group its rows by intervals of values while the grouping columns still contain the real cell values (like MS Outlook - when you group the message list say by the Received column and see groups "Today", "Yesterday", "Two weeks ago", etc while the column still stores the real date-time values). To do that, you need to specify a new **igSortByValueCustomGroups** sort type (the **ESortTypes** enumeration) for the required columns and then substitute the real cell values with their equivalent interval values in a new **CustomGroupValue** event of the following signature:

   ```
   Event CustomGroupValue( _
      ByVal lRow As Long, ByVal lCol As Long, _
      ByRef vGroupValue As Variant, ByRef sGroupText As String)
   ```

   When this event is called, the **vGroupValue** parameter contains the real cell value in the **lRow** row and **lCol** column, but you need to substitute it with your custom value that defines the corresponding group. This custom value also defines the sort order of the new interval groups, but you may not need to show it in the group rows, and because of this you should specify another meaningful title for this group in the **sGroupText** parameter.

   For example, if you need to group a column with numeric values say from 0 to 200 into 3 groups "small", "medium" and "large", your code may look like the following:

```
Private Sub iGrid1_CustomGroupValue(ByVal lRow As Long, ByVal lCol As Long,
vGroupValue As Variant, sGroupText As String)

   Select Case vGroupValue
   Case Is < 50
      vGroupValue = 1
      sGroupText = "Small (< 50)"
   Case Is <= 150
      vGroupValue = 2
      sGroupText = "Medium (50 - 150)"
   Case Else
      vGroupValue = 3
      sGroupText = "Large (> 150)"
   End Select

End Sub
```

Changes Related to the Row Text Column and Row Text Cells

1. [New][Change] This version of iGrid introduces a new paradigm to work with the so called "row text column" which stores the values displayed in the iGrid group rows and row text cells. In the previous version of iGrid you needed to create a special row text column using the **bRowTextCol** parameter of the **AddCol** method, but in this version you do not need to do that because this column is already present in iGrid. This row text column is created automatically when a new instance of iGrid is initialized. To access this column from your code, use the numeric index equal to the total number of normal columns plus one. In the terms of iGrid it is the value of the **ColCount** property + 1, for instance:

   ```
   iGrid1.CellValue(3, iGrid1.ColCount + 1) = "Group row text"
   ```

   To simplify and make this thing more understandable and intuitive, iGrid introduces a new property called **RowTextCol** of the Long data type which always returns the value **ColCount** + 1. Thus, the statement above is equivalent to the following one:

   ```
   iGrid1.CellValue(3, iGrid1.RowTextCol) = "Group row text"
   ```

   As an example, here is a typical VB code that creates a grid with 3 columns and 5 rows, makes the first row a group row, and places the text "Group row" in it:

   ```
   With iGrid1
      .ColCount = 3
      .RowCount = 5
      .RowIsGroup(1) = True
      .CellValue(1, .RowTextCol) = "Group row"
   End With
   ```

   Looking at this concept, the **bRowTextCol** parameter is no longer required, and it was removed from the **AddCol** method.

   The iGrid errors "Attempt to add more than one row text column" (code &H80040205) and "Attempt to add columns after the row text column" (code &H80040209) are also no longer required, so they were removed from iGrid.

2. [New] In the previous version of iGrid you can display row text cells only when row mode is on (the **RowMode** property is True). Now row text cells can be also displayed in cell mode, and you can use all the features available for normal cells including editing with row text cells. To display row text cells in the current version, now it is enough to set the **DrawRowText** property to True. The default value of the **DrawRowText** property was changed to False to prevent iGrid with the default settings from displaying unneeded row text cells if you increase the default row height (when row text cells become visible).

Unicode Editing Support

1. [New] The iGrid cell text editor was rewritten from scratch to support Unicode editing on all Windows NT systems.

Cardinal Cell Structure Improvements

1. [New][Change] This version of iGrid allows you to use more than 1 image list, and the icons from the used image lists may have different sizes. You can use up to 15 different image lists with one iGrid. Each iGrid cell can use icons from different image lists, and you can specify different image lists for a cell's main icon and extra icon. Thus, now you get the ability to display two icons of different sizes in one cell.

   To specify the image lists for iGrid, you should use a new **SetImageList** method. It replaces the **ImageList** property we had in the earlier versions. This replacement also was done because of some VBA-based development environments which do not support Variant properties.

   The **SetImageList** method has the following syntax

   ```
   Sub SetImageList(vImageListStore As Variant)
   ```

   and accepts the same values you could assign to the **ImageList** property:

   ```
   ' We used this in the previous versions:
   ' iGrid1.ImageList = vbalImageList1
   ' The equivalent call in the new version is:
   iGrid1.SetImageList vbalImageList1
   ```

   To upgrade your projects, it is enough to make a global text replacement in which you replace the string ".ImageList =" with ".SetImageList". The same also concerns the **HeaderObject** which uses its own image list.

   To specify several image lists, pass a Variant array of your image list objects instead of one image list:

   ```
   iGrid1.SetImageList Array(imlIcons16, imlIcons32, imlIcons48)
   ```

   To specify what image list is used as the source for the main cell icon, use a new **CellImageList** property. It is a Byte value that accepts values from 0 to 15. 1 means the first image list you specified in the **SetImageList** call, 2 – the second, etc. The default value is 1; it means that the first (or the only) image list specified in **SetImageList** is used. The same is true for the cell extra icon, but you need to use a new **CellExtraImageList** property for that purpose.

   Note that **CellImageList**/**CellExtraImageList** can be zero. In this case the icon/extra icon you specified with the **CellIcon** or **CellExtraIcon** property isn't displayed. It may be useful if you wish to hide your icons temporarily without changing the **CellIcon**/**CellExtraIcon** property – saving it somewhere and then restoring back.

