



MWA Software

Dynamic Database Controls

Issue 1.1
22 July 2024

McCallum Whyman Associates Ltd

Email: info@mccallumwhyman.com, <http://www.mccallumwhyman.com>

Registered in England Registration No. 2624328

COPYRIGHT

The copyright in this work is vested in McCallum Whyman Associates Ltd. The contents of the document may be freely distributed and copied provided the source is correctly identified as this document.

© Copyright McCallum Whyman Associates Ltd (2024)
trading as MWA Software.

Disclaimer

Although our best efforts have been made to ensure that the information contained within is up-to-date and accurate, no warranty whatsoever is offered as to its correctness and readers are responsible for ensuring through testing or any other appropriate procedures that the information provided is correct and appropriate for the purpose for which it is used.

CONTENTS	Page
1 INTRODUCTION.....	1
2 TIBDYNAMICGRID.....	3
2.1 OVERVIEW.....	3
2.2 COLUMN PROPERTIES.....	5
2.3 TIBDYNAMICGRID NEW PROPERTIES.....	6
2.4 TIBDYNAMICGRID NEW EVENTS.....	7
2.5 THE EDITOR PANEL.....	7
2.6 SETTING QUERY PARAMETERS.....	8
3 TDBCNTROLGRID.....	9
3.1 OVERVIEW.....	9
3.2 TDBCNTROLGRID PROPERTIES.....	10
3.3 TDBCNTROLGRID EVENTS.....	11
4 TIBTREEVIEW.....	13
4.1 OVERVIEW.....	13
4.2 TIBTREEVIEW PROPERTIES.....	14
4.3 TIBTREEVIEW METHODS.....	15
4.4 DRAG AND DROP.....	15
4.5 SETTING QUERY PARAMETERS.....	16
5 TIBLOOKUPCOMBOEDITBOX.....	17
5.1 OVERVIEW.....	17
5.2 TIBLOOKUPCOMBOEDITBOX EXAMPLE.....	18
5.2.1 <i>Auto-insert</i>	19
5.3 TIBLOOKUPCOMBOEDITBOX PROPERTIES.....	20
5.4 TIBLOOKUPCOMBOEDITBOX EVENT HANDLERS.....	21
5.5 SETTING QUERY PARAMETERS.....	21
6 TIBARRAYGRID.....	23
6.1 OVERVIEW.....	23
6.2 PROPERTIES.....	23
6.3 EXAMPLES.....	24
6.3.1 <i>Database Creation</i>	24
6.3.2 <i>1D Array Example</i>	25
6.3.3 <i>2D Array Example</i>	25
7 INTERFACE SPECIFICATION.....	27
7.1 THE IDYNAMICSQLDATASET INTERFACE.....	27
7.1.1 <i>TDataset Requirements</i>	28
7.2 THE IDYNAMICSQLCOMPONENT INTERFACE.....	29
7.3 THE IDYNAMICSQLEDITOR INTERFACE.....	29
7.3.1 <i>Methods</i>	30
7.4 THE IDYNAMICSQLPARAM INTERFACE.....	30
7.4.1 <i>Methods</i>	31
7.5 THE IARRAYFIELD INTERFACE.....	31
7.5.1 <i>Methods</i>	31
7.6 THE IARRAYFIELDDEF INTERFACE.....	32
7.6.1 <i>Methods</i>	32

1

Introduction

The Dynamic Data Controls were originally distributed as an integral part of the IBX for Lazarus package and were dependent on the use of IBX for database access. Starting with IBX release 2.7.0, the "ibcontrols" package is no longer dependent on IBX. Pascal (corba) interfaces are used to communicate between the controls and the database access provider (e.g. IBX) and any similar package that provides the same interfaces can now be used with the controls. Developers of other Database Access Providers are encouraged to use these interfaces as defined in the code snippet `IBDynamicInterfaces.pas` and to provide the same functionality in their packages, and hence to allow their use with the Dynamic Database Controls.

The Lazarus IDE pallet tab for these controls has also been renamed from "Firebird Data Controls" to "Dynamic Database Controls". The package name remains "ibcontrols".

The Dynamic Database Controls are:

- `TIBLookupComboEditBox`
- `TIBDynamicGrid`
- `TIBTreeview`
- `TDBControlGrid`
- `TIBArrayGrid`

TIBLookupComboEditBox is a `TDBLookupComboBox` descendent that implements "autocomplete" of typed in text and "autoinsert" of new entries. Autocomplete uses SQL manipulation to revise the available list and restrict it to items that are prefixed by the typed text (either case sensitive or case insensitive). Autoinsert allows a newly typed entry to be added to the list dataset and included in the available list items.

TIBDynamicGrid is a `TDBGrid` descendent that provides for:

- automatic resizing of selected columns to fill the available row length

- automatic positioning and sizing of a "totals" control, typically at the column footer, on a per column basis.
- DataSet resorting on header row click, sorting the dataset by the selected column. A second click on the same header cell reversed the sort order.
- Support for a "Panel Editor". That is on clicking the indicator column, the row is automatically expanded and a panel superimposed on it. The panel can have any number of child controls, typically data aware controls with the same datasource as the grid allowing for editing of additional fields and more complex editors.
- Reselection of the same row following resorting.
- A new cell editor that provides the same functionality as TIBLookupComboEditBox. Its properties are specified on a per column basis and allows for one or more columns to have their values selected from a list provided by a dataset. Autocomplete and autoinsert are also available. The existing picklist editor is unaffected by the extension.

TIBTreeView is a data aware TCustomTreeView.

TDBControlGrid is a lookalike rather than a clone for the Delphi TDBCrtlGrid. TDBControlGrid is a single column grid that replicates a TWinControl - typically a TPanel or a TFrame in each row. Each row corresponds to a row of the linked DataSource. Any data aware control on the replicated (e.g.) TPanel will then appear to have the appropriate value for the row.

Note: TDBControlGrid is not dependent on the availability of the IBDynamicInterfaces and can be used with any Database Access Provider.

TIBArrayGrid is a data aware control derived from TCustomStringGrid and which may be used to display/edit the contents of a one or two dimensional Firebird array Field.

The subsequent chapters of this document describe each control in turn and finally the Pascal interfaces that they depend on. Users of the controls need not read the chapter on the Pascal interfaces; this chapter is intended as a guide for Database Access Provider developers.

Examples of the use of the controls may be found in the IBX "examples" directory.

1.1 Revision History

1.1.1 Issue 1.1

- IdynamicSQLEditor interface expanded to allow orderby column number.

2

TIBDynamicGrid

2.1 Overview

Employee List

Started Before Started After Salary Range

Last Name	First Name	Emp No.	Dept	Located	Started	Salary
Baldwin	Janet	34	Corporate Headquarters / Sales and Mar	USA	21-3-91	\$61,637.81
Bender	Oliver H.	105	Corporate Headquarters / Engineering /	USA	8-10-92	\$212,850.00
Bennet	Ann	28	Corporate Headquarters / Sales and Mar	USA	1-2-91	\$22,935.00
Bishop	Dana	83	Corporate Headquarters / Engineering /	USA	1-6-92	\$62,550.00
Brown	Kelly	109	Corporate Headquarters / Engineering	USA	4-2-93	\$27,000.00
Burbank	Jennifer M.	71	Corporate Headquarters / Engineering /	USA	15-4-92	\$53,167.50
Cook	Kevin	107	Corporate Headquarters / Engineering /	USA	1-2-93	\$111,262.50
De Souza	Roger	29	Corporate Headquarters / Engineering /	USA	18-2-91	\$69,482.63
Ferrari	Roberto	121	Corporate Headquarters / Sales and Mar	Italy	12-7-93	\$99,000,000.00
Fisher	Pete	24	Corporate Headquarters / Engineering /	USA	12-9-90	\$81,810.19
Forest	Phil	9	Corporate Headquarters / Engineering /	USA	17-4-89	\$75,060.00
Glou	Jacques	134	Corporate Headquarters / Sales and Mar	France	23-8-93	\$390,500.00
Green	T.J.	138	Corporate Headquarters / Engineering /	USA	1-11-93	\$36,000.00
Guckenheimer	Mark	145	Corporate Headquarters / Engineering /	USA	2-5-94	\$32,000.00
Hall	Stewart	14	Corporate Headquarters / Finance	USA	4-6-90	\$69,482.63
Ichida	Yuki	110	Corporate Headquarters / Sales and Mar	Japan	4-2-93	\$6,000,000.00
Johnson	Leslie	8	Corporate Headquarters / Sales and Mar	USA	5-4-89	\$64,635.00

Add
Edit
Delete

Save Cancel \$115,542,468.02

Illustration 1: The TIBDynamicGrid

The TIBDynamicGrid is illustrated above using Firebird's example "employee" databases.

