What's New in iGrid 3.0 Release

Contents

I. Completely New Features

I.1. iGrid keeps its internal collection of selected items. This collection can be retrieved in your applications. Some iGrid's internal algorithms were optimized using this collection too. Two new selection-related events were implemented.

I.1.A. The **SelItems** object property was implemented. It allows you to get the total number of selected items in iGrid and enumerate them.

By "selected items" we mean cells if row mode is off and rows if row mode is on.

The **SelItems** property has the **SelItemsObject** type and provides you with the following public members:

```
Property Get Count As Long

Property Get/Let Sorted As Boolean

Public Function GetArray( _
      Optional ByRef lMinRow As Long = -1, _
      Optional ByRef lMaxRow As Long = -1, _
      Optional ByRef lMinCol As Long = -1, _
      Optional ByRef lMaxCol As Long = -1 _
   ) As TSelItemInfo()
```

The first one is the read-only **Count** property that returns the total number of the currently selected items.

The **GetArray** function returns the array of the currently selected items (rows in row mode or cells if row mode is off). Each item of the array is a structure of the **TSelItemInfo** type:

```
Type TSelItemInfo
   Row As Long
   Col As Long
End Type
```

If row mode is off, the **Row** and **Col** fields represent the row and column indices of a selected cell. If row mode is on, the **Row** filed contains the index of a selected row and **Col** always equals 1.

Note that in row mode only one item in that array corresponds one selected row; the rest of the cells in the selected row are not included in that array. This approach simplifies the work with selected rows when row mode is on as you have only one item in the returned array for each row. After you have switched between row mode and cell mode, the **GetArray** function returns the updated set of selected item according to the new mode.

The **Sorted** property determines whether the **GetArray** function returns the sorted list of selected items. By default this property is True, but if you see the **GetArray** function works slow on huge grids with a lot of selected items, you can try to set this property to False to speed up the execution.

If you have at least one selected item in the grid, the **GetArray** function returns the 1-based array and the VB UBound() function for this array in fact returns the total number of selected items. If there is no selected items, **GetArray** returns 0-based array with one element; its **Row** and **Col** fields equal 0 too. So the typical code to enumerate the selected items in iGrid generally looks like the following one (it automatically takes into account the situation when iGrid has no selected items):

```
Dim i As Long
Dim arr() As TSelItemInfo

arr = iGrid1.SelItems.GetArray

For i = 1 To UBound(arr)
   Debug.Print arr(i).Row; arr(i).Col
Next
```

Pay attention to the fact that the **GetArray** function returns the array of selected items at the moment you invoke the method (i.e. the method creates the snapshot of the currently selected items) and this array isn't changed in the future when you select/deselect cells. This can help you a lot if you need to enumerate the selected items and deselect them or save this info because otherwise the returned array would be changed each time you deselect an item.

GetArray also has the ability to calculate the minimum/maximum row and column indices of the selected cells. To do it, simply pass to this function Long variables with values not equal -1. When function finishes execution, your variable will have the corresponding calculated values. Note that you can pass not all parameters to this function but only required.

I.1.B. The **BeforeSelectionChange** and **AfterSelectionChange** events were implemented. These events are raised before and after an operation in which one or several cells change their selection status. Among of such operations are: selecting a rectangular block of cells with the mouse holding down the SHIFT key, selecting a new row when row mode is on (as several cells in the row change their selection status at once), the **ClearSelection** method and so on.

The previous version of iGrid implemented the **CellSelectionChange** event, and this event is still present and works in the current version of the grid like in the previous one, but the current version 'wraps' a series of these events with **BeforeSelectionChange**/**AfterSelectionChange**. Each operation when several cells change their selection status starts with the **BeforeSelectionChange** event, then the series of the corresponding **CellSelectionChange** events is fired, and the **AfterSelectionChange** event is raised after the last **CellSelectionChange** event in this operation.

For example, if you have a grid with 3 columns and select say the 2nd row when row mode is on, the following sequence of selection-related events is fired:

1) BeforeSelectionChange
2) CellSelectionChange (lRow = 2, lCol = 1, bSelected = True)
3) CellSelectionChange (lRow = 2, lCol = 2, bSelected = True)
4) CellSelectionChange (lRow = 2, lCol = 3, bSelected = True)
5) AfterSelectionChange

The following event series will be raised if you click the 4th row after that:

1) BeforeSelectionChange
2) CellSelectionChange (lRow = 2, lCol = 1, bSelected = False)
3) CellSelectionChange (lRow = 2, lCol = 2, bSelected = False)
4) CellSelectionChange (lRow = 2, lCol = 3, bSelected = False)
5) CellSelectionChange (lRow = 4, lCol = 1, bSelected = True)
6) CellSelectionChange (lRow = 4, lCol = 2, bSelected = True)
7) CellSelectionChange (lRow = 4, lCol = 3, bSelected = True)
8) AfterSelectionChange

Another example. Let's suppose you have a grid in which multi-selection mode is turned on, and you have already selected the first cell in the grid. If you click the 2nd cell in the 3rd row holding down the SHIFT key to select this rectangular block of cells, the event sequence will be as follows:

1) BeforeSelectionChange

2) CellSelectionChange (lRow = 1, lCol = 2, bSelected = True)
3) CellSelectionChange (lRow = 2, lCol = 1, bSelected = True)
4) CellSelectionChange (lRow = 2, lCol = 2, bSelected = True)
5) CellSelectionChange (lRow = 3, lCol = 1, bSelected = True)
6) CellSelectionChange (lRow = 3, lCol = 2, bSelected = True)
7) AfterSelectionChange

As you can see, the **BeforeSelectionChange** event is raised to notify you when the current selection in the grid is about to be changed. **AfterSelectionChange** is raised after the last cell changed its selection status in the current operation, and the advantage of this event is in it. If you need to recalculate something in your app when the user changes the set of selected cells or rows (for example, you provide your users with a multi-selection price list based on our grid), the best place to do it is the **AfterSelectionChange** event as the **CellSelectionChange** events can be fired many times in contrast to **AfterSelectionChange**.

The events have the following syntax:

```
Event BeforeSelectionChange()
Event AfterSelectionChange(ByVal bSelContentsChanged As Boolean)
```

The **bSelContentsChanged** parameter in the **AfterSelectionChange** event indicates whether at least one cell or row has been added to or removed from the **SelItems** collection. This parameter is True in most cases; it equals False in such rare cases when the set of the selected cells isn't changed and only row or column indices of some cells have been changed. The latter is true in the situation when say you have two selected cells in the fourth and fifth rows, and you remove the second row from the grid with the **RemoveRow** method. In fact, the set of the selected cells remains the same (those two selected cells) and only their row indices are changed (becomes three and four).

