Apache OpenOffice

Version 4.1

Getting Started Guide

AOO Documentation Team

Copyright

Acknowledgments

This book is updated from *Taming Apache OpenOffice 3.4* (by Jean Hollis Weber), which was adapted from *Getting Started with OpenOffice.org 3.3* (by the OOoAuthors team) with additional material adapted from *Getting Started with LibreOffice 3.4* (by the ODF Authors Team).

Publication date and software version

Published July 14, 2026. Based on Apache OpenOffice 4.1.

Contents

Chapter 12
Getting Started with Macros

Introduction to Macros

An OpenOffice macro, like any macro, is stored computer code. In this case, the code is in an OpenOffice document or in some other location accessible to OpenOffice. The macro runs when triggered by OpenOffice. What the code can do is limited only by the capabilities of the programming language and the macro's author.

Another, perhaps more practical, description of an Open Office macro is that it is computer code used to automate changes to an OpenOffice document. Since making and testing even a simple a macro takes a few minutes, macros are typically efficient tools when they automate a task that will be done many times, especially if that task cannot be done easily with the many powerful OpenOffice tools.

There are two methods for interacting with OpenOffice documents from a macro: The Dispatch Framework (see page 15) and the Application Programming Interface (API). These two methods are usually used in recorded macros and manually written macros, respectively. They result in code that looks quite different.

OpenOffice macros are usually written in a language called OpenOffice Basic, or StarBasic, or just abbreviated Basic (Beginner's All-purpose Symbolic Instruction Code). Although you can learn Basic and the API and write macros, there is a steep learning curve to writing macros from scratch. The usual methods for a beginner are to use macros that someone else has written and to use the built-in macro recorder, which records your keystrokes and saves them for use.

OpenOffice Basic is an interpreted, domain-specific programming language. That means that, rather than have to be compiled or translated to the computer's native language of binary, an interpreted language is carried out one line at a time, from the original source file. OpenOffice Basic is integrated into the Apache OpenOffice suite (and derivatives like LibreOffice) and is used to automate tasks, create macros, and interact with the OpenOffice API. It is similar to VBA (Visual Basic for Applications), allows for GUI dialog creation, and is available in Writer, Calc,Impress, Draw, and Base.

- **Purpose:** Primarily used for automating repetitive tasks, creating custom functions, and developing complex applications within the OpenOffice suite.
- **Integrated IDE:** It includes a built-in Integrated Development Environment with an editor, debugger, and dialog editor.
- **UNO Integration:** The language utilizes the Universal Network Objects (UNO) component model to access API features and interact with documents.
- **Syntax:** It is a dialect of the BASIC programming language, sharing similarities with Microsoft Visual Basic.
- **Scope:** It is used to interact with all the components of the suite, including Writer (word processor), Calc (spreadsheets), and Base (databases).

For those looking to learn more, the Help documents included with OpenOffice include a section on Basic.

Your First Macros

Using a Copied Macro

The first step in learning macro programming is to find and use existing macros. Some online sources of macros are listed at the end of this chapter in Online Resources. For now, use the macro in Listing 1.

Listing 1: Simple macro that says hello.

```
Sub HelloMacro
  Print "Hello"
End Sub
```

In Basic, `Sub` and `End Sub` are keywords used to define the beginning and end of a **Subroutine** (or "Sub procedure"), which is a block of code that performs actions but does not send a value back to the initial routine.
You must create a library and module to contain your macro; this is covered in “Macro Organization” on page 16. Use these steps to create a library to contain your macro:

1) Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the Macro dialog (see Figure 1 and Figure 6).
2) Click **Organizer** to open the Basic Macro Organizer dialog (see Figure 8).
3) Select the *Libraries* tab.
4) Set the *Location* to *My Macros & Dialogs*, which is the default.
5) Click **New** to open the New Library dialog. Enter a library name such as “TestLibrary” and click **OK**.
6) Select the *Modules* tab.

7) In the *Module* list, expand *My Macros* and select TestLibrary. A module named Module1 already exists and can contain your macro. You can click **New** to create another module in TestLibrary.
8) Select the Module1, or the new module that you created, and click **Edit** to open the Integrated Development Environment (IDE).

The IDE is a text editor for macros that allows you to edit and create macros. Copy the macro into the IDE.

When a new module is created, it contains a comment and an empty macro named Main, that does nothing.

Listing 2: Contents of Module1 after it is created.

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

Sub Main

End Sub
```

Add the new macro either before Sub Main or after End Sub. In Listing 3, the new macro has been added before Sub Main. You can also delete Sub Main and its corresponding End Sub.

Listing 3: Module1 after adding the new macro.

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

Sub HelloMacro
  Print "Hello"
End Sub

Sub Main

End Sub
```

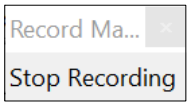
Click on the Run Basic button  in the toolbar to run the first macro in the module. Use the Macro dialog, opened using the Select macro button  or **Tools > Macros > Organize Macros > OpenOffice Basic**, to select and run any macro in the module.

