

# Forms & Dialogs

On the previous sections I have written code to perform specific actions to enhance the capabilities of OOBBase. In the real world, this alone is not all that practical, however. A general user should not be expected to open the Basic editor, and call a desired function or sub routine. Even for an experienced user this is cumbersome and unproductive. The objective of what was learned on the previous sections is to be able to bind it to some sort of User Interface Element such as a menu, tool bar, a form in Writer (OBase Form) or a Dialog.

## Base Forms

Forms can be accessed view the *forms* option on the Base document main window. Much like the other objects, the forms sub window is divided into two sections:

1. Form Options (tasks)--allows creation of forms via the form wizard or from the design view.
2. List of current forms—Double-clicking on the form will open the form in Data Mode which allows the entry manipulation of the underlying data source. The window title will read (*read-only*). This refers to the fact that the form itself cannot be edited. However, you are still able to add, update, and delete records. The form properties such as data source and form controls can be edited by right-clicking on the form name and selecting *Edit* from the context menu. Other options such as *Rename* can also be found on this context menu.

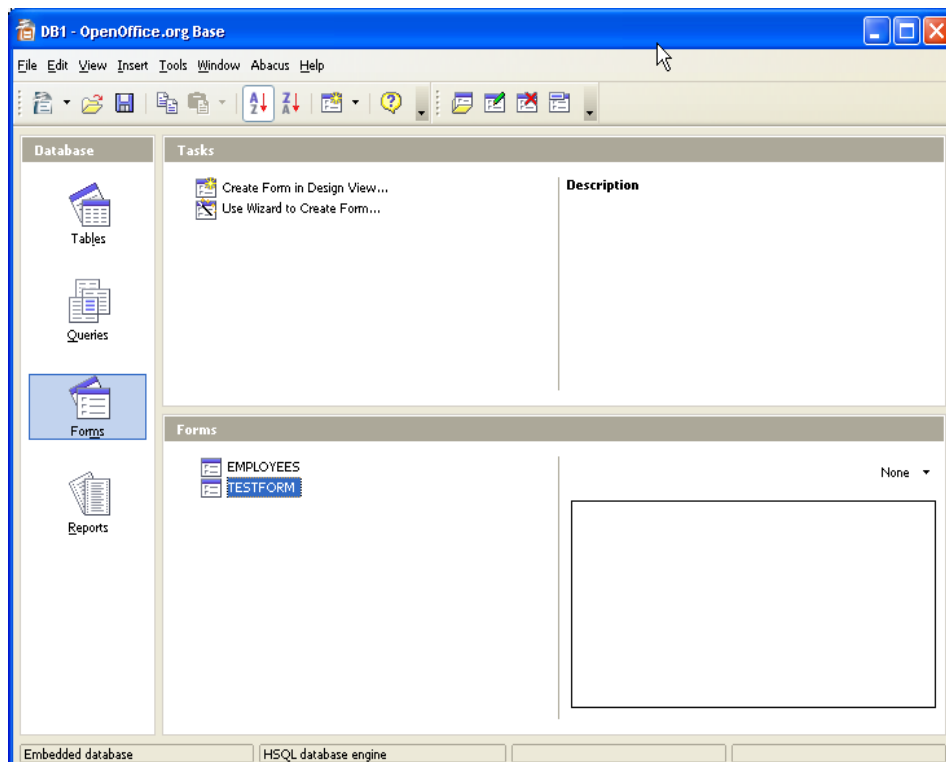


Illustration 1: Base Document

## The Form Wizard

Let us begin the form tour by looking at the form wizard. Start the wizard by selecting the *Use Wizard to Create Form* option.

**Step 1—Field Selection.** Page one of the wizard, shown on illustration 2, allows the selection of the data source (table or query). After the source has been selected, the columns available are displayed on the *List Box* labeled *Available Fields*. Select the desired fields by highlighting the field and clicking on the button with the single arrow pointing to the right. If all the fields are desired, click on the button with two arrows pointing to the right. To remove selected fields from the second list box (Fields in the form) select the desired field(s) and click on the button with single or double arrow pointing to the left (which ever is needed). Note that once the fields are selected, the two buttons on the far right with up and down arrows, respectively, become active; this allows the reordering of fields. After selecting the desired fields, we can click on *Finish* and the default values will be used for the rest of the settings, or we can go to the next step. Steps 2, 3, & 4 are needed when adding sub forms. For now, we are going to create a simple form, so we can go to step 5—Arrange Controls.

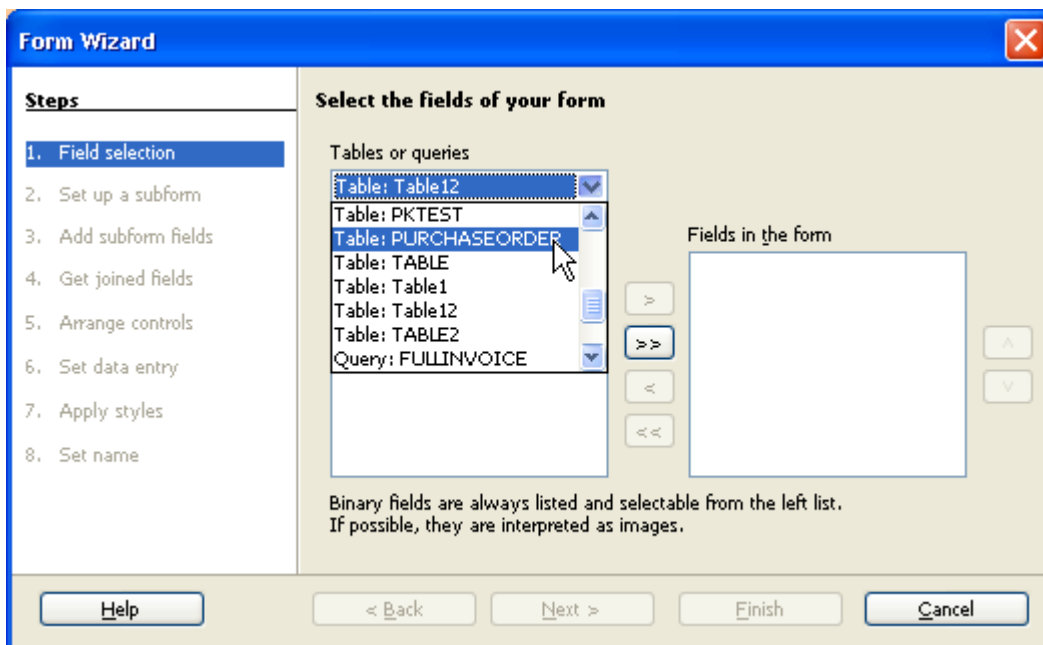


Illustration 2: Form Wizard Page 1

**Step 5—Arrange Controls.** In this step you can select the label text alignment, as well as the control (label plus data control) layout. Additionally, Data Sheet layout is available. The Data Sheet option utilizes the Table Grid Form Control, and basically displays the columns and rows similar to the table view. However, this options allows the binding of macros to control events. See illustration 3 for a screen shot of step 5.

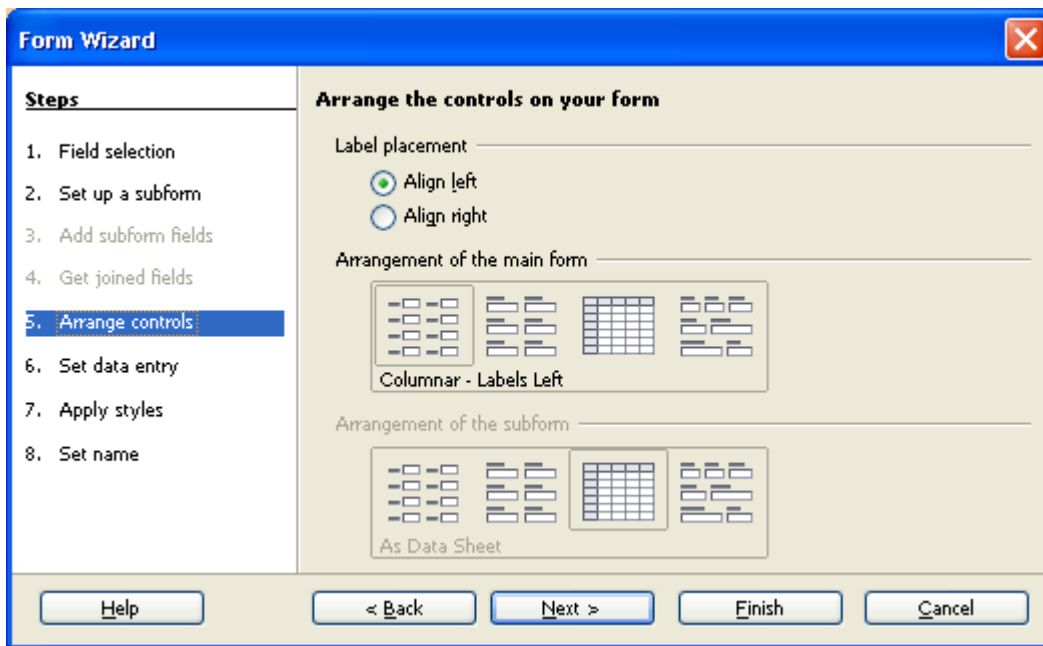


Illustration 3: Form Wizard Step 5

**Step 6—Data Entry.** This step is also quite brief. There are two main options. Select the first to only allow data entry—existing data will not be displayed. Option two displays all data, but also gives the option to add restrictions.

1. Do not allow modification of existing data.
2. Do not allow deletion of existing data.
3. Do not allow addition of existing data. *This might be useful for a search form, to prevent accidental addition of search parameters as a new record.*

**Step 7—Apply Styles.** In this step you can select the look and feel of the form and controls—see illustration 4. Note that the settings are automatically applied to the form on the background.

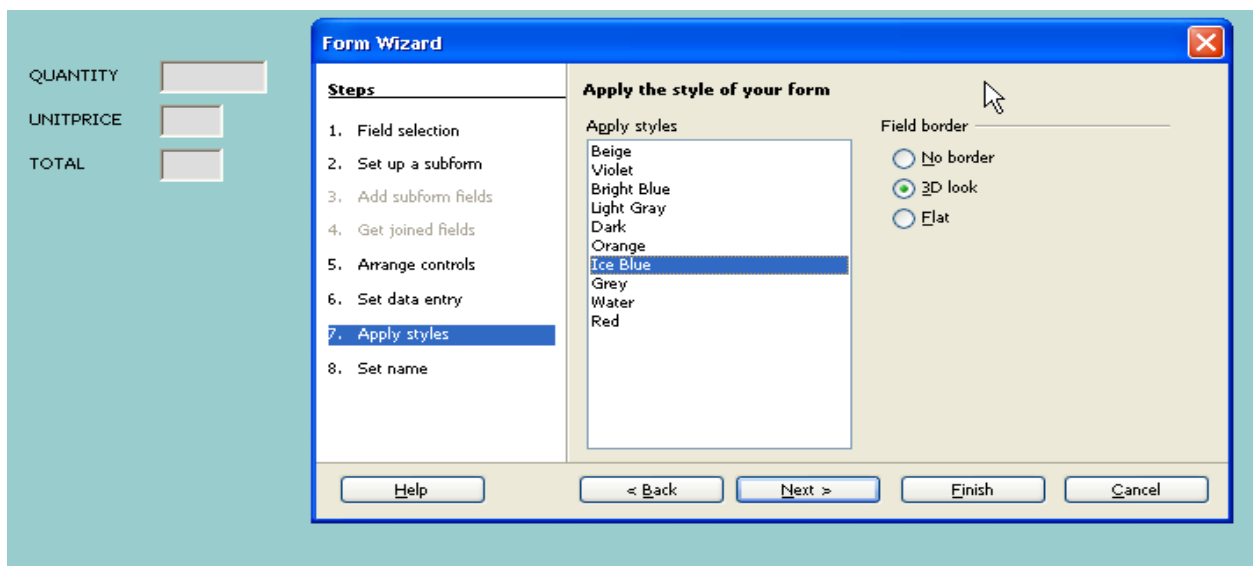
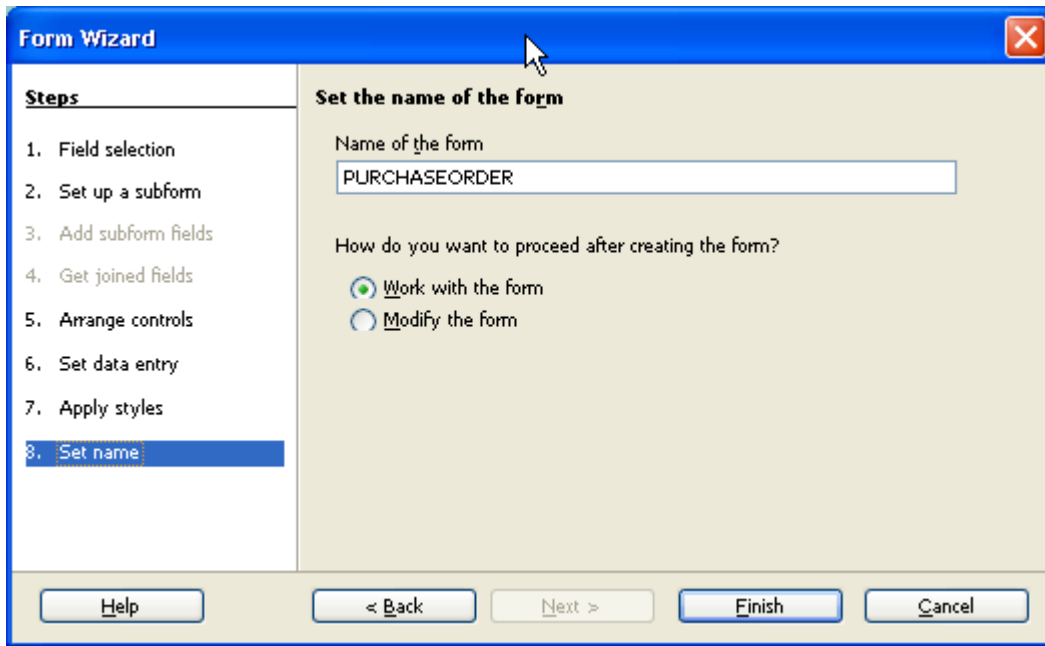


