List boxes (Part II)

By Jean-Yves Fock-Hoon, Quality Assurance Manager, 4D Inc.

Technical Note 06-26

Overview

The purpose of this technical note is to introduce list boxes through the Design mode and through the language. Part II is dedicated to the use of list boxes in through the language.

Creating a list box from the language

In this example, we are going to use a form with a list box that includes several columns and demonstrate the use of 4D commands to set the columns in that list box.

Select "Building a List box" from the menu bar, a dialog is displayed, which uses the [Contacts];"Dial1" form.

The Restart button resets the area to its default settings.

We will start by clicking on the Clear All button. This button uses the **Get number of listbox columns** command to retrieve the number of columns in that list box (invisible or visible). We then use the **DELETE LISTBOX COLUMN** command to delete all columns. The list box area is now empty.

We can now click on the next button, Insert Default. This button is going to insert back the four deleted columns. This implies the insertion of a column and its header title. This is done by calling the M_InsertColumn method, which uses the INSERT LISTBOX COLUMN and BUTTON TEXT commands. The columns are inserted, with default settings for font names, font size and colors.

We can now click on the next button, More Columns. This button is going to insert four more text columns with the use of the same M_InsertColumn.

You have to insert all columns before defining their attributes when executed in the same method. You cannot insert one column, define its attributes, and insert the next column etc, within in the same method call.

Once all columns have been added, we can click the Attributes button. This button calls the M_SetupColumn method, a generic method that uses the SET RBG COLORS, FONT, FONT SIZE and FONT STYLE commands. This method changes attributes such as the background color for each column in the list box. The object method will then revert the background column of the at_city column for display purposes.

The next button is the Grid button. This button uses the SET LISTBOX GRID COLOR and SHOW LISTBOX GRID commands to hide or show the grid or display it with different colors. You can click several times on that button to see the 5 possible settings.

We can now go to the next button: Re-Order. This button is going to reorder our columns. There is no command to move columns within the list box. We need to simulate the order. The trick is very simple: We need to:

- retrieve the number of columns,
- retrieve their information and their current width,
- order the columns,
- delete all current columns
- re-insert them back in the requested order.
- We also need to retrieve the object names and variable names used in that list box, which can be done with the GET LISTBOX ARRAYS command.

To retrieve the current width for each column, we use the Get listbox column width command. In the same loop, we can also initialize an array that will be used later for the order. We can delete all columns in the list box and start to insert each of them back. However, before, we need to redefine the new order by calling the M_Setorder method. We use a local array, $al_newOrder in which we are going to find the object name of the column and assign a column order. If the column has not been found, nothing is done. If a column has not been called, its current order remains the same, as previously initialized with the number of columns.

Once all order has been defined, we can sort all the arrays based on that new order and perform a simple loop to insert each column with INSERT LISTBOX COLUMN. Once all columns have been inserted, we can assign their column header titles and restore their former width.

Our last step is to define some colors and font attributes.

The last button would be the Best Object Size button. We can use the GET LISTBOX ARRAYS command again to retrieve all the columns and perform a loop on each column. BEST OBJECT SIZE will return the best width for both the header and column. The biggest width is then applied to the column in order to fit the longest value.

Moving Columns and Rows

The purpose of this example is to show how to move or resize columns and how to implement a total line at the bottom of the list box. This example uses the [Contacts];”Dial2” form.

The concept is to always track the width of each column after resizing or swapping a column. We need to know the position of the list box. This can be retrieved by using the GET OBJECT RECT command on the list box. We know the number of columns and can retrieve the current width of each of them using the Get listbox column width

command. With the current coordinates of the list box, we can compute the exact horizontal position for each single column since it's a very simple addition.

We know the height of our Total line. If not, we can still use GET OBJECT RECT. Based on the coordinates of our list box, we can compute the vertical coordinates of our Total line in the dialog. The width of the Total line needs to match the width of the columns. We can now retrieve the horizontal coordinates of our Total line in the dialog.

The trick is now to move the Total line each time the list box is modified. We can move the line using the MOVE OBJECT command. We just need to know when to trigger this. This means that we need to detect when a column has been resized or moved. This can be done by testing the current form event. If a column has been resized or moved, we need to detect what has been resized and where in order to move and resize our total line. This is why our object method is triggered only based on those 2 events.

If a column is moved, we need to re-compute the horizontal coordinates of each column in order to move the Total line to match the appropriate column. We also need to know which column our total line must match with. We can use the MOVED LISTBOX COLUMN NUMBER to re-insert the pointer to our column array of the moved column in the appropriate order, re-compute the width for all columns and re-update the position of our Total line.

If a column has been resized, the column order did not change. Only the widths of the two columns affected by the resize have been changed. Since there is no command that can tell us which columns have been resized, it would be safer to re-compute all widths again, and update the position of our Total line.