In use, it looks just like a TDBGrid and is a TDBGrid descendent. Any project that uses IBX and TDBGrid can thus be quickly converted to using TIBDynamicGrid. The control uses SQL Manipulation to manage column sorting.

The above example can be found in "ibx/examples/employee" and illustrates most of the benefits of TIBDynamicGrid.

- Resize the form and you will see how the "Dept" column automatically grows/shrinks to ensure that the grid always fills the available space and how the Salary "Total" control (TDBText) moves so that it is always aligned with the grid. Column resizing is controlled at design time by setting the AutoSizeColumn property for each column that it is to be dynamically resized, with its design time width interpreted as the minimum column width. All other column widths remain unchanged.
- Click on the "Started" column header (or any other column header) and the table will be resorted by that column. A second click on the same header reverses the sort order.
- Select a row and press "F2", or click on "Edit" or the left hand indicator column and the Editor Panel is revealed (See Illustration 2). This allows the row to be edited free of the constraints imposed by a simple column editor.
- After reopening the dataset (e.g. after a re-sort or change of filters) the previously selected row is automatically reselected.
- The filters, such a "salary range", also illustrate how the new IB SQL Parser works with the TIBDynamicGrid. For example, where a salary range is selected, the dataset is re-opened and the filters are applied in the BeforeOpen event handler.
- Each row can still be edited without having to open the panel editor. The column "located" is an example of the use of TIBLookupComboEditBox as a column editor. Note that the country list is dynamically generated and varies according to Job Code (an Employee Database constraint).

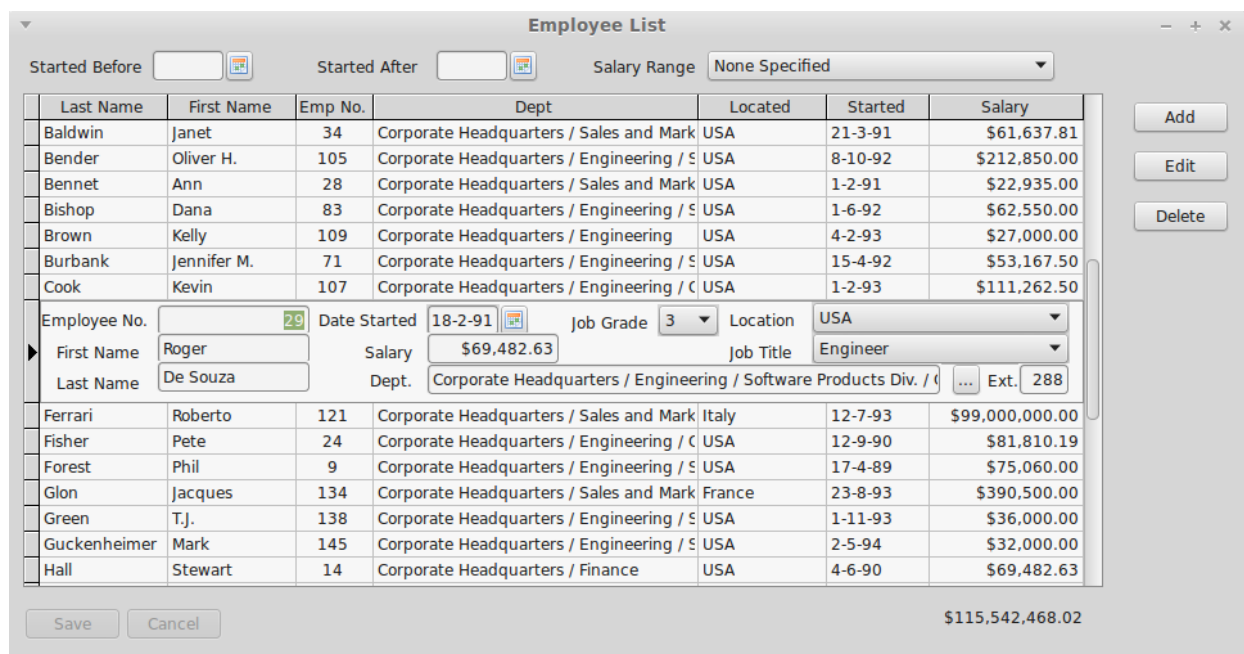


Illustration 2: TIBDynamicGrid with an Editor Panel Visible

2.2 Column Properties

Most of TIBDynamicGrid's new features are accessed via the column editor and are properties of each column in the grid. The new column properties are given below.

AutoSizeColumn	Boolean	If true then the column is automatically resized to fill the grid. More than one column can have this property set to true.
ColumnTotalsControl	TControl	Optional. Used to identify a control (typically a TDBEdit or TDBText) to be kept in vertical alignment with the column, and to have the same width. Note that the horizontal positioning is unaffected by grid resize, and hence the total can be placed either above or below the grid.
InitialSortColumn	Boolean	Identifies the column used to sort the grid when the dataset is first opened.
DBLookupProperties	TDBLookupProperties	These properties are copied to a TIBLookupComboBox when it is used as a column editor. Setting TDBLookupProperties.ListSource implicitly requests this as the column editor instead of a normal pick list. If the TDBLookupProperties.DataFieldName is

		<p>not set then the control works as a “pick list” with its values taken from the List Source DataSet.</p> <p>If the TDBLookupProperties.DataFieldName is set then it works as full lookup list. The DataFieldName identifies a field in the parent TIBDynamicGrid.DataSource.DataSet. This field does not have to be visible in the grid. When the editor completes, the identified field is set to the value of the List Source field identified by TDBLookupProperties.KeyField.</p>
--	--	---

2.3 TIBDynamicGrid New Properties

EditorPanel	TControl	When set, this control (typically a TPanel or TFrame) is used as the Editor Panel (see below).
ExpandEditorPanelBelowRow	Boolean	When set and an editor panel is displayed, the row height is set to the current row height plus the panel height and the Editor Panel placed under the row. That is, the original row is still displayed with the editor panel beneath it. The default is that the editor panel appears to replace the row.
AllowColumnSort	Boolean	Enables column sorting by column header click (default true).
Descending	Boolean	Determines the initial sort order. Default is false i.e. ascending sort order.
DefaultPositionAtEnd	Boolean	Determines the initially selected row when the dataset is first opened. If true then the last row is selected, otherwise the first row. Default: false.
IndexFieldNames	String	<p>This is a semi-colon separated list of one or more dataset fieldnames. Typically this is the primary key for the dataset. Used for automatic reselection of rows after the dataset is reopened.</p> <p>A property editor is available for design time field name selection.</p>

