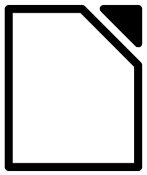


LibreOffice
The Document Foundation

Base Handbook

Chapter 9
Macros

Copyright

Contributors

Jochen Schiffers
Robert Großkopf
Jost Lange
Hazel Russman
Andrew Pitonyak

Feedback

Please direct any comments or suggestions about this document to: documentation@global.libreoffice.org.

Caution

Everything you send to a mailing list, including your email address and any other personal information that is written in the mail, is publicly archived and cannot be deleted.

Acknowledgments

This chapter is based on an original German document and was translated by Hazel Russman.

Publication date and software version

Published 3 June 2013. Based on LibreOffice 3.5.

Note for Mac users

Some keystrokes and menu items are different on a Mac from those used in Windows and Linux. The table below gives some common substitutions for the instructions in this chapter. For a more detailed list, see the application Help.

Windows or Linux	*Mac equivalent*	*Effect*
Tools > Options menu selection	**LibreOffice > Preferences**	Access setup options
Right-click	*Control+click*	Open a context menu
Ctrl (Control)	*⌘ (Command)*	Used with other keys
F5	*Shift+⌘+F5*	Open the Navigator
F11	*⌘+T*	Open the Styles and Formatting window

Contents

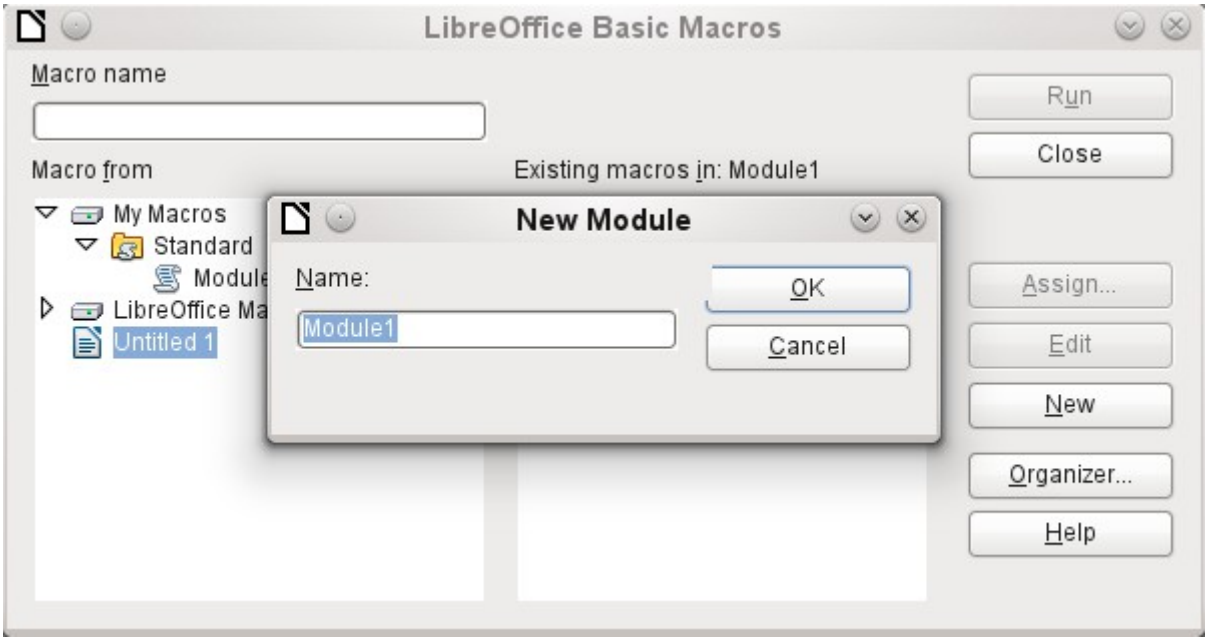
General remarks on macros

In principle a database in Base can manage without macros. At times, however, they may become necessary for:

- More effective prevention of input errors
- Simplifying certain processing tasks (changing from one form to another, updating data after input into a form, and so on)
- Allowing certain SQL commands to be called up more easily than with the separate SQL editor.

You must decide for yourself how intensively you wish to use macros in Base. Macros can improve usability but are always associated with small reductions in the speed of the program, and sometimes with larger ones (when coded poorly). It is always better to start off by fully utilizing the possibilities of the database and the provisions for configuring forms before trying to provide additional functionality with macros. Macros should always be tested on larger databases to determine their effect on performance.

Macros are created using **Tools > Macros > Organize macros > LibreOffice Basic**. A window appears which provides access to all macros. For Base, the important area corresponds to the filename of the Base file.



The **New** button in the LibreOffice Basic Macros dialog opens the New Module dialog, which asks for the module name (the folder in which the macro will be filed). The name can be altered later if desired.

As soon as this is given, the macro editor appears. Its input area already contains the Start and the End for a subroutine:

```
REM  *****  BASIC  *****

Sub Main

End Sub
```

If macros are to be usable, the following steps are necessary:

- Under **Tools > Options > Security > Macro security** the security level should be reduced to Medium. If necessary you can additionally use the Trusted sources tab to set the path to your own macro files to prevent later queries about the activation of macros.
- The database file must be closed and then reopened after the creation of the first macro module.

Some basic principles for the use of Basic code in LibreOffice:

- Lines have no line numbers and must end with a hard return.
- Functions, reserved expressions, and similar elements are not case-sensitive. So "String" is the same as "STRING" or "string" or any other combination of upper and lower case. Case should be used only to improve legibility. Names for constants and enumerations, however, are case sensitive the first time that they are seen by the macro compiler, so it is best to always write those using the proper case.
- There is a basic difference between subroutines (beginning with SUB) and functions (beginning with FUNCTION). Subroutines are program segments without return values. Functions return a value.

For further details see Chapter 13, Getting Started with Macros, in the *Getting Started* guide.

Note	Macros in the PDF and ODT versions of this chapter are colored according to the rules of the LibreOffice macro editor:
	`Macro designation`
	`Macro comment`
	`Macro operator`
	`Macro reserved expression`
	`Macro number`
	`Macro character string`

Improving usability

For this first category of macro use, we show various possibilities for improving the usability of Base forms.

Automatic updating of forms

Often something is altered in a form and this alteration is required to appear in a second form on the same page. The following code snippet calls the reload method on the second form, causing it to refresh.

```
SUB Update
```

First the macro is named. The default designation for a macro is **SUB**. This may be written in upper or lower case. **SUB** allows a subroutine to run without returning a value. Further down by contrast a function is described, which does return a value.

The macro has the name *Update*. You do not need to declare variables because LibreOffice Basic automatically creates variables when they are used. Unfortunately, if you misspell a variable, LibreOffice Basic silently creates a new variable without complaint. Use **`Option Explicit`** To prevent LibreoOffice Basic from automatically creating variables; this is recommended by most programmers.

Therefore we usually start by declaring variables. All the variables declared here are objects (not numbers or text), so we add **AS `OBJECT`** to the end of the declaration. To remind us later of the

type of the variables, we preface their names with an "o". In principle, though, you can choose almost any variable names you like.

```
    DIM oDoc AS OBJECT
    DIM oDrawpage AS OBJECT
    DIM oForm AS OBJECT
```

The form lies in the currently active document. The container, in which all forms are stored, is named **drawpage**. In the form navigator this is the top-level concept, to which all the forms are subsidiary.

In this example, the form to be accessed is named Display. Display is the name visible in the form navigator. So, for example, the first form by default is called Form1.

```
    oDoc = thisComponent
    oDrawpage = oDoc.drawpage
    oForm = oDrawpage.forms.getByName("Display")
```

Since the form has now been made accessible and the point at which it can be accessed is saved in the variable **oForm**, it is now reloaded (refreshed) with the **reload()** command.

```
    oForm.reload()
END SUB
```

The subroutine begins with **SUB** so it must end with **END SUB**.

This macro can now be selected to run when another form is saved. For example, on a cash register (till), if the total number of items sold and their stock numbers (read by a barcode scanner) are entered into one form, another form in the same open window can show the names of all the items, and the total cost, as soon as the form is saved.

Filtering records

The filter itself can function perfectly well in the form described in Chapter 8, Database Tasks. The variant shown below replaces the Save button and reads the listboxes again, so that a chosen filter from one listbox can restrict the choices available in the other listbox.

```
SUB Filter
    DIM oDoc AS OBJECT
    DIM oDrawpage AS OBJECT
    DIM oForm1 AS OBJECT
    DIM oForm2 AS OBJECT
    DIM oFieldList1 AS OBJECT
    DIM oFieldList2 AS OBJECT
    oDoc = thisComponent
    oDrawpage = oDoc.drawpage
```

First the variables are defined and set to access the set of forms. This set comprises the two forms "Filter" and "Display". The listboxes are in the "Filter" form and have the names "List_1" and "List_2".

```
    oForm1 = oDrawpage.forms.getByName("Filter")
    oForm2 = oDrawpage.forms.getByName("Display")
    oFieldList1 = oForm1.getByName("List_1")
    oFieldList2 = oForm1.getByName("List_2")
```

First the contents of the listboxes are transferred to the underlying form using **commit()**. The transfer is necessary, because otherwise the change in a listbox will not be recognized when saving. The **commit()** instruction need only be applied to the listbox that has just been accessed. After that the record is saved using **updateRow()**. In principle, our filter table contains only one record, which is written once at the beginning. This record is therefore overwritten continuously using an update command.

```
    oFieldList1.commit()
    oFieldList2.commit()
    oForm1.updateRow()
```

The listboxes are meant to influence each other. For example, if one listbox is used to restrict displayed media to CDs, the other listbox should not include all the writers of books in its list of authors. A selection in the second listbox would then all too often result in an empty filter. That is why the listboxes must be read again. Strictly speaking, the **`refresh()`** command only needs to be carried out on the listbox that has not been accessed.

After this, form2, which should display the filtered content, is read in again.

```
    oFieldList1.refresh()
    oFieldList2.refresh()
    oForm2.reload()
END SUB
```

Listboxes that are to be influenced using this method can be supplied with content using various queries.