The **bSelContentsChanged** parameter is very useful if you need to process such situations. For instance, if you need to recalculate the total sum for selected items in your grid like we described above, you can avoid unneeded recalculations if you analyze the **bSelContentsChanged** parameter: you should recalculate only when **bSelContentsChanged** equals True. Using this ability, you can improve the performance of your code when the grid has a large amount of rows.

iGrid guarantees that the **BeforeSelectionChange** and **AfterSelectionChange** events are never raised if the current selection isn't changed (for instance, if you click the same cell twice). However, **BeforeSelectionChange** as well as **AfterSelectionChange** can be triggered without a series of the **CellSelectionChange** events. This happens, for instance, when you remove a row that contains at least one selected cell; the **CellSelectionChange** event isn't raised for the removed selected cell(s) in this case, but **BeforeSelectionChange/AfterSelectionChange** are raised to indicate that the selection has been changed.

Notes:

1. Due to the internal highly optimized algorithms of iGrid, the contents of the **SelItems** object may not correspond to the current set of the selected cells if you access this object from the **CellSelectionChange** event, but the **SelItems** object always returns the proper collection of selected items if you access it in the **BeforeSelectionChange** or **AfterSelectionChange** event. So if you need to update something in your app based on the current set of selected items, do it in the **AfterSelectionChange** event querying the **SelItems** object in this event.
2. In multiselection mode the **BeforeSelectionChange** or **AfterSelectionChange** events are always raised after the grid has been sorted if it has at least one selected cell/row even if the selected cell(s)/row(s) has not change their placement in the grid after sorting.

I.1.C. The following actions are performed much faster than in iGrid 2 when multi-selection mode is on (the more cells the more effect, especially if you have a grid with 10'000+ rows):

- Selecting a new cell or row with the mouse or keyboard.
- Selecting several cells at once when you use the Ctrl and/or Shift keys for this purpose.
- Switching between row mode and cell mode.

I.2. iGrid 3.0 implements some new features related to its scroll bars.

I.2.A. With a new **DisplayMode** property of the **VScrollBar** and **HScrollBar** objects you can control the visibility of your scroll bars. It is a property of a new **EScrollBarDisplayMode** enum type and can accept one of the following three values: **igScrollBarDisplayNever**, **igScrollBarDisplayNormal** and **igScrollBarDisplayAlways**.

The 2nd value corresponds to the normal behavior of the scroll bars you saw in the previous versions of iGrid – a scroll bar is visible only if it is required (the default value of the **DisplayMode** property). If you set this property to **igScrollBarDisplayNever**, you'll never see your scroll bar (though you can still change the scroll position with the **Value** property).

I.2.B. The **VScrollBar** and **HScrollBar** object properties have a new **Enabled** property of the Boolean data type which allows you to enable/disable each scroll bar separately. If a scroll bar is disabled, it is drawn as a disabled control and does not contain the thumb box, but note that the user still has the ability to scroll the grid using keyboard commands and you can change the scroll position with the **Value** property of the corresponding **ScrollBarObject**.

iGrid also raises a new **ScrollBarEnabledChanged** event when the availability of a scroll bar on the screen is changed. This event has the following signature:

```
Event ScrollBarEnabledChanged(ByVal eBar As EScrollBar, ByVal bEnabled As
Boolean)
```

Note that this event is fired not only when you change the **Enabled** property, but in some other cases too. For instance, if a scroll bar has the **DisplayMode** property set to **igScrollBarDisplayAlways** and you increase the size of the grid so the scroll bar becomes unneeded, but it should be present in the grid as it must be always displayed always.

I.3. iGrid 3.0 has a new **ResizeColToFillSpace** method you can use to resize a column to occupy the whole available empty space in the grid. This method is useful if you have several columns in your grid and would like to set the width of one column to a maximum value when this column is entirely visible in the grid's area (while the rest of the columns remain unchanged); generally it is done for the last column.

The method accepts one parameter – the index or string key of the column you wish to resize:

```
Sub ResizeColToFillSpace(ByVal vCol As Variant)
```

The method takes into account the minimum and maximum width you might set for the column (with the **ColMinWidth** and **ColMaxWidth** properties respectively). If the total width of the rest of the columns exceeds the grid's visible area, the width of the specified column remains unchanged.

I.4. Now you can freeze beginning columns in iGrid.

I.4.A. You can freeze columns with a new property called **FrozenCols**. To do it, simply assign to this property the number of columns you would like to see always on the screen at the left edge of the grid.

When you assign a non-zero value to this property, iGrid makes the first columns frozen using their order on the screen, not in the grid; invisible columns are also taken into account in this case. Thus you can hide some columns in the frozen area and make them visible at any time when you need it.

Note that if you have frozen columns, iGrid scrolls its columns horizontally using integral scrolling. This means you scroll the grid by columns, not by pixels, and thus you cannot see a column partially cut at left after the last visible frozen column.

I.4.B. You can control the vertical grid line that separates the frozen area from the non-frozen one with new **FrozenColsEdgeWidth** and **FrozenColsEdgeColor** properties. The first of them specifies the width of this line in pixels, and the second one stores the color used to draw this line. By default this grid line has the width of 2 pixels and is drawn using the vbButtonShadow color (the -2147483632 or &H80000010 value).

I.4.C. If the user reorders the columns in a grid and moves a column from the non-frozen area into the frozen one or vice versa, there are 3 possible cases what to do in this case:

1) You may want to have a predefined set of frozen columns in your grid and prohibit such column reordering.
2) You may allow your users to "widen" the frozen columns area by adding a column from the non-frozen area when they drag the column into the frozen area. And vice versa, if the user drags any

column from the frozen area into the non-frozen one, the number of frozen columns may be decreased by one.

3) You may want to make the number of the frozen columns constant. In this case, for instance, if the user moves a column from the non-frozen area into the frozen one, the last column in the frozen area should be "popped" out of this area into the non-frozen one.

iGrid allows you to realize all the 3 scenarios described above.

The default behavior of iGrid in this situation is the 2nd scenario, and you should do nothing to get it working in your iGrid.

The 3rd scenario can be performed with the functionality from the 2nd scenario: you simply move an unnecessary column from the frozen area off after you had added a required column to it. We decided not to implement something special for that in iGrid to simplify the life for you and your users. The user can easily gain the goal with the elementary native operation we described in the 2nd scenario.

And finally, the 1st scenario can be implemented in the code with the existing event, **ColHeaderEndDrag**: in your code you simply set its **bCancel** parameter passed by reference to False if the users is trying to move a column from the frozen area into the non-frozen one or vice versa. A new **bFrozenAreaChange** parameter of the **ColHeaderEndDrag** event will help you to determine this situation (see the description of this parameter in the Members and the Control Behavior section below).

There is just one interesting moment in the 2nd scenario – when you are trying to move the last visible frozen column out of the frozen area. This operation is prohibited in iGrid – if you have frozen columns in your grid, the user can never make it frozen columns free. When you are trying to drag the header of such a column, iGrid simply does nothing. If you need to warn your user about it, you can trace this situation in the **ColHeaderBeginDrag** event; its new in iGrid 3.0 parameter, **bFrozenAreaNotAllowed**, is set to True in this case and will help you to detect this situation. The **bFrozenAreaNotAllowed** parameter is also set to True in the case the user is trying to move the only visible column from the non-frozen area into the frozen one.