2.4 TIBDynamicGrid new Events

OnBeforeEditorHide	This event is called before the Editor Panel is hidden. Can be used to validate changes.
OnEditorPanelShow	This event is called after the Editor Panel is made visible
OnEditorPanelHide	This event is called after the Editor Panel is hidden. Can be used to do any additional tidying up needed.
OnKeyDownHandler	The TIBDynamicGrid uses a KeyDown handler to intercept edit keys while the Editor Panel is active. For example, to process an “escape” key as a cancel edit. You can write your own keydown handler to modify this behaviour.
OnColumnHeaderClick	Called when a column header is clicked and before the dataset is re-sorted. Can be used to modify the column index for the sort.
OnUpdateSortOrder	Called when the dataset select SQL is being modified prior to resorting the dataset. Can be used to modified the SQL “Order by” clause. e.g. to add a subsort column. For example, useful when one column has a “year” and the next column is the “month”. Clicking on “year” can then made to subsort on “month”. Can also return an empty string in order to prevent sorting of the dataset.
OnRestorePosition	<p>Called when the dataset is opened and may be used to override the initially selected record. The event provides a read/write argument (Location) that is an array of variants. This is either an empty zero length array or contains the same number of elements as there are indexnames (See IndexFieldNames property). In the latter case, it contains the index key values for the previously selected row (i.e. when the dataset was last closed). The first time the dataset is opened the array is empty.</p> <p>The location can be inspected and replaced by an alternative location (index key values) or set to empty. In the former case, the grid will attempt to locate the selected row. In the latter case, the default position is selected (see DefaultPositionAtEnd property).</p>

2.5 The Editor Panel

An Editor Panel may be any TControl available on the form. However, in practice, it is typically either a TPanel or a TFrame. The example shows a TPanel being used as an Editor Panel.

You can create an Editor Panel by simply dropping it on to the same form as the TIBDynamicGrid and then selecting it as the value of the TIBDynamicGrid.EditorPanel property.

To be useful, the Editor Panel should be populated with data aware controls that use the same DataSource as the grid and are individually used to edit fields in the same row. The height of the panel should be the minimum necessary as this will determine the row height when it is visible.

At run time, the Editor Panel is automatically hidden until called into use by either:

- a) Pressing “F2” when the Dynamic Grid has the focus.
- b) Clicking on the left hand indicator column, or
- c) Calling the `TIBDynamicGrid.ShowEditorPanel` method.

In order to show the editor panel, the following actions are performed by the `TIBDynamicGrid`:

- The current row is resized to the height of the Editor Panel.
- The Editor Panel is resized and repositioned so that it fits exactly over the current row.
- The Editor Panel is made visible.

The current row can now be edited using the child controls on the Editor Panel – that is as long as their DataSource is the same as the grid's.

The Editor Panel is hidden (and any changes Posted to the DataSet) when:

- a) A different row is selected by the mouse or up/down arrow keys
- b) The Escape Key is Pressed (cancels the changes)
- c) “F2” is pressed.
- d) The `TIBDynamicGrid.HideEditorPanel` method is called.

Once the Editor Panel is hidden, the current row is re-sized back to its correct height.

2.6 Setting Query Parameters

The dataset used as the `TIBDynamicGrid`'s data source may have a select query that contains query parameters. However, in order to perform column sorting, `TIBDynamicGrid` manipulates the SQL query “behind the scenes” to change the “order by” clause. Because of this, the only “safe” place to set values for query parameters is in the dataset's “BeforeOpen” event handler. This is guaranteed to be called every time the grid updates the SQL order by clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the “BeforeOpen” event handler will be lost when the grid updates the SQL and reset to the default null value.

3

TDBControlGrid

3.1 Overview

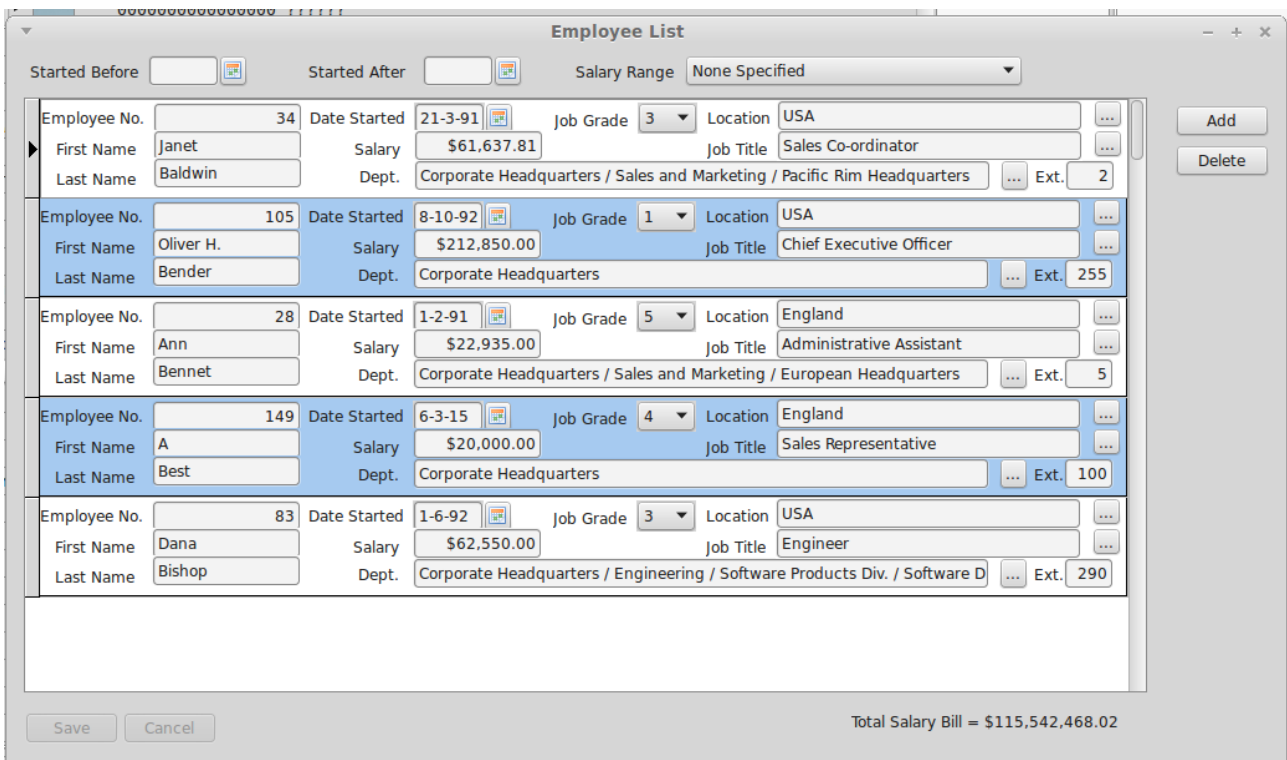


Illustration 3: Example Control Grid

TDBControlGrid is a lookalike rather than a clone for the Delphi TDBCrtGrid. TDBControlGrid is a single column grid that replicates a TWinControl - typically a TPanel or a TFrame in each row.

Each row corresponds to a row of the linked DataSource. Any data aware control on the replicated (e.g.) TPanel will then appear to have the appropriate value for the row.

Unlike the Delphi TDBCtrlGrid, there are no restrictions on which controls can be used on the replicated panel. In principle, any visual control may be used. The "csReplicable" property is not used by TDBControlGrid. However, there can be performance issues with a large number of controls on the panel or when there is a high latency to draw one or more controls.

To use the new control, simply drop it on to a form at design time and size it appropriately. Then separately drop a TPanel on to the same form and populate it with appropriate child controls, typically data aware controls using the same DataSource.