The simplest variant is to have the listbox take its content from the filter results. Then a single filter determines which data content will be further filtered.

```
SELECT "Field_1" || ' - ' || "Count" AS "Display", "Field_1"
FROM ( SELECT COUNT( "ID" ) AS "Count", "Field_1" FROM "Table_Filter_result"
GROUP BY "Field_1" )
ORDER BY "Field_1"
```

The field content and the number of hits is displayed. To get the number of hits, a sub-query is used. This is necessary as otherwise only the number of hits, without further information from the field, will be shown in the listbox.

The macro creates listboxes quite quickly by this action; they are filled with only one value. If a listbox is not NULL, it is taken into account by the filtering. After activation of the second listbox, only the empty fields and one displayed value are available to both listboxes. That may seem practical for a limited search. But what if a library catalog shows clearly the classification for an item, but does not show uniquely if this is a book, a CD or a DVD? If the classification is chosen first and the second listbox is then set to "CD", it must be reset to NULL in order to carry out a subsequent search that includes books. It would be more practical if the second listbox showed directly the various media types available, with the corresponding hit counts.

To achieve this aim, the following query is constructed, which is no longer fed directly from the filter results. The number of hits must be obtained in a different way.

```
SELECT
IFNULL( "Field_1" || ' - ' || "Count", 'empty - ' || "Count" ) AS "Display",
"Field_1"
FROM
( SELECT COUNT( "ID" ) AS "Count", "Field_1" FROM "Table" WHERE "ID" IN
( SELECT "Table"."ID" FROM "Filter", "Table" WHERE "Table"."Field_2" =
IFNULL( "Filter"."Filter_2", "Table"."Field_2" ) )
GROUP BY "Field_1" )
ORDER BY "Field_1"
```

This very complex query can be broken down. In practice it is common to use a **`VIEW`** for the sub-query. The listbox receives its content from a query relating to this **`VIEW.`**

The query in detail: The query presents two columns. The first column contains the view seen by a person who has the form open. This view shows the content of the field and, separated by a hyphen, the hits for this field content. The second column transfers its content to the underlying table of the form. Here we have only the content of the field. The listboxes thus draw their content from the query, which is presented as the filter result in the form. Only these fields are available for further filtering.

The table from which this information is drawn is actually a query. In this query the primary key fields are counted (**SELECT COUNT("ID") AS "Count"**). This is then grouped by the search term in the field (**GROUP BY "Field_1"**). This query presents the term in the field itself as the second column. This query in turn is based on a further sub-query:

```
SELECT "Table"."ID" FROM "Filter", "Table" WHERE "Table"."Field_2" =
IFNULL( "Filter"."Filter_2", "Table"."Field_2" )
```

This sub-query deals with the other field to be filtered. In principle, this other field must also match the primary key. If there are further filters, this query can be extended:

```
SELECT "Table"."ID" FROM "Filter", "Table" WHERE
"Table"."Field_2" = IFNULL( "Filter"."Filter_2", "Table"."Field_2" )
AND
"Table"."Field_3" = IFNULL( "Filter"."Filter_3", "Table"."Field_3" )
```

This allows any further fields that are to be filtered to control what finally appears in the listbox of the first field, "Field_1".

Finally the whole query is sorted by the underlying field.

What the final query underlying the displayed form, actually looks like, can be seen from Chapter 8, Database Tasks.

The following macro can control through a listbox which listboxes must be saved and which must be read in again.

The following subroutine assumes that the “Additional information” property for each listbox contains a comma-separated list of all listbox names with no spaces. The first name in the list must be the name of that listbox.

```
SUB Filter_more_info(oEvent AS OBJECT)
   DIM oDoc AS OBJECT
   DIM oDrawpage AS OBJECT
   DIM oForm1 AS OBJECT
   DIM oForm2 AS OBJECT
   DIM oFieldList1 AS OBJECT
   DIM oFieldList2 AS OBJECT
   DIM sTag AS String
   sTag = oEvent.Source.Model.Tag
```

An array (a collection of data accessible via an index number) is established and filled with the field names of the listboxes. The first name in the list is the name of the listbox linked to the event.

```
   aList() = Split(sTag, ",")
   oDoc = thisComponent
   oDrawpage = oDoc.drawpage
   oForm1 = oDrawpage.forms.getByName("Filter")
   oForm2 = oDrawpage.forms.getByName("Display")
```

The array is run through from its lower bound (**'Lbound()'**) to its upper bound (**'Ubound()'**) in a single loop. All values which were separated by commas in the additional information, are now transferred successively.

```
   FOR i = LBound(aList()) TO UBound(aList())
      IF i = 0 THEN
```

The listbox that triggered the macro must be saved. It is found in the variable **aList(0)**. First the information for the listbox is carried across to the underlying table, and then the record is saved.

```
         oForm1.getByName(aList(i)).commit()
         oForm1.updateRow()
      ELSE
```

The other listboxes must be refreshed, as they now contain different values depending on the first listbox.

```
            oForm1.getByName(aList(i)).refresh()
         END IF
      NEXT
      oForm2.reload()
   END SUB
```

The queries for this more usable macro are naturally the same as those already presented in the previous section.

Searching data records

You can search database records without using a macro. However, the corresponding query that must be set up can be very complicated. A macro can solve this problem with a loop.

The following subroutine reads the fields in a table, creates a query internally, and finally writes a list of primary key numbers of records in the table that are retrieved by this search term. In the following description, there is a table called Searchtmp, which consists of an auto-incrementing primary key field (ID) and a field called Nr. that contains all the primary keys retrieved from the table being searched. The table name is supplied initially to the subroutine as a variable.

To get a correct result, the table must contain the content you are searching for as text and not as foreign keys. If necessary, you can create a VIEW for the macro to use.

```
   SUB Search(stTable AS STRING)
      DIM oDataSource AS OBJECT
      DIM oConnection AS OBJECT
      DIM oSQL_Statement AS OBJECT
      DIM stSql AS STRING
      DIM oQuery_Result AS OBJECT
      DIM oDoc AS OBJECT
      DIM oDrawpage AS OBJECT
      DIM oForm AS OBJECT
      DIM oForm2 AS OBJECT
      DIM oField AS OBJECT
      DIM stContent AS STRING
      DIM arContent() AS STRING
      DIM inI AS INTEGER
      DIM inK AS INTEGER
      oDoc = thisComponent
      oDrawpage = oDoc.drawpage
      oForm = oDrawpage.forms.getByName("Searchform")
      oField = oForm.getByName("Searchtext")
      stContent = oField.getCurrentValue()
      stContent = LCase(stContent)
```

The content of the search text field is initially converted into lower case so that the subsequent search function need only compare lower case spellings.

```
      oDataSource = ThisComponent.Parent.DataSource
      oConnection = oDataSource.GetConnection("","")
      oSQL_Statement = oConnection.createStatement()
```

First it must be determined if a search term has actually been entered. If the field is empty, it will be assumed that no search is required. All records will be displayed without further searching.

If a search term has been entered, the column names are read from the table being searched, so that the query can access the fields.

```
   IF stContent <> "" THEN
      stSql = "SELECT ""COLUMN_NAME"" FROM
""INFORMATION_SCHEMA"".""SYSTEM_COLUMNS"" WHERE ""TABLE_NAME"" = '" + stTable
+ "' ORDER BY ""ORDINAL_POSITION"""
      oQuery_Result = oSQL_Statement.executeQuery(stSql)
```

Note	SQL formulas in macros must first be placed in double quotes like normal character strings. Field names and table names are already in double quotes inside the SQL formula. To create final code that transmits the double quotes properly, field names and table names must be given two sets of these quotes. `stSql = "SELECT ""Name"" FROM ""Table"";"` becomes, when displayed with the command `msgbox stSql`, **`SELECT "Name" FROM "Table"`**

The index of the array, in which the field names are written is initially set to 0. Then the query begins to be read out. As the size of the array is unknown, it must be adjusted continuously. That is why the loop begins with **`'ReDim Preserve arContent(inI)'`** to set the size of the array and at the same time to preserve its existing contents. Next the fields are read and the array index incremented by 1. Then the array is dimensioned again and a further value can be stored.

```
InI = 0
IF NOT ISNULL(oQuery_Result) THEN
   WHILE oQuery_result.next
      ReDim Preserve arContent(inI)
      arContent(inI) = oQuery_Result.getString(1)
      inI = inI + 1
   WEND
END IF
stSql = "DROP TABLE ""Searchtmp"" IF EXISTS"
oSQL_Statement.executeUpdate (stSql)
```

Now the query is put together within a loop and subsequently applied to the table defined at the beginning. All case combinations are allowed for, since the content of the field in the query is converted to lower case.

The query is constructed such that the results end up in the Searchtmp table. It is assumed that the primary key is the first field in the table (**`arContent(0)`**).

```
   stSql = "SELECT """+arContent(0)+""" INTO ""Searchtmp"" FROM """ + stTable
    + """ WHERE "
   FOR inK = 0 TO (inI - 1)
      stSql = stSql+"LCase("""+arContent(inK)+""") LIKE '%"+stContent+"%'"
      IF inK < (inI - 1) THEN
         stSql = stSql+" OR "
      END IF
   NEXT
   oSQL_Statement.executeQuery(stSql)
ELSE
   stSql = "DELETE FROM ""Searchtmp"""
   oSQL_Statement.executeUpdate (stSql)
END IF
```

The display form must be reloaded. Its data source is a query, in this example Searchquery.

```
   oForm2 = oDrawpage.forms.getByName("Display")
   oForm2.reload()
End Sub
```

This creates a table that is to be evaluated by the query. As far as possible, the query should be constructed so that it can subsequently be edited. A sample query is shown:

```
SELECT * FROM "Searchtable" WHERE "Nr." IN ( SELECT "Nr." FROM
"Searchtmp" ) OR "Nr." = CASE WHEN ( SELECT COUNT( "Nr." ) FROM
"Searchtmp" ) > 0 THEN '0' ELSE "Nr." END
```

All elements of the **`Searchtable`** are included, including the primary key. No other table appears in the direct query; therefore no primary key from another table is needed and the query result remains editable.

The primary key is saved in this example under the name **Nr.** The macro reads precisely this field. There is an initial check to see if the content of the **Nr.** field appears in the **Searchtmp** table. The **IN** operator is compatible with multiple values. The sub-query can also yield several records.