See also the description of this parameter and the event in the Changes in the Members and the Control Behavior section below.

I.5. A new **ColHeaderEndDragComplete** event with the following signature was implemented:

```
Event ColHeaderEndDragComplete(ByVal lCol As Long, ByVal bFrozenAreaChange As
Boolean)
```

It was implemented in addition to the existing **ColHeaderEndDrag** event. The difference between these two events is that **ColHeaderEndDrag** occurs when the user finished dragging a column header into a new position and released the mouse button, but the grid has not yet reordered its columns and this event allows you to prohibit this movement with its **bCancel** parameter. The **ColHeaderEndDragComplete** event is fired after the columns in iGrid has been actually reordered.

In the previous version of iGrid you could not execute your code after the columns were actually reordered because in the **ColHeaderEndDrag** event you dealt with the unchanged column order; now you can use the **ColHeaderEndDragComplete** event for this purpose.

The **bFrozenAreaChange** parameter indicates whether the total number of frozen columns has been changed after the column movement (the user moved a column from the non-frozen area into the frozen one or vice versa).

I.6. A new **ColFromPos** function with the following signature was implemented:

```
Function ColFromPos(ByVal lColPos As Long) As Long
```

It returns the numeric index of the column in the specified visible position in the grid. In fact, this function is the inverse function for the **ColPos** property (the **ColOrder** property in the previous version), and thus the expressions iGrid.ColPos(iGrid.ColFromPos(x)) and iGrid.ColFromPos(iGrid.ColPos(x)) always return x.

I.7. The **ColCount** property became a read-write property (it was read-only in the previous version). If you increase this property, the new columns are added to the end of the grid; if decrease, the corresponding number of the last columns is destroyed.

When you add new columns this way, they are created with the default properties as if you issued the **AddCol** method without parameters. The only exclusion is the text of the headers of the new columns. It is created using the template specified in a new **AutoColHeaderTextTemplate** string property. The "#" character in this template is replaced with the index of a new column to get its header text; in the VB6 syntax it is the result of the expression Replace(<AutoColHeaderTextTemplate>, "#", <column index>). By default, this property contains the "Col #" string, and thus the new columns created with the **ColCount** property have the headers "Col 1", "Col 2", etc. If you need to have the headers of the new columns empty, simply assign an empty string to the **AutoColHeaderTextTemplate** property before you increase the **ColCount** property.

I.8. In the previous version of iGrid you could not use the TAB key to move between the iGrid cells, but in the current version you can. By default the TAB key works like the RIGHT ARROW key, i.e. when you press TAB, the next cell in the horizontal direction (or the next row in row mode) becomes current. The same concerns the SHIFT+TAB combination that selects the previous cell/row.

This default behavior can be turned on/off with a new **UseTabKey** Boolean property. If this property is True (the default value), the TAB key works as described; if False, then TAB moves the input focus to the next control if any.

The current version of iGrid also raises the **KeyDown** event for the TAB key if **UseTabKey** is set to True, and thus you can use this event to process the TAB key like any other key. The **KeyDown** event is not raised if **UseTabKey** is False unless iGrid is the only control that can receive the input focus on a form (all the intrinsic controls in VB do so in this case).

Note: By default most (if not all) ActiveX control containers (such as VB forms, VBA UserForms) intercept the TAB key to move the input focus to the next control on the container, and that explains why you cannot simply process this key in your app in such events as KeyDown. To implement the functionality provided by the **UseTabKey** property, we use the standard Microsoft approach – we override the IOleInPlaceActivate COM interface. However, it does not work properly in Visual Basic for Applications in MS Word and MS Excel, and in this environment you still cannot use the TAB key to move between the iGrid cells (though in MS Access this feature works fine). Because of this we cannot guarantee that this functionality will work in another environment that differs from VB6, and thus if you are planning to use this feature in that environment, please, test it first.

I.9. A new **HideSelection** Boolean property was implemented. It determines whether the selected cells are highlighted when the iGrid control loses the focus. If this property is False (the default case), iGrid highlights the selected cells if the control does not have the focus with the colors specified in the **HighlightBackColorNoFocus** and **HighlightForeColorNoFocus** properties; if True, the selected cells are not highlighted in this state.

I.10. 3 new properties, **FocusRectNoFocus**, **FocusRectNoFocusColor1** and **FocusRectNoFocusColor2** were implemented. Using these properties you can force iGrid to draw the focus rectangle even if the control does not have the focus. By default the **FocusRectNoFocus** property is False and iGrid does not draw the focus rectangle in this state, but if you change this property to True, iGrid does it, and the **FocusRectNoFocusColor1** and **FocusRectNoFocusColor2** properties are used as the colors to draw the focus rectangle.

This allows you to specify different colors to draw the focus rectangle when iGrid has and does not have the focus. If iGrid is focused, the **FocusRectColor1** and **FocusRectColor2** properties with the default values vbWindowBackground and vbWindowText are used for that, but if iGrid is not focused, the default colors vbWindowBackground and vbGrayText from the **FocusRectNoFocusColor1** and **FocusRectNoFocusColor2** properties are used. It looks like the focus rectangle becomes grayed when iGrid loses the focus.

I.11. Two new events for cell dynamic formatting, **RowDynamicFormatting** and **CellDynamicFormatting**, were implemented. They are raised by iGrid just before the corresponding item (row or cell) is drawn and allow you to change the colors and fonts of these items dynamically. This ability is very useful if you need to highlight cells with colors and/or fonts based on some conditions, and other approaches such as iterating the entire grid in a loop and changing the colors of the cells with such properties as **CellBackColor**/**CellForeColor** work very slowly (especially on huge grids). These two events are raised only for cells the user sees on the screen at a moment, and this is the answer why the performance increases dramatically when you use these events.

The events have the following syntax:

```
Event RowDynamicFormatting( _
   ByVal lRow As Long, _
   ByRef oForeColor As Long, ByRef oBackColor As Long, _
   ByRef oFont As StdFont)

Event CellDynamicFormatting( _
   ByVal lRow As Long, ByVal lCol As Long, _
   ByVal vValue As Variant, _
   ByRef oForeColor As Long, ByRef oBackColor As Long, _
   ByRef oFont As StdFont)
```

You change the colors and font with the last 3 parameters passed by reference. For instance, if you need to highlight in your grid rows with the companies from the United States, you can use the following event sub:

```
Private Sub iGrid1_RowDynamicFormatting(ByVal lRow As Long, oForeColor As Long
oBackColor As Long, oFont As StdFont)

   If iGrid1.CellValue(lRow, "country") = "USA" Then
      oBackColor = vbGreen
   End If
End Sub
```

For speed optimization, iGrid can pass different values as these ByRef formatting parameters when the event sub is raised, so your code should not rely on these values. They are designated only for changing - if you need to format a row/cell somehow, you assign the required value to the corresponding parameter(s).