Now link to TDBControlGrid DrawPanel property to this panel. The panel should then be repositioned as a child control of the TDBControlGrid and occupying the top and only row of the grid. The row height should be set to the panel height and the panel width will be set to the width of the grid row. The panel can be unlinked at any time.

Now set the TDBControlGrid.DataSource to the common data source for the controls on the panel.

Important Note: It is strongly recommended *not* to open the source DataSet for a DBControlGrid during a Form's "OnShow" event handler. Under GTK2 this is known to risk corrupt rendering of row images when the control is first displayed. If necessary use "Application.QueueAsyncCall" to delay opening of the dataset (see DBControlGrid examples) until the Form's Window has been created. See the example application.

When you build and run your project and open the DataSource's dataset, the TDBControlGrid should show a row for each row in the dataset and the child controls on each row should have the appropriate values for the row.

When the grid has the focus, you can move between rows using the up and down arrow keys, page Up and Page Down, Ctrl+Home and Ctrl+End jump to beginning and end respectively. You can also use the mouse to change between rows, either by clicking on a row or the scroll bar.

Pressing the down arrow key on the last row should append a new row – as long as the "Disable Insert" TDBControlGrid.Option is not selected.

All rows may be edited in situ. Moving between rows should automatically post the changes. The "escape" key may be used to cancel row edits before they are posted.

A row may be deleted by calling the underlying DataSet's Delete method.

See the TDBControlGrid example code for guidance on how to use the control. This example requires IBX and uses the Firebird example employee database.

3.2 TDBControlGrid Properties

DrawPanel	TWinControl	This control will be replicated for each row in the DataSet. Typically a TPanel or a TFrame.
Options	TPanelGridOptions	Similar to a TDBGrid, but limited to: <ul style="list-style-type: none"> • Cancel On Exit

		<ul style="list-style-type: none"> • Disable Insert • Show Indicator Column
DataSource	TDataSource	A row is replicated for every row in this dataset.
DefaultPositionAtEnd	Boolean	When the dataset is opened then it is initially positioned at the last record if this property is true,

3.3 TDBControlGrid Events

OnKeyDownHandler	The TDBControlGrid uses a KeyDown handler to intercept edit keys while the Draw Panel is active. For example, to process an “escape” key as a cancel edit. You can write your own keydown handler to modify this behaviour.
------------------	---

4

TIBTreeView

4.1 Overview

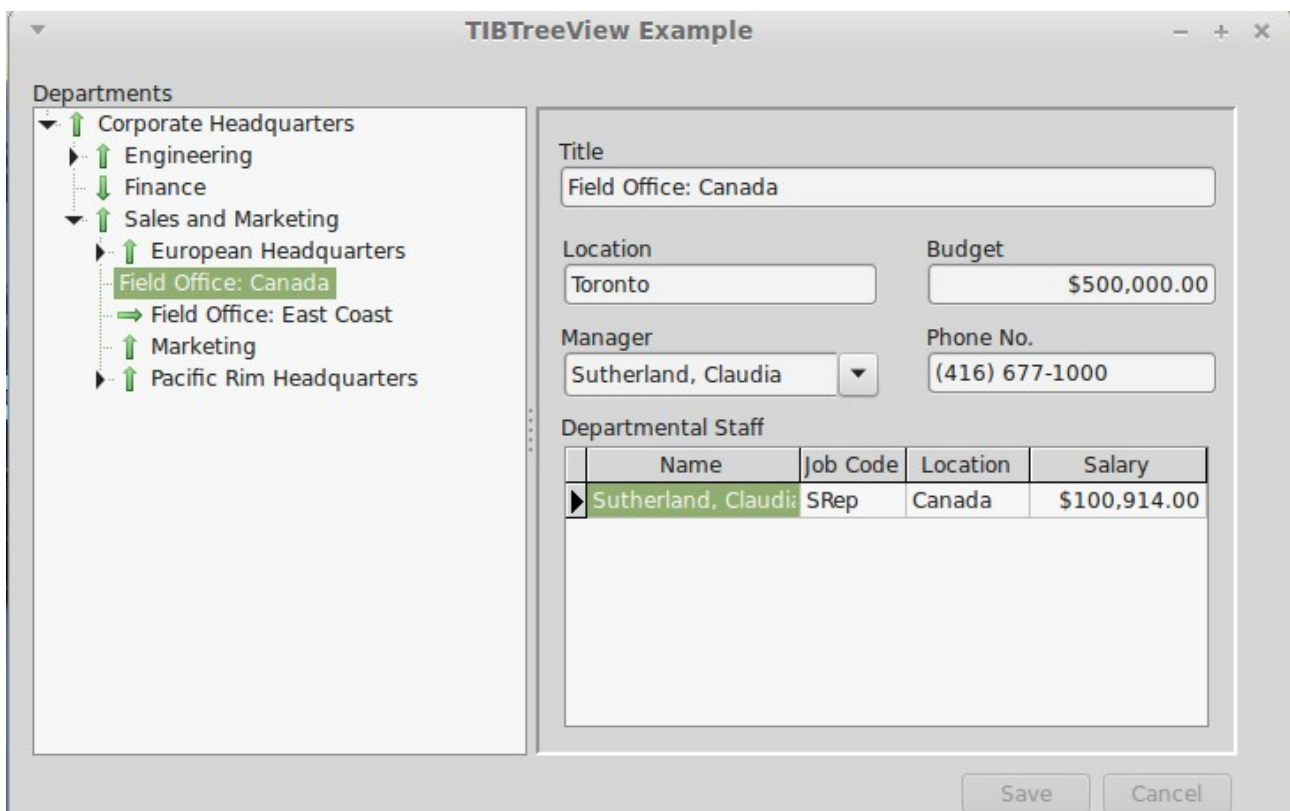


Illustration 4: TIBTreeView Example

TIBTreeView is a data aware descendent of a TCustomTreeView and is used to present a hierarchically organised data set in a tree view. Tree Node Insertion, Deletion and Modification are supported, as is moving (e.g. using drag and drop) nodes from one part of the tree to another. The underlying dataset cursor is always positioned to reflect the currently selected tree node. It can thus be used to select a row for detailed editing. SQL Manipulation is used to load the tree as a series of separate queries.

Illustration 4 Is taken from ibx/examples/ibtreview and uses the Firebird example “employee” database. This database contains a hierarchically organised table “DEPARTMENT” and which is used for the example.

To use a TIBTreeView, simply drop it on to a form, set the DataSource property, and, as a minimum, the TextField, ParentField and KeyField properties as defined below.

The DataSet must have a single primary key field.

4.2 TIBTreeView Properties

DataSource	TDataSource	Identifies the source of the data to present using the tree view
TextField	string	The field name of the column used to source each node's display text
KeyField	string	The field name of the column used to source each node's primary key.
ParentField	string	The field name of the column used to identify the primary key of the parent row. This field is null for a root element.
HasChildField	string	Optional. The field name of the column used to indicate whether or not the row has child nodes. When present, the field should return an integer value with non-zero values implying that child nodes exist.
RelationName	string	Optional. The Child Field is typically the result of joining the table to itself and is a count of child rows. However, this can result in ambiguous column names when the SQL is manipulated. This property should contain the Table Alias used to select the Key, Text and Parent Fields (see example application).
ImageIndexField	string	Optional. If specified then the image index for each node is read from this (integer) field.
SelectedIndexField	string	Optional. If specified then the selected image index for each node is read from this (integer)

		field.
--	--	--------

4.3 TIBTreeView Methods

```
function GetNodePath(Node: TTreeNode): TVariantArray
```

Returns a Variant array containing the primary key values of the Node and its parents from the root node downwards.

```
function FindNode(KeyValuePath: TVariantArray; SelectNode: boolean): TIBTreeNode;
```

Returns the TTreeNode identified by the KeyValuePath. The KeyValuePath is an array comprising a list of primary key values walking the tree down from the root node to the requested node.