For larger amounts of data, value matching by using the **IN** operator quickly slows down. Therefore it is not a good idea to use an empty search field simply to transfer all primary key fields from **Searchtable** into the **Searchtmp** table and then view the data in the same way. Instead an empty search field creates an empty **Searchtmp** table, so that no records are available. This is the purpose of the second half of the condition:

```
OR "Nr." = CASE WHEN ( SELECT COUNT( "Nr." ) FROM "Searchtmp" ) > 0
THEN '-1' ELSE "Nr." END
```

If a record is found in the Searchtmp table, it means that the result of the first query is greater than 0. In this case: **"Nr." = '-1'** (here we need a number which cannot occur as a primary key, so **'-1'**is a good value). If the query yields precisely 0 (which will be the case if no records are present), then **"Nr." = "Nr."**. This will list every record which has a **Nr.** As **Nr.** is the primary key, this means all records.

Comboboxes as listboxes with an entry option

A table with a single record can be directly created by using comboboxes and invisible numeric fields and the corresponding primary key entered into another table.

In previous versions of LibreOffice or OpenOffice.org, it was necessary to make the numeric fields invisible by using a macro. This is no longer necessary in LibreOffice as the 'visible' property is now contained in the GUI.

The Combobox control treats form fields for combined entry and choice of values (comboboxes) as listboxes with an entry option. For this purpose, in addition to the comboboxes in the form, the key field values which are to be transferred to the underlying table are stored in separate numeric fields. These fields, prior to OpenOffice.org 3.3, had to have the ‘invisible’ switch set, as the function was not accessible from form design. This is no longer necessary in LibreOffice and OpenOffice.org 3.3. Fields can now be declared as invisible. The keys from these fields are read in when the form is loaded and the combobox is set to show the corresponding content. If the content of the combobox is changed, it is saved and the new primary key is transferred into the corresponding numeric field to be stored in the main table.

The original of this module is in the example database, as expanded with macros.

Text display in comboboxes

This subroutine is to show text in the combobox according to the value of the invisible foreign key fields from the main form. It can also be used for listboxes which refer to two different tables. This might happen if, for example, the postcode in a postal address is stored separately from the town. In that case the postcode might be read from a table that contains only a foreign key for the town. The listbox should show postcode and town together.

```
SUB TextDisplay(NameForm AS STRING, NameSubform AS STRING, NameSubSubform AS
STRING, NameField AS STRING, NameIDField AS STRING, NameTableField1 AS
STRING, NameTableField2 AS STRING, Fieldseparator AS STRING, NameTable1 AS
STRING, OPTIONAL NameTable2 AS STRING, OPTIONAL NameTab12ID AS STRING,
OPTIONAL Position AS INTEGER )
```

This macro should be bound to the following form events: 'When loading' and 'After record change'.

The following parameters are optional and need not be given when the subroutine is called. To prevent a runtime error, default values must be predefined.

```
IF isMissing(NameTable2) THEN NameTable2 = ""
IF isMissing(NameTab12ID) THEN NameTab12ID = ""
IF isMissing(Position) THEN Position = 2
```

The **IF** condition here is only one line, so it does **not** require an **END IF**.

After this, the variables are declared. Some variables have already been declared globally in a separate module and are not declared here again.

```
DIM oForm AS OBJECT
DIM oSubForm AS OBJECT
DIM oSubSubForm AS OBJECT
DIM oField AS OBJECT
DIM oFieldList AS OBJECT
DIM stFieldValue AS STRING
DIM inID AS INTEGER
DIM oCtlView AS OBJECT
oDoc = thisComponent
oDrawpage = oDoc.Drawpage
oForm = oDrawpage.forms.getByName(NameForm)
```

The position of the field in the corresponding form is determined. All forms are held on the **Drawpage** of the current document **thisComponent**. This subroutine uses arguments to cover the possibility that the field is contained in a subform of a subform, that is two levels down from the actual form. The field that contains the foreign key is named **oField**. The combobox, which now exists instead of a listbox is named **oFieldList**.

```
IF NameSubform <> "" THEN
   oSubForm = oForm.getByName(NameSubform)
   IF NameSubSubform <> "" THEN
      oSubSubForm = oSubForm.getByName(NameSubSubform)
      oField = oSubSubForm.getByName(NameIDField)
      oFieldList = oSubSubForm.getByName(NameField)
   ELSE
      oField = oSubForm.getByName(NameIDField)
      oFieldList = oSubForm.getByName(NameField)
   END IF
ELSE
   oField = oForm.getByName(NameIDField)
   oFieldList = oForm.getByName(NameField)
END IF
oFieldList.Refresh()
```

The combobox is read in again using **Refresh()**. It might be that the content of the field has been changed by the entry of new data. This must be made possible.

After this, the value of the foreign key is read. Only if a value is entered here will a connection be made to the data source.

```
IF NOT IsEmpty(oField.getCurrentValue()) THEN
   inID = oField.getCurrentValue()
   oDataSource = ThisComponent.Parent.CurrentController
   IF NOT (oDataSource.isConnected()) Then
      oDataSource.connect()
   End IF
   oConnection = oDataSource.ActiveConnection()
   oSQL_Statement = oConnection.createStatement()
```

The SQL statement is formulated according to the fields used in the combobox. This requires the testing of various combinations. The most general is that the combobox must be provided with a query that refers to two table fields from different tables. This subroutine is not designed for further possibilities. The test begins with this most important possibility.

```
   IF NameTableField2 <> "" THEN
```

If a second table field exists,

```
IF NameTable2 <> "" THEN
```

... and if a second table exists, the following SQL code will be produced:

```
IF Position = 2 THEN
   stSql = "SELECT """ + NameTable1 + """."" + NameTableField1 + """||'" +
Fieldseparator + "'||""" + NameTable2 + """."" + NameTablenField2 + """ FROM
""" + NameTable1 + """,""" + NameTable2 + """ WHERE ""ID""='" +inID+ "' AND
""" + NameTable1 + """."" + NameTab12ID + """=""" + NameTable2 + """.""ID"""
ELSE
   stSql = "SELECT """ + NameTable2 + """."" + NameTableField2 + """||'" +
Fieldseparator + "'||""" + NameTable1 + """."" + NameTableField1 + """ FROM
""" + NameTable1 + """,""" + NameTable2 + """ WHERE ""ID""='" + inID + "' AND
""" + NameTable1 + """."" + NameTab12ID + """=""" + NameTable2 + """.""ID"""
END IF
```

When written out in the form required by Basic, this SQL command is pretty incomprehensible. Each field and each table name must be written into the SQL input with two sets of double quotes as shown above. As double quotes are normally interpreted by Basic as indicating text, they disappear when the code is transferred. Only when a second set of double quotes is used are terms passed on in simple quotes. **""ID""** therefore means that the **"ID"** field (with a single set of quotation marks for the SQL relationship) is accessed in the query. A partial simplification for this subroutine is that all primary keys in this database carry the name ID.

The code is further complicated by the fact that, as well as requiring duplicate quotation marks, some of the included table fields are entered as variables. These are not text; they are simply concatenated with the preceding text by using + . But these variables must also be masked. That is why these variables are shown above with three sets of quotation marks: **"""+NameTable1+"""."""+NameTableField1+"""** finallly translates into the familiar query language **"Table1"."Field1"**. Fortunately when we create the macro, the coloring of the code shows if any double quotes have been omitted. The quotation marks change their color immediately if they are not recognised by the macro as string delimiters.

```
   ELSE
```

... and if the second table does not exist, the following SQL code is created:

```
   IF Position = 2 THEN
      stSql = "SELECT """ + NameTableField1 + """||'" + Fieldseparator +
"'||""" + NameTableField2 + """ FROM """ + NameTable1 + """ WHERE ""ID""='" +
inID + "'"
   ELSE
      stSql = "SELECT """ + NameTableField2 + """||'" + Fieldseparator +
"'||""" + NameTableField1 + """ FROM """ + NameTable1 + """ WHERE ""ID""='" +
inID + "'"
   END IF
   END IF
ELSE
```

If a second table field does not exist there can be only one table. This yields the following query:

```
   stSql = "SELECT """ + NameTableField1 + """ FROM """ + NameTable1 +
""" WHERE ""ID""='" + inID + "'"
END IF
```

The query stored in the variable **stSql** is now run and the result of the query is stored in the variable **oQuery_result**.

```
oQuery_result = oSQL_Statement.executeQuery(stSql)
```

The query result is read in a loop. Here, as in GUI queries, several fields and records can be created. But the construction of this query will produce only one result. This result will be found in the first column **(1)** of the query. It is the record which provides the content which the combobox is

to display. The content is a text string (**'getString()'**), so here we see **'oQuery_result.getString(1)'**.

```
   IF NOT IsNull(oQuery_result) THEN
      WHILE oQuery_result.next
         stFieldValue = oQuery_result.getString(1)
      WEND
```

The combobox must now be set to the text values resulting from the query. This requires access to the Controller.

```
      oDocCrl = ThisComponent.getCurrentController()
      oCtlView = oDocCrl.GetControl(oFieldList)
      oCtlView.setText(stFieldValue)
   END IF
   END IF
```

If there is no value in the field for the foreign key 'oField', it means that the query has failed. The combobox is then set to an empty display.

```
      IF IsEmpty(oField.getCurrentValue()) THEN
         oDocCrl = ThisComponent.getCurrentController()
         oCtlView = oDocCrl.GetControl(oFieldList)
         oCtlView.setText("")
      END IF
   END SUB
```

This subroutine handles the contact between the foreign keys in a hidden field and the combobox. This is sufficient for the display of the correct values in the combobox. To store a new value requires a further subroutine.

Transfering a foreign key value from a combobox to a numeric field

If a new value is entered into the combobox (and this after all is the purpose for which this macro was constructed), the corresponding primary key must be entered into the form's underlying table as a Foreign key.

```
   SUB TextChoiceValueSave(NameForm AS STRING, NameSubform AS STRING,
   NameSubSubform AS STRING, NameField AS STRING, NameIDField AS STRING,
   NameTableField1 AS STRING, NameTableField2 AS STRING, FieldSeparator AS
   STRING, NameTable1 AS STRING, OPTIONAL NameTable2 AS STRING, OPTIONAL
   NameTab12ID AS STRING, OPTIONAL Position AS INTEGER )
```

This macro should be bound to the following form event: 'Before record action'.