Note that the **oForeColor** and **oBackColor** parameters in both events in fact have the OLE_COLOR type that allows you to use for these parameters such system-specific values as vbWindowText. The fully equivalent Long data type is used instead because the events with the parameters of the OLE_COLOR type cannot be used in the MS Access environment.

I.12. In addition to cell dynamic formatting with the help of the **RowDynamicFormatting** or **CellDynamicFormatting** events you can specify the cell text dynamically in a new **CellDynamicText** event of the following syntax:

```
Event CellDynamicText(ByVal lRow As Long, ByVal lCol As Long, _
   ByVal vValue As Variant, ByRef sText As String)
```

The event is raised just before the grid will use the cell text (in drawing routine or somewhere else), and the **sText** parameter passed to this event sub by reference allows you to override the cell text iGrid is about to use. For instance, the following event sub can be used to apply the required format to the numeric data in a grid depending on the user's choice in the option button optbtnFullFormat:

```
Private Sub iGrid1_CellDynamicText(ByVal lRow As Long, ByVal lCol As Long,
ByVal vValue As Variant, sText As String)

   If optbtnFullFormat.Value = 1 Then
      sText = Format(vValue, "$#,##0.00")
   Else
      sText = Format(vValue, "#,##")
   End If
End Sub
```

This event is helpful in many cases. Among of them are:

- You need to apply a complex formatting algorithm to your data, and the **CellFmtString** property that delegates its work to the intrinsic VB Format function is not enough. Note that in this case you still store in your cells raw data and can sort them.
- You wish to provide your users with the ability to change the format of the cells on the fly (like in the example above).

- You store in the cells the IDs of your objects, but you need to display on the screen the corresponding names of these objects which are stored in a separate data structure. It could be for instance a customer ID and its full name stored in an array.
- Your grid should have a column with values which are never sorted when the user sorts the grid. It could be say row numbers in the first column, and this task is performed very easily now:

```
Private Sub iGrid1_CellDynamicText(ByVal lRow As Long, ByVal lCol As Long,
ByVal vValue As Variant, sText As String)

   If lCol = 1 Then sText = lRow
End Sub
```

Note that this event works only for normal text cells.

I.13. A new **Refresh** method was introduced, and it can be used to redraw the entire contents of the grid. This can be useful, for instance, if you highlight your cells with the **RowDynamicFormatting** or **CellDynamicFormatting** events based on some conditions, and you need to redraw the entire grid when something has been changed in your conditions.

I.14. New **Click** and **ClickNoDblClick** events with the following identical signatures were implemented:

```
Event Click(ByVal lRow As Long, ByVal lCol As Long)
Event ClickNoDblClick(ByVal lRow As Long, ByVal lCol As Long)
```

They both are raised when a cell is clicked, but there is a difference between them. If you use the standard mouse events, the normal sequence of these events in the classic VB environment is:

- for single-click: MouseDown, MouseUp, Click;
- for double-click: MouseDown, MouseUp, Click, DblClick, MouseUp.

The **Click** event is raised when the user simply clicks the grid or double-clicks it, and there is no simple way to distinguish clicks and double-clicks if you need to perform two different tasks for these two different actions. The **ClickNoDblClick** event was introduced specially for this case, and it is raised ONLY if the user has clicked the grid but has not double-clicked it. If you need to distinguish clicks and double-clicks in your grid, the **ClickNoDblClick** event will help you in this situation in pair with the existing **DblClick** event.

However, you can notice a short delay in raising the **ClickNoDblClick** event by iGrid. That is because in MS Windows there is an interval during which two mouse clicks are considered a double-click, and of course we cannot raise the **ClickNoDblClick** event till this interval is elapsed. So if you need to process only user clicks without double-clicks as fast as possible, use the **Click** event as it is raised just after the first **MouseUp** event.

I.15. New features were added to the **FillFromRS** method that now has the following syntax:

```
Public Sub FillFromRS( _
      ByVal RS As Object, _
      Optional ByVal eMode As EFillFromRSMode = igFillRSColsIfEmptyAndRows, _
      Optional ByVal vRowKeyField As Variant)
```

All the parameters except the first one were added to the method's signature, and you can enable the new features through these parameters.

The **eMode** parameter allows you to change the way the method populates the grid. This parameter can accept one of the following values from the new **EFillFromRSMode** enumeration:

Member name	Value	Description
igFillRSRecreateColsOnly	0	Only the columns corresponding to the fields in the specified recordset are added to the grid. The current column set is cleared if it exists before the method is issued. This mode is useful if you need to set some properties of the column default cells before you add real rows to the grid. In this case you call **FillFromRS** in this mode to populate the grid with the columns only, then set the required properties of the column default cells accessing them by the column keys (which are equal the filed names), and then issue again **FillFromRS** in the default mode (**igFillRSColsIfEmptyAndRows**).
igFillRSColsIfEmptyAndRows	1	The default mode in which iGrid creates the required column set (if it was not created before) and adds the records from the specified recordset. It is the mode that corresponds to the behavior of the **FillFromRS** method in the previous versions.
igFillRSAddToExistingRows	2	Allows you to not clear the existing structure and rows in the grid and add the rows from the specified recordset. It is useful when you perform several queries with different conditions but of the same field structure, and you wish to display the results of these queries in one grid.

The **vRowKeyField** parameter can be used to assign row keys while the **FillFromRS** method is enumerating the records in the recordset. It can be useful, for instance, if one of the fields stores the unique identifiers for the records, and you are going to search your records by these IDs in iGrid using such properties as **RowIndex**. In this parameter you can specify the string name or numeric index of the recordset's filed used for this purpose. If the field is not of the string data type, its values are converted to strings using the standard VB rules. If the field contains repeated values, no error will be generated but only the first unique key will be stored in the first row in which it appeared. If there is no the specified field, the method raises the 'Invalid procedure call or argument' error.

I.16. A new ability that allows you to store the initial row number in the row's key or tag was implemented. By initial row number we understand the order number of a row in the grid when this row is created.

This ability can be turned on with a new property **StoreInitRowNumbers** that accepts one of the following values from a new enumeration called **EStoreInitRowNumbersPlace**:

Member	Value	Description
igStoreInitRowNoNowhere	0	Initial row numbers are not stored. This is the default mode.
igStoreInitRowNoInRowKey	1	Initial row numbers are stored as row keys (they are converted to strings before that).
igStoreInitRowNoInRowTag	2	Initial row numbers are stored as row tags in its native Long format.
igStoreInitRowNoIn1stCol	3	Initial row numbers are stored as the cell values of the cells in the 1st grid column.

This feature can be used if you want to numerate the rows created in the grid using their original numbers, and use these data later.