   An example of the use of two icons of different sizes in the iGrid cell is the following:

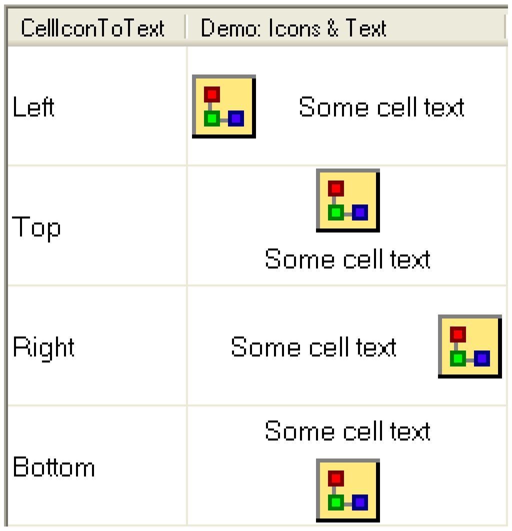
   ```
   iGrid1.SetImageList Array(iml16, iml32, iml48)
   iGrid1.CellImageList(1, 1) = 3
   iGrid1.CellIcon(1, 1) = 1
   iGrid1.CellExtraImageList(1, 1) = 2
   iGrid1.CellExtraIcon(1, 1) = 4
   ```

   In this sample the first cell displays two icons. The main icon is the first icon of the 3rd image list (iml48), and the extra icon is the 4th icon from the 2nd image list (iml32).

   Please, pay attention to one more change in iGrid v4. The cell icons are indexed starting from 1, not 0 as in the previous versions, which is more natural for people.

To specify the default cell image lists, two new Byte properties, **btImageList** and **btExtraImageList**, were added to the default column cell object.

2. [New] Now cell icon(s) can be placed not only to the left of cell text, but also to the top/right/bottom of it:



CellIconToText	Demo: Icons & Text
Left	Some cell text
Top	Some cell text
Right	Some cell text
Bottom	Some cell text

This layout of the cell is controlled through a new **CellIconToText** property. It accepts one of the values of the following new **ECellIconToText** enumeration:

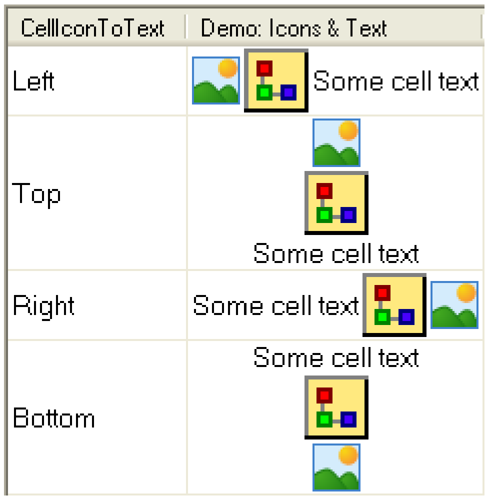
```
Public Enum ECellIconToText
   igCellIconToTextLeft = 0
   igCellIconToTextTop = 1
   igCellIconToTextRight = 2
   igCellIconToTextBottom = 3
End Enum
```

By default **CellIconToText** is set to **igCellIconToTextLeft** that corresponds the behavior of iGrid in the previous versions.

The column default cell object was also supplemented with the corresponding **eIconToText** property you can use to specify the default value for this setting.

If the cell has two icons (the main and extra ones), they are considered as one "combined" icon. In this virtually combined icon the cell icons are placed depending on the **CellIconToText** setting. If it is **igCellIconToTextLeft,** then icons are placed horizontally from left to right – the first is the extra icon, and then the main one. If this setting is **igCellIconToTextTop**, the icons are placed vertically from top to bottom – the main icon under the extra icon. And so on for the other two **CellIconToText** setting.

The screenshot below illustrates this concept. The main cell icon has a yellow background and is bigger than the extra icon with a blue background:



CellIconToText	Demo: Icons & Text
Left	Some cell text
Top	Some cell text
Right	Some cell text
Bottom	Some cell text

3. [New][Change] In this version the contents of the cells are aligned using two new properties, **CellAlignH** and **CellAlignV**. They accept one of the values from the following two new enumerations, **EHAlignment** and **EVAlignment**, respectively:

```
Public Enum EHAlignment
   igAlignHLeft = 0
   igAlignHCenter = 1
   igAlignHRight = 2
End Enum

Public Enum EVAlignment
   igAlignVTop = 0
   igAlignVCenter = 1
   igAlignVBottom = 2
End Enum
```

As a result, the following flags which did this work in the earlier versions were removed from the **ETextFormatFlags** enumeration: **igTextTop**, **igTextLeft**, **igTextCenter**, **igTextRight**, **igTextVCenter**, **igTextBottom**.

In the previous version of iGrid the icon was always drawn in the top-left corner of the cell, but now you have more options to specify its position with the **CellAlignV**/**CellAlignH** properties.

If the cell has some text, the cell icon is displayed at the edge of the cell specified in the **CellIconToText** property, but it is aligned vertically or horizontally together with the cell text according to the **CellAlignV** or **CellAlignH** setting. If **CellIconToText** is set to the "left" or "right" options, the cell vertical alignment is applied to the icon; if **CellIconToText** is "top" or "bottom", then the horizontal alignment is in effect.

If the cell has no text, you have even more control and can place the cell icon into any of the 9 possible positions defined by the horizontal and vertical alignment options in the **CellAlignH**/**CellAlignV** properties. The **CellIconToText** setting isn't taken into account in this case.

If the cell has two icons (the main and extra ones), they are considered as one "combined" icon, and the same alignment rules are applied to it. The main and extra icons inside this combined one are placed according to the setting specified in the **CellIconToText** property.

The **CellAlignH**/**CellAlignV** options are also applicable to the check box cells. For the cells of this type they define the position of the check box control. In the current version the check box control is top- and left-aligned in the cell by default, and if you need to have its position like in the previous version (centered horizontally and top-aligned), you need to specify the **igAlignHCenter** horizontal alignment in the **CellAlignH** property.

4. [New][Change] The changes in the cell contents alignment (the **CellAlignH**/**CellAlignV** properties functionality) were also implemented for the column headers, but you should use the equivalent **ColHeaderAlignH**/**ColHeaderAlignV** properties for them. The **AddCol** method was also supplemented by two new optional arguments, **eHeaderAlignH** and **eHeaderAlignV**, you can use for that purpose when you define a new column.