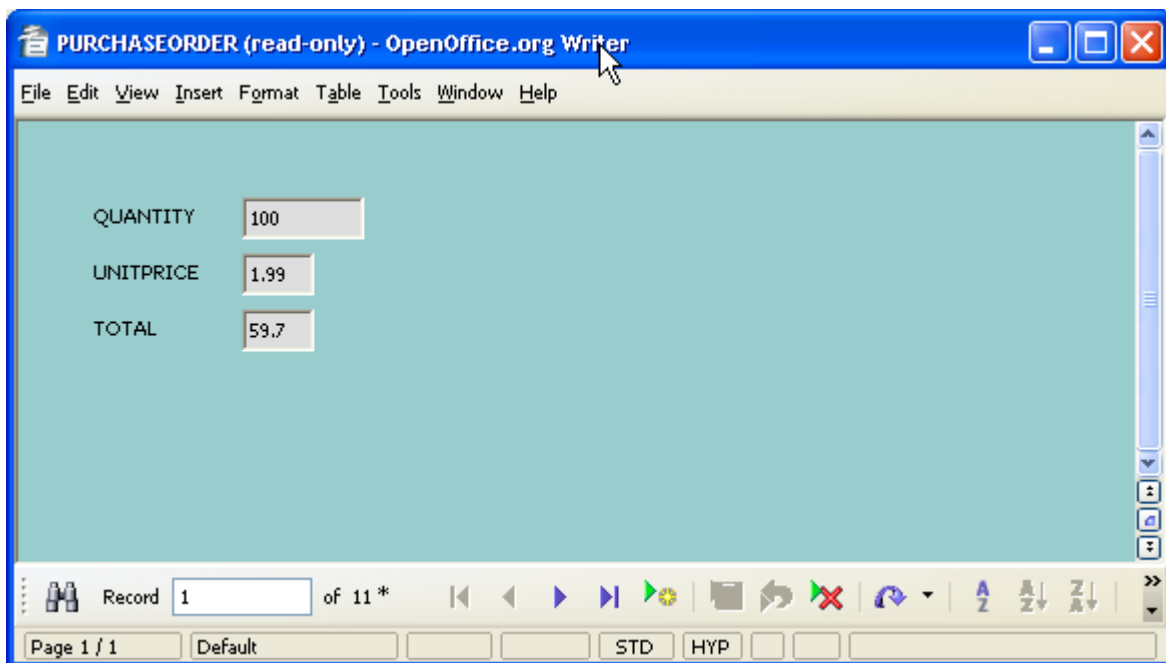
Illustration 4: Form Wizard Step 6

**Step 8—Set Name.** Finally, you can set the name of the form, and opt to open the form in Edit or Data Mode. See illustration 5



*Illustration 5: Form Wizard Step 8*

Illustration 6 shows the finished form. In this screen shot I have manually removed all the useless tool bars. This is one of my biggest quarrels with the current version of OpenOffice.org—Some toolbars are difficult to remove once they are opened (e.g. Form controls, drawing controls, etc). Or at least that is the case on my installation. I will show you a possible solution at the end of the chapter for removing unwanted toolbars..



*Illustration 6: Finished Base Form*

## Form States/Modes

After the last step of the form wizard, the form is completed and ready for use. However, it is seldom sufficient to use a wizard to create exactly what you need. Generally, it is necessary to open the form document in design view and make modification—especially code binding.

A form can be in one of three states or modes:

1. **Edit Mode**—When a form is in edit mode, you can modify the form settings as well as add/remove form controls and edit the form control settings. As a programmer/database designer, this is the mode with which you will be most involved.
2. **Data Mode**—This is the mode used by the end user. The data mode allows the modification of the data source which is bound to the form. Additionally, you can search records with specified criteria and filter the data source.
3. **Filter Mode**—When the form is in Data Mode, you can switch to Filter Mode, and enter filter criteria on the columns to filter the underlying data source. Filtering is different than searching for records matching certain criteria. When performing a search, the cursor will move to the first record that matches the search criteria. When using the filter, however, only the records that match the search criteria will be accessible through the form; this is similar to creating a query object on a particular table. The originally data source can be retrieved by removing the filter.

As this guide is concerned with database programming/development, greater emphasis will be placed on the edit mode; I will briefly discuss the other two, however, to demonstrate when it is best to use the built in features rather than writing code.

Before diving into form design, I will manually create the same form as in the previous section. Start by going to *Forms* and then select the *Create form In Design View*. This will open a form document (writer document). Form here on is up to you what you want to do with the form—see illustration 7. The first step is of course to select a data source.

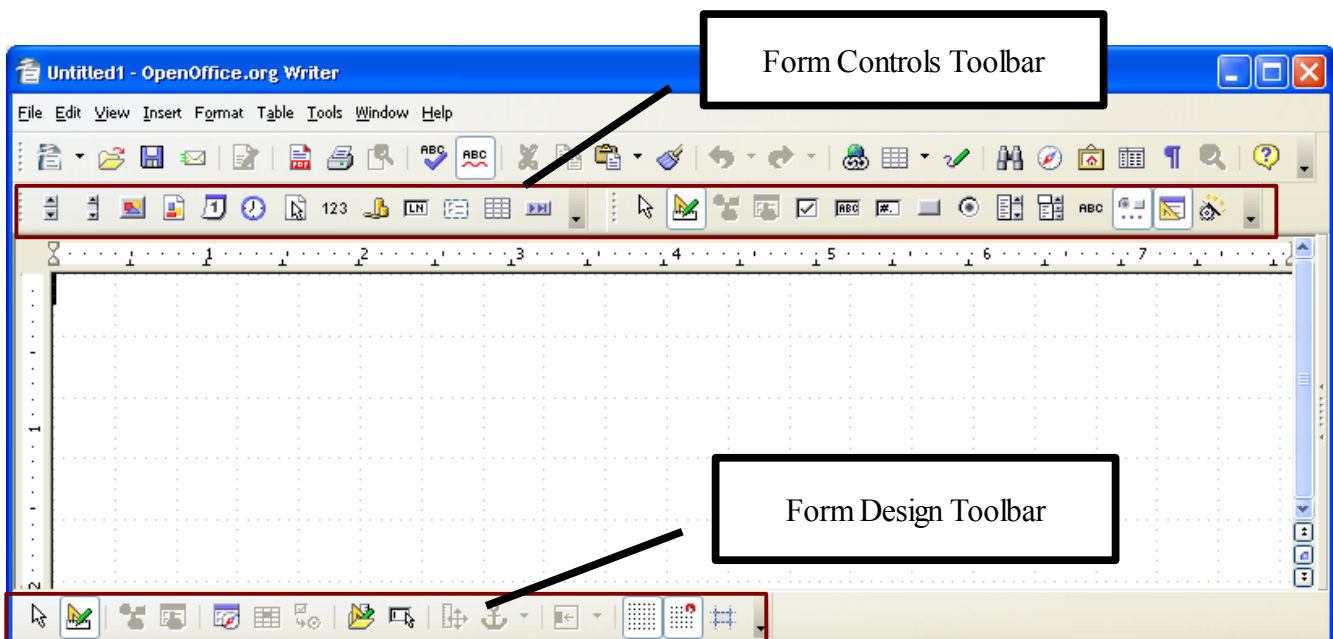


Illustration 7: Creating A Manual Form

As with most GUIs, an element can be modified by opening the properties dialog. However, when creating a form manually, the properties dialog button is disabled for the form and form controls.—see illustration 8.

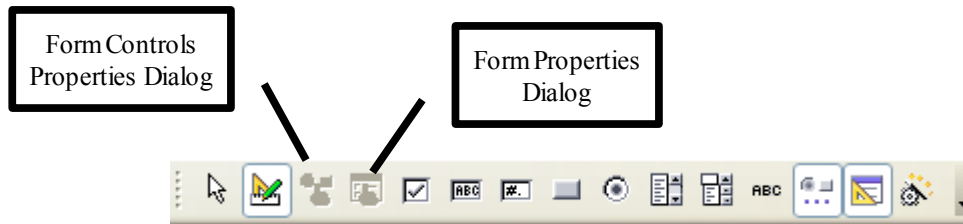


Illustration 8: Disabled Properties Dialog

This is due to the fact that, at the moment, there are no forms in the form document. A form will be added when a form control is dropped in the design area (the white grid on illustration 7), or by opening the *Form Navigator* (Form Design Toolbar) and adding a form manually—The *Form Navigator* as well as multiple forms and sub forms will be covered later in this chapters. After the form is added to the form document, the properties dialog button becomes enabled, and the form attributes can be modified. The form properties dialog consists of three tabs. General, Data, and Events. At the moment I want to select a data source for the form, so it is only logical to open the Data Tab. See illustration 9. The *Content type* field is one of three options: Table, Query or SQL Command. The *Content* field specifies the actual data content/source.

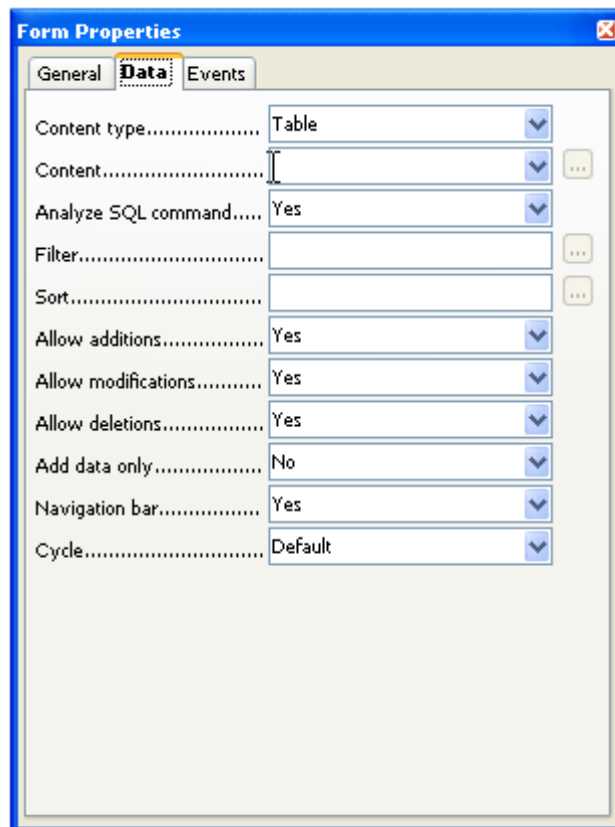
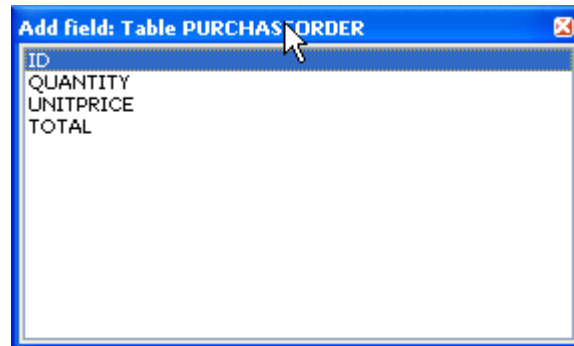


Illustration 9: Form Properties Dialog--Data Tab

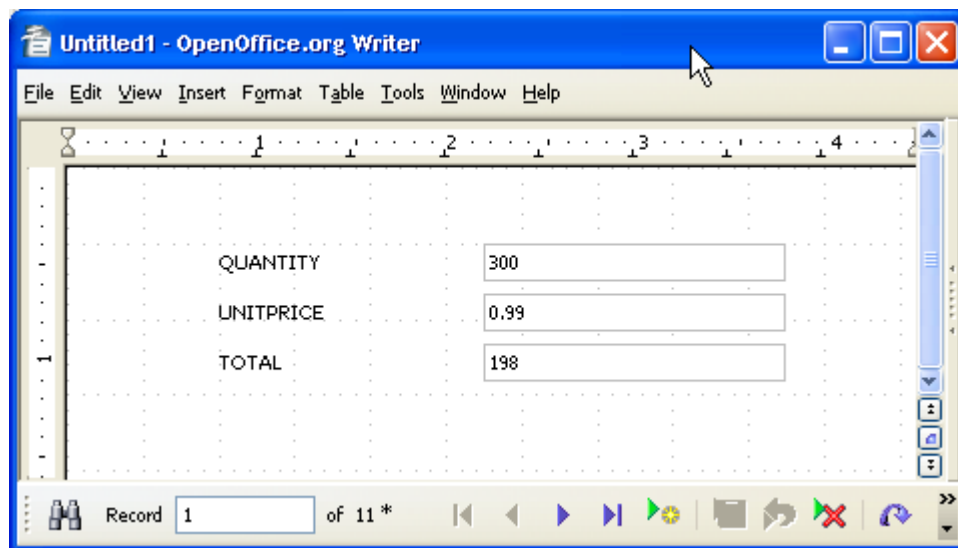
If Table or Query is selected as the content type, then the content field will have a combo box showing the tables or queries (respectively) available in the database. If SQL Command is selected, the push button with the ellipse next to the content field will be enabled. If you click on this button, the query builder will open such that you can easily generate the SQL Command. But you may also enter the SQL by hand.