If SelectNode is true then the returned node is also selected.

This function can be used to select the tree node using the node path returned by an earlier call to the function GetNodePath.

```
function FindNode(KeyValue: variant): TIBTreeNode;
```

Returns the tree node with the primary key given by KeyValue. Note: this forces the whole tree to be loaded by a call to TCustomTreeView.FullExpand.

4.4 Drag and Drop

Drag and drop is supported by TCustomTreeView without the need for additional support from TIBTreeView. In the example, drag and drop is enabled by:

- DragMode set to automatic
- The OnDragOver Event handled by:

```
procedure TForm1.IBTreeView1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source = Sender;
end;
```

- The OnDragDrop Event Handled by:

```
procedure TForm1.IBTreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var Node: TTreeNode;
    tv: TTreeView;
begin
  if Source = Sender then {Dragging within Tree View}
  begin
    tv := TTreeView(Sender);;
    Node := tv.GetNodeAt(X,Y); {Drop Point}
    if assigned(tv.Selected) and (tv.Selected <> Node) then
    begin
      if Node = nil then
        tv.Selected.MoveTo(nil,naAdd) {Move to Top Level}
      else
        begin
```

```
    if ssCtrl in GetKeyShiftState then
    begin
        Node.Expand(false);
        tv.Selected.MoveTo(Node, naAddChildFirst)
    end
    else
        tv.Selected.MoveTo(Node, naInsertBehind)
    end;
end;
end;
end;
end;
```

Note that the above applies the convention that if the “control” key is held down while the node is “dropped” then it is added as a child node. Otherwise, it is added as a sibling.

4.5 Setting Query Parameters

The dataset used as the TIBTreeView's data source may have a select query that contains query parameters. However, in order to determine child nodes, TIBTreeView manipulates the SQL query “behind the scenes” to change the “Where” clause in order to select only the current node and its child nodes. Because of this, the only “safe” place to set values for query parameters is in the dataset's “BeforeOpen” event handler. This is guaranteed to be called every time the control updates the SQL where clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the “BeforeOpen” event handler will be lost when the control updates the SQL and reset to the default null value.

5

TIBLookupComboEditBox

5.1 Overview

TIBLookupComboEditBox is a TDBLookupComboBox descendent that implements "autocomplete" of typed in text and "autoinsert" of new entries.

- Autocomplete uses SQL manipulation to revise the available list and restrict it to items that are prefixed by the typed text (either case sensitive or case insensitive).
- Autoinsert allows a newly typed entry to be added to the list dataset and included in the available list items.

Although TDBLookupComboBox also supports auto-complete, the benefit of using TIBLookupComboEditBox comes with long lookup lists as typing in one or more characters forces the list to be queried again and restricted to list members beginning with the same characters. The list of alternatives becomes much shorter.

Auto-insert normally uses the list dataset's insert query to add a new row and depends upon the dataset's "After Insert" event handler to set the other fields of the row to appropriate values and/or the generator assigned to the dataset.

5.2 TIBLookupComboEditBox Example

The screenshot shows a window titled "IBLookup Combo Box Demo". At the top, there is a label "Employee Name" above a text box containing "Baldwin, Janet" and a small downward arrow. Below this is a section titled "Employee Details" which contains a grid of fields: "First Name" (Janet), "Last Name" (Baldwin), "Grade" (3), "Location" (USA), "Hire Date" (21-3-91), "Salary" (\$61,637.81), "Job Title" (Sales Co-ordinator), "Department" (Corporate Headquarters / Sales and Marketing / Pacific Rim Headquarters), and "Ext." (2). At the bottom of the window are three buttons: "Delete", "Save", and "Cancel".

Illustration 5: Using the TIBLookupComboEditBox

The above example can be found in `ibx/examples/lookupcombobox` and uses the Firebird "employee" example database. The "Employee Name" is a TIBLookupComboEditBox and is used here to:

- a) Select an employee record for editing
- b) Initiate the entry of a new employee record.

First, you should explore the use of the new control. Click on the drop down arrow and a drop down list of all employee names (in lastname/firstname syntax) will be shown. This is typically longer than can be displayed on a single screen.

The screenshot shows a window titled "IBLookup Combo Box Demo". At the top, there is a text box labeled "Employee Name" containing "Page, Mary" and a small downward arrow. Below this is a section titled "Employee Details" which contains several input fields: "First Name" (Mary), "Last Name" (Page), "Grade" (4), "Location" (USA), "Hire Date" (13-4-93), "Salary" (\$48,000.00), "Job Title" (Engineer), "Department" (Corporate Headquarters / Engineering / Consumer Electronics Div. / Research and ...), and "Ext." (845). At the bottom of the window are three buttons: "Delete", "Save", and "Cancel".

Illustration 6: Selection of a Different Employee

Now close the drop down list, select all characters in the Employee Name edit box and enter "pa". After a short (600ms) delay, after you stop typing, the employee details should change to that shown in Illustration 6 i.e. for the first employee with a lastname beginning with "pa", i.e. Mary Page.

Of course, auto-complete to the first employee beginning "pa" may not get the actual employee you want. Now click on the drop down list and this will show all employees with a last name starting with "pa". This is a much shorter list than the full list and allows you to quickly focus in on the employee you want.

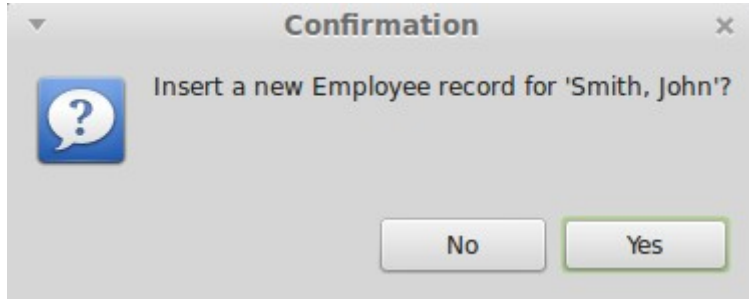
Indeed, this can also be done from the keyboard. Start again, and enter "pa", now press the down arrow and you can cycle quickly through all employees starting "pa". The up arrow also works. Use the Enter key to select the employee record.

Alternatively, after entering "pa" and seeing the entry for Mary Page, then press "r" to extend the entry to "par" and you get the record for Bill Parker.

To return to the full list, just press the escape key while the control has the focus.

5.2.1 Auto-insert

Auto-insert allows quick insertion of new employee records. For example, start by selecting all text in the Employee Name edit box and enter the name of the new employee (e.g. Smith, John), and press the "Enter" key. You should now get a prompt confirming the entry of the new employee record:



If you click on “yes” then a new employee record is created and displayed as show below.