   Pay attention to the fact that when you specify a vertical alignment for the column header text, the column header icon (if it was defined) is aligned vertically accordingly; if there is no column header text, the column header icon can use the horizontal alignment like in the iGrid cells.

   The third part of the column header, sort info (the sort icon and possible sort key text), does not use the specified column header alignment. It is always centered vertically and displayed at the right edge of the column header - in contrast to the previous versions when it was top-aligned vertically and displayed just after the column header text or icon if the text is absent. This allows us to have a better and standard look for all possible alignment options for the iGrid column header.

5. [New] In the earlier versions of iGrid you could not align cell or column header text vertically unless you specify the **igTextSingleLine** format flag, and thus vertical alignment was not possible for multiline text. In the current version there is no such a restriction. You can use any **EHAlignment**/**EVAlignment** options with any cell or column header text contents.

   Just one performance tip. If you are using single-line text and you wish to align it vertically, iGrid may draw its contents a little bit faster if you specify the **igTextSingleLine** flag.

6. [New][Change] Now iGrid allows you to adjust the left/top/right/bottom indents in its cells (the previous version allowed you to do that only for the left and right indents). It is done with the **CellIndentLeft**/**CellIndentTop**/**CellIndentRight**/**CellIndentBottom** properties of iGrid. The **CellIndentLeft** and **CellIndentRight** properties correspond to the old **CellIndent** and **CellRightIndent** properties; **CellIndentTop** and **CellIndentBottom** are new to this version. The corresponding properties of the column default cell object (see the **CellObject** type) are called **btIndentLeft**, **btIndentTop**, **btIndentRight** and **btIndentBottom** respectively; **btIndentLeft** and **btIndentRight** replaces **iIndent** and **iRightIndent** in the previous version.

   The data type of the cell indent properties was changed from Integer to Byte – that is why the cell indent properties of the **CellObject** has the "bt" prefix.

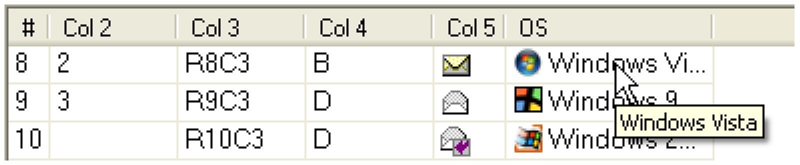
   The default values for the cell left and right indents are set to 1 instead of 2 in the previous version.

7. [Enhancement] Now iGrid uses the cell space more effectively when you turn off the grid lines and/or focus rectangle. In the previous version the grid lines were simply hidden and there was 1-pixel unused space at the right and bottom part of the cell. The same concerns the focus rectangle – if you turn it off, you have an additional 1-pixel space near each edge of the cell.

New Built-in Cell and Column Header Tooltips

1. [New] This version of iGrid has a built-in mechanism that displays the full text of a cell in a tooltip if the cell text is not fully visible by some reason. For instance, it may happen if the cell is not enough wide to display the cell text. iGrid also displays such a tooltip even if the cell text isn’t truncated by the cell's boundary, but it is partially hidden by the edge of iGrid or its scroll bars.

   The full cell text is displayed in a native Windows tooltip window when the mouse pointer is stationary within such a cell for a period of time:



#	Col 2	Col 3	Col 4	Col 5	OS
8	2	R8C3	B	✉	Windows Vi...
9	3	R9C3	D	✉	Windows 9
10		R10C3	D	✉	Windows 2...

Windows Vista

The same tooltips are displayed for the iGrid column headers with truncated texts. All the facts described for the cell built-in tooltips here are true for the column header tooltips.

2. [New] You can control what text will be displayed in the tooltip window, and whether it will be displayed at all. This is done through a new **RequestCellToolTip** event which has the following syntax:

```
Event RequestCellToolTip( _
   ByVal lRow As Long, ByVal lCol As Long, _
   ByRef sTipText As String, _
   ByRef eTipIcon As EToolTipIcon, ByRef sTipTitle As String)
```

This event is raised each time when the pointer enters a cell (practically the same as the **MouseEnter** event), and the **sTipText** parameter of this event contains the string which will be displayed in the tooltip window. If the cell under the mouse pointer is truncated, **sTipText** contains the full text of the cell which will be displayed in the tooltip; if not, then this parameter is an empty string, and the tooltip window won't be displayed.

This approach with the **sTipText** parameter allows you to change the tooltip text on the fly if you need that. Note also that the **RequestCellToolTip** event is raised for all cells, not only for the cells with truncated texts, and this fact can be used to display your own tooltips for all cells or prohibit the displaying of the tooltip window at all if you set the **sTipText** parameter to an empty string in a handler of this event.

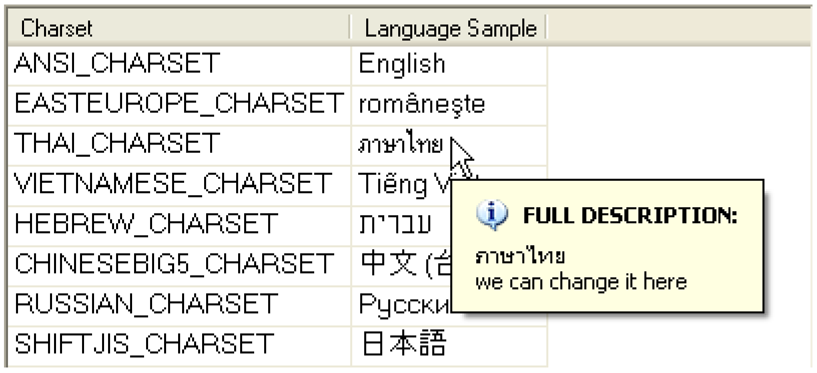
Pay attention to the last two parameters of the **RequestCellToolTip** event. They are used to add one of the standard Windows icon and a title to your tooltip.

The title for the tooltip text is specified in the **sTipTitle** parameter. It is a string displayed as bold above the tooltip text. The **eTipIcon** parameter can be a value of the following new enumeration **EToolTipIcon**:

```
Enum EToolTipIcon
   igToolTipIconNone = 0
   igToolTipIconInfo = 1
   igToolTipIconWarning = 2
   igToolTipIconError = 3
End Enum
```

The names of the members are self-descriptive and they mean the corresponding standard built-in Windows icons. By default no icon is displayed, and this parameter is set to **igToolTipIconNone**.