In this case, I will select *Table* and *PURCHASEORDER* as the content type and content respectively. Now that the form has a content or source, a button has become enabled (see form design toolbar next to the form navigator). This button allows you to add field from the table selected. See Illustration 10



*Illustration 10: Add Field Window*

To add a field, select the desired field and drag it to the design area, or simply double click on it. Note that Base did several things for you. For each field, it created both a label and a text box. You can move the label/field pair anywhere on the design area. Illustration 11 shows the form with all the desired fields added. It is quite similar to the form generated with the wizard, except for the look and feel attributes.



*Illustration 11: Manual Form*

This of course can be modified by modifying the properties of each form control. The properties dialog for a control be accessed by double clicking on the desired control, right-clicking and selecting *control*

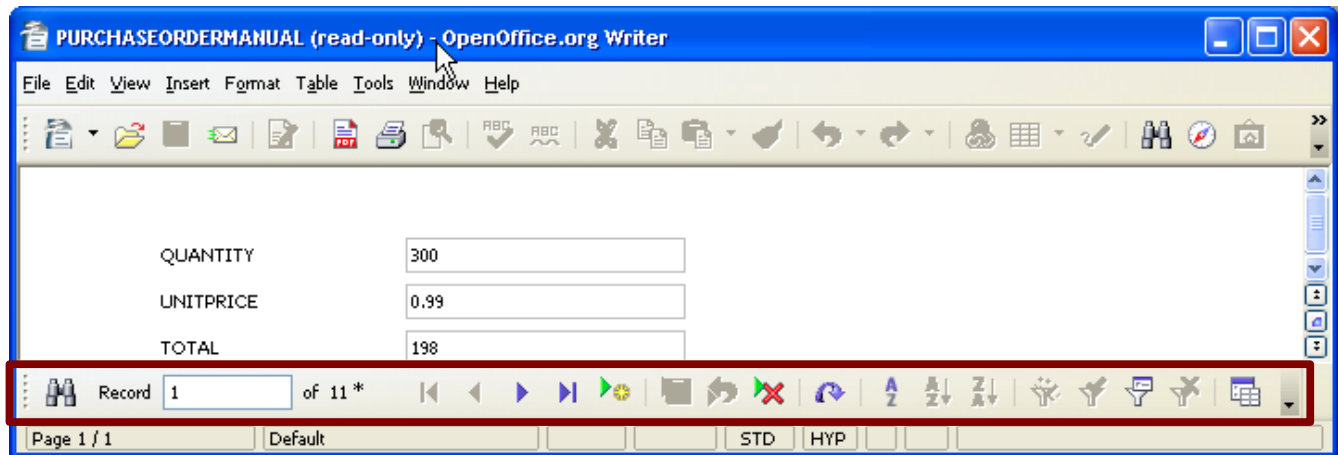
from the context menu, or by clicking on the property control property dialog button on the form control toolbar (see illustration 8). There is a small problem, however. If you click on the first field, the properties dialog only shows a few properties. This is due to the fact that when the wizard or the *Add Field* dialog is used as we need in the previous steps, the field label and the data control are grouped. Therefore, the properties dialog only shows the properties which both controls have in common. You can easily solve this problem by right-clicking on the grouped controls and selecting un-group from the context menu. Now you can open the properties dialog for the text box.

Generally, it is best to use the wizard to get the form started, and then open it in edit mode to personalize it to your needs. If, however, the form needs to be very specific, you might be best use starting from scratch, as wizards are intended to general purpose use. Aside from form and form control colors, we now have arrived at the same place as with the wizard generated form. Therefore, it is time to dive into other form aspects.

I will discuss adding controls from the toolbars as well as modifying them later when I discuss each form control in detail.

## Form Filters

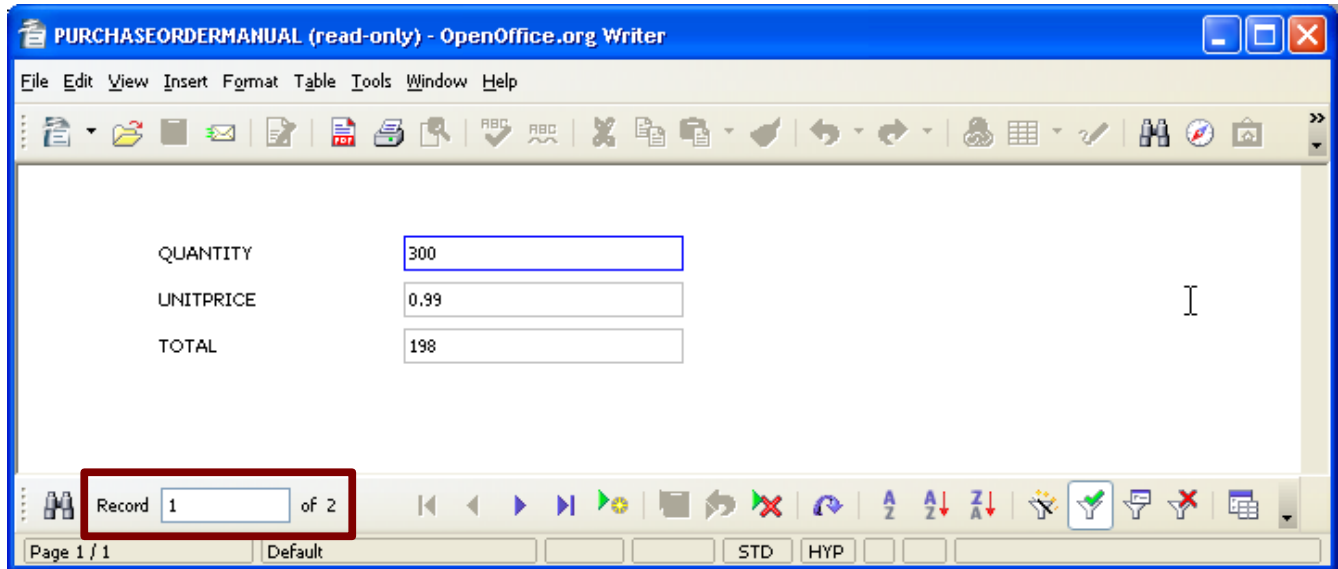
One of the basic form needs is to query data. This can be done with a query object. However, the query object is not really designed for heavy data manipulation. Therefore, it is best to use the form's filter mode. There are two ways to switch a form to filter mode. The first is by using the Auto Filter. To do so, click on the button with a magic wand and funnel, which is located on the right side of the form navigation Toolbar. This will use the currently selected form field as the filter criteria. Consider the purchase order form shown on illustration 12. It is showing records 1 of 11 (as seen on the navigation bar); the quantity for record 1 is 300. I have made sure that there are at least 2 such records on the table.



*Illustration 12: Form Navigation Toolbar*

Now, if I select the quantity field and click on the Auto Filter button, the form now is displaying record 1 of 2—see illustration 13.

The data has not disappeared, it is simply hidden for the moment. To retrieve the original data set, simply remove the filter—click on the button with the red X and funnel which is also located on the right side of the form navigation toolbar.



*Illustration 13: Filtered Form*

Note that this was all done automatically. There was no need of entering data, as the data currently in the field was utilized. Furthermore, the form was automatically ready to be used again. This is great as it is a fast way of narrowing your data set. However, is quite limited. You may want to filter by multiple fields or use relational operators (<,>, etc). This can be accomplished with a standard Form-Based Filter. Click on the button with a small form and a funnel, and the form will be switched to *Filter-Mode*. Unlike the auto filter, however, all the form controls/fields are now blank. The data entered in the fields will be used as query criterion and NOT be added as a new record such as when the form is in data-mode. Additionally, you will not that the form navigation bar has disappeared, and another small toolbar has appeared—see illustration 14.



*Illustration 14: Form-Based Filter Toolbar*

To apply the filter, click on the button with the funnel. To cancel the filter/search, click on the *Close* button. This will return you to the form in data-mode. Note that all data entered in to the fields at this point will modify your data source.

As I mentioned on the previous paragraph, you can create complex queries. In fact you can use, you can use the same operators as in the query object or as utilized when using the API (AND, OR, LIKE >,>=,<,<=, etc). In the purchase order example, I could have searched for all records with quantity > 300 or >100 AND <250, or whatever query construct would yield the desired result. For strings (VARCHAR) you may also use the LIKE operator. Note that when using the LIKE operator, you may also want to use the wild card character—the asterisk (\*) for Base. Let us know consider the employees form I created on the previous chapter. I have made sure that there are at least three employees with

category of 'Delivery.' Those are Philip Fry, Zap Brannigan, and Bender Rodriguez. Consider now that I want to get all employees with category of 'Delivery' but also that the last name begins with Bran or is Fry (the practical application of such a query is beyond me, but it puts the query features of Base forms to practice). To do so, simply click on the *Form-Based Filters* button and:

1. Enter LIKE Bran\* OR Fry on the LASTNAME field
2. Select 'Delivery' from the category combo box
3. Click on the *Apply Filter* button

The form now shows 1 of two records—Fry and Brannigan. Of course, given that the last name was being limited to Bran\* or Fry, the category parameters is unnecessary. However, it illustrates the point that when in filter-mode, a combo box source list can be utilized to select filter criterion. See illustration 15 for an example of a Base form in *Filter-Mode*.

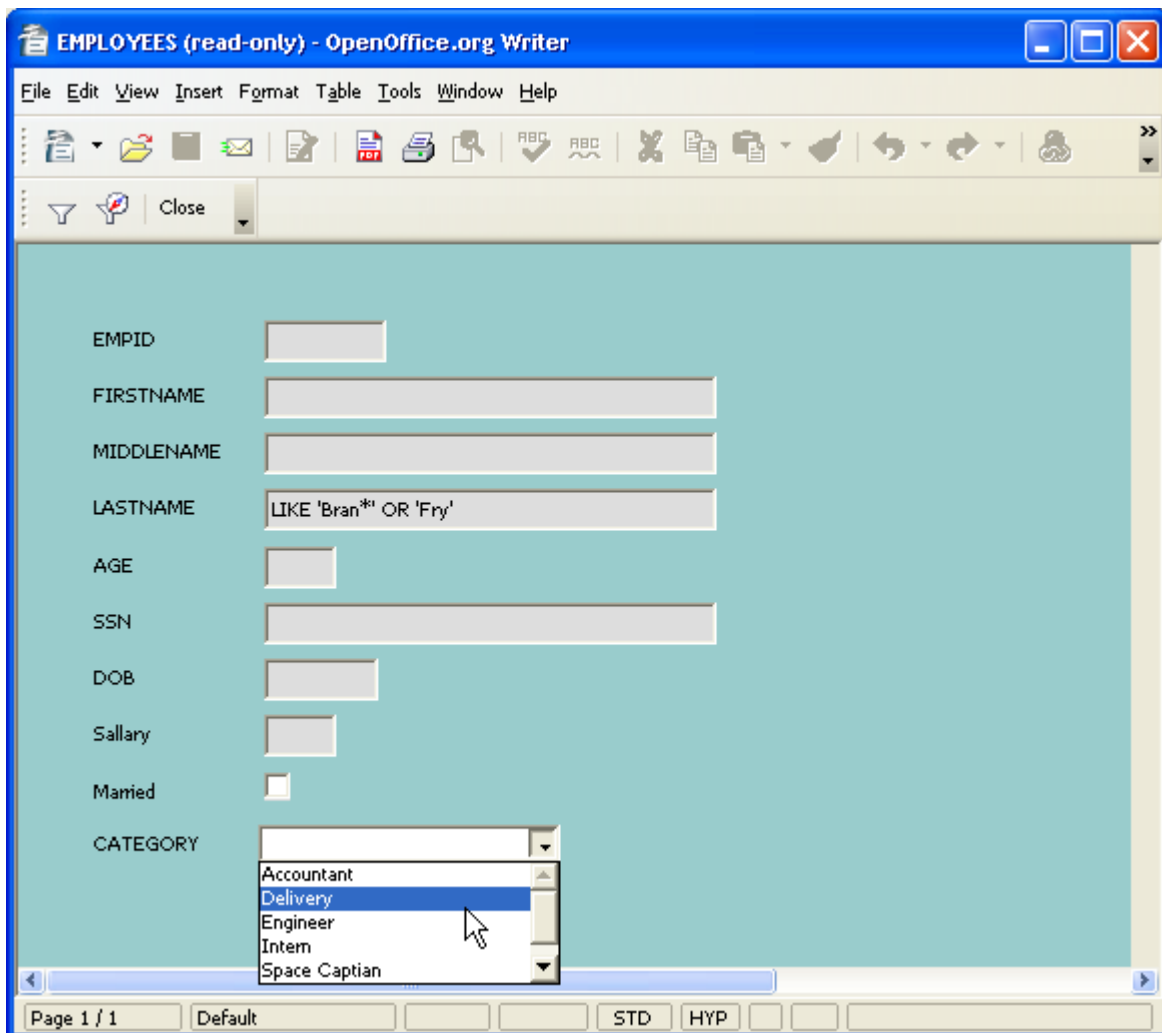


Illustration 15: Form in Filter Mode

There is another option on *the Form-Based Filters Toolbar* which I have not discussed. This is the Filter-Navigation. I discovered this option after frantically trying to figure out why I was getting strange query results. The Filter-Navigation Dialog shows the structure of your query. Note that parenthesis (structure) are of great importance in query construction. Illustration 16 shows the Filter-Navigation Dialog/Navigator.