Recording a Simple Macro

Imagine repeatedly entering simple information. While you can store the information in the clipboard, if you use the clipboard for something else, its contents are changed. Storing such contents as a macro is a simple solution. (In some simple cases, including the example used here, a better solution is to use AutoText.)

1) Use **Tools > Macros > Record Macro** to start recording a macro.

   A small window is displayed so you know that OpenOffice is recording.



2) Type the desired information or perform an appropriate series of operations. In this case, we typed the name *Andrew Pitonyak*.
3) Click the **Stop Recording** button to stop recording, save the macro, and display the OpenOffice Basic Macros dialog shown in Figure 1.

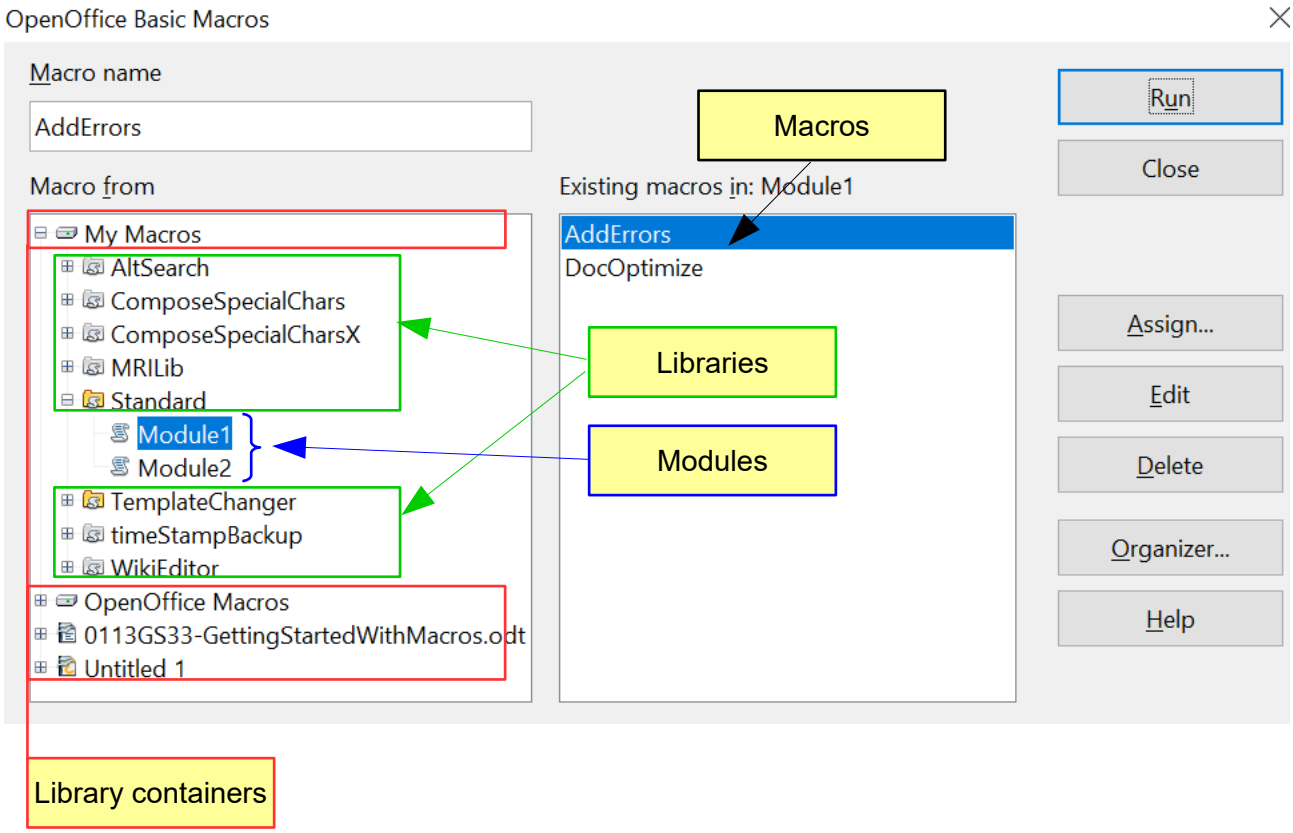


Figure 1: Macro Organizer dialog, Standard library selected

Be certain to open the library container named *My Macros*. Find the library named *Standard* under My Macros.

Caution

Be forewarned; *every* library container has a library named Standard. Select the Standard library and click **New Module** to create a new module to contain the macro.

4) The default module name is Module1. Type a descriptive name and click **OK** to create the module. The OpenOffice Basic Macros dialog is displayed again, showing the new module.

Figure 2: Give your module a meaningful name

5) Highlight the newly created module. In the upper left corner, type the macro name to use, such as “EnterMyname”, and then click **Save** to save the macro.

If you followed all of the steps, the Standard library now contains a module named Recorded, which contains the EnterMyName macro, as shown in Figure 3. When OpenOffice creates a new module, it automatically adds the macro named Main.