The start of this function corresponds in principle with the previously described subroutine. Here too there are optional variables.

```
      IF isMissing(NameTable2) THEN NameTable2 = ""
      IF isMissing(NameTab12ID) THEN NameTab12ID = ""
      IF isMissing(Position) THEN Position = 2
```

After this the other variables are declared, those that have not yet been declared outside the subroutine or function. Then the form is loaded and the relevant fields are allocated to variables. The field **oField** contains the foreign key and **oFieldList** shows the corresponding text.

```
      DIM oForm AS OBJECT
      DIM oSubForm AS OBJECT
      DIM oSubSubForm AS OBJECT
      DIM oField AS OBJECT
      DIM oFieldList AS OBJECT
      DIM stContent AS STRING
      DIM stContentField2 AS STRING
      DIM a_stPart() AS STRING
      DIM stmsgbox1 AS STRING
      DIM stmsgbox2 AS STRING
```

```
DIM inID1 AS INTEGER
DIM inID2 AS INTEGER
DIM Field1Length AS INTEGER
DIM Field2Length AS INTEGER
```

The maximum length allowed for an entry is determined using the function Columnsize, described below. Just setting a limit on the size of the combobox is not safe here, as we need to include the possibility of entering two fields together.

```
Field1Length = Columnsize(NameTable1,NameTableField1)
IF NameTableField2 <> "" THEN
   IF NameTable2 <> "" THEN
      Feld2Length = Columnsize(NameTable2,NameTableField2)
   ELSE
      Feld2Length = Columnsize(NameTable1,NameTableField2)
   END IF
ELSE
   Feld2Length = 0
END IF
```

The form is loaded, and the comobox is read.

```
oDoc = thisComponent
oDrawpage = oDoc.Drawpage
oForm = oDrawpage.Forms.getByName(NameForm)
IF NameSubform <> "" THEN
   oSubForm = oForm.getByName(NameSubform)
   IF NameSubSubform <> "" THEN
      oSubSubForm = oSubForm.getByName(NameSubSubform)
      oField = oSubSubForm.getByName(NameIDField)
      oFieldList = oSubSubForm.getByName(NameField)
   ELSE
      oField = oSubForm.getByName(NameIDField)
      oFieldList = oSubForm.getByName(NameField)
   END IF
ELSE
   oField = oForm.getByName(NameIDField)
   oFieldList = oForm.getByName(NameField)
END IF
stContent = oFieldList.getCurrentValue()
```

The displayed content of the combobox is read. Leading and trailing spaces and non-printing characters are removed if necessary.

```
StContent = Trim(stContent)
IF stContent <> "" THEN
   IF NameTableField2 <> "" THEN
```

If a second table field exists, the content of the combobox must be split. To determine where the split is to occur, we use the field separator provided to the function as an argument.

```
      a_stPart = Split(stContent, FieldSeparator, 2)
```

The last parameter signifies that the maximum number of parts is 2.

Depending on which entry corresponds to field 1 and which to field 2, the content of the combobox is now allocated to the individual variables. "Position = 2" serves here as a sign that the second part of the content stands for Field 2.

```
      IF Position = 2 THEN
         stContent = Trim(a_stPart(0))
         IF UBound(a_stPart()) > 0 THEN
            stContentField2 = Trim(a_stPart(1))
         ELSE
            stContentField2 = ""
         END IF
```

```
            stContentField2 = Trim(a_stPart(1))
         ELSE
            stContentField2 = Trim(a_stPart(0))
            IF UBound(a_stPart()) > 0 THEN
               stContent = Trim(a_stPart(1))
            ELSE
               stContent = ""
            END IF
            stContent = Trim(a_stPart(1))
         END IF
      END IF
```

It can happen that with two separable contents, the installed size of the combobox (text length) does not fit the table fields to be saved. For comboboxes that represent a single field, this is normally handled by suitably configuring the form control. Here by contrast, we need some way of catching such errors. The maximum permissible length of the relevant field is checked.

```
   IF (Field1Length > 0 AND Len(stContent) > Field1Length) OR (Field2Length >
0 AND Len(stContentField2) > Field2Length) THEN
```

If the field length of the first or second part is too big, a default string is stored in one of the variables. The character **CHR(13)** is used to put in a line break .

```
      stmsgbox1 = "The field " + NameTableField1 + " must not exceed " +
Field1Length + "characters in length." + CHR(13)
      stmsgbox2 = "The field " + NameTableField2 + " must not exceed " +
Field2Length + "characters in length." + CHR(13)
```

If both field contents are too long, both texts are displayed.

```
      IF (Field1Length > 0 AND Len(stContent) > Field1Length) AND
(Field2Length > 0 AND Len(stContentField2) > Field2Length) THEN
         msgbox ("The entered text is too long." + CHR(13) + stmsgbox1 +
stmsgbox2 + "Please shorten it.",64,"Invalid entry")
```

The display uses the **msgbox()** function. This expects as its first argument a text string, then optionally a number (which determines the type of message box displayed), and finally an optional text string as a title for the window. The window will therefore have the title "Invalid entry" and the number '64' provides a box containing the Information symbol.

The following code covers any further cases of excessively long text that might arise.

```
      ELSEIF (Field1Length > 0 AND Len(stContent) > Field1Length) THEN
         msgbox ("The entered text is too long." + CHR(13) + stmsgbox1 +
"Please shorten it.",64,"Invalid entry")
      ELSE
         msgbox ("The entered text is too long." + CHR(13) + stmsgbox2 +
"Please shorten it.",64,"Invalid entry")
      END IF
   ELSE
```

If there is no excessively long text, the function can proceed. Otherwise it exits here.

First variables are preallocated which can subsequently be altered by the query. The variables inID1 and inID2 store the content of the primary key fields of the two tables. If a query yields no results, Basic assigns these integer variable a value of 0. However this value could also indicate a successful query returning a primary key value of 0; therefore the variable is preset to -1. HSQLDB cannot set this value for an autovalue field.

Next the database connection is set up, if it does not already exist.

```
      inID1 = -1
      inID2 = -1
      oDataSource = ThisComponent.Parent.CurrentController
      If NOT (oDataSource.isConnected()) Then
         oDataSource.connect()
```

```
        End If
        oConnection = oDataSource.ActiveConnection()
        oSQL_Statement = oConnection.createStatement()
        IF NameTableField2 <> "" AND NOT IsEmpty(stContentField2) AND
NameTable2 <> "" THEN
```

If a second table field exists, a second dependency must first be declared.

```
           stSql = "SELECT ""ID"" FROM """ + NameTable2 + """ WHERE """ +
NameTableField2 + """='" + stContentField2 + "'"
           oQuery_result = oSQL_Statement.executeQuery(stSql)
           IF NOT IsNull(oQuery_result) THEN
              WHILE oQuery_result.next
                 inID2 = oQuery_result.getInt(1)
              WEND
           END IF
           IF inID2 = -1 THEN
              stSql = "INSERT INTO """ + NameTable2 + """ (""" +
NameTableField2 + """) VALUES ('" + stContentField2 + "') "
              oSQL_Statement.executeUpdate(stSql)
              stSql = "CALL IDENTITY()"
```

If the content within the combobox is not present in the corresponding table, it is inserted there. The primary key value which results is then read. If it is present, the existing primary key is read in the same way. The function uses the automatically generated primary key fields (**IDENTITY**).

```
              oQuery_result = oSQL_Statement.executeQuery(stSql)
              IF NOT IsNull(oQuery_result) THEN
                 WHILE oQuery_result.next
                    inID2 = oQuery_result.getInt(1)
                 WEND
              END IF
           END IF
```

The primary key for the second value is temporarily stored in the variable **inID2** and then written as a foreign key into the table corresponding to the first value. According to whether the record from the first table was already available, the content is freshly saved (**INSERT**) or altered (**UPDATE**):

```
           IF inID1 = -1 THEN
              stSql = "INSERT INTO """ + NameTable1 + """ (""" +
NameTableField1 + """,""" + NameTab12ID + """) VALUES ('" + stContent + "','"
+ inID2 + "') "
              oSQL_Statement.executeUpdate(stSql)
```

And the corresponding ID directly read out:

```
              stSql = "CALL IDENTITY()"
              oQuery_result = oSQL_Statement.executeQuery(stSql)
              IF NOT IsNull(oQuery_result) THEN
                 WHILE oQuery_result.next
                    inID1 = oQuery_result.getInt(1)
                 WEND
              END IF
```

The primary key for the first table must finally be read again so that it can be transferred to the form's underlying table.

```
           ELSE
              stSql = "UPDATE """ + NameTable1 + """ SET """ + NameTab12ID +
"""='" + inID2 + "' WHERE """ + NameTableField1 + """ = '" + stContent + "'"
              oSQL_Statement.executeUpdate(stSql)
           END IF
        END IF
```

In the case where both the fields underlying the combobox are in the same table (for example Surname and Firstname in the Name table), a different query is needed:

```
        IF NameTableField2 <> "" AND NameTable2 = "" THEN
           stSql = "SELECT ""ID"" FROM """ + NameTable1 + """ WHERE """ +
NameTableField1 + """='" + stContent + "' AND """ + NameTableField2 + """='"
+ stContentField2 + "'"
           oQuery_result = oSQL_Statement.executeQuery(stSql)
           IF NOT IsNull(oAbfrageergebnis) THEN
              WHILE oQuery_result.next
                 inID1 = oQuery_result.getInt(1)
              WEND
           END IF
           IF inID1 = -1 THEN
```

... and a second table does not exist:

```
           stSql = "INSERT INTO """ + NameTable1 + """ (""" + NameTableField1 +
""","""" + NameTableField2 + """) VALUES ('" + stContent + "','" +
stContentField2 + "') "
           oSQL_Statement.executeUpdate(stSql)
```

Then the primary key is read again.

```
           stSql = "CALL IDENTITY()"
           oquery_result = oSQL_Statement.executeQuery(stSql)
           IF NOT IsNull(oQuery_result) THEN
              WHILE oQuery_result.next
                 inID1 = oQuery_result.getInt(1)
              WEND
           END IF
        END IF
     END IF
     IF NameTableField2 = "" THEN
```

Now we consider the simplest case: The second table field does not exist and the entry is not yet present in the table. In other words, a single new value has been entered into the combobox.

```
        stSql = "SELECT ""ID"" FROM """ + NameTable1 + """ WHERE """ +
NameTableField1 + """='" + stContent + "'"
        oQuery_result = oSQL_Statement.executeQuery(stSql)
        IF NOT IsNull(oQuery_result) THEN
           WHILE oQuery_result.next
              inID1 = oQuery_result.getInt(1)
           WEND
        END IF
        IF inID1 = -1 THEN
```

If there is no second field, the content of the box is inserted as a new record.

```
           stSql = "INSERT INTO """ + NameTable1 + """ (""" + NameTableField1 +
""") VALUES ('" + stContent + "') "
           oSQL_Statement.executeUpdate(stSql)
```

… and the resulting ID directly read out.

```
           stSql = "CALL IDENTITY()"
           oQuery_result = oSQL_Statement.executeQuery(stSql)
           IF NOT ISNULL(oQuery_result) THEN
              WHILE oQuery_result.next
                 inID1 = oQuery_result.getInt(1)
              WEND
           END IF
        END IF
     END IF
```

The value of the primary key field must be determined, so that it can be transferred to the main part of the form.