For instance, it is useful if you populate your grid using the **FillFromRS** method that enumerates the rows in the specified recordset from top to bottom using their native order and add these rows to the grid. In this case the stored order number of each row in the grid will correspond to the order number of the record in the recordset (or its AbsolutePosition property), and this value can be used later to find the row in the recordset that corresponds the required row in the grid - even after such operations as sorting when the original row order has been lost.

Another example of the use of this feature is the ability to save anywhere the initial order of the rows and then restore it in the future. You lose this information after the user has sorted the grid, and to have the ability to "undo" sorting and restore the initial row order, you could use these saved initial row numbers say in row tags. To restore the initial row order, you simply perform custom sorting in iGrid and compare these initial row numbers stored say in row tags.

Note that if you add/remove rows after you have created the initial set of rows, you can have two initial row numbers stored in row tags. For instance, if you initially have 3 rows (say you created them by assigning 3 to the **RowCount** property), their initial row numbers are 1, 2, 3. If you remove the second row, you have two rows with the initial numbers 1 and 3. If you add after that a new row to the end, you'll have 3 grid rows with the following initial numbers: 1, 3, 3. This situation is possible if you store initial row numbers in row tags, but it is impossible if you store them in row keys. In this case non-unique initial row numbers are simply ignored and only the 2 first rows will have the initial row numbers "1" and "3", the row key for the third row will be empty.

I.17. Two new events that allow you to perform your own actions when the control is resized, **ResizeBeforeRedraw** and **ResizeAfterRedraw**, were implemented. Both events are raised just after the control has been resized, but there is a small difference between them.

The grid should redraw its contents when its size has been changed, and the **ResizeBeforeRedraw** event is raised before this redraw operation is performed, but **ResizeAfterRedraw** is raised after that. More over, when **ResizeBeforeRedraw** is raised, the updating in iGrid is turned off (as if the **Redraw** property were set to False), and this allows you to change the parts of iGrid without redrawing the contents twice. If iGrid didn't that, then you would need to wrap your resize code with two calls to the **Redraw** property in which you set the redrawing to False and then True, and the grid would redraw its contents after that again in its internal code as the result of the current resize operation.

The behavior described above allows you to write your resize code in the **ResizeBeforeRedraw** event without additional calls to the **Redraw** property and two redraw operations that eliminates flickering in the grid. This event and this feature are effectively used, for instance, if you resize the columns to fit the entire client area of the grid (like in MS Outlook when the columns are resized proportionally or you resize the last column to make it as wide as possible in the client are while the other columns keep their width).

Conversely, the **ResizeAfterRedraw** event is useful if you need to deal with the iGrid parts when they have been updated after the resize operation.

The events do not have parameters.

I.18. A new **DefaultRowNormalCellHeight** property was implemented. The value of this property is used as the default height for the normal cells in each new created row (i.e. the default value for the **RowNormalCellHeight** property for each row).

In iGrid, if you wish to display the row text cell in a row, you need to set the height of the row (the **RowHeight** property) to a value greater then the height of the normal cells in this row (the **RowNormalCellHeight** property). In the previous version of iGrid, you can set the default row height before you populate the grid with the **DefaultRowHeight** property, but you won't see row text cells because by default the height of the normal cells in each row is set to the row height and you need to iterate through each row in a loop to decrease the value of the **RowNormalCellHeight** property for each row.

Now you can avoid this additional loop that can dramatically decrease performance of your app on huge grids. By default, the **DefaultRowNormalCellHeight** and **DefaultRowHeight** properties have the same value, but to gain the goal we are discussing, you just need to increase the **DefaultRowHeight** property before you populate the grid. In this case the **RowNormalCellHeight** property for each row will be less then the height of the entire row, and you will have some space in each row under the normal cells to display row text cells in this area.

I.19. A new method **MoveRow** that allows you to reorder rows in iGrid was implemented. It has the following syntax:

```
Sub MoveRow(ByVal vRow As Variant, ByVal vRowNewPos As Variant)
```

In this method the **vRow** parameter specifies the row you move, and **vRowNewPos** is used to specify a new position for this row.

Note that you can use row numeric indices or string keys as the values of these parameters.

I.20. A new function, **Sys**, was implemented. With the help of this function you can retrieve some system parameters stored and used internally in iGrid. The function has the following syntax:

```
Function Sys(ByVal eSysParm As ESystemParameters) As Long
```

Its only argument, **eSysParm**, specifies the system parameter you wish to retrieve. The **eSysParm** argument can be one of the following value list, defined in the new **ESystemParameters** enumeration:

Member	Value	Description
igSysCellsAreaStartRow	0	The index of the first row visible in the cells area (the row can be partially visible).
igSysCellsAreaStartRowTopY	1	The y-coordinate of the top edge of the first row visible in the cells area.
igSysCellsAreaStartCol	2	The index of the first column visible in the cells area (the column can be partially visible).
igSysCellsAreaStartColLeftX	3	The x-coordinate of the left edge of the first column visible in the cells area.
igSysCellsAreaEndRow	4	The index of the last row visible in the cells area (the row can be partially visible).
igSysCellsAreaEndRowBottomY	5	The y-coordinate of the bottom edge of the last row visible in the cells area.
igSysCellsAreaEndCol	6	The index of the last column visible in the cells area (the column can be partially visible).
igSysCellsAreaEndColRightX	7	The x-coordinate of the right edge of the last column visible in the cells area.
igSysCellsAreaWidth	8	The width of the cells area.
igSysCellsAreaHeight	9	The height of the cells area.
igSysClientAreaWidth	10	The width of the client area of iGrid.
igSysClientAreaHeight	11	The height of the client area of iGrid.
igSysColsFirstVisCol	12	The index of the first visible column in the grid.
igSysColsLastVisCol	13	The index of the last visible column in the grid.
igSysColsLastVisFrozenCol	14	The index of the last visible frozen column.
igSysColsFirstVisScrollCol	15	The index of the first visible scrollable column.
igSysColsVisFrozenCount	16	The total number of all visible frozen columns.
igSysColsVisScrollCount	17	The total number of all visible scrollable columns.
igSysColsVisFrozenWidth	18	The total width of all visible frozen columns.
igSysColsVisScrollWidth	19	The total width of all visible scrollable columns.
igSysRowsFirstVisRow	20	The index of the first visible row in the grid.
igSysRowsLastVisRow	21	The index of the last visible row in the grid.
igSysRowsVisScrollCount	22	The total number of all visible rows.
igSysRowsVisScrollHeight	23	The total height of all visible rows.
igSysGroupRowsCount	24	The total number of the group rows.
igSysMemMngAllocatedRows	25	The total number of rows really allocated for iGrid in memory by the internal memory manager.

Notes:

1. The client area of iGrid is the entire area of the control without the border, i.e. it is the area in which iGrid places all its constituent parts (header, scroll bars, cells, etc). The cells area is the client area available for drawing cells, i.e. it is the client area without such parts as header and scroll bars.
2. If you have frozen columns, the rest non-frozen columns you can scroll are called "scrollable" columns. If you do not have frozen columns at all, all the columns in the grid are scrollable, and such parameters as **igSysColsFirstVisScrollCol, igSysColsVisScrollCount** and

igSysColsVisScrollWidth allow you to retrieve the information about all the columns in the grid.