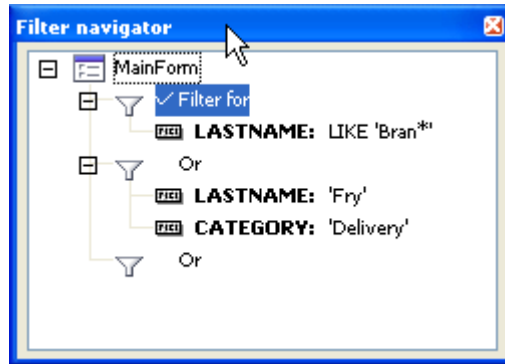


Illustration 16: Filter Navigator

The parenthesized version would be:

LASTNAME LIKE 'Bran\*' OR (LASTNAME LIKE 'Fry' AND CATEGORY='Delivery')

You can delete a whole node (group) by right-clicking on that node and selecting *Delete*. The entries can also be modified or deleted by right-clicking and selecting the appropriate action. Additionally, you may drag an entry into a different group. Of course, the best way to clear them all is to click on *Remove Filter* from the Form Navigation Toolbar.

While in design mode, the form filter can also be modified by clicking on the button next the filter field (Data Tab). Illustration 17 shows the filter editing dialog.

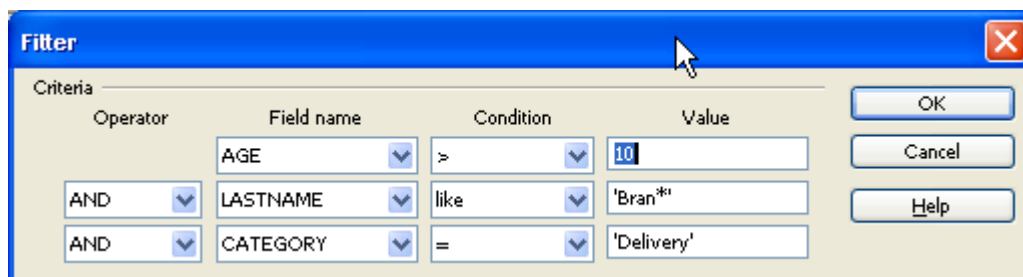


Illustration 17: Form Filter Editor--Design Mode

## Finding Records

In the previous section I showed how to located records of interest by using form filters—a variation of a query object. Base forms provides an additional way of finding records. This is similar to typical search features throughout OpenOffice.org. You can open the *Find Record* dialog by clicking on the button with the binoculars located on the left side of the Form Navigation Toolbar. The *Find Record* and the *Form Filters* differ in that the filter narrows the number of records available on the form to

those that match the filter/query criterion, whereas the *Find record* feature simply advances the cursor (moves to) the first match. The find record dialog is quite simple and self-explanatory, and therefore will not go into details on it. Once the desired searched parameters are selected simply click on the *Search* button. Dialog will not close, and you can continue to click on *Search* to find the next match. See illustration 18 for an example of the search dialog.

This wraps up the section on form filters and searching. More advanced filters/search features can be created with the help of macros. I will cover that later, as I first want to illustrate what features are available that do not require code. As a programmer my first instinct is to write code. However, that is not always the best approach; more fun, definitely, but I have found that it is best to first see what is available before investing time on re-inventing the wheel.

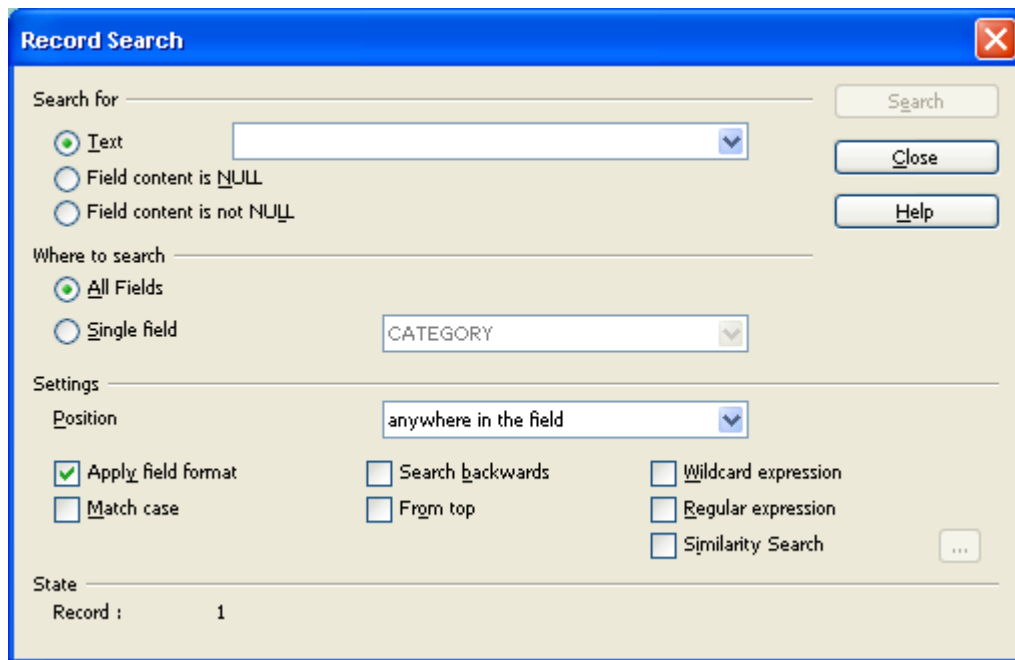


Illustration 18: Find Record Dialog

## Form Events

Form, as well as all form objects have what is called events or event handlers—this is the same for all graphical user interfaces. As the name suggests, these are events that, at one point or another, are initiated by the user. The interesting point of course is that code can be associated with these events to carry out a desired task. The code associated with a GUI event is similar to the code in the previous sections, with a couple of exceptions. First, the parameter event must be passed. While this is not required, the code will not be able to refer to the calling GUI element without this parameter. The second point, which is related to the first, is that most code that is associated to a GUI element is required to perform a function that in one way or another refers to its calling element or other related elements. For example, a Push Button may be associated with a macro, which reads data from form fields, and performs a function such as creating a record in a different table, or supplying data for calculated table columns.

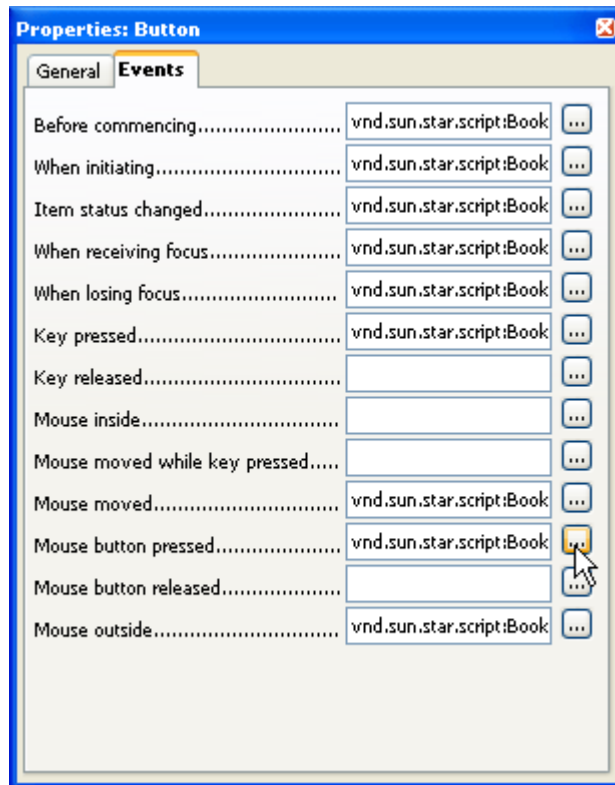
The subroutine does not take any other parameters, as any data required can be obtained from GUI Elements via the event parameter. Note that *event* is just a name. You may choose to call the variable

something else, but it will be holding an event object. However, the function/subroutine MUST take one parameter. Additionally, the subroutine does not return a value, as there is no one to properly accept such a value. The exception is *Boolean*. An event handler can return true or false, where true indicates that it is OK to continue and false means to cancel the action. Note that the subroutine or function must be defined to return a boolean, else the events will continue regardless of the value returned. See code listing 1 for an example of an event handler.

### Listing 1

```
Sub BeforeReset(event As Object)
    Dim control As Object
    control=event.Source.getByname("FORMDUMP")
End Sub
```

To assign a macro (OOBasic Code) to a form or form control, open the properties dialog and select the *Events* tab. Next to every event option is a text box which is read only, and next to that box is a push button—see illustration 19. Clicking on that button will open the *Assign Action* dialog—see illustration 20.



*Illustration 19: Assigning macro to control event: Step 1*

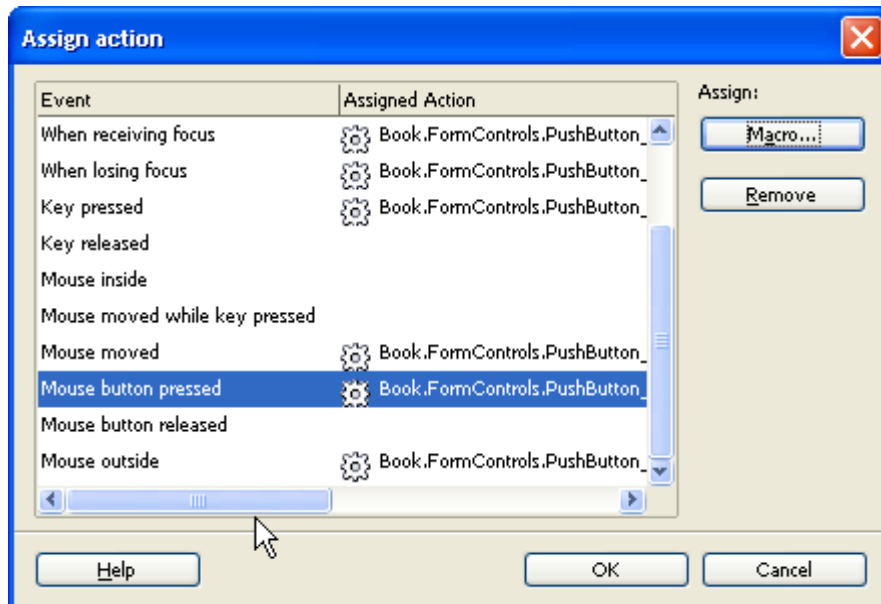


Illustration 20: Assigning macro to control: Step 2

Note that the Action Assign dialog has the same list of events on the previous dialog, but has a few more options. You can either remove or assign an action. To assign a macro, make sure the desired event is still selected, and click on the *Macro* button. This will open a third and final dialog, The *Macro Selector*—see illustration 21.

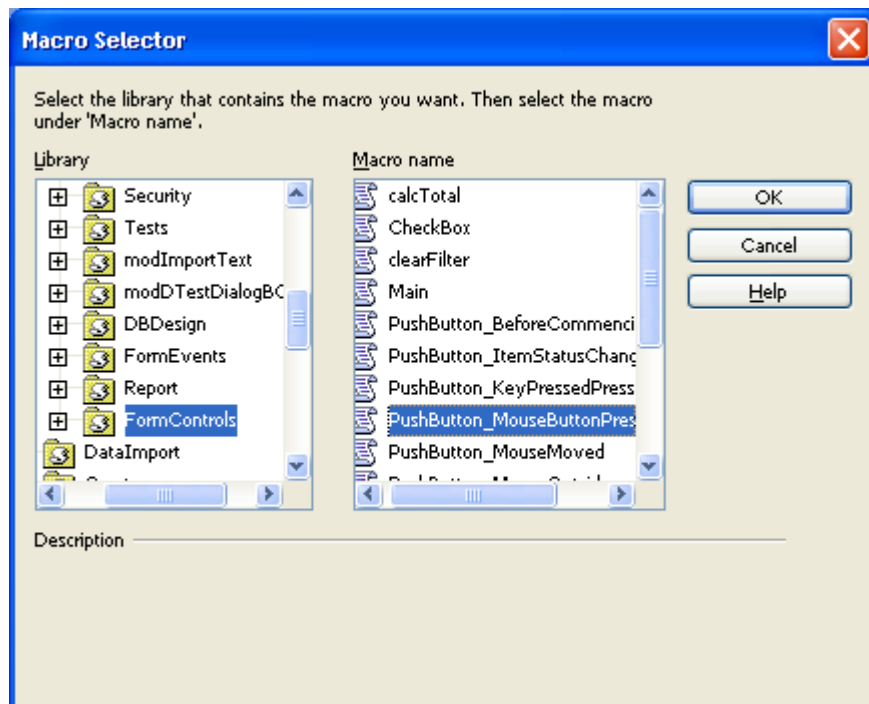


Illustration 21: Assigning macro to control: Step 3

It is in this dialog where you can finally browse through your macro libraries and assign a function or subroutine of choice. The library options consist of *My Macros*, which is a list of libraries you have written; *OpenOffice.org Macros*, which consists of macros that come pre-installed, and can not be

modified by the user; as well as a list of all the currently open OOo documents (this includes Base Forms), as each of these can hold its own set of macro libraries. Storing the code on base forms allows for easy portability. However, this will cause that pesky message asking if you want to enable or disable macros. This check can be disabled, but of course, now you have a security issue. Personally, I prefer to just store it under *My Macros*, and simply import the library into the client's computer.