Next the primary key value that has resulted from all these loops is transferred to the invisible field in the main table and the underlying database. The table field linked to the form field is reached by using **'BoundField'**. **'updateInt'** places an integer (see under numerical type definitions) in this field.

```
            oField.BoundField.updateInt(inID)
         END IF
      ELSE
```

If no primary key is to be entered, because there was no entry in the combobox or that entry was deleted, the content of the invisible field must also be deleted. **updateNull()** is used to fill the field with the database-specific expression for an empty field, **NULL**.

```
         oField.BoundField.updateNull()
      END IF
   END SUB
```

Function to measure the length of the combobox entry

The following function gives the number of characters in the respective table column, so that entries that are too long do not just get truncated. A **FUNCTION** is chosen here to provide return values. A **SUB** has no return value that can be passed on and processed elsewhere.

```
FUNCTION ColumnSize(Tablename AS STRING, Fieldname AS STRING) AS INTEGER
   oDataSource = ThisComponent.Parent.CurrentController
   If NOT (oDataSource.isConnected()) Then
      oDataSource.connect()
   End If
   oConnection = oDataSource.ActiveConnection()
   oSQL_Statement = oConnection.createStatement()
   stSql = "SELECT ""COLUMN_SIZE"" FROM
""INFORMATION_SCHEMA"".""SYSTEM_COLUMNS"" WHERE ""TABLE_NAME"" = '" +
Tablename + "' AND ""COLUMN_NAME"" = '" + Fieldname + "'"
   oQuery_result = oSQL_Statement.executeQuery(stSql)
   IF NOT IsNull(oQuery_result) THEN
      WHILE oQuery_result.next
         i = oQuery_result.getInt(1)
      WEND
   END IF
   Columnsize = i
END FUNCTION
```

Calling the subroutine for displaying texts

The subroutine for creating the combobox is called each time a record is changed. The following example shows this for the sample database.

```
SUB Form_Reader_Input_Load
   REM TextDisplay(NameForm AS STRING, NameSubForm AS STRING, NameSubSubForm
AS STRING, NameField AS STRING, NameIDField AS STRING, NameTableField1(from
Table 1) AS STRING, NameTableField2(from Table 1 or from Table 2) AS STRING,
FieldSeparator AS STRING, NameTable1 AS STRING, OPTIONAL NameTable2 AS
STRING, OPTIONAL NameIDFromTable2InTable1 AS STRING, OPTIONAL Field Position
from the table2 in the Combobox AS INTEGER [1 or 2] )
   TextDisplay ("Filter", "Form", "Address", "comStr", "numStrID", "Street",
"", "", "Street")
   TextDisplay ("Filter", "Form", "Address", "comPlcTown", "numPlcTownID",
"Postcode", "Town", " ", "Postcode", "Town", "Town_ID", 2)
END SUB
```

The comment lines show the list of parameters that need to be provided for the subroutine. An empty parameter is represented by a pair of double quotes. The last three parameters are optional as they are covered where necessary by default values in the subroutine.

The **TextDisplay** subroutine is called. The form that contains the fields is **Filter > Form > Address**. We are therefore dealing with the subform of a subform.

The first combobox, in which the street is entered, is called **comStr**, the hidden foreign key field in the table underlying the form is called **numStrID**. In this first combobox the field **Street** is displayed. The table, which will contain the entries from the combobox, is also called **Street**.

The town and postcode are entered into the second combobox. This is called **comPlcTown**. The hidden foreign key field is called **numPlcTownID**. This second combobox contains the **Postcode** and **Town**, separated by a space (**" "**). The first table has the name **Postcode**, the second table the name **Town**. In the first table the foreign key for the second table is called **Town_ID**. The field for the second table is the second element in the combobox, that is position **2**.

Calling the subroutine for text storage

To store the entry, we use the subroutine TextChoiceValueSave.

```
SUB Form_Reader_Output_Save
   REM TextChoiceValueSave(NameForm AS STRING, NameSubForm AS STRING,
NameSubSubForm AS STRING, NameField AS STRING, NameIDField AS STRING,
NameTableField1(from Table 1) AS STRING, NameTableField2(from table 1 or
table 2) AS STRING, FieldSeparator AS STRING,NameTable1 AS STRING, OPTIONAL
NameTable2 AS STRING, OPTIONAL NameIDFromTable2InTable1 AS STRING, OPTIONAL
Position of the field from Table2 in the Combobox AS INTEGER [1 or 2] )
```

The comment is similar to the one in the previous subroutine. The variables being passed are also the same.

```
   TextChoiceValueSave ("Filter", "Form", "Address", "comStr", "numStrID",
"Street", "", "", "Street", "", "")
   TextChoiceValueSave ("Filter", "Form", "Address", "comPlcTown",
"numPlcTownID", "Postcode", "Town", " ", "Postcode", "Town", "Town_ID", 2)
END SUB
```

To safely store the value, the form must be informed about changes. As far as the user is concerned, the change takes place only to the text inside the combobox to which the form has no further connection. Rather it is the numeric field containing the foreign key that is the link to the database. So the foreign key is simply assigned the value -1, which an autovalue field cannot legally have. This ensures a change in the field content. This field content is then copied over by the normal operation of the listbox.

```
SUB RecordAction_produce(oEvent AS OBJECT)
```

This macro should be bound to the following event for the listbox: 'On focus'. It is necessary so that the save operation takes place in all cases where the listbox content changes. Without this macro there will be no change in the table, that Base can recognise, since the combobox is not linked to the form.

```
    DIM oForm AS OBJECT
    DIM oSubForm AS OBJECT
    DIM oSubSubForm AS OBJECT
    DIM oField AS OBJECT
    DIM stTag AS String
    stTag = oEvent.Source.Model.Tag
    aForms() = Split(stTag, ",")
```

An array is populated; the field name comes first and then the form names, with the main form preceding the subform.

```
    oDoc = thisComponent
    oDrawpage = oDoc.Drawpage
    oForm = oDrawpage.Forms.getByName(aForms(1))
    IF UBound(aForms()) > 1 THEN
       oForm = oForm.getByName(aForms(2))
       IF UBound(aForms()) > 2 THEN
          oForm = oForm.getByName(aForms(3))
       END IF
    END IF
    oField = oForm.getByName(aForms(0))
    oField.BoundField.updateInt(-1)
END SUB
```

Navigation from one form to another

A form is to be opened when a particular event occurs.

In the form control properties, on the line "Additional information" (tag), enter the name of the form. Further information can also be entered here, and subsequently separated out by using the **`Split()`** function.

```
SUB From_form_to_form(oEvent AS OBJECT)
   DIM stTag AS String
   stTag = oEvent.Source.Model.Tag
   aForm() = Split(stTag, ",")
```

The array is declared and filled with the form names, first the form to be opened and secondly the current form, which will be closed after the other has been opened.

```
   ThisDatabaseDocument.FormDocuments.getByName( Trim(aForm(0)) ).open
   ThisDatabaseDocument.FormDocuments.getByName( Trim(aForm(1)) ).close
END SUB
```

If instead, the other form is only to be opened when the current one is closed, for example where a main form exists and all other forms are controlled from it using buttons, the following macro should be bound to the form with **Tools > Customize > Events > Document closed**:

```
SUB Mainform_open
   ThisDatabaseDocument.FormDocuments.getByName( "Mainform" ).open
END SUB
```

If the form documents are sorted within the ODB file into directories, the macro for changing the form needs to be more extensive:

```
SUB From_form_to_form_with_folders(oEvent AS OBJECT)
   REM The form to be opened is given first.
   REM If a form is in a folder, use "/" to define the relationship
   REM so that the subfolder can be found.
   DIM stTag AS STRING
   stTag = oEvent.Source.Model.Tag 'Tag is entered in the additional
information
   aForms() = Split(stTag, ",")  'Here the form name for the new form comes
first, then the one for the old form
   aForms1() = Split(aForms(0),"/")
   aForms2() = Split(aForms(1),"/")
   IF UBound(aForms1()) = 0 THEN
      ThisDatabaseDocument.FormDocuments.getByName( Trim(aForms1(0)) ).open
   ELSE
      ThisDatabaseDocument.FormDocuments.getByName(
Trim(aForms1(0)) ).getByName( Trim(aForms1(1)) ).open
   END IF
```

```
    IF UBound(aForms2()) = 0 THEN
       ThisDatabaseDocument.FormDocuments.getByName( Trim(aForms2(0)) ).close
    ELSE
       ThisDatabaseDocument.FormDocuments.getByName(
Trim(aForms2(0)) ).getByName( Trim(aForms2(1)) ).close
    END IF
END SUB
```

Form documents that lie in a directory are entered into the Additional Information field as directory/form. This must be converted to:

`...getByName("Directory").getByName("Form")`.

Removing distracting elements from forms

Toolbars have no function in forms. They are more likely to irritate the normal user, as the form is not being edited while data is input. These macros allow toolbars to be removed and subsequently reinstated. However, depending on the LibreOffice version, menu bars in text form can only be temporarily removed and then reappear.

```
Sub Toolbar_remove
   DIM oFrame AS OBJECT
   DIM oLayoutMng AS OBJECT
   oFrame = thisComponent.CurrentController.Frame
   oLayoutMng = oFrame.LayoutManager
   oLayoutMng.visible = false
   oLayoutMng.showElement("private:Resource/menubar/menubar")
End Sub

Sub Toolbar_restore
   DIM oFrame AS OBJECT
   DIM oLayoutMng AS OBJECT
   oFrame = thisComponent.CurrentController.Frame
   oLayoutMng = oFrame.LayoutManager
   oLayoutMng.visible = true
End Sub
```

When a toolbar is removed, all bars are affected. However as soon as a form control is clicked, the menu bar reappears. This is a safety precaution so that the user does not end up in a jam. To prevent constant toggling, the menu bar reappears.