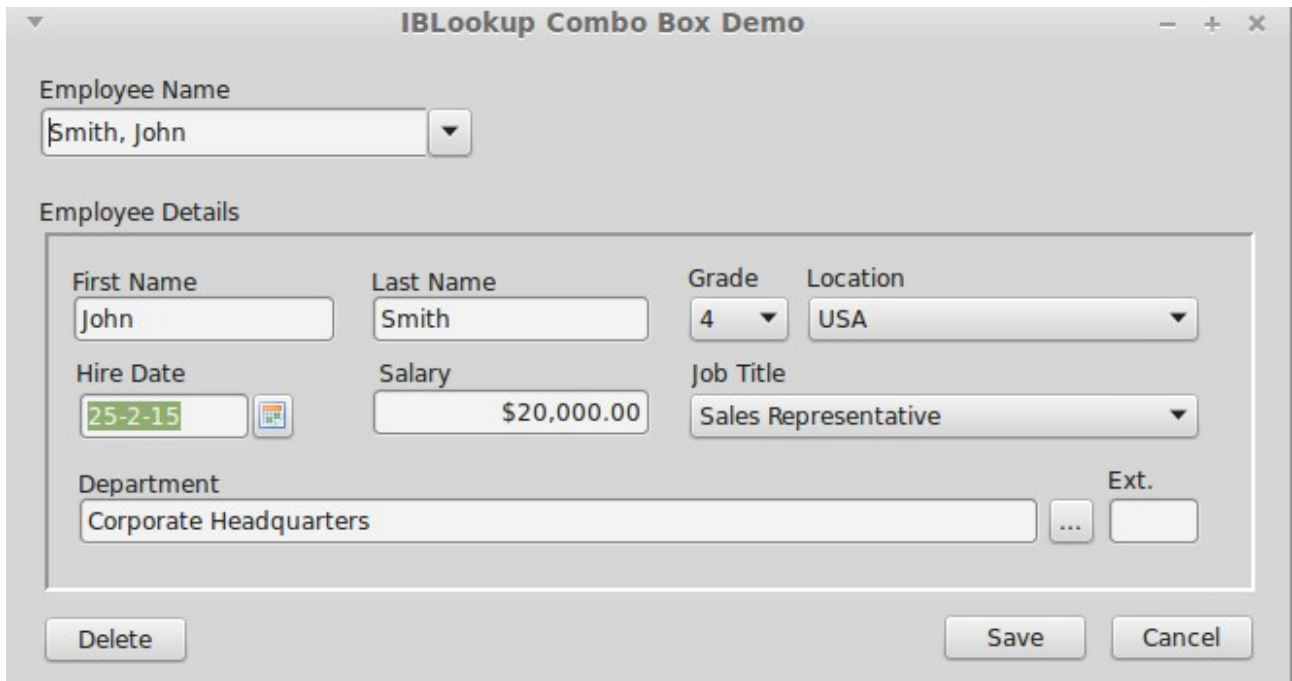


Illustration 7: New Employee Record

The employee name is parsed from the text entered into the Employee Name box. The remaining fields come from defaults taken from the “OnInsert” event handler. You can now amend the defaults as required.

5.3 TIBLookupComboEditBox Properties

TIBLookupComboEditBox inherits TDBLookupComboBox properties. In addition, it defines:

AutoInsert	Boolean	Set to true to enable auto-insert
AutoComplete	Boolean	Default: true in TIBLookupComboEditBox
KeyPressInterval	Integer	Delay in milliseconds between last key press and auto-complete (Default: 500ms).
RelationName	String	TIBLookupComboEditBox updates the “Where” clause in the ListSource select SQL query in order to refine the list, and uses the value of the

		<p>“ListField” property as the column name. If this name is ambiguous in the SQL query then the “RelationName” property must be set to the name of the table or table alias to qualify the column name and remove the ambiguity.</p>
--	--	--

5.4 TIBLookupComboEditBox Event Handlers

OnAutoInsert	<p>TIBLookupComboEditBox will normally use the ListSource's Insert query to perform auto-insert. If this is not possible or inappropriate then an OnAutoInsert handler must be provided to perform the insertion. The handler is provided with the value of the display text to insert and must return the new key value.</p>
OnCanAutoInsert	<p>This handler is called immediately before auto-insertion is performed and is typically used to validate the insert and obtain user agreement (e.g. via a dialog box). The handler is provided with the value of the display text to insert and must set the “Accept” boolean on return to true to accept the insert or to false to reject it.</p>

5.5 Setting Query Parameters

The dataset used as the TIBLookupComboEditBox data source may have a select query that contains query parameters. However, in order to perform auto-complete, TIBLookupComboEditBox manipulates the SQL query “behind the scenes” to change the “Where” clause in order to select only the rows matching the current text (as a prefix). Because of this, the only “safe” place to set values for query parameters is in the dataset's “BeforeOpen” event handler. This is guaranteed to be called every time the control updates the SQL where clause and re-executes the query.

Parameter values set before the dataset is opened and outside of the “BeforeOpen” event handler will be lost when the control updates the SQL and reset to the default null value.

6

TIBArrayGrid

6.1 Overview

TIBArrayGrid is a visual control that can be linked to a TIBArrayField and used to display/edit the contents of a one or two dimensional Firebird array. It may be found in the “Firebird Data Controls” palette.

To use a TIBArrayGrid, simply drop it onto a form and set the DataSource property to the source dataset and the DataField property to the name of an array field. The grid should then be automatically sized to match the dimensions of the array.

Note that the array bounds can be refreshed at any time in the IDE, by right clicking on the control and selecting "Update Layout" from the pop up menu.

At runtime, the TIBArrayGrid will always display/edit the value of the array element in the current row. If this element is null then the array is empty. However, data can be inserted into an empty array. When the row is posted, the field will be set to the new/updated array.

6.2 Properties

Most TIBArrayGrid properties are the same as for TStringGrid. The following are specific to TIBArrayGrid. Note that you cannot set the Row or column counts directly as these are always set to match the array field.

Public Properties

DataSet	The DataSet provided by the DataSource (read only).
Field	The source field (must provide the IArrayField interface)

Published:

DataField	The name of the array column.
DataSource	The data source providing the source table.
ReadOnly	Set to true to prevent editing
ColumnLabels	A string list that provides the labels for each column in the grid. Provide one line per column. If non empty then a column label row is created as a fixed row at the top of the grid.
ColumnLabelAlignment	Sets the text alignment for column Labels
ColumnLabelFont	Sets the font used for column labels
RowLabels	A string list that provides the labels for each row in the grid. Provide one line per row. If non empty then a row label column is created as a fixed column to the left of the grid.
RowLabelAlignment	Sets the text alignment for row Labels
RowLabelFont	Sets the font used for row labels
RowLabelColumnWidth	Width of the Fixed Column used for row labels.
TextAlignment	Alignment of all cells other than those containing labels.

6.3 Examples

Example applications are provided for both one and two dimensional arrays. In each case, the example applications create their own database and populate it with test data when first run. Note that you will typically need to run the application before accessing database properties in the IDE. This is in order to create the database referenced by the IDE.