Running the Macro

Use **Tools > Macros > Run Macro** to open the Macro Selector dialog. Select the newly created macro and click **Run**.

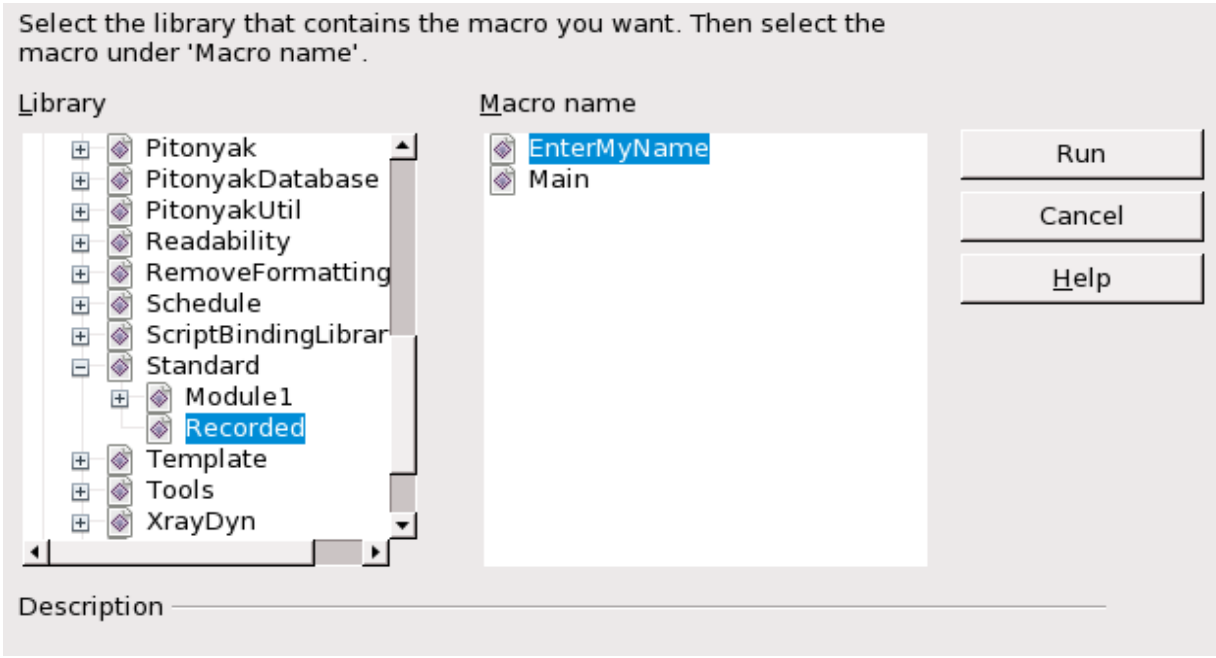


Figure 3: Select your macro and click Run

There are other methods to run a macro. For example, use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the macro organizer, which contains a **Run** button as well.

Viewing and Editing the Macro

You can view and edit the macro that was just created. Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the OpenOffice Basic Macros dialog (see Figure 1). Select the new macro and click **Edit** to open the macro in the Basic IDE (Integrated Development Environment).

Listing 4: Generated "EnterMyname" macro.

```
REM  *****  BASIC  *****
Sub Main

End Sub

sub EnterMyName
rem ----------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem ----------------------------------------------------------------
dim args1(0) as new com.sun.star.beans.PropertyValue
```

```
args1(0).Name = "Text"
args1(0).Value = "Andrew Pitonyak"

dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args1())
end sub
```

Explanation of the Recorded Macro

The macro in Listing 4 is not as complicated as it first appears. Learning a few things helps significantly in understanding generated macros. The discussion starts with features near the top of the macro listing and describes them. If you like, simply change the text “Andrew Pitonyak” in the macro above to what you want to insert at the current cursor position.

Comments Start with REM

The keyword REM, short for *remark*, starts a macro comment. All text after REM on the same line is ignored when the macro runs. As a short cut, the single quote character can also be used to start a comment. Comments explain what the code is doing.

Tip	OpenOffice Basic is not case-sensitive for keywords, so REM, Rem, and rem all start a comment. If you use symbolic constants as defined by the API, it is safer to assume that the names are case-sensitive. Symbolic constants are an advanced topic not usually needed by people that use the macro recorder. Note that symbolic constants are simply names given to values that won't change, such as pi.

Defining Subroutines with SUB

Individual macros are stored in blocks of code called subroutines, defined with the keyword SUB. The end of a subroutine is indicated by the words END SUB. The sample code we're using starts by defining the subroutine named Main, which is empty and does nothing. The next subroutine, EnterMyName, contains the generated code.

Tip	OpenOffice always creates an empty subroutine named Main when it creates a module. You can delete it, but be sure to delete both the *Sub Main* and the *End Sub.*

Defining Variables Using DIM

The `Dim` (short for Dimension) statement in BASIC is used to declare variables and allocate computer storage space. It explicitly defines a variable's name and data type, usually at the beginning of a program. The data types of variables are categories like integer, string (that is, text), or objects. They are discussed further in the section Variables.