Database tasks expanded using macros

Making a connection to a database

```
oDataSource = ThisComponent.Parent.DataSource
IF NOT oDataSource.IsPasswordRequired THEN
   oConnection = oDataSource.GetConnection("","")
```

Here it would be possible to provide a username and a password, if one were necessary. In that case the brackets would contain ("Username","Password"). Instead of including the username and a password in clear text, the dialog for password protection is called up:

```
ELSE
   oAuthentication = createUnoService("com.sun.star.sdb.InteractionHandler")
   oConnection = oDataSource.ConnectWithCompletion(oAuthentication)
END IF
```

If however a form within the same Base file is accessing the database, you only need:

```
oDataSource = ThisComponent.Parent.CurrentController
```

```
IF NOT (oDataSource.isConnected()) Then
   oDataSource.connect()
End IF
oConnection = oDataSource.ActiveConnection()
```

Here the database is known so a username and a password are not necessary, as these are already switched off in the basic HSQLDB configuration for internal version.

For forms outside Base, the connection is made through the first form:

```
oDataSource = Thiscomponent.Drawpage.Forms(0)
oConnection = oDataSource.activeConnection
```

Securing your database

It can sometimes happen, especially when a database is being created, that the ODB file is unexpectedly truncated. Frequent saving after editing is therefore useful, especially when using the Reports module.

When the database is in use, it can be damaged by operating system failure, if this occurs just as the Base file is being terminated. This is when the content of the database is being written into the file.

In addition, there are the usual suspects for files that suddenly refuse to open, such as hard drive failure. It does no harm therefore to have a backup copy which is as up-to-date as possible. The state of the data does not change as long as the ODB file remains open. For this reason, safety subroutines can be directly linked to the opening of the file. You simply copy the file using the backup path given in **Tools > Options > LibreOffice > Paths**. This macro begins to overwrite the oldest version after five copies have been made.

```
SUB Databasebackup
   REM The database file *.odb is copied into the Backup directory.
   REM The maximum number of copies is set to 5. After that, the oldest copy
is replaced.
   REM This method does not cover:
   REM - data entry into a database that is already open as the data are
written into the *.odb file only when it is closed.
   DIM oPath AS OBJECT
   DIM oDoc AS OBJECT
   DIM sTitle AS STRING
   DIM sUrl_end AS STRING
   DIM sUrl_start AS STRING
   DIM i AS INTEGER
   DIM k AS INTEGER
   oDoc = ThisComponent
   sTitle = oDoc.Title
   sUrl_start = oDoc.URL
   oPath = createUnoService("com.sun.star.util.PathSettings")
   FOR i = 1 TO 6
      IF NOT FileExists(oPath.Backup & "/" & i & "_" & sTitle) THEN
         IF i > 5 THEN
            FOR k = 1 TO 4
            IF FileDateTime(oPath.Backup & "/" & k & "_" & sTitle) <=
FileDateTime(oPath.Backup & "/" & k+1 & "_" & sTitle) THEN
               i = k
               EXIT FOR
            END IF
            NEXT
         END IF
         EXIT FOR
      END IF
```

```
    NEXT
    sUrl_end = oPath.Backup & "/" & i &"_" & sTitle
    FileCopy(sUrl_start,sUrl_end)
END SUB
```

You can also do a backup while Base is running, provided that the data can be written back out of the cache into the file before the Databasebackup subroutine is carried out. It might be useful to do this, perhaps after a specific elapsed time or when an on-screen button is pressed. This cache-clearing is handled by the following subroutine:

```
SUB Write_data_out_of_cache
    REM Writes the data out of the table directly to disk while Base is
running.
    DIM oData AS OBJECT
    DIM oDataSource AS OBJECT
    oData = ThisDatabaseDocument.CurrentController
    IF NOT ( oData.isConnected() ) THEN oData.connect()
    oDataSource = oData.DataSource
    oDataSource.flush
END SUB
```

Database compaction

This is simply a SQL command (**SHUTDOWN COMPACT**), which should be carried out now and again, especially after a lot of data has been deleted. The database stores new data, but still reserves the space for the deleted data. In cases where the data have been substantially altered, you therefore need to compact the database.

Once compaction is carried out, the tables are no longer accessible. The file must be reopened. Therefore this macro closes the form from which it is called. Unfortunately you cannot close the document itself without causing a recovery when it is opened again. Therefore this function is commented out.

```
SUB Database_compaction
    DIM stMessage AS STRING
    oDataSource = ThisComponent.Parent.CurrentController  ' Accessible from
the form
    IF NOT (oDataSource.isConnected()) THEN
       oDataSource.connect()
    END IF
    oConnection = oDataSource.ActiveConnection()
    oSQL_Statement = oConnection.createStatement()
    stSql = "SHUTDOWN COMPACT" ' The database is being compacted and shut down
    oSQL_Statement.executeQuery(stSql)
    stMessage = "The database is being compacted." + CHR(13) + "The form will
now close."
    stMessage = stMessage + CHR(13) + "Following this, the database file
should be closed."
    stMessage = stMessage + CHR(13) + "The database can only be accessed after
reopening the database file."
    msgbox stMessage
    ThisDatabaseDocument.FormDocuments.getByName( "Maintenance" ).close
    REM The closing of the database file causes a recovery operation when you
open it again.
'   ThisDatabaseDocument.close(True)
END SUB
```

Decreasing the table index for autovalue fields

If a lot of data is deleted from a table, users are often concerned that the sequence of automatically generated primary keys simply continues upwards instead of starting again at the highest current

value of the key. The following subroutine reads the currently highest value of the "ID" field in a table and sets the next initial key value 1 higher than this maximum.

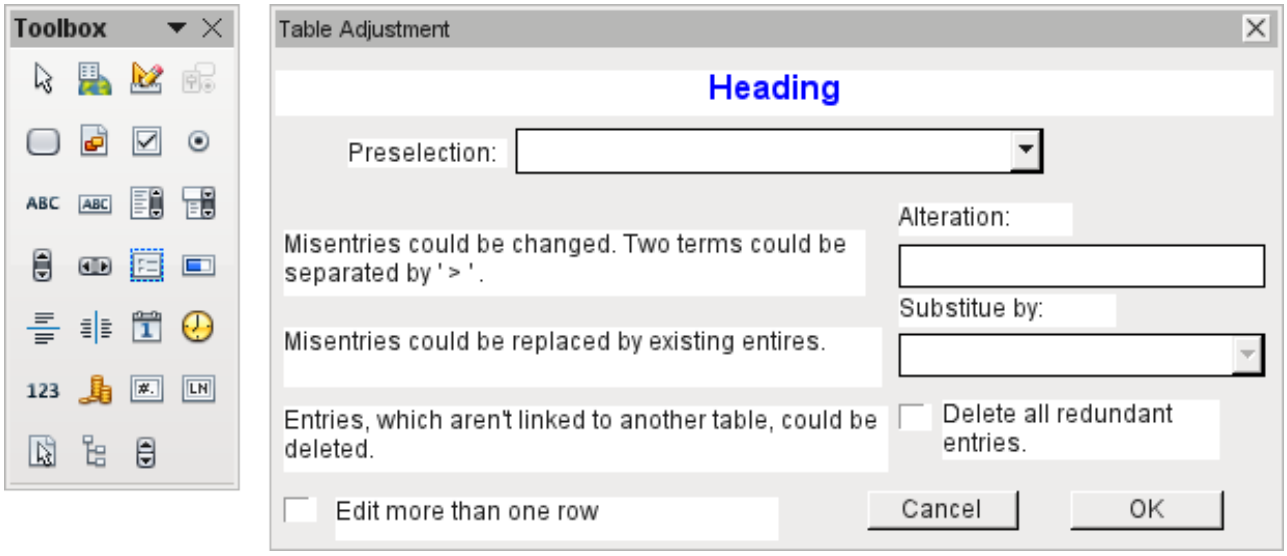
If the primary key field is not called ID, the macro must be edited accordingly.

```
SUB Table_index_down(stTable AS STRING)
   REM This subroutine sets the automatically incrementing primary key field
mit the preset name of "ID" to the lowest possible value.
   DIM inCount AS INTEGER
   DIM inSequence_Value AS INTEGER
   oDataSource = ThisComponent.Parent.CurrentController  ' Accessible through
the  form
   IF NOT (oDataSource.isConnected()) THEN
      oDataSource.connect()
   END IF
   oConnection = oDataSource.ActiveConnection()
   oSQL_Statement = oConnection.createStatement()
   stSql = "SELECT MAX(""ID"") FROM """+stTable+"""  ' The highest value in
"ID" is determined
   oQuery_result = oSQL_Statement.executeQuery(stSql) ' Query is launched and
the return value stored in the variable oQuery_result
   IF NOT ISNULL(oQuery_result) THEN
      WHILE oQuery_result.next
      inCount = oQuery_result.getInt(1)   ' First data field is read
      WEND  ' next record, in this case none as only one record exists
      IF inCount = "" THEN ' If the highest value is not a value, meaning the
table is empty, the highest value is set to -1
         inCount = -1
      END IF
      inSequence_Value = inCount+1  ' The highrst value is increased by 1
      REM A new command is prepared for the database. The ID will start
afresh from inCount+1.
      REM This statement has no return value, as no record is being read
      oSQL_statement = oConnection.createStatement()
      oSQL_statement.executeQuery("ALTER TABLE """ + stTable + """ ALTER
COLUMN ""ID"" RESTART WITH " + inSequence_Value + "")
   END IF
END SUB
```

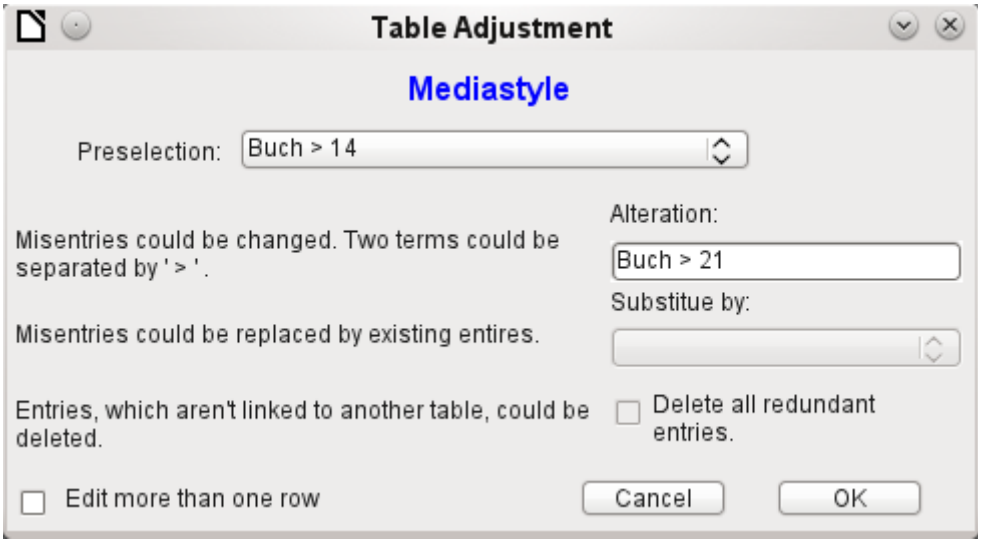
Dialogs

Input errors in fields are often only noticed later. Often it is necessary to modify identical entries in several records at the same time. It is awkward to have to do this in normal table view, especially when several records must be edited, as each record requires an individual entry to be made.