Note that if you wish to display one of the standard Windows icons in your tooltip, you should assign a title string to the **sTipTitle** parameter. As an example, here is a screenshot of an extended tooltip you can display for all cells (even non-truncated) using this built-in feature:



Charset	Language Sample
ANSI_CHARSET	English
EASTEUROPE_CHARSET	românește
THAI_CHARSET	ภาษาไทย
VIETNAMESE_CHARSET	Tiếng V
HEBREW_CHARSET	עברית
CHINESEBIG5_CHARSET	中文 (台
RUSSIAN_CHARSET	Русски
SHIFTJIS_CHARSET	日本語

FULL DESCRIPTION:
ภาษาไทย
we can change it here

Here is the equivalent code:

```
Private Sub iGrid1_RequestCellToolTip( _
     ByVal lRow As Long, ByVal lCol As Long, _
     sTipText As String, eTipIcon As EToolTipIcon, sTipTitle As String)

   sTipText = iGrid1.CellValue(lRow, lCol) & vbCrLf & "we can change it here"
   eTipIcon = igToolTipIconInfo
   sTipTitle = "FULL DESCRIPTION:"

End Sub
```

The same things are applicable to the column header tooltips, except the fact that iGrid raises the **RequestColHeaderToolTip** event for them:

```
Event RequestColHeaderToolTip( _
   ByVal lCol As Long, ByRef sTipText As String, _
   ByRef eTipIcon As EToolTipIcon, ByRef sTipTitle As String)
```

3. [New] Two new iGrid properties, **ToolTipDelayTime** and **ToolTipVisibleTime**, can be used to adjust some time parameters of the built-in cell tooltips. These are Long and specify values in milliseconds.

   The **ToolTipDelayTime** property returns/sets the amount of time the mouse pointer must remain stationary within a cell's bounding rectangle before the tooltip window appears. The **ToolTipVisibleTime** property returns/sets the amount of time the tooltip window remains visible if the pointer is stationary within the cell.

   Both properties support the "-1" value which means that the corresponding default system parameters are used. The default values for the **ToolTipDelayTime** and **ToolTipVisibleTime** properties are -1 and 20000 respectively - this means that the tooltip window for will be displayed after the default system period of time and will remain visible for 20 second.

4. [New] To determine whether the text in a cell or column header is clipped the way iGrid does that, you can use the following two new Boolean functions of iGrid:

```
Public Function IsCellTextClipped( _
   ByVal vRow As Variant, ByVal vCol As Variant, _
   Optional ByVal bCheckPartiallyHidden As Boolean = True) As Boolean

Public Function IsColHeaderTextClipped(ByVal vCol As Variant, _
   Optional ByVal bCheckPartiallyHidden As Boolean = True) As Boolean
```

   If the **bCheckPartiallyHidden** parameter of these functions is set to False, iGrid checks whether the cell or column header text is clipped by the cell's or column header's boundary only. If you set this parameter to True, iGrid also checks whether the text is partially visible because it may be hidden by the edges of iGrid or its constituent controls (such as scroll bars). In fact, when iGrid needs to determine whether the tooltip for a cell or column header should be displayed, it calls the corresponding function internally setting the **bCheckPartiallyHidden** parameter to True.

5. A couple of final notes regarding this built-in cell tooltips functionality.

   First, some ActiveX host containers may provide the iGrid control with their own tooltip property (for instance, in Visual Basic 6 you can find the **ToolTipText** property when iGrid is on a VB6 form). We do not recommend to use the built-in tooltip functionality together with such tooltip properties because you may have two tooltip windows visible in the control at the same time. If you prefer to use the built-in functionality, don't use the host's tooltips; if you need to display only the host's tooltips, suppress all the built-in tooltips by setting the **sTipText** parameter of the **RequestCellToolTip** event to an empty string as it was described above.

   Second, the tooltip text may consist of several lines of text. If you need a line break in your tooltip text, specify the **vbCrLf** constant in the required place in the **sTipText** parameter.

Drawing and Behavior Improvements

1. [Improvement] The drawing of the iGrid column headers was enhanced. In the new version the space is used more effectively (you can see more info in the same space), the sort info has priority over icon and text, and the sort info isn't hidden when there is no space to fully draw it.
2. [Improvement] The behavior of iGrid in virtual mode was enhanced a lot. First of all, now the vertical scroll bar represents the real number of already requested rows, and iGrid doesn't request new rows as you are dragging the thumb box down. The latter fact also caused a bad side effect in the previous versions when iGrid had many rows – it might start to request new rows by big portions, and in some cases you even could not stop this until the last row was requested. Now new rows are requested only when the user reached the last row and presses the DOWN ARROW or PAGE DOWN key, or is trying to scroll more clicking the down button on the vertical scroll bar (the equivalent effect has the mouse wheel rotation). In each case the corresponding number of new rows is requested in one portion. For instance, when the user presses DOWN ARROW or the down button on the scroll bar, only one row is requested; for the PAGE DOWN key it is the number of rows with the default row height which can be displayed in the visible part of iGrid (i.e. in one page).
3. [Improvement] The **AutoWidthCol** and **EvaluateTextWidth** methods were optimized and now work faster. The same speed improvement you see when you double-click a column divider to automatically fit the width of the column. In all these operations the width for the cells with multiline text is calculated accurately and thus more precisely than in the previous versions.
4. [Improvement] The behavior of the PAGE UP and PAGE DOWN keys became smarter, and now they work practically like the same keys in the file list pane of the Windows Explorer. For instance, if you press the PAGE DOWN key, iGrid checks whether the last fully visible row in the viewport is selected. If this is not the case, iGrid just selects it without any vertical scrolling; otherwise iGrid scrolls the contents up so this row becomes the first fully visible row and selects the new last fully visible row at the bottom. The similar behavior is used for the PAGE UP key.
5. [Improvement] To better separate cells visually when a row has row text cell, the vertical grid lines for the normal cells above the row text cell are drawn when the **GridLines** property is set to **igGridLinesVertical**. In the previous version they were drawn only when **GridLines** was **igGridLinesBoth**.