Pulling the Macro Together

The following explanation has a lot of details; It isn't necessary to understand them to use the macro, but understanding the code will allow you to modify macros for a new purpose, or adjust them if they do not do exactly what you want.

The first line defines the macro's start.

```
sub EnterMyName
```

The next lines declare two variables of the object type:

```
dim document   as object
dim dispatcher as object
```

Objects in macros can contain either simple values or complex sub-objects, both of which are referred to as *properties* of the object. A property of an object can be referred to by giving the name of the parent object, followed by a period, and then the name of the sub-object. For example, if an object variable named Doc refers to a Writer document, the Text sub-object is Doc.Text. Similarly, an object may be able to take actions using a *method*, a function associated with the object. A method is called using the name of the parent object, a period, and then the name of the method.

The next line of code assigns an object called *Frame* to the *document* variable.

```
document = ThisComponent.CurrentController.Frame
```

ThisComponent is a special term in OpenOffice Basic that refers to the current document. Like the variables *document* and *dispatcher*, it is an object. The above line of code refers to ThisComponent, its sub-object CurrentController, and the Frame sub-object of the CurrentController.

The CurrentController property of a document refers to a service that “controls” the document. For example, when you type, it is the current controller that notices. The current controller then dispatches any changes or other edits to the document’s frame.

The code now assigns an object to the *dispatcher* variable.

```
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")
```

Most tasks in OpenOffice are accomplished by dispatching (that is, sending) a command. OpenOffice includes a dispatch helper service, which does most of the work of using dispatches in macros. The method createUnoService accepts the name of a service, and then tries to create an instance (that is, an occurrence) of that service.

The next line of code

```
dim args1(0) as new com.sun.star.beans.PropertyValue
```

declares an array variable.

The section Common DIM Examples describes arrays in detail.

So, for the moment, think of *args1(0)* as simply a variable's name. The type of the variable is *com.star.beans.PropertyValue*. That is a kind of object that has two properties, a *Name* and a *Value.*

Tip	An array is a kind of aggregate data type. A single ordinary variable (a "scalar") could be considered as a zero-dimensional array. A one-dimensional array is also known as a "vector" Elements of an array are usually stored contiguously. Arrays are appropriate for storing data which are logically related. The values might form a series that will be used sequentially or they might be used all at once to set the values of several properties of an object.

In the code below, we've given the property the name “Text” and the value “Andrew Pitonyak”. That latter is the text that is inserted when the macro is run.

```
args1(0).Name = "Text"
args1(0).Value = "Andrew Pitonyak"
```

The last line of the code performs the text insertion. The *dispatcher* object is told to act on the object stored in the *document* variable using the *uno:InsertText* action according to the PropertyValue stored in the args1 array.

```
dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args1())
```

Finally, the end of the subroutine.

```
end sub
```

A More Complicated Example

When creating a macro, it is important to ask two questions before recording:

1) Can the task be written as a simple set of commands?
2) If the macro will be run repeatedly during each use, can the steps be arranged so that the last command leaves the cursor ready for the next command?

These questions are best answered if you're familiar with the many keystrokes that aid movement in a document. For example, a Writer macro will be more flexible and shorter if you move to the end of the line by pressing the *End* key once rather than pressing the right arrow key many times.

We'll work with an example of copying rows and columns of data from a web site and formatting them as a table in a text document. First, we copy the table from the web site to the clipboard. To avoid strange formatting and fonts, paste the text into a Writer document as unformatted text. We reformat the text with tabs between columns so that we can use **Table > Convert > Text to Table** to convert to a table.

We now inspect the text to see if we can record a macro to format the text (remember the two questions that above). As an example, we copied the FontWeight constants group from the OpenOffice web site. The first column indicates the constant name. Each name is followed by a space and a tab.

DONTKNOW	The font weight is not specified/known.
THIN	specifies a 50% font weight.
ULTRALIGHT	specifies a 60% font weight.
LIGHT	specifies a 75% font weight.
SEMILIGHT	specifies a 90% font weight.
NORMAL	specifies a normal font weight.
SEMIBOLD	specifies a 110% font weight.
BOLD	specifies a 150% font weight.
ULTRABOLD	specifies a 175% font weight.
BLACK	specifies a 200% font weight.

We want the first column to contain the numeric value, the second column the name, and the third column the description. The desired work is easily accomplished for every row except for DONTKNOW and NORMAL, which do not contain a numeric value—but we know that the values are 0 and 100, so we will enter those manually.

The data can be cleaned in multiple ways—all of them easy. This example uses keystrokes that assume the cursor is at the start of the line with the text THIN. Two of the keystrokes used in the macro may be unfamiliar to many users. *Ctrl+Right Arrow* moves the cursor from its current position to the start of the next word. *Ctrl+Shift+Right Arrow* makes the same movement and also selects the intervening text.