As you may have noticed, the dialog can be resized. The Total line is vertically movable but horizontally fixed. If you resize the dialog vertically, 4D automatically moves the object. However, the horizontal grow attribute has been disabled for technical reasons: Increasing the resize is very easy. Once the cursor has been released, you can re-compute the widths of each column and resize the Total line. This part works perfectly. A problem occurs when you try to downsize the dialog. You cannot decrease the size of a dialog because of the fixed width of the Total line. A quick workaround would be to make those variables grow horizontally. With this setting, we can decrease the size of the dialog further but we are still stuck since you cannot resize and define the Subject line to a width less than a few pixels. Since this solution is not perfect, this is why the automatic horizontally resizing has been disabled.

Simple Drag and Drop

The purpose of that example is to show how to implement a simple drag and drop between two list boxes within the same dialog. This example uses the [Contacts];"Dial3" form. In the form, we have two list boxes, MyListBox and MyListBox2.

We want to drag and drop/insert items from MyListBox2 to MyListBox.
To do so, we need to enable MyListBox2 as draggable. If this check box is unchecked, the MyListBox object won't receive the drag and drop events. MyListBox also needs to have the Droppable feature enabled in order to receive those events. Once done, MyListBox can now test both "On Drag over" and "On Dropped" form events. During the "On Drag Over" form event, we can execute the DRAG AND DROP PROPERTIES command and see where the 'drop' is coming from and what is dropped. Since we only want to authorize a drop from our second list box, we can test the current pointer to see if it is really coming from MyListBox2. If the item is coming from MyListbox2, we can now check if the current value has already been dropped. If yes, we are going to deny the 'drop'. To do so, we are going to deny all 'drops' as default behavior. If the item comes from MyListBox2 and is not a duplicated item, we can then allow the drop. To allow the 'drop', just set $0 to 0. To deny the 'drop', set $0 to 1.
If you deny the drop, the object will not be redrawn with that "droppable area" visual effect.

Drag and Drop between processes

The purpose of this example is to show how to drag and drop an item between two list boxes located in two separate processes This example uses the [Orders];"Input" and [Contacts];"DialProduct" forms. This example is very similar to the previous example. The first process is the product list process, a very simple list box that contains all available products from our data file. The second process executes the ADD RECORD command with a list box. We can drag and drop an item in that list box. Using the commands DRAG AND DROP PROPERTIES, RESOLVE POINTER and GET PROCESS VARIABLE, we can retrieve the values to be inserted in the list box. This part is very easy and very short in terms of code, unlike the code for the synchronization of the invoice itself.

Once an item has been inserted, you can change its quantity. Enter 0 to remove that item from the invoice or enter another value. If you enter a non-null value, you have to re-compute the total of that line. If you enter 0, we have to remove all elements related to that product in our arrays. In both cases, we have to re-compute the total of the invoice.

To do so, we have to compute the total without the current line. This can be done from the "On getting focus" event when editing the quantity. If we delete the line or if we change the quantity, it will be then easy to compute the current total of the invoice. If the line has been deleted, the current total is the previous total. If the quantity has been changed, the current total is the previous total plus the new total of

the line. All of this can be computed during the “On Data Change” form event of our list box.

Editing a Cell

The purpose of this example is to show how to scroll down rows or edit a row in a list box. This example uses the form [Contacts];”Dial5”. Our form has a list box with 4 columns. 2 columns are visible and 2 are invisible. We are going to display in that list box the current list of contacts from the data file.
When displaying this dialog, you can only see two columns. We can try to edit/modify any name. As soon as we are editing a name, we can see that the two other columns are visible. This can be done by testing on the “On Getting Focus” form event and by using the SET VISIBLE command. When displaying the dialog, the “On getting Focus” form event is also triggered. This is why we are using the vOnLoad variable to prevent the display of those columns when displaying the dialog for the first time.

After modification, we need to hide those columns again. We can use the SET VISIBLE command again from the form events “On Data Change” or “On Losing Focus”. There are two ways to stop editing an entry. You can hit the Enter key to validate your entry or you can change the current focus by clicking somewhere else such as a button without changing any data. This is why we are executing this in both form events.

If we click on any of these buttons, we are going to re-order the arrays and find the first name starting with the current letter. The sort can be done with MULTI SORT ARRAY. This is not mandatory but logical if we want to edit the first name starting by that letter. Once a name has been found, we are going to scroll down to show the row that contains the first value. This can be done by executing the SCROLL LINES command. We can now edit the item using the EDIT ITEM command. Both the SCROLL LINES and EDIT ITEM commands require a row number, returned by the Find in array command. Since we are editing an item, we should not forget to set our two other columns to visible again.
If we click on the J button while editing a name, we can see that there is no name starting by J and that we are no longer editing any name. This is because we lost the focus when clicking on that button and since there was no name found in our array, we executed neither SCROLL LINES nor EDIT ITEM.

Form Events

The purpose of this example is to demonstrate how to use the six new form events specifically implemented for list boxes. This example uses the form [Contacts];”Dial6”.