Other Changes

1. [New] You can change and/or localize all the text strings iGrid displays for the user. These are the items of the header's and cell's default context menus and the error message box displayed during the built-in input validation.

   This is done through a new **UIStrings** property of iGrid. "UIStrings" stands for "User Interface Strings" and has the following syntax:

   ```
   Property Let/Get UIStrings(ByVal lStringIndex As Long) As String
   ```

   The **UIStrings** property is indexed by the numeric values which means the corresponding UI strings. iGrid supports the following 11 UI strings:

Index	UI String
The iGrid header's default context menu	
1	"Sort Ascending"
2	"Sort Descending"
3	"Group By This Column"
4	"Don't Group By This Column"
5	"Remove Grouping"
The iGrid cell's default context menu	
6	"Collapse All Rows"
7	"Expand All Rows"
8	"Collapse All Child Rows"
9	"Expand All Child Rows"
The built-in input validation error message box	
10	"Invalid value:" (the beginning part of the message box text; the ending one is the conversion error text got from the VB runtime)
11	"Input validation" (the caption of the message box)

   Thus, for example, if you wish to change the first item in the cell's default context menu from "Collapse All Rows" to "Collapse All Nodes", your code will look like the following:

   ```
   iGrid1.UIStrings(6) = "Collapse All Nodes"
   ```

2. [New][Change][Removed] iGrid no longer supports the **Redraw** property used to turn off the redrawing of the grid while you are performing a series of changes in the grid. Now you should use the **BeginUpdate**/**EndUpdate** calls for that. These new methods should be used instead of setting the **Redraw** property to False and True respectively, for example:

   ```
   With iGrid1
      .BeginUpdate ' instead of .Redraw = False in the previous versions

      .ColCount = 6
      .RowCount = 30
      .CellValue(1, 1) = "Some text"

      .EndUpdate ' instead of .Redraw = True in the previous versions
   End With
   ```

   But these new methods can do more: the calls of these methods can be nested - what is not possible with the simple **Redraw** property. This means that all **BeginUpdate** calls should be paired with the corresponding **EndUpdate** calls, and only after that the grid is redrawn. This is very useful if you have some subs that can update the grid, and these subs can be called from other subs which also update the grid. Then you can have **BeginUpdate**/**EndUpdate** in all these subs, and the grid will be updated only after the last **EndUpdate** method is invoked.

To know whether iGrid is currently redrawn, use a new Long read-only property called **BeginUpdateCount**. In fact, it returns the number of currently active **BeginUpdate** statements that were not closed with the corresponding **EndUpdate** calls. iGrid is redrawn if **BeginUpdateCount** equals zero, the grid is redrawn; if not, then all your updates are cached until you call the last **EndUpdate** method. The **BeginUpdateCount** property is the equivalent for the same **Redraw** property in the older versions which may also return a Boolean value indicating whether the grid is redrawn now.

To upgrade your projects, it is enough to make a global text replacement: you need to replace the assignment statements ".Redraw = False" and ".Redraw = True" with the ".BeginUpdate" and ".EndUpdate" calls respectively. Note that it should be exactly the assignments but not expressions when you read and compare the value of the **Redraw** property like in the statement "If iGrid1.Redraw = False Then". These calls when you read the value of the **Redraw** property should be replaced on something like that: "If iGrid1.BeginUpdateCount > 0 Then".

3. [New] The **SortGroupObject** class used to define sort and group criteria implements the following new methods and properties which simplify its usage.

   First, a **Clear** method was added to clear the contents (all defined columns) in a **SortGroupObject**.

   Second, now you can add new columns to sort/group by using a new **AddItem** method with the following syntax:

   ```
   Sub AddItem( _
      ByVal vCol As Variant, _
      Optional ByVal eSortOrder As ESortOrders = igSortAsc, _
      Optional ByVal eSortType As ESortTypes = igSortByValueGeneral)
   ```

   For instance, if you wish to define a group criteria when iGrid is grouped by the 1st column ascending and the 4th column descending, you can use a block of code like this:

   ```
   With iGrid1.GroupObject
      .Clear
      .AddItem 1
      .AddItem 4, igSortDesc
   End With
   ```

   Note that string column keys are also supported as the values of the first parameter of this method.

   Third, you can use a new **RemoveItem** method to remove the specified item from a **SortGroupObject**:

   ```
   Sub RemoveItem(ByVal lSortIndex As Long)
   ```

   Note that you specify what item of the **SortGroupObject** you remove but not the index of a column from the grid. If you need to find the index of the specified grid column in the **SortGroupObject**, use the following fourth new method called **FindCol**:

   ```
   Function FindCol(ByVal vCol As Variant) As Long
   ```

   This function returns the position of the specified grid column in the sort/group criteria defined in the **SortGroupObject**, or 0 if it cannot be found.

   And last, you can create a full copy of the required **SortGroupObject** with a new **Clone** method:

   ```
   Function Clone() As SortGroupObject
   ```

   It can be useful, for instance, if you need to store in a variable the current sort criteria and then restore it later. Note that it is not allowed to use a **SortGroupObject** defined for one iGrid with another iGrid.

4. [New] The **Sort** method was supplemented with two new parameters, **vRowStart** and **vRowEnd**, which can be used to specify a subrange of the iGrid rows to sort. The updated syntax of this method is the following:

```
Sub Sort( _
   Optional ByVal vCols As Variant, _
   Optional vRowStart As Variant, _
   Optional vRowEnd As Variant _
)
```

If you omit the start row parameter, iGrid is sorted from the first row; if you omit the end row parameter, the last iGrid row is implied. Both parameters are optional, so it is possible to specify only one of them.

In the previous version you can sort the rows between group rows independently, but in the current version this feature was turned off (to optimize the performance) when you specify **vRowStart** and/or **vRowEnd**.

The same changes were made in the **DoDefaultSort** method which has the following updated syntax:

```
Sub DoDefaultSort( _
   ByVal vCol As Variant, _
   Optional ByVal iShift As Integer = 0, _
   Optional ByVal eSortOrder As ESortOrders = -1, _
   Optional vRowStart As Variant, _
   Optional vRowEnd As Variant _
)
```

5. [Change] The type of the **RowHeight**, **RowNormalCellHeight**, **DefaultRowHeight** and **DefaultRowNormalCellHeight** properties was changed from Long to Integer. Some internal structures that store the information about each row were also compacted. As a result of all these changes, the current version of iGrid uses less memory to store the rows data.