There are many types of events, such as `MouseEvent`, `KeyEvent`, etc. All events are based on the generic `EventObject`. This has one attribute called *Source* which points to the object that fired it (button, form, text box, etc). All other events will have additional attributes. A `MouseEvent`, for example, will have attributes that hold mouse information, such as which button was clicked, modifiers, etc.

Attribute	Type	Description
Source	Object	Form or form control that fired event
Modifiers	Integer	Modifying keys such as Ctr or Alt
KeyCode	Integer	ASCII code for key pressed
KeyChar	Char	Corresponding Unicode character
KeyFunc	Integer	Function type. Constant in constant group <code>com.sun.star.awt.KeyFunction</code> . See table 3 for description

*Table 1: KeyEvent Object*

When using the `KeyFunction` constant group, you may use the full name:

```
if (Event.KeyFunc=com.sun.star.awt.KeyFunction .PRINT )
```

or you may use the numeric representation:

```
if(Event.KeyFunc=5)
```

As I discussed in the previous chapter, it is up to you which method you use; one allows for less typing but requires you to remember which code belongs to which function, where as the other requires a lot more typing, but is self explanatory.

Name	Numeric Value	Description
SHIFT	1	Refers to both SHIFT keys.
MOD1	2	Refers to the CTRL key—usually. May differ in Apple Keyboard
MOD2	3	Refers to the ALT key-usually.

*Table 2: com.sun.star.awt.KeyModifier Constant Group*

Name	Numeric Value
DONTKNOW	0
NEW	1
OPEN	2
SAVE	3
SAVEAS	4
PRINT	5
CLOSE	6
QUIT	7
CUT	8
COPY	9
PASTE	10
UNDO	11
REDO	12
DELETE	13
REPEAT	14
FIND	15
FINDBACKWARD	16
PROPERTIES	17
FONT	18

Table 3: *com.sun.star.awt.KeyFunction Constant Group*

Attribute Name	Data Type	Description
Source	Object	Object that fired event
Modifiers	Integer	Modifying key
Buttons	Integer	Which mouse button was pressed
X	Integer	X value (coordinate) for mouse position
Y	Integer	Y value (coordinate) for mouse position
Clickcount	Integer	How many clicks were associated with the event
PopupTrigger	Boolean	Event is pop-up (context) menu trigger. Such as when right-clicking to obtain context menu for an object/control.

Table 4: *MouseEvent*

These two are the most common events. I will discuss others as they come along. Generally, I mainly utilized the *source* attribute of the event, which is contained by all events. This gives me access to the form and form controls, from which I can obtain data and perform whatever desired function.

Table 5 shows some of the most common form events.

Event Name	Event Type	Description
Prior To Reset	EventObject	Triggered before the form is reset. The form is reset when the cursor is moved to the <i>insert row</i>
After Reset	EventObject	Triggered after the form is reset
When Loading	EventObject	Triggered when the form is loaded for the first time
Before Reloading	EventObject	Triggered before the form is reloaded—either from the reload button on the form navigation toolbar or programmatically
When Reloading	EventObject	Triggered when the form is reloading
Before Unloading	EventObject	Triggered before the form starts to
When Unloading	EventObject	Triggered when the form is unloading
Confirm Deletion	com.sun.star.sdb.RowChangeEvent	Triggered when a records is selected to be deleted. See table 6 for a description of the <i>RowChangeEvent</i> .
Before Record Action	com.sun.star.sdb.RowChangeEvent	Triggered when a record is either inserted, updated, or deleted.
After Record Action	EventObject	Triggered after an action is performed on an record. Note: neither the before or after event will fire until the form is moved out of the modified record or the <i>Save Record</i> button is pressed.
Before Record Change	EventObject	This event is triggered before the form moves to another record.
After Record Change	EventObject	Triggered after the form moves to another record.

*Table 5: Common Form Events*

When testing *the Before Unloading*, as well as *When Unloading* events I found a rather peculiar behavior. The *Before/When Unloading* events fire when *the Design Mode On/Off* button is pressed, but only if the form was opened in edit mode, and pressed the *Design Mode On/Off* button to get out of design mode. Neither event seems to fire when the form document is closed regardless of how the form/form document was opened. This does not seem practical, as a user will not (should not) be messing with the *Design On/Off* button. Rather, a general user should open the form directly in Data-Mode.

Attribute Name	Data Type	Description
Source	Object	object/element that triggered the event.
Action	Long	This is the action that took place. The possible values are INSERT, UPDATE, DELETE which correspond to numeric values of 1,2 & 3 respectively. These are in the <i>RowChangeAction</i> constant group in com.sun.star.sdb
Rows	Long	Indicates the number of rows affected by action/event

Table 6: com.sun.star.sdb.RowChangeEvent

When working with events, it is not sufficient to know where to attach your code. It is also important to know when exactly the events are firing/being triggered. For example, When the form document is opened, the *Before Record Change* is the first event to fire. *When Loading* is the second and then, finally, *After Record Change*.

For some reason, I do not know if it is a feature or a bug, the *Before Record Action* fires twice for every action (INSERT, UPDATE, DELETE). Therefore, you must make sure that the desired action is not performed twice—unless that is what you want. It appears as though the first time it is triggered by the *FormController Service* (controls interaction between form and user) and the second time by the *DataForm Service*. If you try to access the form components the first time the event fires, you will get an error, as the form controller does not have the form controls. The form is the FormController's model, so you may access the form controls that way. See listing 2 for an example.

### Listing 2

```

1 Sub BeforeRecordAction(event As Object)
2     Dim Form As Object
3     Dim Control As Object
4     Form=event.Source.Model
5     Control=Form.getByName("UNITPRICE")
6     MsgBox "The Unit Price is $" & Control.Text
7 End Sub

```

This is all well and good for the first time around, as the event was triggered by the form controller. However, the second time it will be triggered by the form itself. I can change it around to handle the form triggered event, but the the first will fail. So, what to do? Solution: Every service as an implementation name, and a property that holds that name. Therefore, I can use that property to determine who triggered the event. See listing 3.

### Listing 3

```
1 Sub BeforeRecordAction(event As Object)
2     Dim Form As Object
3     Dim Control As Object
4     Dim ControllerName As String
5     Dim FormName As String
6     ControllerName="com.sun.star.form.FmXFormController"
7     FormName="com.sun.star.comp.forms.ODatabaseForm"
8     If Event.Source.ImplementationName=ControllerName Then
9         Form=Event.Source.Model
10        Control=Form.getByName("UNITPRICE")
11        MsgBox "FROM FORM CONTROLLER: " & Control.Text
12    Elseif Event.Source.ImplementationName=FormName Then
13        Form=Event.Source
14        Control=Form.getByName("UNITPRICE")
15        MsgBox "FROM FORM: " & Control.Text
16    End If
17 End Sub
```

This will make sure that the form is obtained. However, it still executes twice. To solve this problem, simply test for the implementation name of your choice, and bail out if it is anything else. This can be a crucial step if you are modifying a data table in your code. Running the code twice may corrupt your data.

Another point of note is that, when deleting a record, the *Confirm Deletion* event will also fire. The event sequence is:

1. Confirm Deletion
2. Before Record Action (FormController Service)
3. Before Record Action (DataForm Service—the form)
4. After Record Action

Remember that the *RowChangeEvent* can be any of three actions: INSERT, UPDATE, DELETE. However, you may only want to execute code during one of the three, but not the other two. This can be easily solved by checking for the appropriate event action. See listing 4

### Listing 4

```
1 Sub BeforeRecordAction_INSERT(event As Object)
2     Dim Form As Object
3     Dim FormImpName As String
4     FormImpName="com.sun.star.comp.forms.ODatabaseForm"
5
6     If Event.Source.ImplementationName<>FormImpName Then
7         Exit Sub REM WILL GET IT NEXT TIME AROUND
8     End If
9     If Event.Action=1 Then REM com.sun.star.sdb.RowChangeAction.INSERT=1
10        MsgBox "Inserting"
11    End If
12 End Sub
```

This concludes the section on form events. I will have a more detailed discussion in the sections to come regarding the form document structure/hierarchy, as this is quite important in GUI programming.

## Form API

One of the most common functions when working programmatically with forms is to access its controls to insert or extract data, or perform some other function on a particular control. Form controls can be accessed via the API by three different ways: By Name, Index, and by Enumeration.

Examples:

1. Index Access: *Control=Form.GetByIndex(1)*
2. Name Access: *Control=Form.GetName("TXTBOX\_FIRSTNAME")*
3. Enumeration Access:  
*Controls=Form.createEnumeration()*  
*While Controls.hasMoreElements()*  
*Control=Controls.nextElement()*  
*Wend*

Note that in each example above control is a specific control. Therefore, when iterating over all the controls, it is important to keep in mind that different controls have different APIs. For example, a combo box has the property *SelectedItems*, but a text box does not, and attempting to call such property on a text box would course cause an error. When iterating over the controls, either test for the control type (check the *ImplementationName* property) before calling methods or properties for that control, or make sure you are only calling properties or methods that all controls have in common; such as *Name* or *ImplementationName*.

At first sight, the form API appears quite daunting. However, it becomes a bit more clearer, when you consider its make up. First, there is the form service, which has a group/container of *FormComponets*. The form components are of course the various components or controls that can be used to interact with the form. Additionally, the purpose of the form is to access a data source. Therefore, it also has all the services discussed on the previous chapter—*ResultSet*, *RowSet*, *DataSource*, *Connection*, etc. Table 7 has some of properties for the form service. As you an see, a lot of the properties are the same ones I discussed on the previous sections—row set and result set. I specially have found the *ActiveConnection* property quite useful. Consider for example, that you are entering data into a form, and want to add data to another related table based on this entry. You can quite easily use this property to create a statement to do the insert. Consider code listing 4. The following code could be added at line 10:

```
Form=Event.Source.Model  
Statement=Form.ActiveConnection.createStatement()
```

At this point you may proceed to use the statement service as usual. In Microsoft Access® you can refer to the current database via the *CurrentDB()* function. Although some more labor is involved, the *ActiveConnection* serves the same purpose. Much like the generic functions I provided in the *ResultSet* and *RowSet* sections, I can also just as easily provide a to obtain the active connection. In certain respects I am quite lazy and proud of it—whenever possible, I will try to create general purpose code such that I do not have to keep writing the same code over and over. Instead, let the computer do the work. As the man said:

*Ask not what you can do for the computer, but what the computer can do for you.*

<b>Name</b>	<b>Description</b>
ActiveConnection	Connection Service.
AllowDeletes	Boolean. Determines if deletes are allowed.
AllowInserts	Boolean. Determines if inserts are allowed.
AllowUpdates	Boolean. Determines if updates are allowed.
ApplyFilter	Boolean. Determines if filter specified by filter property should be applied.
Columns	Column Service. This is the same service as discussed in the previous chapter (ResultSet and RowSet). Column information and data for current row.
Command	Command name. Same as in RowSet. The command can be a table/query name, or a valid SQL command.
CommandType	Type of command. Can be TABLE, QUERY or COMMAND. Each of which is a member of the com.sun.star.sdb.CommandType constant group
ControlModels	Object array containing the models for all form controls. Generally these are accessed by name, index, or enumeration.
Count	Number of controls in the form.
ElementNames	String array containing the names of all controls in form. As the form supports name access, it is often best to use the <i>hasByName(...)</i> method of the form.
FetchSize	Number of rows to fetch/cache from source.
Filter	String containing filter criteria. Any valid SQL statement is valid—the WHERE clause of a SELECT statement
IsModified	Boolean. Indicates if the current record has been modified.
IsNew	Boolean. Indicates if the current record is a new entry
Order	String indicating how to sort the form data
Parent	Object pointing to the form's parent. This can be the forms collection or another form. Forms & Sub forms will be discussed in greater detail in the upcoming chapter.
ResultSetConcurrency	Determines if the result set can be update or if it is to be read only.
ResultSetType	Determines navigation. This was also discussed

Name	Description
	in the result set section.
Row	Current row index.
RowCount	Number of rows in result set.