1) Use **Tools > Macros > Record Macro** to start recording.
2) Press *Ctrl+Right Arrow* to move the cursor to the start of “specifies”.
3) Press *Backspace* twice to remove the tab and the space.
4) Press *Tab* to add the tab without the space after the constant name.
5) Press *Delete* to delete the lower case s and then press *S* to add an upper case S.
6) Press *Ctrl+Right Arrow* twice to move the cursor to the start of the number.
7) Press *Ctrl+Shift+Right Arrow* to select and move the cursor before the % sign.
8) Press *Ctrl+C* to copy the selected text to the clipboard.
9) Press *End* to move the cursor to the end of the line.
10) Press *Backspace* twice to remove the two trailing spaces.
11) Press *Home* to move the cursor to the start of the line.
12) Press *Ctrl+V* to paste the selected number to the start of the line.
13) Pasting the value also pasted an extra space, so press *Backspace* to remove the extra space.
14) Press *Tab* to insert a tab between the number and the name.
15) Press *Home* to move to the start of the line.
16) Press *down arrow* to move to the next line.
17) Stop recording the macro and save the macro.

It takes much longer to read and write the steps than to record the macro. Work slowly and think about the steps as you do them. After a bit, this will become instinctive.

The generated macro has been modified to contain the step number in the comments to match the code to the step above. The use of the keyword *rem* in the code below gives you a closer look at the relationship between the steps that will go into the macro, how they'll affect it, and how it will function.

Listing 5: Copy the numeric value to the start of the column.

```
sub CopyNumToCol1
rem ----------------------------------------------------------------
rem define variables
dim document   as object
dim dispatcher as object
rem ----------------------------------------------------------------
rem get access to the document
document   = ThisComponent.CurrentController.Frame
dispatcher = createUnoService("com.sun.star.frame.DispatchHelper")

rem (2) Press Ctrl+Right Arrow to move the cursor to the start of
“specifies”.
dispatcher.executeDispatch(document, ".uno:GoToNextWord", "", 0, Array())

rem (3) Press Backspace twice to remove the tab and the space.
dispatcher.executeDispatch(document, ".uno:SwBackspace", "", 0, Array())

rem ----------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:SwBackspace", "", 0, Array())

rem (4) Press Tab to add the tab without the space after the constant name.
dim args4(0) as new com.sun.star.beans.PropertyValue
args4(0).Name = "Text"
args4(0).Value = CHR$(9)

dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args4())

rem (5) Press Delete to delete the lower case s ....
dispatcher.executeDispatch(document, ".uno:Delete", "", 0, Array())

rem (5) ... and then press S to add an upper case S.
dim args6(0) as new com.sun.star.beans.PropertyValue
args6(0).Name = "Text"
args6(0).Value = "S"

dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args6())

rem (6) Press Ctrl+Right Arrow twice to move the cursor to the number.
dispatcher.executeDispatch(document, ".uno:GoToNextWord", "", 0, Array())

rem ----------------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:GoToNextWord", "", 0, Array())

rem (7) Press Ctrl+Shift+Right Arrow to select the number.
dispatcher.executeDispatch(document, ".uno:WordRightSel", "", 0, Array())

rem (8) Press Ctrl+C to copy the selected text to the clipboard.
dispatcher.executeDispatch(document, ".uno:Copy", "", 0, Array())
```

```
rem (9) Press End to move the cursor to the end of the line.
dispatcher.executeDispatch(document, ".uno:GoToEndOfLine", "", 0, Array())

rem (10) Press Backspace twice to remove the two trailing spaces.
dispatcher.executeDispatch(document, ".uno:SwBackspace", "", 0, Array())

rem ----------------------------------------------------------
dispatcher.executeDispatch(document, ".uno:SwBackspace", "", 0, Array())

rem (11) Press Home to move the cursor to the start of the line.
dispatcher.executeDispatch(document, ".uno:GoToStartOfLine", "", 0, Array())

rem (12) Press Ctrl+V to paste the selected number to the start of the line.
dispatcher.executeDispatch(document, ".uno:Paste", "", 0, Array())

rem (13) Press Backspace to remove the extra space.
dispatcher.executeDispatch(document, ".uno:SwBackspace", "", 0, Array())

rem (14) Press Tab to insert a tab between the number and the name.
dim args17(0) as new com.sun.star.beans.PropertyValue
args17(0).Name = "Text"
args17(0).Value = CHR$(9)

dispatcher.executeDispatch(document, ".uno:InsertText", "", 0, args17())

rem (15) Press Home to move to the start of the line.
dispatcher.executeDispatch(document, ".uno:GoToStartOfLine", "", 0, Array())

rem (16) Press down arrow to move to the next line.
dim args19(1) as new com.sun.star.beans.PropertyValue
args19(0).Name = "Count"
args19(0).Value = 1
args19(1).Name = "Select"
args19(1).Value = false

dispatcher.executeDispatch(document, ".uno:GoDown", "", 0, args19())
end sub
```

Running the Macro Quickly

It is tedious to repeatedly run the macro using **Tools > Macros > Run Macro** (see Figure 3). The macro can be run from the IDE. Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the Basic Macro dialog. Select your macro and click **Edit** to open the macro in the IDE.