6. [New] Two new read-only properties, **RowStartY** and **ColStartX**, were added. They allow you to know the Y-coordinate of the top edge of the specified row and the X-coordinate of the left edge of the specified column respectively. These are absolute values calculated from the very first row/column, and they do not depend on the current scrolling position. In fact, the **RowStartY** value is the sum of the heights of all the visible rows before this row (the same concerns **ColStartX**).

   Note that the Y-coordinate of the top edge of the first row is always 0 even if the header is visible. This is useful if you need to use the **RowStartY** values for custom scrolling algorithms when you assign these values to the **VScrollBar.Value** property. In this case, if you need to display a row at the top of the grid, it is enough to assign the **RowStartY** value for this row to **VScrollBar.Value**.

7. [New] A new **DoKeyboardCommand** was implemented. It allows you to imitate the iGrid default actions as if the user pressed keys on the keyboard. The method has the following definition:

```
Sub DoKeyboardCommand(ByVal KeyCode As Integer, Optional ByVal Shift As
Integer)
```

   Its parameters are used to specify the code of the key to imitate (**KeyCode**) and optionally the state of the modifier keys (**Shift**).

   You pass the same values as you get in the **KeyDown** event using these parameters. For instance, if you need to imitate the behavior of the PAGE DOWN key, use a statement like this:

```
iGrid1.DoKeyboardCommand vbKeyPageDown
```

8. [Change] The functionality of the **RowIndex** and **ColIndex** functions was changed a little bit. In the previous versions they raised the "Subscript out of range" error if you passed a non-existing row or column key into them, now just zero is returned in this case.

   Due to this feature your code will look simpler if you need to test whether a row or column with the required key exists. In the previous version you should have written:

```
Dim lRowIndex As Long
On Error Resume Next
lRowIndex = iGrid1.RowIndex("row_key")
If Err.Code = 0 Then
   ' Do something with the lRowIndex row
End If
On Error GoTo 0
```

Now it looks much simpler:

```
If iGrid1.RowIndex("row_key") <> 0 Then
   ' Do something with the lRowIndex row
End If
```

9. [Change] In the previous versions of iGrid you needed to specify the **igTextWordBreak** flag from the **ETextFormatFlags** enumeration if you wish to enable multi-line editing in text cells. Now you do not need to do that – iGrid allows you to enter the multi-line text without this option when you press CTRL+ENTER.

   The cells and column headers are not formatted with the **igTextSingleLine** flag now. This allows you to display the CR/LF characters in their texts as line breaks without additional settings (if you inserted them into the text in code for instance with the **vbCrLf** constant or manually while editing by pressing CTRL+ENTER). To suppress these line breaks (like in the previous versions by default), you should use this formatting constant.

10. [New] A new **SelectionAlphaBlend** property was implemented. It allows you to control whether the selection in iGrid will be semi-transparent. This property has the Byte data type and it stores the level of transparency of the selection color. By default, the value of this property is 255 which means the selection is not transparent. If you decrease this value, the selection becomes transparent – the less the value, the more the transparency. The 0 value means the selection is fully transparent.

    Note that alpha-blend selection functionality is available only if your video card outputs a true-color signal (16- or 32-bit) which is ok for all modern video cards. You can test whether this feature is available using a new **igSupFeatAlphaBlendSel** member from the **ESupportFeatures** enumeration in the **Supports** function. Pay also attention to the fact that if you expect your users can change the bits per pixel parameter for your display, iGrid does not update this parameter (because there is no good non-consuming resource method for that) and you need to restart iGrid to update this value. You can do that when one of your top-level windows receives the WM_SETTINGCHANGE or WM_DISPLAYCHANGE Windows message.

11. [New] When iGrid draws a cell icon, there is a small gap between the icon and the cell text. By default it is a gap of 2 pixels, but now you can change it using a new Long **CellIconGap** property. This setting affects the gap next to the cell extra icon as well.

12. [New] A new **SelectAll** method was implemented you can use to select all cells (or rows in row mode) in one statement.

13. [New] Now iGrid allows you to create new rows (by increasing the **RowCount** property or calling the **AddRow** method) even if you have not defined any column, and never generates the error "Attempt to add rows with no columns" (the error code is &H80040204) in such cases. This is possible because in the current version you always have one invisible row text column and you can access its cells even if you do not have real columns. However, you are still advised to create the column set first before you create new rows as iGrid works much faster in this case.

14. [New] You can specify the default sort order for a column using a new **ColSortOrderDefault** property:

```
Property Get/Let ColSortOrderDefault(ByVal vCol As Variant) As ESortOrders
```

By default, iGrid sorts a column ascending, but using this property you can specify that the column should be sorted descending when the user sorts it and it has not been sorted on that moment.

You can also make this setting when you create a new column with a new **eSortOrderDefault** parameter of the **AddCol** method.

This setting is taken into account when you sort the grid by calling its **Sort** method when you specify the set of columns to sort as its first **vCols** parameter. When you do that, the sort order isn't specified, and the corresponding **ColSortOrderDefault** value is used for each specified column.

15. [New] A new **ScrollBarBeforeScroll** event was implemented:

```
Event ScrollBarBeforeScroll( _
   ByVal eBar As EScrollBar, _
   ByVal eScrollReason As EScrollReasons, _
   ByVal bMouseWheel As Boolean, _
   ByRef lNewValue As Long, _
   ByRef bDoDefault As Boolean _
)
```

This event is raised each time when the user is trying to scroll the iGrid contents and allows you to prohibit the current scrolling operation if required. The **eBar** parameter of this event indicates what scroll bar is used for the current scrolling operation, and the **eScrollReason** parameter tells you how the user is about to scroll the control. **eScrollReason** takes one of the following values from a new **EScrollReasons** enumeration:

```
Public Enum EScrollReasons
   igScrollReasonLineUpOrLeft = 0
   igScrollReasonLineDownOrRight = 1
   igScrollReasonPageUpOrLeft = 2
   igScrollReasonPageDownOrRight = 3
   igScrollReasonThumbPosition = 4
   igScrollReasonThumbTrack = 5
End Enum
```

The first two items are used when the user clicks the arrow buttons on a scroll bar, the third and fourth items are used when the user clicks the shaded shaft between the thumb box and an arrow button on a scroll bar. The **igScrollReasonThumbPosition** and **igScrollReasonThumbTrack** items are used to indicate that the user finished the scrolling using the thumb box or is doing that respectively.