3. All coordinate parameters are measured in pixels.
4. iGrid does not cache all the parameters listed above and can perform internal calculations each time when you request those parameters, so you can get some performance gain if you request a system parameter once and store it in a local variable if you are planning to get the same value of this parameters several times.
5. If the requested parameter concerns an iGrid item that does not exist at the moment, zero is returned (for instance, it is the case if you request the **igSysColsLastVisFrozenCol** parameter and there is no visible frozen columns or there is no frozen columns at all).
6. A row/column is considered visible if its visibility is set to True and its height/width is greater than zero.

I.21. A new Boolean property, **ChangeColorsWhenDisabled**, was implemented. In the previous versions of iGrid, if you disable it by setting its **Enabled** property to False, the background colors of the entire grid and each cell, and the color of the grid lines are changed to special "disabled" values to indicate this state, and you have no ability to prevent iGrid from doing that. In iGrid 3.0 you can set the **ChangeColorsWhenDisabled** property to False to disable this behavior, and iGrid will not change the color of the elements mentioned above when the grid is disabled. It is useful if you change the background colors of some cells and would like to see them when the grid is disabled (for instance, in the case if you need to prohibit any user changes in the grid, and the **Enabled** property is the easiest way to do that). The default value for the **ChangeColorsWhenDisabled** property is True that corresponds the behavior of iGrid in the previous versions.

II. Changes in the Members and the Control Behavior

II.1. The syntax of the **CustomSort** event in the previous version of iGrid was

```
Event CustomSort(ByRef v1 As Variant, ByRef v2 As Variant, ByRef bIsGreater As
Boolean)
```

In the current version it was supplemented with the **lCol**, **lRow1** and **lRow2** parameters, and its Boolean **bIsGreater** parameter was replaced on Long **lCompareResult**. The current syntax of this event is:

```
Event CustomSort( _
   ByVal lCol As Long, _
   ByVal lRow1 As Long, _
   ByVal lRow2 As Long, _
   ByRef v1 As Variant, _
   ByRef v2 As Variant, _
   ByRef lCompareResult As Long)
```

The **lCol** parameter contains the index of the column from which the **v1** and **v2** values are compared. In the previous version of iGrid, if you had more than one custom sort column, you needed to know what column is sorted in this event, and it was hard to do this without the **lCol** parameter.

The **lRow1** and **lRow2** parameters indicate the row indices for the values **v1** and **v2** respectively. This can be useful if you need to know additional information stored in the same row when you compare these sorted cell values (in the previous version you get only the values), or if you need to sort the grid by other "non-cell" values – for instance, row keys or row tags.

The last change in this event is that now you should return to iGrid the result of comparison with the Long **lCompareResult** parameter. You should set it to a positive value if **v1** is greater than **v2**, to a negative value if **v1** is less than **v2** or to zero if both values are equal. This extended information is used internally by iGrid to speed up sorting.

When you change your existing code in this event, it is simple to replace the existing code if you compare numeric or string values for **v1** and **v2**. If you compared two Long values got from **v1** and **v2**, you had a code like this:

```
bIsGreater = FuncAsLong(v1) > FuncAsLong(v2)
```

Now it looks like

```
lCompareResult = FuncAsLong(v1) - FuncAsLong(v2)
```

For string values you could have

```
bIsGreater = FuncAsString(v1) > FuncAsString(v2)
```

and now it is

```
lCompareResult = StrComp(FuncAsString(v1), FuncAsString(v2))
```

Caution! The optimized sorting algorithm of iGrid operates with the internal array that store row properties while the grid contents is being sorted, and if you try to access such row properties as row key or row tag while sorting in the **CustomSort** event, you may get improper values. This means you cannot use this event to sort the grid by row properties.

II.2. New sort criteria were added to the grid, and some previous sort criteria were changed (see the **ESortTypes** enumeration that contains all the possible sort criteria in iGrid).

II.2.A. By default the sort type of a column is the generic **igSortByValueGeneral** sort type. It replaces the default sort type **igSortByValue** in iGrid 2.5 and works practically as its predecessor; the only difference that string values are compared without case information (i.e. the sorting is case-insensitive). The goal of introducing this sort type is to get a special value you can use to sort your data like a man expects in most cases.

The two previous sort types, **igSortByTextNoCase** and **igSortByText**, were renamed to **igSortByCellTextNoCase** and **igSortByCellTextUseCase** respectively for better understanding of what iGrid is doing in these cases. Please, remember, that in these cases iGrid deals with cell texts (the **CellText** property) but not real cell values (the **CellValue** property) when it sorts its contents; cell texts can differ from cell values if you use say format strings (the **CellFmtString** property).

Two new sort types, **igSortByValueUseCase** and **igSortByValueNoCase**, were introduced. If you specify one of these sort types for a column, iGrid always compare the cell values as strings using case-sensitive or case-insensitive comparison respectively. In the previous version, you should specify the **igSortByTextNoCase** flag if you wish to sort your string data using case-insensitive sort, but the new sort types work faster (up to 1.5 times) because they work directly with cell values – the previous **igSortByTextNoCase** and **igSortByText** flags in fact work with cell texts (iGrid needs some additional time to get the text representation of a cell by its value).

You can also use **igSortByValueUseCase** for backward compatibility if you port existing code based on iGrid 2.x – it works like **igSortByValue** in the previous versions of iGrid.

Note: If you sort string values, **igSortByValueGeneral** and **igSortByValueNoCase** are equivalent. The **igSortByValueNoCase** allows you to adjust sorting if you sort data of different data types and even mix them. For instance, if you store in a column the Boolean values True and False, by default (**igSortByValueGeneral**) they sorted as their numeric equivalents in VB – True precedes False, but if you specify **igSortByValueNoCase**, these values are converted to strings before comparison and as the result you get False before True. A more interesting effect you also get if you mix in the same column Boolean values with numeric values and/or strings – you can experiment in such particular cases to see how iGrid sorts them.

II.2.B. A new sort criteria, **igSortNone**, were added to the **ESortTypes** enumeration. In fact, this new criteria is not a sort criteria, it just tells iGrid that a column should not be sorted.

Note that the **SortObject** property of iGrid never contains a column with the **igSortNone** sort type after a sorting operation (done interactively or in code) because the columns of the **SortObject** should reflect the current sort criteria column-by-column, and such a non-sorted column simply isn't included in sorting. Even if you specified non-sorted columns in the **SortObject**, they will be removed from this object after you issue the **Sort** method.