Table 7: Some Form Properties

I will provide two approaches. Each has drawbacks and advantages—though the drawbacks are trivial. Throughout all the code thus far, I have showed one method of navigating through the form hierarchy. That is to start with the event, get the source, and start crawling up through that object's parent. This is my method of choice, but is not the only way.

#### Listing 5

```

1 Function ThisConnection(Event As Object) As Object
2   On Error Goto HandleErr
3   Dim obj As Object
4   Dim FormImpName As String
5   FormImpName="com.sun.star.comp.forms.ODatabaseForm"
6   On Error Goto GetObject
7   If IsNull(obj) Then
8     obj=Event.Source.Model
9   Else
10    Exit Function
11  End If
12  While True
13    If obj.ImplementationName=FormImpName Then REM FOUND THE FORM
14      ThisConnection=obj.ActiveConnection REM get the active connection
15      Exit Function
16    End If
17    obj=obj.Parent REM TRY TO GET PARENT
18  Wend
19  Exit Function
20 GetObject:
21  On Error Goto HandleErr
22  obj=Event.Source
23  Resume Next
24 HandleErr:
25  REM some error occurred
26  Exit Function
27 End Function

```

As you can see, the function is quite simple. I first start with the model for the calling object. I will assume that the source has the property *Model*. If this fails, then the source itself is the *Model*, therefore I will goto line 20 and get the model this way. On line 12 I will begin to iterate. The first test is to check the implementation name for the object. If this is the form implementation name (defined on line 5), then I get the *ActiveConnection* and exit. Else, get the parent and start next iteration. If the object does not have a *Parent* property, the error will be caught by the general error handler *HandleErr* on line 24 and bail out.

The function can be used as follows:

```
Conn=ThisConnection(Event)
```

Of course, the event must be a valid event object; else, the function will return null. The second approach eliminates the need to pass this parameter, and provides a method more comparable to the *CurrentDB()* function in MS Access® . See listing 6

#### Listing 6

```
1 Function ThisConnection2() As Object
2 REM GET ACTIVE CONNECTION OF THE FIRST FORM
3 On Error Goto HandleErr
4     Dim Forms As Object
5     Dim FirstForm As Object
6     Dim Doc As Object
7     Doc=ThisComponent
8     If Doc.ImplementationName<>"SwXTextDocument" Then REM not a writer form
9         Exit Function
10    End If
11    Forms=Doc.DrawPage.Forms
12    If Forms.Count<1 Then REM NO FORMS.
13        Exit Function
14    End If
15    ThisConnection2=Forms.getByIndex(0).ActiveConnection
16 HandleErr:
17     Exit Function
18 End Function
```

As you can see, this approach is more straight forward. The first test is on line 8, to see if the document is a writer document. It is not necessary to have a writer document to have a form, however, a Base form must be writer document. The next check is on line 12. It may be that the writer document does not have forms. In this case, the function must exit and return null. Finally, I get the *ActiveConnection* on line 15. If the form has not been assigned the function will return null, as there is no active connection.

There is one caveat with this approach, however. As it is relying on the global variable *ThisComponent*, there is a risk that this variable will change. Consider the use case on listing 7.

#### Listing 7

```
1 Sub PushButton_MouseButtonPressed(Event As Object)
2     Dim url As String
3     Dim Conn As Object
4     url=ConvertToURL("Introduction.odt")
5     StarDesktop.loadComponentFromURL(url,"_blank",0,Array())
6     Conn=ThisConnection2()
7 End Sub
```

This code is linked to the mouse *mouse button pressed* event of a push button. Consider that, for

whatever reason, you need to open another OpenOffice.org document as on line 5. Then On line 6, you get the current connection. What you want to do, of course, is to open a document, and later get the current connection. The current connection being being the *ActiveConnection* of the form whose control triggered the code. However, after opening the other document, *ThisComponent* no longer refers to the form, but to the most recently opened document. The function *ThisConnection2()* will then try to find a form and active connection the new document, and quite likely fail, or get the wrong connection.

The most obvious solution is to get the connection first, and then open the other document. However, the same problem will occur if the code takes a while to execute, and you manually open another document, or even activate a different window. Again, it is quite unlikely that you are execute such a long function or subroutine that gives you time to go wonder around before the code is done executing, but the possibility is there. Obtaining the connection on the first few lines will assure that the code gets executed quickly, before you get the chance to modify *ThisComponent*. Alternatively, you can use the first method, and live with the burden of passing that pesky event parameter. Of course, you can always get it from the form every time you need it. Most likely you will be working with the form, and already have obtained a reference to the form object—so this should be not much of a problem. As you can see, there are many options; it is up to you which approach you want to utilize.

The list of methods for the form service is quite extensive. However, much like the properties, most are those of the included services such as the row set service. There are minor differences however, in functionality for methods that provide means of navigation (next, previous, absolute, etc). In the row and result set services, these methods moved the row cursor. In the form service, they must also update the data for all controls to reflect the current table row; this is the function performed by the *First*, *Previous*, *Next*, and *Last Record* buttons on the Form Navigation Toolbar.

Table 8 shows some of the methods for the form service. Generally, the *getByName(...)* method is the most widely used to get access to form control models. I will now show you a brief example using the *createResultSet()* method. The form has a built-in search feature which moves the row cursor to the first row matching the search criteria. Consider, however, that you need to do this programmatically. Listing 8 demonstrates an example of how to accomplish this.

### Listing 8

```
1 Function findFirst(Form As Object, ColumnName As String, Target As Variant, _
2     Optional StartPos As Integer) As Integer
3     REM FIND THE FIRST OCCURRENCE OF TARGET
4     Dim rs as Object
5     Dim ColIndex As Integer
6     Dim Column As Object
7     Dim FirstPos As Integer
8     Dim ColumnValue As Variant
9     Dim FormImpName As String
10
11     FormImpName="com.sun.star.comp.forms.ODatabaseForm"
12     REM ONLY WORK WITH FORM FOR THE MOMMENT
13     REM COULD HVE IT WORK FOR RESULTSET AND ROWSET AS WELL
14     If Form.ImplementationName<>FormImpName Then
15         Exit Function
16     End If
17     If IsMissing(StartPos) Then
18         StartPos=1
19     End If
20     rs=Form.createResultSet()
```

```

21     ColIndex=rs.findColumn(ColumnName)
22     If ColIndex<1 Or ColIndex > rs.Columns.Count Then
23         Exit Function
24     End If
25     rs.absolute(StartPos)
26     REM IF NOT BOTTOM GUARDED LOOP, FIRST ITERATION WILL BE SKIPPED
27     REM THIS IS NOT GOOD AS THIS COULD BE THE FIRST OCCURRENCE
28     Do
29         Column=rs.Columns.getByName(ColumnName)
30         ColumnValue=getXXX(Column)
31         If ColumnValue=Target Then
32             findFirst=rs.Row
33             Exit Function
34         End If
35     Loop While rs.next()
36 End Function

```

The function takes four parameters—the form object(*Form*), Name of column to search (*ColumnName*), the target search string (*Target*), and an optional starting position (*StartPos*). On line 14 I check to make sure that the parameter named *form* is actually a form object. On line 17 I check to see if the starting position was passed; if not it defaults to 1. A clone of the form's result set is created on line 20. I could have just as well have the function use the form itself to perform the search. However, remember that when you use the form navigation methods, the whole form is updated on each move. This slows down the search and makes for quite odd behavior when the cursor starts moving up and down the result set before the user's eyes. Therefore, it is best to do all this work on a clone of the result set. One last check is made on line 21 before the actual search begins. As it is possible that the column name passed is not a valid name, I want to check by using the `findColumn(...)` method the column is there. If it is not, the function exits and returns 0. Generally, you may want to start searching from the start of the result set. However, this may not always be the case. Therefore the row cursor of the result set is moved to the starting position indicated by *StartPos*. The final showdown is on lines 28 to 35 where I get the column and then pass it to the `getXXX(...)` method I created on the previous chapter; and finally check to see if this column's value matches the target search string. If so, return the current row and exit, else continue looking. Listing 9 shows a quick use case.

### Listing 9

```

1 Sub PushButton_MouseButtonPressed(Event As Object)
2     Dim Form As Object
3     Dim Pos As Integer
4
5     Form= Event.Source.Model.Parent
6     pos=findFirst(Form,"QUANTITY",4)
7     If pos>0 Then
8         Form.absolute(pos)
10    End If
11 End Sub

```

Consider that you want to keep looking for the next match. In this case pass a starting position which is one plus the last position returned. For example:

```
pos=findFirst(Form,COLUMN_NAME,TARGET_SEARCHSTRING,pos+1)
```

Name	Description
Close()	Closes the underlying connection NOT the form window.
CreateEnumeration()	Creates an enumeration of form control models.
CreateResultSet()	Creates a clone of the underlying result set.
GetByName(aName As String)	Gets the model for the control specified by <i>aName</i> .
GetByIndex(index As Long)	Get the model for the control specified by the index.
GetConrolModels()	Gets an object array consisting of all control models in the form.
GetCount()	Gets the number of controls in the form.
GetElementNames()	Gets an array consisting of all control names in the form.
hasByName(aName As String)	Returns true if a control is found with specified name, false other wise.
Load()	Loads the data into form. This is done automatically when you open the form, so there is little use. If for whatever reason you called the close() method, you can get the connection/data back by calling the load() method.
RefreshRow()	Refreshes the number of rows specified by the <i>FetchSize</i> property. This method gets the most recent data from the database, therefore, if it is called before the current row is updated, all changes made to the current row will be lost. Also, frequent calls to this method may slow performance, specially if the database resides across a network.
Reload()	Reloads the form. This will return the form to the same state as when it was originally opened. Much like the refreshRow() method, all changes will be lost if called prior to updating the current row.

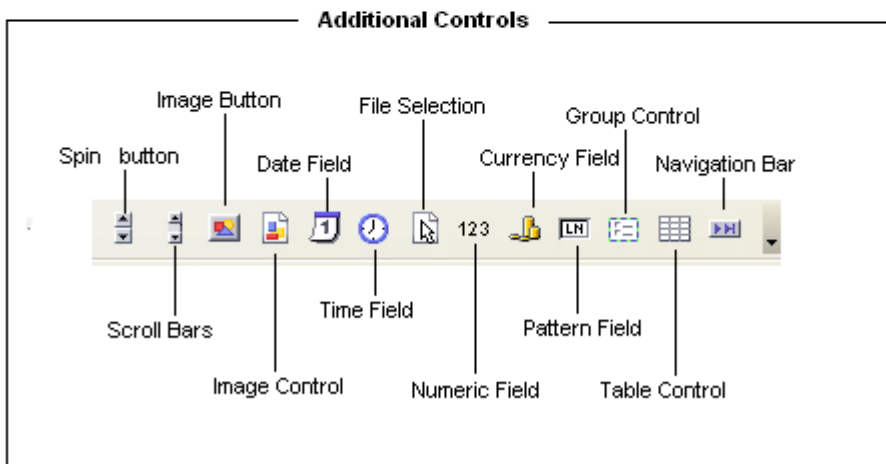
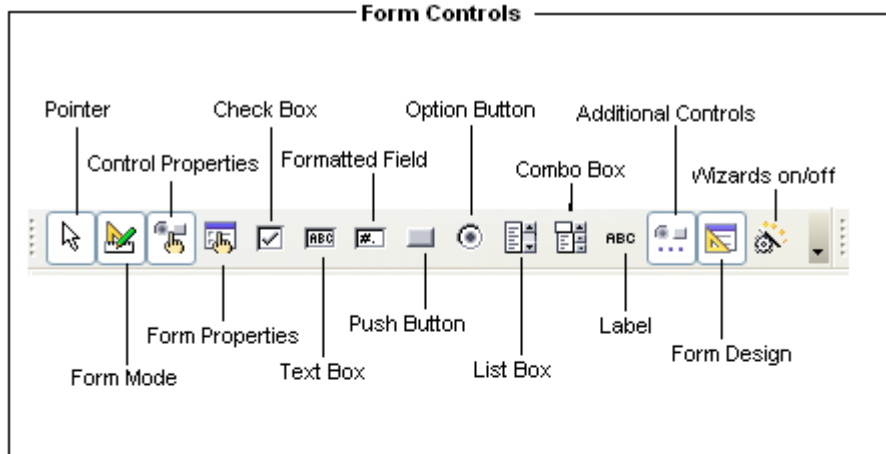
Table 8: Some Form Methods

[ ... ]

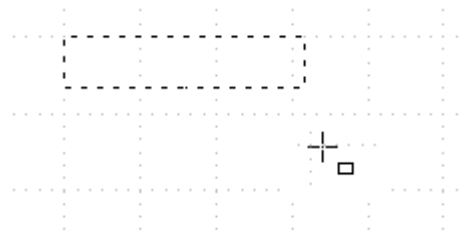
## Form Controls

The form, of course, would not be of much use without the form controls. In the previous section I quickly demonstrated how to manually duplicate a form that was created by the Form Wizard. I will now discuss the form controls in more detail. When you open a base form, the form controls toolbar will automatically open. Should it not be visible, for whatever reason, simply select it by going to View | Toolbars | Form Controls. This toolbar contains the most basic controls (top toolbar on illustration 22. More controls are available by clicking on the *More Controls* button to open additional controls. See bottom of illustration 22.