Tip	An Integrated Development Environment (pronounced *I-D-E* or *eye-dee-ee*) is software that provides a comprehensive, centralized workspace for developers to write, test, and debug code efficiently. It combines essential tools into a single graphical user interface.

The IDE has a **Run Basic** icon in the toolbar that runs the first macro in the IDE. Unless you change the first macro, it is the empty macro named Main. Modify Main so that it reads as shown in Listing 6.

Listing 6: Modify Main to call CopyNumToCol1.

```
Sub Main
  CopyNumToCol1
End Sub
```

Because it is referenced within the ”Main”module, CopyNumToCol1 can be run simply by repeatedly clicking the **Run Basic** icon in the toolbar of the IDE. Such a technique works especially well for temporary macros that will be used a few times and then discarded.

Sometimes the Macro Recorder Fails

Understanding what goes on under the hood of OpenOffice helps to clarify how and why the macro recorder can fail. The main reason is the relationship between the dispatch framework and the macro recorder.

The Dispatch Framework

The purpose of the dispatch framework is to provide uniform access to documents for commands that correspond to menu items. We can save a document using **File > Save** from the menu, the shortcut keys *Ctrl+S*, or click on the **Save** toolbar icon. All of these commands are translated into the same “dispatch command”, which is sent to the current document.

The dispatch framework can also be used to send “commands” back to us, through the User Interface. For example, after saving the document, the File Save command is disabled. As soon as the document has been changed, the File Save command is enabled.

How the Macro Recorder Uses the Dispatch Framework

The macro recorder is relatively simple to implement, and logs commands so that they can be available for later use. The problem is that not all dispatched commands are complete. For example, inserting an object generates the following code:

```
dispatcher.executeDispatch(document, ".uno:InsertObject", "", 0, Array())
```

It is not possible to specify what kind of object to create or insert. If an object is inserted from a file, you cannot specify from which file the insertion should be drawn .

If you record a macro and use **Tools > Options** to open and modify configuration items, the generated macro does not record any configuration changes; in fact, the generated code is commented so it will not even be run.

```
rem dispatcher.executeDispatch(document,
    ".uno:OptionsTreeDialog", "", 0, Array())
```

Caution	If a dialog is opened, a command to open the dialog is likely to be generated. Any work done inside the dialog is not usually recorded. Examples include macro organization dialogs, inserting special characters, and similar types of dialogs. Other possible problems using the macro recorder include things such as inserting a formula, setting user data, setting filters in Calc, actions in database forms, and exporting a document to an encrypted PDF file. You never know for certain what will work unless you try it, However. actions from the search dialog are usually properly captured.

Other Options

When the macro recorder is not able to solve a specific problem, the usual solution is to write code using OpenOffice Application Programming. Unfortunately, there is a steep learning curve for the API. It is usually best to start with simple examples and then branch out slowly as you learn more.

Macro Organization

In OpenOffice, macros are grouped in modules, modules are grouped in libraries, and libraries are grouped in library containers. A library is usually used as a major grouping for either an entire category of macros, or for an entire application. Modules usually split functionality, such as user interaction and calculations. Individual macros are subroutines and functions.

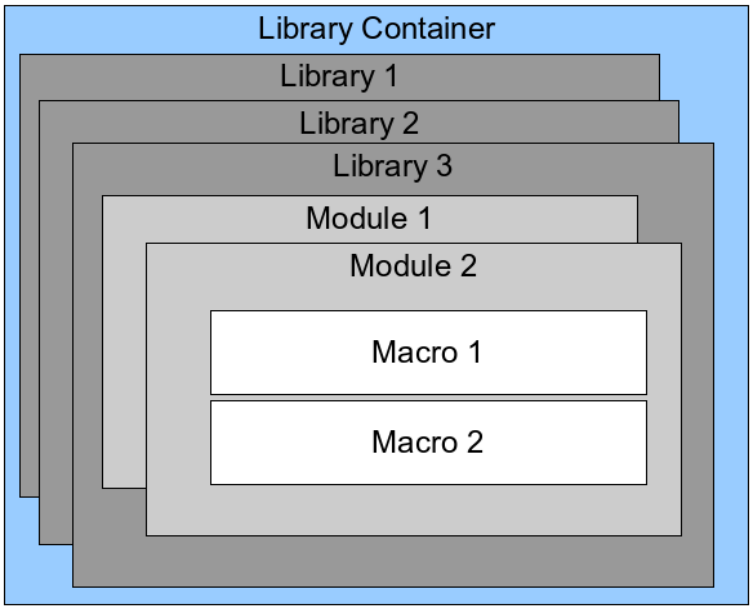


Figure 4: Macro Library hierarchy

A computer scientist could use Figure 5 to describe the situation. The text “1..*” means one or more, and “0..*” means zero or more. The black triangle means composed of or contains.