Forms can use macros to do this kind of thing, but to do it for several tables, you would need identically constructed forms. Dialogs can do the job. A dialog can be supplied at the beginning with the necessary data for appropriate tables and can be called up by several different forms.



Dialogs are saved along with the modules for macros. Their creation is similar to that of a form. Very similar control fields are available. Only the table control of forms is absent as a special entry possibility.



The appearance of dialog controls is determined by the settings for the graphical user interface.

The dialog shown above serves in the example database to edit tables which are not used directly as the basis of a form. So, for example, the media type is accessible only through a listbox (in the macro version it becomes a combobox). In the macro version, the field contents can be expanded by new content but an alteration of existing content is not possible. In the version without macros, alterations are carried out using a separate table control.

While alterations in this case are easy to carry out without macros, it is quite difficult to change the media type of many media at once. Suppose the following types are available: "Book, bound", "Book, hard-cover", "Paperback" and "Ringfile". Now it turns out, after the database has been in use for a long time, that more active contemporaries foresaw similar additional media types for printed works. The task of differentiating them has become excessive. We therefore wish to reduce them, preferably to a single term. Without macros, the records in the media table would have to be found (using a filter) and individually altered. If you know SQL, you can do it much better using a SQL command. You can change all the records in the Media table with a single entry. A second SQL command then removes the now surplus media types which no longer have any link to the

Media table. Precisely this method is applied using this dialog's Replace With box – only the SQL command is first adapted to the Media Type table using a macro that can also edit other tables.

Often entries slip into a table which with hindsight can be changed in the form, and so are no longer needed. It does no harm simply to delete such orphaned entries, but they are quite hard to find using the graphical user interface. Here again a suitable SQL command is useful, coupled with a delete instruction. This command for affected tables is included in the dialog under *Delete all superfluous entries*.

If the dialog is to be used to carry out several changes, this is indicated by the *Edit multiple records* checkbox. Then the dialog will not simply terminate when the OK button is clicked.

The macro code for this dialog can be seen in full in the example database. Only excerpts are explained below.

```
SUB Table_purge(oEvent AS OBJECT)
```

The macro should be launched by entering into the *Additional information* section for the relevant buttons:

```
0: Form, 1: Subform, 2: SubSubform, 3: Combobox or table control, 4: Foreign
key field in a form, empty for a table control, 5: Table name of auxiliary
table, 6: Table field1 of auxiliary table, 7: Table field2 of auxiliary
table, or 8: Table name of auxiliary table for table field2
```

The entries in this area are listed at the beginning of the macro as comments. The numbers bound to them are transferred and the relevant entry is read from an array. The macro can edit listboxes, which have two entries, separated by **">"**. These two entries can also come from different tables and be brought together using a query, as for instance in the Postcode table, which has only the foreign key field Town_ID for the town, requiring the Town table to display the names of towns.

```
DIM aForeignTable(0, 0 to 1)
DIM aForeignTable2(0, 0 to 1)
```

Among the variables defined at the beginning are two arrays. While normal arrays can be created by the **Split()** command during execution of the subroutine, two-dimensional arrays must be defined in advance. Two-dimensional arrays are necessary to store several records from one query when the query itself refers to more than one field. The two arrays declared above must be able to interpret queries that refer to two table fields. Therefore they are defined for two different contents by using *0 to 1* for the second dimension.

```
stTag = oEvent.Source.Model.Tag
aTable() = Split(stTag, ", ")
FOR i = LBound(aTable()) TO UBound(aTable())
   aTable(i) = trim(aTable(i))
NEXT
```

The variables provided are read. The sequence is that set up in the comment above. There is a maximum of nine entries, and you need to declare if an eighth entry for the table field2 and a nineth entry for a second table exist.

If values are to be removed from a table, it is first necessary to check that they do not exist as foreign keys in some other table. In simple table structures a given table will have only one foreign key connection to another table. However, in the given example database, there is a Town table which is used for both the place of publication of media and the town for addresses. Thus the primary key of the Town table is entered twice into different tables. These tables and foreign key names can naturally also be entered using the Additional Information field. It would be nicer though if they could be provided universally for all cases. This can be done using the following query.

```
   stSql = "SELECT ""FKTABLE_NAME"", ""FKCOLUMN_NAME"" FROM
""INFORMATION_SCHEMA"".""SYSTEM_CROSSREFERENCE"" WHERE ""PKTABLE_NAME"" = '"
+ aTable(5) + "'"
```

In the database, the INFORMATION_SCHEMA area contains all information about the tables of the database, including information about foreign keys. The tables that contain this information can be

accessed using "INFORMATION_SCHEMA"."SYSTEM_CROSSREFERENCE". KTABLE_NAME" gives the table that provides its primary key for the connection. FKTABLE_NAME gives the table that uses this primary key as a foreign key. Finally FKCOLUMN_NAME gives the name of the foreign key field.

The table that provides its primary key for use as a foreign key is in the previously created array at position 6. A the count begins with 0, the value is read from the array using **`aTable(5)`**.

```
inCount = 0
stForeignIDTab1Tab2 = "ID"
stForeignIDTab2Tab1 = "ID"
stAuxiltable = aTable(5)
```

Before the reading of the arrays begins, some default values must be set. These are the index for the array in which the values from the auxiliary table will be written, the default primary key if we do not need the foreign key for a second table, and the default auxiliary table, linked to the main table, for postcode and town, the Postcode table.

When two fields are linked for display in a listbox, they can, as described above, come from two different tables. For the display of Postcode and town the query is:

```
SELECT "Postcode"."Postcode" || ' > ' || "Town"."Town" FROM "Postcode", "Town" WHERE "Postcode"."Town_ID" = "Town"."ID"
```

The table for the first field (Postcode), is linked to the second table by a foreign key. Only the information from these two tables and the Postcode and Town fields is passed to the macro. All primary keys are by default called ID in the example database. The foreign key of Town in Postcode must therefore be determined using the macro.

In the same way the macro must access each table with which the content of the listbox is connected by a foreign key.

```
oQuery_result = oSQL_Statement.executeQuery(stSql)
IF NOT ISNULL(oQuery_result) THEN
   WHILE oQuery_result.next
      ReDim Preserve aForeignTable(inCount,0 to 1)
```

The array must be freshly dimensioned each time. In order to preserve the existing contents, they are backed up using (Preserve).

```
      aForeignTables(inCount,0) = oQuery_result.getString(1)
```

Reading the first field with the name of the table which contains the foreign key. The result for the Postcode table is the Address table.

```
      aForeignTables(inCount,1) = oQuery_result.getString(2)
```

Reading the second field with the name of the foreign key field. The result for the Postcode table is the field Postcode_ID in the Address table.

In cases where a call to the subroutine includes the name of a second table, the following loop is run. Only when the name of the second table occurs as the foreign key table for the first table is the default entry changed. In our case this does not occur, as the Town table has no foreign key from the Postcode table. The default entry for the auxiliary table therefore remains Postcode; finally the combination of postcode and town is a basis for the Address table, which contains a foreign key from the Postcode table.

```
      IF UBound(aTable()) = 8 THEN
         IF aTable(8) = aForeignTable(inCount,0) THEN
            stForeignIDTab2Tab1 = aForeignTable(inCount,1)
            stAuxiltable = aTable(8)
         END IF
      END IF
      inCount = inCount + 1
```

As further values may need to be read in, the index is incremented to redimension the arrays. Then the loop ends.

```
        WEND
    END IF
```

If, when the subroutine is called, a second table name exists, the same query is launched for this table:

```
    IF UBound(aTable()) = 8 THEN
```

It runs identically except that the loop tests whether perhaps the first table name occurs as a foreign key table name. That is the case here: the Postcode table *c*ontains the *f*oreign key Town_ID from the Town table. This foreign key is now assigned to the variable stForeignIDTab1Tab2, so that the relationship between the tables can be defined.

```
            IF aTable(5) = aForeignTable2(inCount,0) THEN
               stForeignIDTab1Tab2 = aForeignTable2(inCount,1)
            END IF
```

After a few further settings to ensure a return to the correct form after running the dialog (determining the line number of the form, so that we can jump back to that line number after a new read), the loop begins, which recreates the dialog when the first action is completed but the dialog is required to be kept open for further actions. The setting for repetition takes place using the corresponding checkbox

```
    DO
```

Before the dialog is launched, first of all the content of the listboxes is determined. Care must be taken if the listboxes represent two table fields and perhaps even are related to two different tables.

```
        IF UBound(aTable()) = 6 THEN
```

The listbox relates to only one table and one field, as the argument array ends at Tablefield1 of the auxiliary table.

```
            stSql = "SELECT """ + aTable(6) + """ FROM """ + aTable(5) + """
ORDER BY """ + aTable(6) + """"
        ELSEIF UBound(aTable()) = 7 THEN
```

The listbox relates to two table fields but only one table, as the argument array ends at Tablefield2 of the auxiliary table.

```
            stSql = "SELECT """ + aTable(6) + """||' > '||""" + aTable(7) + """
FROM """ + aTable(5) + """ ORDER BY """ + aTable(6) + """"
        ELSE
```

The listbox is based on two table fields from two tables. This query corresponds to the example with the postcode and the town.

```
            stSql = "SELECT """ + aTable(5) + """.""" + aTable(6) + """||' >
'||""" + aTable(8) + """.""" + aTable(7) + """ FROM """ + aTable(5) + """,
""" + aTable(8) + """ WHERE """ + aTable(8) + """.""" + stForeignIDTab2Tab1 +
""" = """ + aTable(5) + """.""" + stForeignIDTab1Tab2 + """ ORDER BY """ +
aTable(6) + """"
        END IF
```

Here we have the first evaluation to determine the foreign keys. The variables stForeignIDTab2Tab1 and stForeignIDTab1Tab2 start with the value ID. For stForeignIDTab1Tab2 evaluation of the previous query yields a different value, namely the value of Town_ID. In this way the previous query construction yields exactly the content already formulated for postcode and town – only enhanced by sorting.