II.3. A new parameter, **bFrozenAreaNotAllowed**, has been added to the **ColHeaderBeginDrag** event. Now this event has the following signature:

```
Event ColHeaderBeginDrag(ByVal lCol As Long, ByRef bCancel As Boolean, ByVal
bFrozenAreaNotAllowed As Boolean)
```

If you have one visible frozen column in your grid, the user can never drag this column into the non-frozen area and make the grid frozen columns free. When the user is trying to drag the header of such a column, iGrid does nothing, and the **bFrozenAreaNotAllowed** parameter is set to True in this case (see also the description of this situation above when we are describing the frozen columns functionality in iGrid). The **bCancel** parameter is also automatically set to True in this case, and your changes of this parameter have no effect.

The **bFrozenAreaNotAllowed** parameter is also set to True in the case the user is trying to move the only visible column from the non-frozen area into the frozen one.

II.4. The **ColHeaderEndDrag** event was supplemented with a new **bFrozenAreaChange** Boolean parameter and now has the following syntax:

```
Event ColHeaderEndDrag(ByVal lCol As Long, ByVal lColBefore As Long, bCancel
As Boolean, ByVal bFrozenAreaChange As Boolean)
```

The **bFrozenAreaChange** parameter indicates whether the total number of frozen columns will be changed after the column movement (the user moves a column from the non-frozen area into the frozen one or vice versa).

II.5. The **ColOrder** property was renamed to **ColPos**. It also generates a new "Illegal operation" error if you try to move programmatically the only column outside of the frozen or non-frozen area (this movement is not allowed in iGrid).

II.6. The **EnsureVisibleCol** method cannot be used if your grid has frozen columns because this method operates with pixels, but in this frozen columns mode iGrid is scrolled horizontally only by columns. If you try to issue the method in this mode, you get the "Illegal operation" error. Use the **EnsureVisibleCell** method instead.

II.7. The **StartColWidthChange** event was renamed to **ColWidthStartChange**, and now all the events related to the column width changing process (**ColWidthStartChange, ColWidthChanging, ColWidthChanged**) are grouped together in any sorted list – in the help index, in the IntelliSense list, etc.

II.8. The **AddRow** method has a new optional parameter, **lCount**, that allows you to add more than one row at a time:

```
Sub 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 lGroupColStartIndex As Long = 0, _
      Optional ByVal vTag As Variant, _
      Optional ByVal lCount As Long = 1 _
   )
```

If you specify the **sKey** parameter, it is applied only to the first added row because two different rows cannot have the same string key.

If you insert several adjacent rows in your application using a loop and the **AddRow** method inside this loop, and your rows have the same properties, we recommend that you will replace this loop on the equivalent call to this method with the corresponding value of the **lCount** parameter because in the general case this approach works faster.

The **AddCol** method was also supplemented with the **lCount** parameter and now has the following syntax:

```
Public Function AddCol( _
      Optional ByVal sKey As String, _
      Optional ByVal sHeader As String, _
      Optional ByVal eHdrTextFlags As ETextFormatFlags = igTextLeft Or _
                     igTextSingleLine Or igTextEndEllipsis, _
      Optional ByVal iIconIndex As Integer = -1, _
      Optional ByVal lWidth As Long = -1, _
      Optional ByVal bVisible As Boolean = True, _
      Optional ByVal vColBefore As Variant, _
      Optional ByVal bIncludeInSelect As Boolean = True, _
      Optional ByVal bRowTextCol As Boolean = False, _
      Optional ByVal eSortType As ESortTypes = igSortByValueGeneral, _
      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
```

Using the **lCount** parameter of this method you can create several columns with the same parameters in one statement. If you specify such parameters as **lWidth**, **bAllowSizing**, etc, they will be applied to all the columns created with this method when the value of the **lCount** parameter is greater than 1. Note that if you create several columns this way, the **sKey** parameter is applied only to the first created column. You cannot also create several row text columns with this method (by setting the **bRowTextCol** parameter to True) – an error will be raised because iGrid can have no more than one row text column.

II.9. Now the **RemoveCol** method works slightly faster and has a new optional parameter, **lCount**, that allows you to specify the number of columns you remove:

```
Sub RemoveCol(ByVal vCol As Variant, Optional ByVal lCount As Long = 1)
```

The **lCount** parameter specifies the number of the adjacent columns in the cell array (note that these columns can be displayed on the screen in arbitrary positions because the user might reorder them).

The **RemoveRow** method was also supplemented with the **lCount** parameter:

```
Sub RemoveRow(ByVal vRow As Variant, Optional ByVal lCount As Long = 1)
```

If you remove several adjacent columns/rows in your grid in a loop, we recommend that you replace it on the corresponding call to **RemoveCol**/**RemoveRow** with the proper value of the **lCount** parameter because in the general case it will work faster.

II.10. The **KeyDown** event is raised for the TAB key if the **UseTabKey** property is True (see the description of this property). The event is also raised even if the control is disabled when its **Enabled** property is False (in the previous version the event isn't triggered in this state).

II.11. The **MemMngAllocatedRows** property was removed, you should use the **Sys(igSysMemMngAllocatedRows)** expression instead The internal data structure in iGrid has been changed, and now this value can be only retrieved but not changed.

II.12. The **GetStartEndCell** method was removed. Now you should use the equivalent call to the new **Sys()** function instead. The system parameters from **igSysCellsAreaStartRow (0)** to **igSysCellsAreaEndColRightX (7)** correspond the values you can get with this method in the previous version of iGrid.

II.13. Some changes in the iGrid behavior and events related to mouse clicks were made.

II.13.A. When the user presses the left mouse button in multi-selection mode when the mouse pointer is over a selected cell, the cell under the mouse pointer becomes current. In the previous version this is done when the user releases the left mouse button.

II.13.B. The signatures of the **MouseDown** and **MouseUp** events were changed. Now they have the following syntax:

```
Event MouseDown(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,
ByVal bOverCellCtrl As Boolean)

Event 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)
```

Now in the **MouseDown** event the **Button** parameter is passed by reference (it is passed by value in the previous version). This means you can change the mouse behaviour in your app by substituting mouse buttons in this event, i.e. you can make the right button working like the left button (MS Windows Explorer mode) simply by setting the **Button** parameter to 1 in this event if **Button** equals 2:

```
Private Sub iGrid1_MouseDown(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, ByVal bOverCellCtrl As Boolean)

   If Button = 2 Then
      Button = 1
   End If
End Sub
```

The last parameter of the event named in the previous version **bUnderControl** (which means there is a cell control, check box or combo button, under the mouse pointer) was renamed to a much more appropriate name **bOverCellCtrl**.

In the **MouseUp** event the first two parameters, **Button** and **Shift**, are also passed by reference now.