To add a form control to the form, click on the control of choice (the mouse pointer turns into cross-hairs and a small rectangle), then click (and hold) where you want to drop it on the form. Now, while holding the left-mouse button, drag to size the control. A dashed rectangle will show the current size of the control. Once you have the desired size, release the mouse button. See illustration 23.



*Illustration 22: Form Controls*

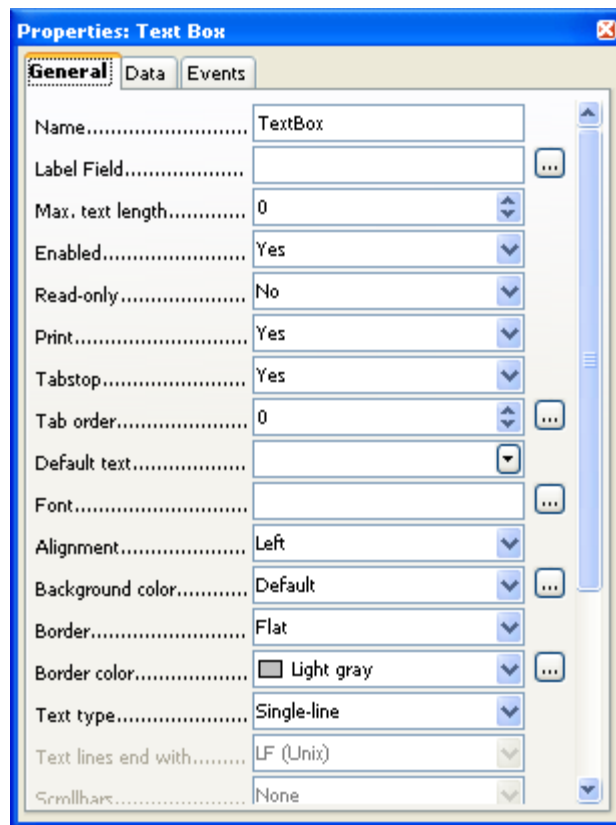


*Illustration 23: Dropping Form Control Into Form*

The control can be resized and repositioned after as well. To access the control properties dialog, either double click on the control, or select the control by single clicking, and the click on the control properties dialog button. This is located on the form controls as well as form design toolbars

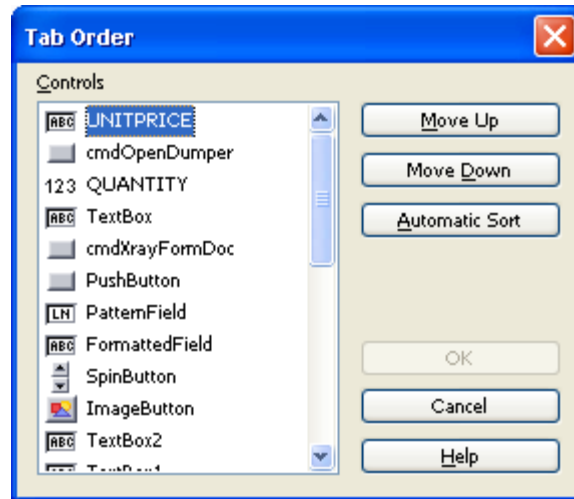
Most some similar attributes such as name, font options, enable (true/false), background color, border color/style, and a label or associated label field. There are many other attributes specific to controls, these will be discussed individually in the following sections. See illustration 24 for an example of the

properties dialog. This dialog is available for both forms as well as form controls. Most controls, as well as the form, have three tabs in the properties dialog—General, Data and Events. There are a few which can not be bound to table columns and therefore do not have a data tab. As you can note from the properties dialog, some properties require a simple text or numeric input, while others have options from which to select the setting. Additionally, some have a button with an ellipse label. This is for more complex input. Click on this button and a dialog will open to make the selection. For example, most data bound controls (those that can be bound/linked to a table column) have a label. You can either enter a label, or click on the button next to the input box to select a label field in that form. All event handler options have this button, which allows you to browse for a sub routine or function to serve as an event handler for the given event—see previous section for detailed instructions on assigning macros to events.



*Illustration 24: Sample Properties Dialog*

Most controls, aside from the label control, have a tab stop attribute. This indicates the sequence in which controls receive the focus the when tab key is pressed. To set the tab order, enter the sequence on the field next to tab order, use the up & down arrows next to the field, or click on the push button



*Illustration 25: Tab Order Dialog*

next to the field. This button opens the tab order dialog. This dialog can also be accessed via the form design toolbar. To modify the order, select a control and select the move up or move down buttons. Additionally, you can sort them automatically by clicking on the *Automatic Sort* button.

## **Label Field**

The Label Field is one of the simplest form controls. Its use is basically just to serve as a label for other form controls. The properties consist of basic font, border, background, and alignment options. A label can not be bound to a database table column, and therefore does not have a data tab. However, it does have an events tab. Available events are the mouse and keyboard events. Personally I have never had the need to add event handlers to a label fields, but needs are quite diverse, so if you ever need an event handler for a label fields, the option is there. I will cover mouse and keyboard event handlers for text boxes and other data bound, as I can think of more practical applications for those controls.

## **Push Button**

The push button, commonly referred to as command button or just button, has one purpose. That is to execute code. Generally, it is used to perform some function with the form data, navigate through the form, or open another form or report.

Name	Description
Label	Label for Push Button. Label for a push button is located in the button itself, you can not select a label field in the form as label. Enter the label on the field.
Enable	Yes/No. Allows you to enable/disable the button. This is useful if you want to prevent the user from clicking the button too many times, and thus executing to code too many times. You can disable the button at the start of the event handler, and re enabled it at the end.
Toggle	Yes/No. Specifies if the button should act as a toggle button—when pressed it remains selected (pushed) until it is pressed again.
Action	You can select one of several predefined function such as record navigation or a function on the form data.
Default Button	Yes/No. If a button is the default button, pressing the enter key has the same effect as a single click on that button. If more than one button is set as the default button, the one with the lowest tab order sequence will be used as the default button.
Graphics	You can select a graphic (icon) rather than a label.
Help Text	The text entered into this field appears as a tool tip when you mouse over the push button.

Table 9: Some Push Button General Properties



Illustration 26: Push Button

If the action selected for the *Action* property is *OpenDocument/Web Page*, The URL will be activated. Enter the URL or browse for the document. The URL is a web page, the web page will be opened with the default web browser. Make sure that if you are selecting an action, you do not have a macro linked to the *Mouse Button Pressed* event, and vice versa. In my tests, the *Mouse Button Pressed* code did not execute when I had inadvertently selected an action from the predefined options.

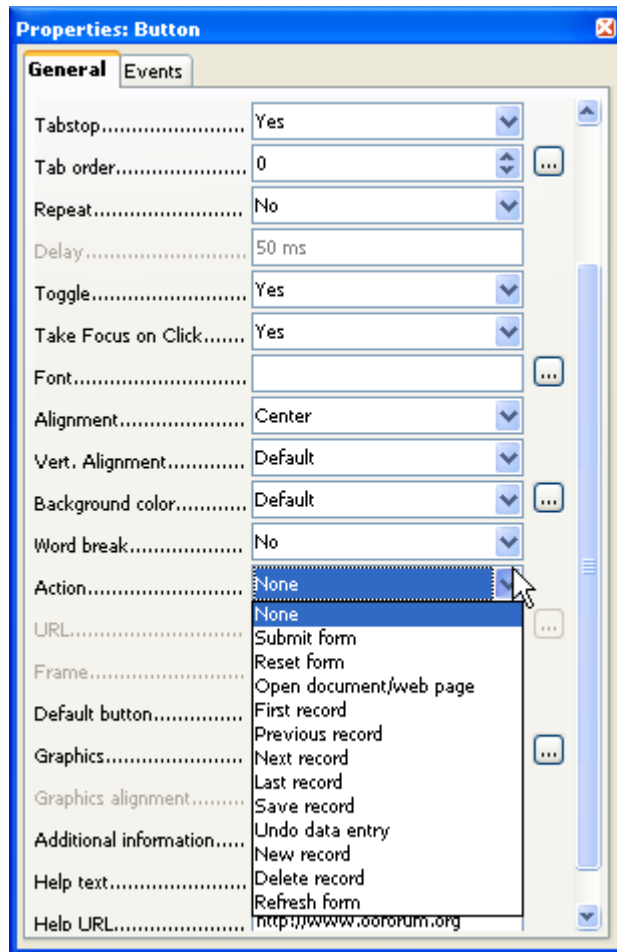


Illustration 27: Predefined Push Button Actions

Event Name	Event Type
Before Commencing	EventObject
When Initiating	EventObject
Item Status Changed	EventObject
When Receiving Focus	EventObject
When Losing Focus	EventObject
Key Pressed	KeyEvent
Key Released	KeyEvent
Mouse Inside	MouseEvent
Mouse Moved While Key Pressed	MouseEvent
Mouse Moved	MouseEvent
Mouse Button Pressed	MouseEvent
Mouse Button Released	MouseEvent
Mouse Outside	MouseEvent

Table 10: Push Button Events

The different event objects were discussed in the previous section; please refer to them if you need a reference. The event names appear to be quite descriptive of what they do and/or when they fire. However I did find some peculiar behavior with a couple of the events (Before Commencing & When Initiating), at least in respect to my expectations. From the names it suggests that these two events will fire first. However, these two events do not appear to be related to the button events but rather form events, when the button's *Action* is selected from the predefined options. The objective/use of these events is for validation, as they give you chance to bail out should the form data fail validation. Remember that you can do the same with the *Before Record Action* event which was discussed in the previous section.

The most common push button event is the Mouse Button Pressed event. I do not remember having a need for any of the other events for a button; but again, should the need arrive, they are there. In Base forms, the mouse button pressed event is equivalent to the *onClick* event on some system. However, note that some systems have a separate events for single and double click as well as different mouse buttons. If you refer back to the previous section in which the mouse event was discussed, you will note that the moused pressed event has all the information to determine which button was pressed as well as number of clicks.

### **Text Box**

The text box is one of the most widely used form controls. As the name suggests, it is a box where you can enter text. Additionally, there are several derivations of a text box, which will be covered in the upcoming sections. These are the Pattern Field, Formatted Field, and Numeric Field. Generally, these are all combined into one. But for some reason reason, OOo has several—perhaps to best comply with the Single Responsibility Principle (SRP). Having the text box object handle all the cases mentioned above would make it unnecessarily bulky, when all you need is one of the four.

Name	Description
Label Field	You can not enter a label directly for the text box. You must select one of the available label fields in the form. It is not necessary to select a label field for a text box. You can just place one next to it.
Max. Text Length	Maximum number of characters to allow—when the limit is reached, the control will not accept more input. 0 indicates no limit. This is useful when linking the input box to a database field, as sending a value that exceeds the defined length in a database column would give an error.
Read-Only	Does not allow data entry. Useful for system generated values, and where user meddling could corrupt the data.
Text Type	The options are single-line, multi-line, or multi-line with formatting. Generally, single-line (default) is sufficient. If however, your linked column is a memo field, the multi-line may be more appropriate. Note that when multi-line is selected, the EOL character as well as scroll bars options become available. Currently, the multi-line with formatting does not allow the text box to be linked with a column.
Password character	Character to use when the data is a password—this character is echoed rather than the actual character entered. This is only available with the text type is single-line.
Data Field	Bound data field (Data Tab). This can be selected from a drop down list available to the parent form.

*Table 11: Some Properties for the Text Box*

All the properties listed above are accessible through the properties dialog. These can also be accessed through the API. There is another property which is very useful when using the API. This is the *Text* property. This can be used to set the text for the text box or get the text currently in the text box. Additionally, the text box control has the *CurrentValue* property which also contains the text/value currently in the text box.

Often the reason to access the content of a text box is to perform some sort mathematical operation with the content of the text box. Consider the following example.

#### **Listing 4**

```

1 Sub TextBoxTests(Event As Object)
2     Dim Form As Object
3     Dim TextBox As Object
4     Form=Event.Source.Model.Parent
5     TextBox=Form.GetByName("TextBox1")
6     MsgBox TextBox.Text + 10.50
7 End Sub

```

Consider now that the text box has current value of 2.3. You would expect the result to be 12.80. However, the result on line 6 is 2.310.50. The reason is that the value of a text box is a string. Therefore, the values were concatenated, not added mathematically. To be used in a mathematical operation, the value of a text box must be casted. If lines six on listed 4 is replaced with

```
MsgBox Cdbl( TextBox.Text) + 10.50
```

you will get the sum of 2.3 and 10.50 desired. Table 10 has the functions that the be used to cast data

types.

Function Name	Casted Value Returned
Cdbl	Double
CInt	Integer
CLng	Long
CSng	Single
CStr	Single

Table 12: Type Conversion/Casting Functions

Sometimes it is necessary to insert data into the text box programmatically. For example, consider that you have Quantity and Unit Price text boxes, and want to calculate the total and place it in another text box. This is as easy as assigning that total to the Text property—see listing 5.