6.3.1 Database Creation

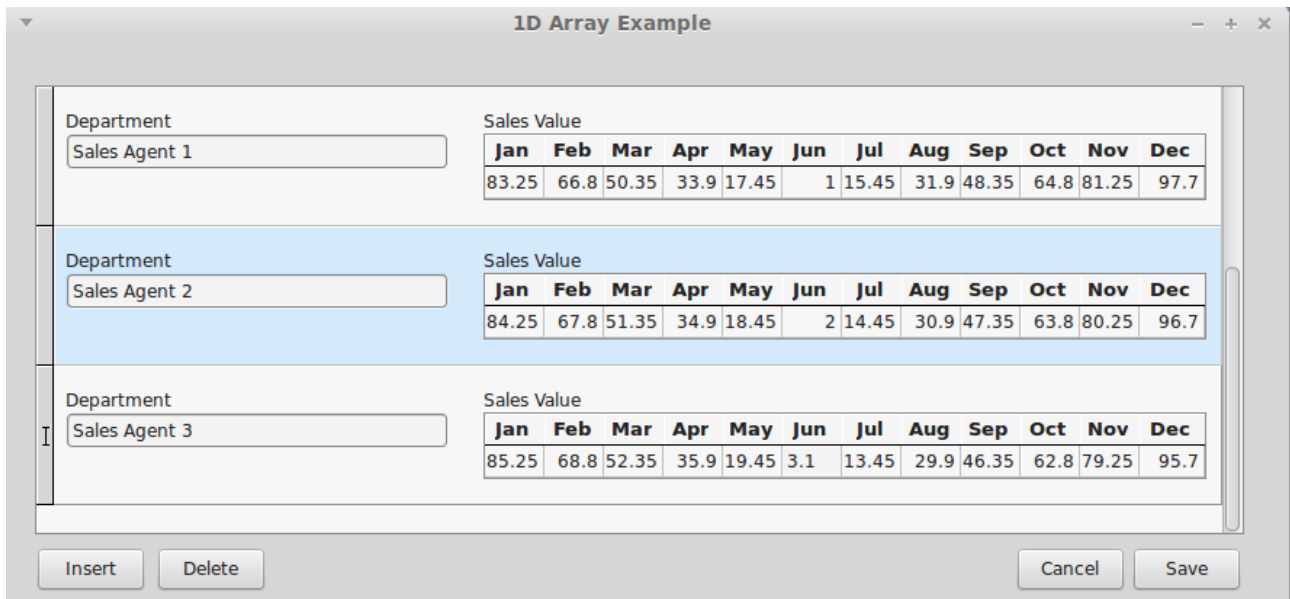
The TIBDatabase property “CreateIfNotExists” is set to true in both examples. This means that if the database does not exist when an attempt is made to connect to it then the database is created. After it is created, the “OnCreateDatabase” event handler is used to add a table to the newly created database and to populate it with test data. The application then continues as if the database already existed.

By default, the database is created in the defined temporary directory. This behaviour can be overridden by editing the example's “unit1” unit to remove the “{\$DEFINE LOCALDATABASE}” directive and setting the const “sDatabaseName” to the required path e.g.

```
const
  sDatabaseName = 'myserver:/databases/test.fdb';
```

6.3.2 1D Array Example

A screenshot from this example program is illustrated below.



In this case, the test data table is defined as

```
Create Table TestData (
  RowID Integer not null,
  Title VARCHAR(32) Character Set UTF8,
  MyArray Double Precision [1:12],
  Primary Key(RowID)
);
```

Each row includes a floating point array with twelve elements. In the example application, the table is displayed and edited using a DBControlGrid. The title field is interpreted as a “Department” and displayed using a TDBEdit control. The array field is interpreted as sales by month and displayed as a one dimensional TIBArrayGrid with column labels. The example allows both the Department Name and monthly sales values to be updated and changes saved. New rows can be inserted and existing rows deleted.

Note: there is an LCL bug (<http://bugs.freepascal.org/view.php?id=30892>) which will cause the 1D array example to render incorrectly under Windows. That is only the focused row will show the array. The bug report includes an LCL patch to fix this problem. It is believed to be fixed in Lazarus 1.8.0.

6.3.3 2D Array Example

A screenshot from this example program is illustrated below.

2D Array Example

Row 1

	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8									
Row 1	A0	A9	A18	A27	A36	A45	A54	A63	A72	A81	A90	A99	A108	A117	A126	A135	A144
Row 2	A1	A10	A19	A28	A37	A46	A55	A64	A73	A82	A91	A100	A109	A118	A127	A136	A145
Row 3	A2	A11	A20	A29	A38	A47	A56	A65	A74	A83	A92	A101	A110	A119	A128	A137	A146
	A3	A12	A21	A30	A39	A48	A57	A66	A75	A84	A93	A102	A111	A120	A129	A138	A147
	A4	A13	A22	A31	A40	A49	A58	A67	A76	A85	A94	A103	A112	A121	A130	A139	A148
	A5	A14	A23	A32	A41	A50	A59	A68	A77	A86	A95	A104	A113	A122	A131	A140	A149
	A6	A15	A24	A33	A42	A51	A60	A69	A78	A87	A96	A105	A114	A123	A132	A141	A150
	A7	A16	A25	A34	A43	A52	A61	A70	A79	A88	A97	A106	A115	A124	A133	A142	A151
	A8	A17	A26	A35	A44	A53	A62	A71	A80	A89	A98	A107	A116	A125	A134	A143	A152

In this case, the test data table is defined as

```

Create Table TestData (
  RowID Integer not null,
  Title VarChar(32) Character Set UTF8,
  MyArray VarChar(16) [0:16, -1:7] Character Set UTF8,
  Primary Key(RowID)
);

```

Each row includes a two dimensional string array with indices 0..16 and -1 to 7. The grid interprets the first index as a column index and the second as a row index (i.e. x,y Cartesian co-ordinates).

The example program displays a row at a time with a navigation bar providing the means to scroll through the dataset, as well as saving or cancelling changes, inserting and deleting rows.

This example illustrates the use of both column and row labels.

7

Interface Specification

The Dynamic Database Controls both provide a Pascal Interface to a Database Access Provider (e.g. IBX) and expect the Database Access Provider to provide a Pascal Interface. These interfaces are specified below and are used to support the dynamic aspects of the controls.

The purpose of these interfaces is to define a set of interfaces that permit the IBControls package to be independent of the IBX package. These interfaces are provided by IBX but may also be provided by other database adapters for Firebird and other databases.

Note the use of CORBA interfaces. In this mode, the interface is not a managed type. Any objects providing such an interface must be both explicitly created and destroyed.

7.1 The IDynamicSQLDataset interface

The IDynamicSQLDataset interface is used to register/unregister a dynamic SQL component with a dataset providing this interface. The component must provide the IDynamicSQLComponent. Interface (see below) Otherwise an exception is raised when RegisterDynamicComponent is called.

Declaration:

```

type
  TDynamicDatasetCapability = (dcChangeDatasetOrder, {supports OrderBy, GetOrderByClause
                                                    and SetOrderByClause interfaces}
                              dcUpdateWhereClause,  {supports Add2WhereClause interface}
                              dcSetParams);          {supports IDynamicSQLParam}

  TDynamicDatasetCapabilities = set of TDynamicDatasetCapability;

  IDynamicSQLDataset = interface
  ['{c94afb6a-a28d-4b2b-b62e-8611816cf21e}']
  procedure RegisterDynamicComponent(aComponent: TComponent);
  procedure UnRegisterDynamicComponent(aComponent: TComponent);
  function GetCapabilities: TDynamicDatasetCapabilities;
end;

```

A Database Access Provider is always a TDataset descendent and provided to the control via a TDataSource. At runtime, a control uses the Pascal “is” operator to see if this dataset provides the IDynamicSQLDataset interface. If this interface is not provided then this is silently ignored - but reduced functionality may be observed (see table below).

If the interface is provided then the Pascal “as” operator is used to access the interface and to test the interface's capabilities by calling the GetCapabilities method. If the required capabilities are not provided then this is also silently ignored as above.

Note: this interface is not used by TDBControlGrid and TIBArrayGrid.

The capabilities required by each control are given in the following table:

Control	Required Capabilities	Reduced Functionality when not available
TIBDynamicGrid	dcChangeDatasetOrder	Dynamic column sorting not available.
TDBControlGrid	None	
TIBTreeView	dcUpdateWhereClause, dcSetParams	Control is effectively unusable
TIBLookupComboEditBox	dcUpdateWhereClause, dcChangeDatasetOrder	Reverts to TDBLookupComboBox
TIBArrayGrid	N/A	

If the IDynamicSQLDataset interface is provided and required capabilities are available then the control registers with the dataset by calling the IDynamicSQLDataset.RegisterDynamicComponent interface.