The next parameter of the event, **bMouseWheel**, can be used to know whether the user is scrolling the grid using the mouse wheel. Such a scrolling operation is equivalent to the scrolling with the help of the scroll bars, so the **eScrollReason** parameter is still can be used to know in what direction and how iGrid is being scrolled. Just note that not all mouse drivers generate the corresponding system message (WM_MOUSEWHEEL) while scrolling with the mouse wheel, especially non-Microsoft ones, so **bMouseWheel** parameter may be False even if you scroll the grid with the mouse wheel.

The **lNewValue** parameter allows you to know the new position of the thumb box. It is the future value of the **Value** property of the **VScrollBar** or **HScrollBar** object property if the current scrolling operation will be done. This parameter is passed by reference so you can correct it on the fly in this event if required (for instance, if you implement your own scrolling algorithm).

The **bDoDefault** parameter is passed by reference and allows you to prohibit the current scrolling operation - change its default True value to False for that.

16. [New] A new event **TextEditStarted** was implemented. It is raised after the editing of a text box cell has been started. iGrid uses the native Windows text box control to edit its text cells, and this event is raised

when this control has been initialized and displayed so you can access its WinAPI handle through the **TextEditHwnd** property. If you need to do some manipulations with it before the user is editing the text (for instance, set some specific WinAPI flags), this event is the best place to do that.

17. [New] The **TextEditDblClick** event was supplemented with the **Button**, **Shift**, **x** and **y** parameters which allows you to get the detailed information about the mouse event like other mouse related events (**TextEditMouseDown**, **TextEditMouseUp**, **TextEditMouseMove**). The event has the following definition now:

    ```
    Event TextEditDblClick(ByVal lRow As Long, ByVal lCol As Long, ByVal Button
    As Integer, ByVal Shift As Integer, ByVal x As Single, ByVal y As Single)
    ```

18. [Change] The column header icons are now indexed starting from 1, not 0 as it was in the previous versions. The default value of the icon index for the iGrid column headers is set to 0 meaning that the icon is absent.
19. [New][Change] In this version of iGrid you need to specify an image list for your combo lists explicitly because of the fact that different cells may have different image lists. You should use a new **SetImageList** method of the **ComboObject** for that purpose. However, this approach allows you to do more than you could do in the previous versions as now the image list used in a combo list and the image list in a cell it is attached to can differ. For instance, to save screen space, you can display small icons in cells and large more informative icons in the combo lists attached to them.
20. [Change] In the previous versions the default row height (the **DefaultRowHeight** property) is used as the minimal row height in the **AutoHeightRow** method if you omit the **lMinimumHeight** parameter. In the current version there is no minimal row height limitation if you omit **lMinimumHeight**.
21. [New][Change] Two new events, **BeforeContentsSorted** and **AfterContentsSorted**, were implemented. They are raised before and after every sorting operation.

    The previous versions of iGrid had the **ContentsSorted** event which was raised only after interactive sorting operation (when you clicked a column header), but now this event is abolished as the **AfterContentsSorted** event is a more general event for this situation. Note that **AfterContentsSorted** also works when you group the grid in code but not only interactively.

22. [Change][Renaming] Two parameters of the **AddCol** method were renamed for commonality. These are **iIconIndex** and **eHdrTextFlags**. They are now called **iHeaderIcon** and **eHeaderTextFlags** respectively. Thus, now all the parameters of the **AddCol** method related to the column header, contains the "Header" word.

    The order of the parameters has been also changed so they are placed in a more native order accordingly to the frequency of use, and it helps to simplify the code if you are not using named arguments. The new syntax of the **AddCol** method is the following:

```
Function AddCol( _
   Optional ByVal sKey As String, _
   Optional ByVal sHeader As String, _
   Optional ByVal lWidth As Long = -1, _
   Optional ByVal eHeaderAlignH As EHAlignment = igAlignHLeft, _
   Optional ByVal eHeaderAlignV As EVAlignment = igAlignVTop, _
   Optional ByVal eHeaderTextFlags As ETextFormatFlags = igTextEndEllipsis, _
   Optional ByVal iHeaderIcon As Integer = 0, _
   Optional ByVal bVisible As Boolean = True, _
   Optional ByVal vColBefore As Variant, _
   Optional ByVal bIncludeInSelect As Boolean = True, _
   Optional ByVal eSortType As ESortTypes = igSortByValueGeneral, _
   Optional ByVal eSortOrderDefault As ESortOrders = igSortAsc, _
   Optional ByVal lMinWidth As Integer = -1, _
   Optional ByVal lMaxWidth As Integer = -1, _
   Optional ByVal bAllowSizing As Boolean = True, _
   Optional ByVal vTag As Variant, _
   Optional ByVal lCount As Long = 1 _
) As CellObject
```

23. [New][Change] The new edition of the **MouseDown** event allows you to determine what part of the cell has been clicked – its text part, its icon or extra icon, or the cell control (tree button, check box, combo button). In the previous version you can only detect that the cell control has been clicked using the **bOverControl** Boolean parameter, but in the current version a new **eCellPart** parameter is used for that:

```
Event MouseDown(Button As Integer, Shift As Integer, ByVal x As Single, ByVal
y As Single, ByVal lRow As Long, ByVal lCol As Long, ByVal eCellPart As
ECellParts, ByRef bDoDefault As Boolean)
```

The **eCellPart** argument contains the values from a new **ECellParts** enumeration; its members indicate the corresponding parts of the iGrid cell:

```
Public Enum ECellParts
   igCellPartNone
   igCellPartText
   igCellPartIcon
   igCellPartExtraIcon
   igCellPartComboButton
   igCellPartCheckBox
   igCellPartTreeButton
End Enum
```

24. [New] A new **HitTest** function was implemented. It allows you to know what cell part is at the specified point in the coordinate system of iGrid:

```
Public Function HitTest( _
   ByVal xPixels As Long, _
   ByVal yPixels As Long, _
   Optional ByVal lRow As Long = 0, _
   Optional ByVal lCol As Long = 0 _
) As ECellParts
```

In the general case, the first two arguments are enough to do that, but if you know the cell the specified point belongs to, you can pass this information in the two last optional arguments to speed up the execution of this function on big grids. This is possible to do if you call this function from such events as **MouseMove** which already provide you with the row and column indices of the cell under the mouse pointer.