### Listing 5

```
1 Sub calcTotal(Event As Object)
2   Dim Form As Object
3   Dim Quantity As Object
4   Dim UnitPrice As Object
5
6   Form=Event.Source.Model.Parent
7   Quantity=Form.GetByName("QUANTITY")
8   UnitPrice=Form.GetByName("UNITPRICE")
9   Form.GetByByName("TOTAL").Text=CInt(Quantity)*Cdbl(UnitPrice)
10 End Sub
```

Remember that when binding code to events, it is quite important that the code is aware of how and when it is going to be executed. For example, the code in listing listing 5 is expecting the caller to be a push button. But, what if you want to execute it on a text box event instead? Line 6 would fail, since the source for a text box event object is the model itself, and does not have a property called Model. Therefore the line `Form=Event.Source.Model.Parent` would fail. The proper call to obtain the parent (form) for a text box is `Form=Event.Source.Parent`. Furthermore, what event is the best to use? When the control loses the focus? When the text is modified? Table 11 shows some events for the text box control.

As most other controls, it has the basic keyboard events as well as mouse events. Therefore, I will only discuss here those that are particular to the text box control. Specially, I would like to comment on the *Changed* and *Text Modified* events. They appear to be the same, but the key difference is that the *Changed* event does not fire until the control loses the focus, and the content has changed of course. This would be the best event to use for situations such as that posed in listing 5. The *Text Modified* event, on the other hand, fires every time the text is modified, even if the focus is not lost. This event could be used to count the number of characters entered, and move the focus the the next control if a requisite number of characters has been entered.

Event Name	Source	Description
Changed		The content of the text box has changed and the focus was lost.
Text Modified		The text is modified. This fires even if the control retains the focus.
When Receiving Focus		When the control receives the focus. Also contained by most other controls
When Losing Focus		When the control loses the focus. Also contained by most other controls
Before/After Updating		These two events fire before and after (respectively) the bound data field is updated.

Table 13: Some Text Box Control Events

Consider the following scenario. You want to filter a form based on data/text entered in a text box. There are a few options. You can either bind the code to the *Changed* or the *Text Modified* events, or you can use a push button. As mentioned above, the difference is that one fires after the object loses the focus while the other fires every time the text is modified, whether or not the control loses the focus. Additionally, note that the source for these two events is the text box view, not the model. Therefore, the form must be accessed via *Event.Source.Model.Parent*, not by *Event.Source.Parent*. The choice of using the *Changed*, *Text Modification*, or *mouse button pressed* (push button event) depends on the use. Let us consider the different functionalities obtained from the different approaches:

1. **Changed**—As this event fires after the text box loses the focus, this is best used when the *flow* of the form is such that the user would, or is expected, to naturally move the focus out of the text box and into another control. For example, consider the controls QUANTITY, UNITPRICE, TOTAL, where the user enters a quantity, unit price, and finally moves to the TOTAL control. There is a natural flow, and therefore, it is reasonable to use the *Changed* event on the UNITPRICE control to bind code that calculates the total. In the scenario posed above, this may not be the best choice, as the case is that the user enters data, and expects the form to be filtered. The focus may or may not be moved instinctively out of the text box.
2. **Text Modified**—This event has more of a real-time function, as it fires after every key stroke. This may work in the case posed above. However, it would further depend on the use. For instance, this would work fine if the data entered is a string, and you want to filter the form by any part of the string entered. Consider the table below for a sequential analysis of how this event would function in a search for city names. However, consider that you do not want to perform a search after every key stroke, but wait the user has entered. In this case, using the *Changed* event would be best, but see caveat noted for the *Changed* event.

Key Stroke #	Text Entered	Partial SQL	Results
1	H	LIKE '*H*'	Houston, Harlingen, Humble, Hondo
2	Ho	LIKE '*Ho*'	Houston, Hondo
3	Hou	LIKE '*Hou*'	Houston

Table 14: Search Example Using the Text Modified Event

3. **Mouse Button Pressed (Push Button)**--This of course is an entirely different control, so the

user is expected to enter the data into the box and then click on the push button. This may be ideal, if the form layout is the search text box, with the standard data form below. Illustration 28 a sample search form.

City	
▶	Houston
	Bronson
	San Antonio
	Cut & Shoot
	Sugar Land
	Austin
	Dallas
	El Paso
	Corpus Cristi
	Johnson City
	Paris
	Rome
	Hondo
+	

Record 1 of 13

*Illustration 28: Search Form Example*

I would like to make a couple of points about the *Before* and *After Updating* events. First, in the before updating event, The text box control will have the new value, however, the bound data field will still have the old value. It is not until the after updating event fires that both the control and data field will have the same value. Second, This new value is not actually written back to the database until the cursor (record pointer) is moved to the next record or the *Save Record* button (Form Navigation Toolbar) is pressed.

### **Formatted Field**

The Formatted Field is quite similar to the text box. The key difference is that the contents are formatted in a specific way—similar to a cell in a spreadsheet. Table 12 has some properties for the formatted field.

Name	Description
Value Min.	Minimum Value that will be accepted.
Value Max	Maximum value that will be accepted.
Formatting	Field content format
Spin Button	Spin button to increment/decrement the field value
Data Field	Bound data field (Data Tab). This can be selected from a drop down list available to the parent form.

Table 15: Some Formatted Field Properties

Illustration 28 shows the format dialog. Note the similarities to the table as well as Calc cell format options dialog. Additionally, it is important to note that this control is only for numeric values. Note that date and time are considered numeric. If a non-numeric value is entered it will get converted to 0 or its date equivalent of 12/30/99 and time equivalent of 12:00:00 AM. If the value entered is below the minimum, it will be automatically replaced by the minimum specified and the maximum if it falls above the maximum value specified.

In all other respects, the Formatted Fields is the Text Box. While this control works with numbers, when performing calculations with the contained in the *Text* property, it still must be casted. Else it will be treated as a string much like in the text box. However, you may use the *CurrentValue* property which eliminates the need for casting.

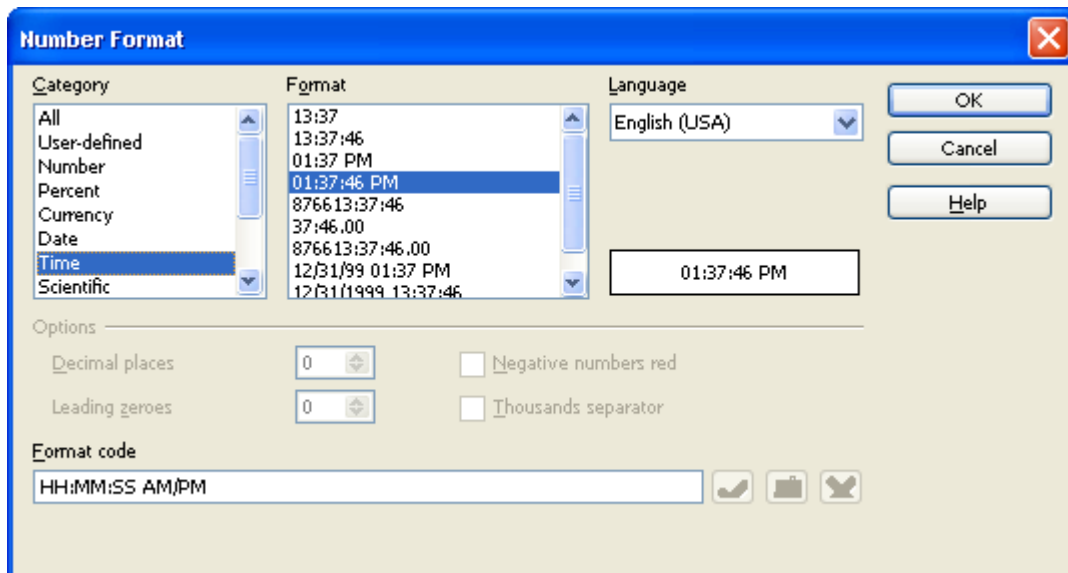


Illustration 29: Format Options/Selector for Formatted Field

## Numeric Field

The numeric field is quite similar to the formatted field. However, it does not accept dates or times. The formatting options available are limited to the number of decimals and thousands separator.

Another key difference is that, unlike the formatted field which converts non-compliant values to 0, the numeric field simply does not accept non-compliant values. Additionally, the API does not have a *Text* property, only *Value* and *CurrentValue*, and there is no need for casting.

### Pattern Field

The pattern field is the last one in the family of text box controls. Much like the numeric field is only suitable for numbers, the pattern field is only suitable for text (which may include numbers but as strings). The features of this control are usually included into the text box in other systems and is referred to as an *Input Mask*. Again, this controls is quite similar to the text box control, with the addition of two properties, the *Edit Mask* and the *Literal Mask*.

The *Edit Mask* contains the format codes that determine what kind of characters are valid. Table 13 (straight out of the OpenOffice.org help) contains a the format codes that can be utilized. The literal mask contains the default values for the control, and serve as a hint for the user as to what the expected format and required input is. For example: Consider that you need to enter a telephone number of the form (713)867-5309. The user should only have to enter the numbers, and the control should handle the parenthesis and dash. The first character then must be a text constant (L), followed by three numbers (NNN), followed by another text constant (L), then three more numbers (NNN), another text constant (L), and finally, the last four numbers (NNNN). The edit mask would like like this:

LNNNLNNNLNNN

Now, the literal mask must be created. The first character must be an open parenthesis. This is the text constant denoted by the L on the edit mask. Note that this characters cannot be edited. When the control encounters a text constant, the text cursor is advanced until a non text constant is found. For non-text constants, you can have whatever character that seems fitting. For example, the edit mask is for a phone number, I can either have the # character as it indicates that a number must be entered. However, an underscore ( \_ ) is traditionally utilized as a place holder. The full literal mask would look like this (using the underscore for non constants)

( ) \_ \_ - \_ \_ \_

Note that there is one place holder for each character defined in the edit mask. Illustration 29 shows a sample pattern field with a telephone number mask.

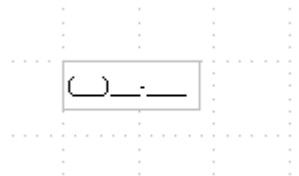


Illustration 30: Pattern Field Example

<b>Character</b>	<b>Meaning</b>
L	A text constant. This position cannot be edited. The character is displayed at the corresponding position of the Literal Mask.
a	The characters a-z and A-Z can be entered. Capital characters are not converted to lowercase characters.
A	The characters A-Z can be entered. If a lowercase letter is entered, it is automatically converted to a capital letter
c	The characters a-z, A-Z, and 0-9 can be entered. Capital characters are not converted to lowercase characters.
C	The characters A-Z and 0-9 can be entered. If a lowercase letter is entered, it is automatically converted to a capital letter
N	Only the characters 0-9 can be entered.
x	All printable characters can be entered.
X	All printable characters can be entered. If a lowercase letter is used, it is automatically converted to a capital letter.

*Table 16: Edit Mask Codes—taken from the OpenOffice.org Help*

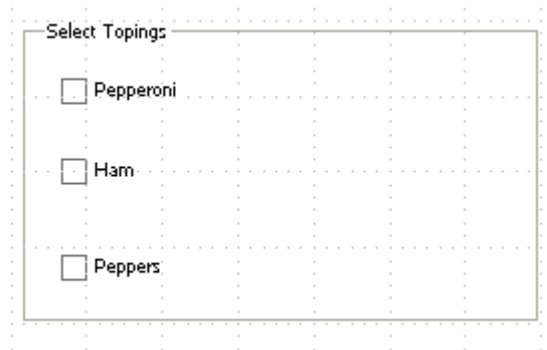
Note that both the Numeric as well as the Pattern field also have the Data Field property, which allows binding to a table column.

### **Check Box**

Check boxes are generally associated with boolean—Yes/No values. The basic function is to indicate a selection. See illustration 30 for an example.

<b>Name</b>	<b>Description</b>
Default Status	Default Status/Value. Can be selected or not selected
Tristate	Turn tristate on or off. In normal mode, the check box checked (true) and unchecked (false) on successive click. On tristate mode, the goes from unchecked to checked to checked which is grayed-out.
Data Field	Bound data field (Data Tab). This can be selected from a drop down list available to the parent form.

*Table 17: Some Checkbox Properties*



*Illustration 31: Check Box Example*

There is one property that I have most useful in the check box API; that is the `CurrentValue` value property. You can use this to test if the box was checked or unchecked. The possible values are 0 for unchecked, 1 for checked, and 2 if the tristate mode was selected—this is the grayed-out checked state.

The events for the check box are the typical mouse and keyboard events mentioned on the previous sections. However, I would like to make a couple of points. The source for the *Item Status Changed* event is the check box view (controller), not the model. Remember that the parent form (and thereby the rest of the controls) is accessed through the model. Therefore, the parent form is accessed by `Event.Source.Model.Parent` much like the push button. Whenever you have a problem accessing the parent form, it could be that the source is the callers view and not the model. This event is useful to monitor when the box is checked or unchecked. In this event, the value for the *CurrentValue* property is the future value, in other words, the changed value. For example: when for form loads, a check box is unchecked (value 0). When you check it (the *Item Status Changed* event fires), the `CurrentValue` property at this point would be 1—as the event name suggests, the status has changed, and therefore makes sense the value changed from 0 to 1. Another point of note is that when testing the *When Receiving Focus* event, the code kept executing. It seems as though the event kept firing, until I manually move the focus to another control. This was impossible on my first test, when the code opened a message box, as it did not give me enough time to move the focus away from the check box. Personally, the *Item Status Changed* is the event I have found of greater use.

## **Option Button**

Generally, this control has been known as a *Radio Button*.