These changes allow you to change the normal behavior of iGrid and can be useful in many situations. For instance, iGrid allows you to select several cells (or rows in row mode) in multi-selection mode if you click the cells/rows while holding down the CTRL key. If you need to select cells/rows without holding CTRL simply by clicking them (in other controls this mode can be named as "simple selection"), then in iGrid 3.0 you can do this by setting the **Shift** parameter of the **MouseDown**/**MouseUp** events to 2 (that means the CTRL key, or use the vbCtrlMask constant in VB for better readability instead):

```
Private Sub iGrid1_MouseDown(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, ByVal bOverCellCtrl As Boolean)
   Shift = 2
End Sub

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)
   Shift = 2
End Sub
```

II.14. In the previous version of iGrid you can lose the changes made by the user during the editing of a text cell in some cases. Among of them are clicking a column header to sort the grid, scrolling the grid, etc. In many cases in a real-world app we need to save the user's changes, but the fact is that according to our event model iGrid should raise the **BeforeCommitEdit** event before we can do that, and you should have the ability to validate the new entered value in this event in the general case and return to editing if the new value is not valid (by setting the **eResult** parameter of **BeforeCommitEdit** to **igEditResProceed**). In the situations described above there is no chance to proceed the text editing, and thus we simply cancelled the user's changes in the previous version (raising the **CancelEdit** event as well) in order to not save the possible improper values if you validate them.

Side note: This happens because we use a text box control to edit the iGrid cells, we place it over a cell and activate it when the user is editing the cell, and that text box control can lose the input focus in such "problem" situations, and there is no universal way to prevent it from losing the focus in the general case.

In the current release we improved the situation: now the entered value is saved by default in those "problem" situations, and you can even validate it with the **BeforeCommitEdit** event though the only thing you cannot do in this case is still to return to editing if the value is not correct. We have also added a new Boolean **bCanProceedEditing** parameter to this event to indicate this situation; its value equals True if you can return to the editing process. Now the **BeforeCommitEdit** event has the following syntax:

```
Event BeforeCommitEdit(ByVal lRow As Long, ByVal lCol As Long, ByRef eResult
As EEditResults, ByVal sNewText As String, ByRef vNewValue As Variant, ByVal
lConvErr As Long, ByVal bCanProceedEditing As Boolean)
```

The **bCanProceedEditing** parameter also indicates for check box and combo box cells that editing cannot be proceeded in such cells, and this information can be useful if you write a **BeforeCommitEdit** event handler in the general case for cells of any type and you need to know whether you can proceed editing.

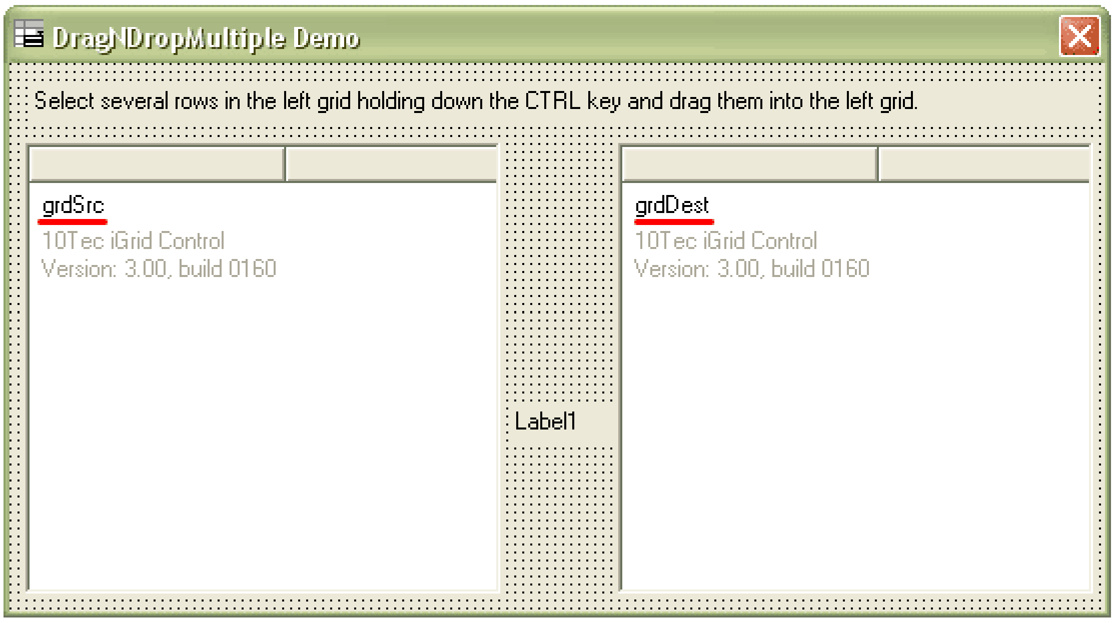
II.15. If you work in cell mode, the HOME key moves the current cell selection to the first visible column in the current row, the END key moves the current cell selection to the last visible column in the current row. In the previous version of iGrid they move the current cell selection to the first/last visible cell in the current column.

In row mode HOME/END work like in the previous version: they make the first/last visible row the current row.

II.16. In iGrid 3.0 you can use two new keys to edit cells in addition to the iGrid standard keys: F2 can be used to edit a text cell, and F4 opens the drop-down list in a combo box cell (these keys are the standard keys for these actions in many Windows grids and controls).

III. Design time changes

III.1. Now at design time in the cells area the name you assigned to the control in the Property Editor is also displayed in addition to the full version of the control including the build number:



Select several rows in the left grid holding down the CTRL key and drag them into the left grid.

grdSrc
10Tec iGrid Control
Version: 3.00, build 0160

grdDest
10Tec iGrid Control
Version: 3.00, build 0160

IV. Performance Improvements

IV.1. Due to optimization we made internally in iGrid, it sorts its contents up to 10 times faster. This concerns general cases and some particular cases (for instance, if a column contains many identical values).

IV.2. The **RowCount** property works much faster (up to 8 times) if you add to the grid new rows by increasing the value of this property.

IV.3. Adding and removing rows with the **AddRow** and **RemoveRow** methods also became faster, and you can notice a performance gain of up to 2.5 times on huge grids.

IV.4. The **FillFromRS** method became 30% faster.

IV.5. The drawing code of iGrid was optimized more, and now iGrid is redrawn faster. If you heavily update iGrid and it should be redrawn on the screen after each update, you may notice less CPU usage - the drawing can be three times faster in this case.

V. Fixed bugs and incompatibilities, new supported hardware configurations

V.1. As you may know, iGrid still has the current cell even if you work in row mode. In row mode, if you have enough many columns (and thus the horizontal scroll bar is visible), and click a cell in row mode, then scroll the grid horizontally to make the clicked cell off the visible area, and then press the HOME, END, PAGEUP or PAGEDOWN key, the previous version of iGrid moves the current row to the new appropriate position but also scrolls its contents horizontally to make the clicked cell (which is also current) visible.

In the current version of the grid this does not happen – your horizontal position remains unchanged when you press one of those keys, iGrid just scrolls its contents vertically to make the new current row visible.

V.2. The current version of iGrid now works properly on computers with multiple monitors (for instance, you may have some problems with displaying drop-down lists in combo box cells on secondary monitors in the previous versions, but now they are displayed in the proper position).