Now we must make contact with the listboxes, to supply them with the content returned by the queries. These listboxes do not yet exist, since the dialog itself has not yet been created. This dialog is created first in memory, using the following lines, before it is actually drawn on the screen.

```
DialogLibraries.LoadLibrary("Standard")
oDlg = CreateUnoDialog(DialogLibraries.Standard.Dialog_Table_purge)
```

Next come the settings for the fields of the dialog. Here, for example, is the listbox which is to be supplied with the results of the above query:

```
oCtlList1 = oDlg.GetControl("ListBox1")
oCtlList1.addItems(aContent(),0)
```

Access to the fields of the dialog is accomplished by using **`GetControl`** with the appropriate name. In dialogs it is not possible for two fields to use the same name as this would create problems when evaluating the dialog.

The listbox is supplied with the contents of the query, which have been stored in the array aContent() . The listbox contains only the content to be displayed as a field, so only the position 0 is filled.

After all fields with the desired content have been filled, the dialog is launched.

```
   Select Case oDlg.Execute()
   Case 1'Case 1 means the "OK" button has been clicked
   Case 0'If it was the "Cancel" button
      inRepetition = 0
   End Select
LOOP WHILE inRepetition = 1
```

The dialog runs repeatedly as long as the value of "inRepetition" is 1. This is set by the corresponding checkbox.

Here, in brief, is the content after the "OK" button is clicked:

```
Case 1
   stInhalt1 = oCtlList1.getSelectedItem() 'Read value from Listbox1 ...
   REM ... and determine the corresponding ID-value.
```

The ID value of the first listbox is stored in the variable "inLB1".

```
stText = oCtlText.Text  ' Read the field value.
```

If the text field is not empty, the entry in the text field is handled. Neither the listbox for a replacement value nor the checkbox for deleting all orphaned records are considered. This is made clear by the fact that text entry sets these other fields to be inactive.

```
IF stText <> "" THEN
```

If the text field is not empty, the new value is written in place of the old one using the previously read ID field in the table. There is the possibility of two entries, as is also the case in the listbox. The separator is **>**. For two entries in different tables, two UPDATE-commands must be launched, which are created here simultaneously and forwarded, separated by a semicolon.

```
ELSEIF oCtlList2.getSelectedItem() <> "" THEN
```

If the text field is empty and the listbox 2 contains a value, the value from listbox 1 must be replaced by the value in listbox 2. This means that all records in the tables for which the records in the listboxes are foreign keys must be checked and, if necessary, written with an altered foreign key.

```
stInhalt2 = oCtlList2.getSelectedItem()
REM Read value from listbox.
REM Determine ID for the value of the listbox.
```

The ID value of the second listbox is stored in the variable inLB2. Here too, things develop differently depending on whether one or two fields are contained in the listbox, and also on whether one or two tables are the basis of the listbox content.

The replacement process depends on which table is defined as the table which supplies the foreign key for the main table. For the obove example, this is the Postcode table, as the Postcode_ID is the foreign key which is forwarded through Listbox 1 and Listbox 2.

```
      IF stAuxilTable = aTable(5) THEN
         FOR i = LBound(aForeignTables()) TO UBound(aForeignTables())
```

Replacing the old ID value by the new ID value becomes problematic in n:m-relationships, as in such cases, the same value can be assigned twice. That might be what you want, but it must be prevented when the foreign key forms part of the primary key. So in the table rel_Media_Author a medium cannot have the same author twice because the primary key is constructed from Media_ID and Author_ID. In the query, all key fields are searched which collectively have the property UNIQUE or were defined as foreign keys with the UNIQUE property using an index.

So if the foreign key has the UNIQUE property and is already represented there with the desired future inLB2, that key cannot be replaced.

```
   stSql = "SELECT ""COLUMN_NAME"" FROM
   ""INFORMATION_SCHEMA"".""SYSTEM_INDEXINFO"" WHERE ""TABLE_NAME"" = '" +
   aForeignTables(i,0) + "' AND ""NON_UNIQUE"" = False AND ""INDEX_NAME"" =
   (SELECT ""INDEX_NAME"" FROM ""INFORMATION_SCHEMA"".""SYSTEM_INDEXINFO"" WHERE
   ""TABLE_NAME"" = '" + aForeignTables(i,0) + "' AND ""COLUMN_NAME"" = '" +
   aForeignTables(i,1) + "')"
```

' "NON_UNIQUE" = False ' gives the names of columns that are UNIQUE. However not all column names are needed but only those which form an index with the foreign key field. This is handled by the Subselect with the same table names (which contain the foreign key) and the names of the foreign key fields.

If now the foreign key is present in the set, the key value can only be replaced if other fields are used to define the corresponding index as UNIQUE. You must take care when carrying out replacements that the uniqueness of the index combination is not compromised.

```
      IF aForeignTables(i,1) = stFieldname THEN
         inUnique = 1
      ELSE
         ReDim Preserve aColumns(inCount)
         aColumns(inCount) = oQuery_result.getString(1)
         inCount = inCount + 1
      END IF
```

All column names, apart from the known column names for foreign key fields as Index with the UNIQUE property, are stored in the array. As the column name of the foreign key field also belongs to the group, it can be used to determine whether uniqueness is to be checked during data modification.

```
   IF inUnique = 1 THEN
      stSql = "UPDATE """ + aForeignTables(i,0) + """ AS ""a"" SET """ +
   aForeignTables(i,1) + """='" + inLB2 + "' WHERE """ + aForeignTables(i,1) +
   """='" + inLB1 + "' AND ( SELECT COUNT(*) FROM """ + aForeignTables(i,0) +
   """ WHERE """ + aForeignTables(i,1) + """='" + inLB2 + "' )"
      IF inCount > 0 THEN
         stFieldgroup = Join(aColumns(), """||  ||""")
```

If there are several fields, apart from the foreign key field, which together form a 'UNIQUE' index, they are combined here for a SQL grouping. Otherwise only "aColumns(0)" appears as "stFieldgroup".

```
         stFieldname = ""
         FOR ink = LBound(aColumns()) TO UBound(aColumns())
            stFieldname = stFieldname + " AND """ + aColumns(ink) + """ =
   ""a"".""" + aColumns(ink) + """ "
```

The SQL parts are combined for a correlated subquery.

```
         NEXT
      stSql = Left(stSql, Len(stSql) - 1)
```

The previous query ends with a bracket. Now further content is to be added to the subquery, so this closure must be removed again. After that, the query is expanded with the additional conditions.

```
      stSql = stSql + stFeldbezeichnung + "GROUP BY (""" + stFeldgruppe + """) ) < 1"
   END IF
```

If the foreign key has no connection with the primary key or with a UNIQUE index, it does not matter if content is duplicated.

```
   ELSE
      stSql = "UPDATE """ + aForeignTables(i,0) + """ SET """ +
   aForeignTables(i,1) + """='" + inLB2 + "' WHERE """ + aForeignTables(i,1) +
   """='" + inLB1 + "'"
   END IF
   oSQL_Statement.executeQuery(stSql)
   NEXT
```

The update is carried out for as long as different connections to other tables occur; that is, as long as the current table is the source of a foreign key in another table. This is the case twice over for the Town table: in the Media table and in the Postcode table.

Afterwards the old value can be deleted from listbox 1, as it no longer has any connection to other tables.

```
   stSql = "DELETE FROM """ + aTable(5) + """ WHERE ""ID""='" + inLB1 + "'"
   oSQL_Statement.executeQuery(stSql)
```

In some cases, the same method must now be carried out for a second table that has supplied data for the listboxes. In our example, the first table is the Postcode table and the second is the Town table.

If the text field is empty and listbox 2 also contains nothing, we check if the relevant checkbox indicates that all surplus entries are to be deleted. This means the entries which are not bound to other tables by a foreign key.

```
   ELSEIF oCtlCheck1.State = 1 THEN
      stCondition = ""
      IF stAuxilTable = aTable(5) THEN
         FOR i = LBound(aForeignTables()) TO UBound(aForeignTables())
            stCondition = stCondition + """ID"" NOT IN (SELECT """ +
   aForeignTables(i,1) + """ FROM """ + aForeignTables(i,0) + """) AND "
         NEXT
      ELSE
         FOR i = LBound(aForeignTables2()) TO UBound(aForeignTables2())
            stCondition = stCondition + """ID"" NOT IN (SELECT """ +
   aForeignTables2(i,1) + """ FROM """ + aForeignTables2(i,0) + """) AND "
         NEXT
      END IF
```

The last AND must be removed, since otherwise the delete instruction would end with AND.

```
      stCondition = Left(stCondition, Len(stCondition) - 4) '
      stSql = "DELETE FROM """ + stAuxilTable + """ WHERE " + stCondition + ""
      oSQL_Statement.executeQuery(stSql)
```

As the table has already been purged once, the table index can be checked and optionally corrected downwards. See the subroutine described in one of the previous sections.

```
   Table_index_down(stAuxilTable)
```

Afterwards, if necessary the listbox in the form from which the Table_purge dialog was called can be updated. In some cases, the whole form needs to be reread. For this purpose, the current record is determined at the beginning of the subroutine so that after the form has been refreshed, the current record can be reinstated.

```
      oDlg.endExecute() 'End dialog ...
```

```
    oDlg.Dispose() '... and remove from storage
END SUB
```

Dialogs are terminated with the endExecute() command and completely removed from memory with Dispose().