7.1.1 TDataset Requirements

A dataset providing the IDynamicSQLDataset interface shall:

1. Test that each registered control provides the IDynamicSQLComponent interface (using the Pascal “is” operator) and raise an exception if a control not providing this interface attempts to register with it.
2. Keep a list of controls that are currently registered with it and remove a control from the list if the control subsequently calls the interface's Unregisterdynamiccomponent method.
3. Override its TDataset.DoBeforeOpen method and
 - a) Perform any required SQL Filtering before
 - b) calling each registered control's IDynamicSQLComponent.UpdateSQL method
 - c) calling the inherited OnBeforeUpdate method and finally

- d) If the `dcSetParams` capability is supported, calling each registered control's `IDynamicSQLComponent.SetParams` method.
- 4. When the `UpdateSQL` method is called, the dataset provides a `IDynamicSQLEditor` interface. This allows the control to edit the SQL text using the `IDynamicSQLEditor` methods are constrained by the supported capabilities.
- 5. When the `SetParams` is called, the dataset provides an `IDynamicSQLParam` interface. This allows the control to set query parameters using IBX parameter name conventions.

7.2 The `IDynamicSQLComponent` interface

The `IDynamicSQLComponent` interface is provided by a Dynamic SQL Component such as the Dynamic Database Controls. This interface is used by a Dynamic SQL Dataset to tell the component when it should Update the SQL and when it should set parameters. The `UpdateSQL` procedure is called before the dataset is opened and before the `OnBeforeOpen` event is called. The `SetParams` procedure is called before the dataset is opened and after the `OnBeforeOpen` event is called. This sequence allows a user to set dataset parameters in an `OnBeforeOpen` event handler, while allowing the component priority over setting any parameter values - typically those included in conditional parts to the Where Clause added by the component.

It also allows the user to perform additional SQL editing as part of an `OnBeforeOpen` event handler.

```
IDynamicSQLComponent = interface
  ['{4814f5fd-9292-4028-afde-0106ed00ef84}']
  procedure UpdateSQL(SQLEditor: IDynamicSQLEditor);
  procedure SetParams(SQLParamProvider: IDynamicSQLParam);
end;
```

Note: a component must provide both methods even when it does not use them. In such cases, the method implementation is empty.

The `UpdateSQL` method may be used to perform SQL editing. For example, an `TIBDynamicGrid` may change the dataset sort order to correspond with the last time a user clicked on a column heading.

The `SetParams` method may be used to set a query parameter. For example, a `TIBTreeView` uses this method to set the current root element.

7.3 The `IDynamicSQLEditor` interface

The `IDynamicSQLEditor` interface allows a user to update the dataset's select SQL and is provided as a parameter to `IDynamicSQLComponent.UpdateSQL`.

In the IBX implementation, this interface is provided by an instance of the `TSelectSQLParser` class, created by the dataset for the purpose of SQL editing.

```
IDynamicSQLEditor = interface
  ['{3367a89a-4059-49c5-b25f-3ff0fa4f3d55}']
  procedure OrderBy(fieldname: string; ascending: boolean); overload;
  procedure OrderBy(columnNo: integer; ascending: boolean); overload;
  procedure Add2WhereClause(const Condition: string; OrClause: boolean=false;
    IncludeUnions: boolean = false);
  function QuoteIdentifierIfNeeded(const s: string): string;
```

```
function SQLSafeString(const s: string): string;
function GetOrderByClause: string;
procedure SetOrderByClause(const Value: string);
end;
```

7.3.1 Methods

Method	Required Action	Capability
OrderBy	the select SQL Order By clause is updated to order the dataset by the given field name or column number in ascending or descending order.	dcChangeDatasetOrder
Add2WhereClause	<p>the select SQL Where clause is updated to add the provided condition. This condition may include parameters in the IBX parameter syntax i.e. a valid SQL identifier preceded by a colon.</p> <p>Note this is the same syntax as used in procedural SQL.</p> <p>When "OrClause" is true, the condition is added as an "Or" to any existing condition. Otherwise, it is added as an "And".</p> <p>When "IncludeUnions" is true, the condition is added to the Where clause of every union in the select statement. Otherwise, only the first is updated.</p>	dcUpdateWhereClause
QuoteIdentifierIfNeeded	The returned value is the same as that given by the parameter but double quoted if the parameter is not a valid SQL Identifier.	
SQLSafeString	parses a text string and adds escapes any unsafe SQL sequences e.g.. embedded single quotes.	
GetOrderByClause	Returns the current text for the Select SQL Order By clause	dcChangeDatasetOrder
SetOrderByClause	Replaces the current text for the Select SQL Order By clause with "Value".	dcChangeDatasetOrder

7.4 The IDynamicSQLParam interface

The IDynamicSQLParam interface is provided by a Dynamic SQL Dataset. This allows the caller to set query parameter values to any valid value for the parameter type, including the "null" value.

The interface is provided as a parameter to the `IDynamicSQLComponent.SetParams` method. This method is only called when the dataset provides the `dcSetParams` capability.

Parameters are accessed by parameter name where a name is a valid SQL identifier preceded by a colon (':').

```
IDynamicSQLParam = interface
  ['{02dc5296-25e0-4767-95f5-9a4a29a89ddb}']
  function GetParamValue(ParamName: string): variant;
  procedure SetParamValue(ParamName: string; ParamValue: variant);
end;
```

7.4.1 Methods

Method	Required Action
GetParamValue	Returns the current value, if any, of the named parameter.
SetParamValue	Updates the the current value of the named parameter to that provided.

7.5 The IArrayField interface

The `IArrayField` interface provides access to a `TField` instance that is for an array field.

When a field is assigned to a `TIBArrayGrid`, an exception is raised if it does not provide the `IArrayField` interface.

```
IArrayField = interface(IArrayFieldDef)
  ['{1c2492a4-09c7-4515-852e-f6affc6f78da}']
  function IsEmpty: boolean;
  function GetEltAsString(index: array of integer): string;
  procedure SetEltAsString(index: array of integer; aValue: string);
end;
```

FPC itself does not provide a `TArrayField` type as a standard subclass of `TField`. However, IBX does provide a `TIBArrayField` as a `TField` subclass for support of Firebird array columns. The `IArrayField` interface provides access to the additional methods of `TIBArrayField` whilst avoiding an IBX dependency. In principle, another Database Access Provider could also provide its own version of `TIBArrayField` and support the `TIBArrayGrid` control by also providing the `IArrayField` interface.

7.5.1 Methods

Method	Required Action
IsEmpty	Returns true if the array is empty.
GetEltAsString	Returns the current value, as a text string, of the identified array element. Note the index is an array of integers, one integer for each

	dimension of the array.
SetEltAsString	Updates the the current value of the identified array element. Note the index is an array of integers, one integer for each dimension of the array.

7.6 The IArrayFieldDef interface

IBX provides a TIBArrayDef class as a subclass of TFieldDef. This class provides access to additional information about the array. The IArrayFieldDef interface allows a TIBArrayGrid to access this information whilst avoiding an IBX dependency.

TIBArrayGrid searches the dataset's field defs to find the fielddef for the array field (by name) and expects this fielddef to provide the IArrayFieldDef interface. An exception is raised if it does not.

```
IArrayFieldDef = interface
  ['{10d1c460-168f-40a8-b98c-05c6971c09f5}']
  function GetArrayDimensions: integer;
  function GetArrayLowerBound(dim: integer): integer;
  function GetArrayUpperBound(dim: integer): integer;
end;
```

7.6.1 Methods

Method	Required Action
GetArrayDimensions	Returns the number of dimensions in the array. Currently TIBArrayFrid supports a maximum of two dimensions.
GetArrayLowerBound	Gets the index of the lowest element in the given dimension.
GetArrayUpperBound	Gets the index of the highest element in the given dimension.