- The “On Column moved” form event:
This form event is used with the MOVED LISTBOX COLUMN NUMBER command. We can detect which column has been moved from where to where. This form event and command is useful to track down the order of columns. In this example, we can keep track of the column order and compute their width. Based on that data, we can move

a variable to the bottom of the list box, the SumAmount variable, following the position of the Amount column.

- The "On Column Resize" form event:
This form event allows us to detect the resizing of any column. In our example, the column order has not changed but their widths did. We can compute them and move or resize the SumAmount variable; the resizing of a column may have pushed the Amount column.

The move of that variable can become very tricky to execute. In both cases, the Amount column can be very small or can even be outside of the visible list box. In both cases, the variable must not be displayed. This is why we are executing the GET OBJECT RECT and GET FORM PROPERTIES commands to retrieve the position of the list box and the size of the form. We have the new width of the Amount column and its position. If the position is greater than the width of the list box, it means that the column his hidden. In this case, we are going to move our variable outside of the visible dialog. This is why we need the width of the form. The problem is that this also requires to not display the horizontal scrollbar. If there is a horizontal scrollbar, we won't be able to detect its use. If the column becomes now visible, we won't be able to detect it and display the variable.

A workaround would be to display that variable with fixed coordinates when the Amount column becomes hidden or when the variable becomes too small and still have the horizontal scrollbar. However, that also means that this variable would be moved under the Amount Column only when the Horizontal scrollbar is removed by 4D.

Note: In the code, $x1 and $x2would be the final coordinates of our variable. They are computed based on the position of the column and the size of the list box. We are removing 15 from the list box width because of the display of the vertical scrollbar.

- The "On Header Click" form event:
This form event allows us to detect the click on the header of any column. From this form event, you can execute your own code such as changing some text attributes. If the column is sortable, the click on that column header will sort that column. You can deny the sort by executing $0:=1.
If you click on the Amount column header, we can see that the form event is executed because the title becomes blue. The object method also tests the name of the variable and if it is the vAmount variable, a $0:=1 is executed. The sort on that column is then denied. Since no sort has been performed, the "On After Sort" form event is not triggered.

- The "On After Sort" form event:
This form event is executed once a sort has been performed after clicking on the header of the column. This event can be useful to execute some specific code such as automatically selecting the first new row or using the values of the first rows for specific computations. In our case, we have previously defined an array of pointers on each column header variables. You can also generate that array by executing the GET

LISTBOX ARRAYS command. If the variable contains 0, the variable has not been clicked. If it contains 1 or 2, a click or a reverse-order click has been performed on that variable. We know which column header has been clicked. In that case, we are going to add an extra style to our header title: Italic or Bold.

- The "On Before Data Entry" form event:
This form event is executed when we are about to edit a cell. In our example, we are going to see which row we are currently editing and deny the edit if it's one of the first rows by posting the Escape character to the current process.

- The "On Row Moved" form event:
This form event is executed when a row is moved. You can detect which row has been moved using the MOVED LISTBOX ROW NUMBER command.
In our example, we are just going to display the old and new positions of the moved row.

Multiple pages

The purpose of this example is to show how to display multiple list boxes on multiple pages in a form. This example uses the form [Contacts];"Dial7". We have the list boxes MyListBox, MyListBoxP1, MyListBoxP2 and MyListBoxP3 respectively on page 0, page 1, page 2 and page 3.

The use of this example is very simple. Select any item from the list box on page 0 and all others list boxes will be updated. You can click on the tab control to switch pages. You can display as many list boxes as you want. However, there is a rule that you should not forget. Variables must be unique. You cannot use the same variables or arrays in the same list box or in different list boxes on the same page (current page AND page 0). Displaying many times the same variable on the same page may work for now but this feature is not supported.

This example also shows all the types of arrays that list boxes can display and re-uses some of he features already covered in the Preview example such as defining a small icon next to the column header label, or the use of choice list in the Car column.

When changing pages, a call to the M_BestObjectSize is executed. This method computes the best object size for the Mileage, Total and Margin columns and redefines the widths of these columns based on these sizes.

Puzzle

The purpose of that example is to show another way to use list boxes. This example uses the form [Contacts];"DialPuzzle". This is a puzzle with a picture made of 15 other pictures. Click on the reset button to shuffle all pictures and try to rebuild the logo. This list box contains 8 columns. 4 columns contain a picture loaded from the picture library. The 4 other columns are LONGINT columns that will contain the position of those pictures in the matrix, especially the position of the empty spot that allows the

move of each other squares. The code used to track each move is pretty simple. We are just searching where the empty spot is, in order to perform a swap between both elements. As we can see, no other commands specific to list boxes are required. Everything is a matter of swaps. The list box has been used just as a frame, a display for our arrays.

Summary

A list box is a very powerful object that allows you to display your arrays with more controls than scrollable areas. Part I of this technical note explored the attributes of list boxes as you define them in the Design environment. Part II explored the multiple possibilities offered by handling list boxes through the language.