- A library container contains one or more libraries, and each library is contained in one library container.
- A library contains zero or more modules, and each module is contained in one library.

- A module contains zero or more macros, and each macro is contained in one module.

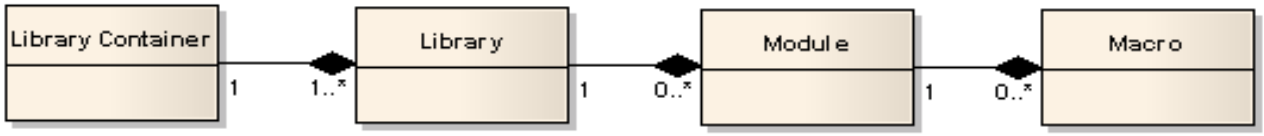


Figure 5: Macro Library hierarchy

Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the OpenOffice Basic Macros dialog (see Figure 6). All available library containers are shown in the *Macro from* list. Every document is a library container, capable of containing multiple libraries. The application itself acts as two library containers, one container for macros distributed with OpenOffice called OpenOffice Macros, and one container for personal macros called My Macros. As shown in Figure 6, only two documents are currently open.

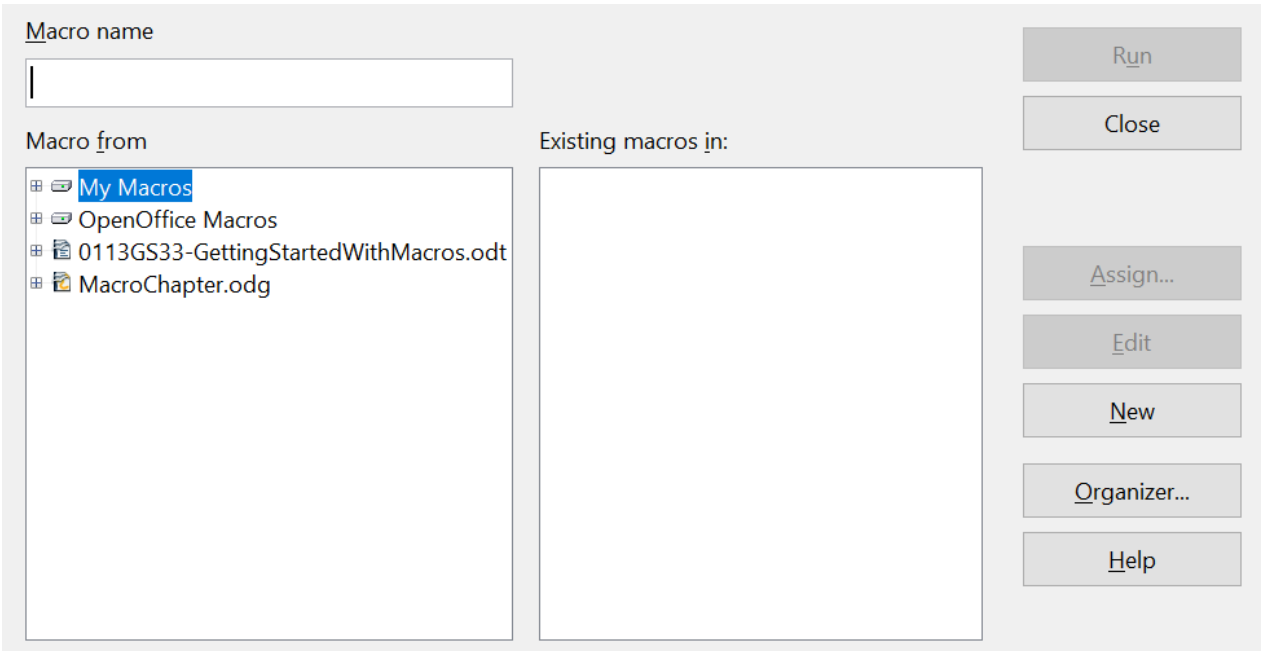


Figure 6: Library containers are shown on the left

The macros in the library named OpenOffice Macros are stored with code that executes at runtime. As such, it may not be editable, unless you are an administrator. But that's,ok; these macros should not be changed. Neither should you store your own macros in the OpenOffice Macros container.

Unless your macros are applicable to a single document, and only to a single document, your macros will probably be stored in the My Macros container. The My Macros container is stored in your user profile.

Every library container contains a library named *Standard*. It is better to create your own libraries with meaningful names than to use the Standard library. Not only are meaningful names easier to manage, but they can also be imported into other library containers. The Standard library cannot.

Caution

OpenOffice allows you to import libraries into a library container, but it will not allow you to overwrite the library named Standard. Therefore, if you store your macros in the Standard library, you cannot import them into another library container.