25. [New] Two new Boolean properties, **EnsureVisibleAutoHScroll** and **EnsureVisibleAutoVScroll**, were implemented. If you select a cell which is partially visible or completely outside of the viewport of iGrid,

by default iGrid scrolls its contents automatically to display this cell in its viewport. Using these two new properties, you can prevent iGrid from automatic horizontal and vertical scrolling respectively in this case. To do that, change the default value True of the corresponding property to False.

These properties affect both the interactive and programming cell selection (using such methods as **SetCurCell**), and thus affect the **EnsureVisibleCell**, **EnsureVisibleCol** and **EnsureVisibleRow** methods so they work according to your **EnsureVisibleAutoHScroll**/**EnsureVisibleAutoVScroll** settings. Note that even if you set these properties to False, you can still scroll the grid manually using its scroll bars.

26. [New] By default iGrid automatically scrolls its contents after sorting to display the current cell in the viewport, but now this behavior can be turned on/off using a new Boolean property **SortScrollToCurCell** (the default value is True).

27. [New] The special vertical line used as the divider between the frozen and non-frozen columns now can be drawn in the grid even if the vertical and/or horizontal grid lines are turned off. This mode is turned on or off using a new Boolean property called **FrozenColsEdgeDrawAlways.** The default setting is False.

28. [New] Now you can use two new events, **BeforeCellContentsDraw** and **AfterCellContentsDraw**, to modify the standard cell drawing in iGrid. This is useful when you need to change what iGrid draws for text or combo box cells a little bit, and you do not want to redraw the entire cell contents from scratch as if you do that for custom draw cells.

    The new events have the following signatures:

```
Event BeforeCellContentsDraw( _
   ByVal lRow As Long, ByVal lCol As Long, _
   ByVal hdc As Long, _
   ByVal lLeft As Long, ByVal lTop As Long, _
   ByVal lRight As Long, ByVal lBottom As Long, _
   ByVal bSelected As Boolean)
Event AfterCellContentsDraw( _
   ByVal lRow As Long, ByVal lCol As Long, _
   ByVal hdc As Long, _
   ByVal lLeft As Long, ByVal lTop As Long, _
   ByVal lRight As Long, ByVal lBottom As Long, _
   ByVal bSelected As Boolean)
```

    Note that they are raised for all cells except custom drawing ones - in contrast to the **CustomDrawCell** event which is raised only for custom draw cells. The **BeforeCellContentsDraw** event is raised just before the grid is about to draw the cell contents and can be useful, for instance, if you need to draw your own cell background. The **AfterCellContentsDraw** event is raised when iGrid has just finished the drawing of the cell contents and can be used, for instance, if you need to draw such things as custom cell borders.

29. [New] The **SelItems** object property, which is used to access the list of the selected cells or rows, has a new method called **GetString**. It is equivalent to the **GetArray** method but it returns the list of the selected objects as a string. This method is useful for some non-VB development environments which do not support functions which return complex objects or types like **GetArray** (it returns an array of the **TSelItemInfo** structures).

    The returned string is in the form "(<row1>,<column1>)[;(<row2>,<column2>)...]". For instance, if you selected the cells in the first two columns in the 4th and 5th rows, the method returns "(4,1);(4,2);(5,1);(5,2)" if the grid works in cell selection mode. In row mode you can select only whole rows, and only the first cells from the selected rows are returned. In our example, if the 4th and 5th rows are selected in row mode, the method returns "(4,1);(5,1)".

30. [Renaming] The **igSortByCustomer** member of the **ESortTypes** enumeration was renamed to a more appropriate name **igSortCustom**.
31. [Renaming] Some internal limitations of Visual Basic do not allow to display the proper help topics when you press F1 to get the context help if two or more members have the same name (for instance, a method and an event, such as the **RequestEdit** method and the **RequestEdit** event). To avoid this problem in the current version and display the proper help topics while using the context help, the following member name changes were made:
    - The **RequestEdit** and **CancelEdit** methods were renamed to **RequestEditCurCell** and **CancelEditCurCell** respectively (the corresponding event names remain the same). For commonality, the **CommitEdit** method was also renamed to **CommitEditCurCell** (but it has no event of the same name).
    - The **TextEditCustomContextMenu** event was changed to **TextEditShowCustomContextMenu** (the corresponding property wasn't renamed).

Obsolete Removed Features

1. [Removed] The very rarely used ability to store objects in the cell values and combo box items was removed. This feature also caused misunderstanding when the developer assigned an **ADODB Field** object to a cell value trying to store the field value in a statement like this:

   ```
   iGrid1.CellValue(1, 1) = rsCustomers("ID")
   ```

   In this case iGrid displayed nothing in the cell instead of the default **Value** property of this **Field** object as the entire **ADODB Field** object was stored in the cell value.

   Note that the **RowTag** and **ColTag** properties still allow you to store objects (using the VB "Set" keyword), so you can store in these properties such values as VB collections, etc.
2. [Removed] iGrid no longer supports no-clip cells and the corresponding **igTextNoClip** flag (the **ETextFormatFlags** enumeration, the **CellTextFlags** property).
3. [Removed] Due to the new iGrid grouping functionality, iGrid no longer needs the **RowGroupStartCol** property used in the previous versions to implement grouping, and this member was removed. The corresponding optional parameter **lGroupColStartIndex** of the **AddRow** method was also removed.
4. [Removed] The **EvaluateTextWidth** method no longer has the useless optional **bForceNoModify** parameter.