Just as it makes good sense to give your libraries meaningful names, it is prudent to use meaningful names for your modules. By default, OpenOffice uses names such as Module1. It will make them more recognizable and accessible if you use your own meaningful names for both modules and libraries.

As you create your macros, you must decide where to store them. Storing a macro in a document is useful if the document will be shared and you want the macro to be included with the document. Macros stored in the application library container named My Macros, however, are globally available to all documents.

Macros are not available until the library that contains them is loaded. The Standard library and Template library, however, are automatically loaded. A loaded library is displayed differently from a library that is not loaded. To load the library and the modules it contains, double-click on the library. Figure 7 shows three libraries in the My Macros library container. The MRLib and Standard libraries have been loaded and the TemplateChanger library has not been loaded.

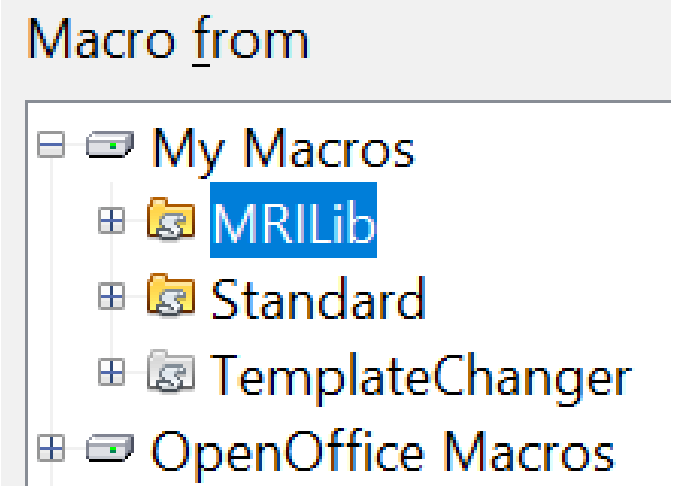


Figure 7: Two loaded and one unloaded libraries.

Where are Macros Stored?

OpenOffice stores user-specific data in a directory under the user's home directory. The location is operating system specific. Use **Tools > Options > OpenOffice > Paths** to view where other configuration data is stored. On Windows, this is `C:\Users\<username>\AppData\Roaming\OpenOffice\4\user`. User macros are stored in the *basic* directory of that folder. Each library is stored in its own directory under the *basic* directory.

It is not important to understand where macros are stored for casual use. If you know where they are stored, however, you can create a backup, share your macros, or inspect them if there is an error.

Use **Tools > Macros > Organize Dialogs** to open the OpenOffice Macro Organizer dialog shown in Figure 8. Another common way to open this dialog is to use **Tools >**

Macros > Organize Macros > OpenOffice Basic to open the OpenOffice Macros dialog and then click the **Organizer** button.

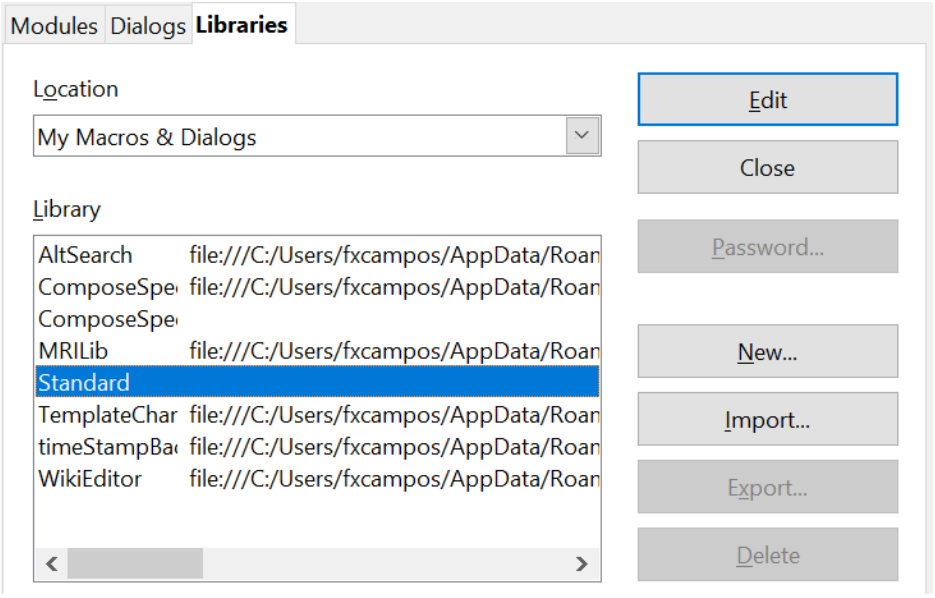


Figure 8: The macro organizer dialog

Importing Macros

The OpenOffice Macro Organizer dialog provides the ability to import, create, delete, and rename libraries, modules, and dialogs. Select the library container to use and then click the **Import** button to import macro libraries (see Figure 9).

Tip	You cannot import the library named Standard.

Tip	The folders containing macros are normally set to be hidden. You may have to type the full file path in the *File name* box rather than use the usual graphical navigation icons.

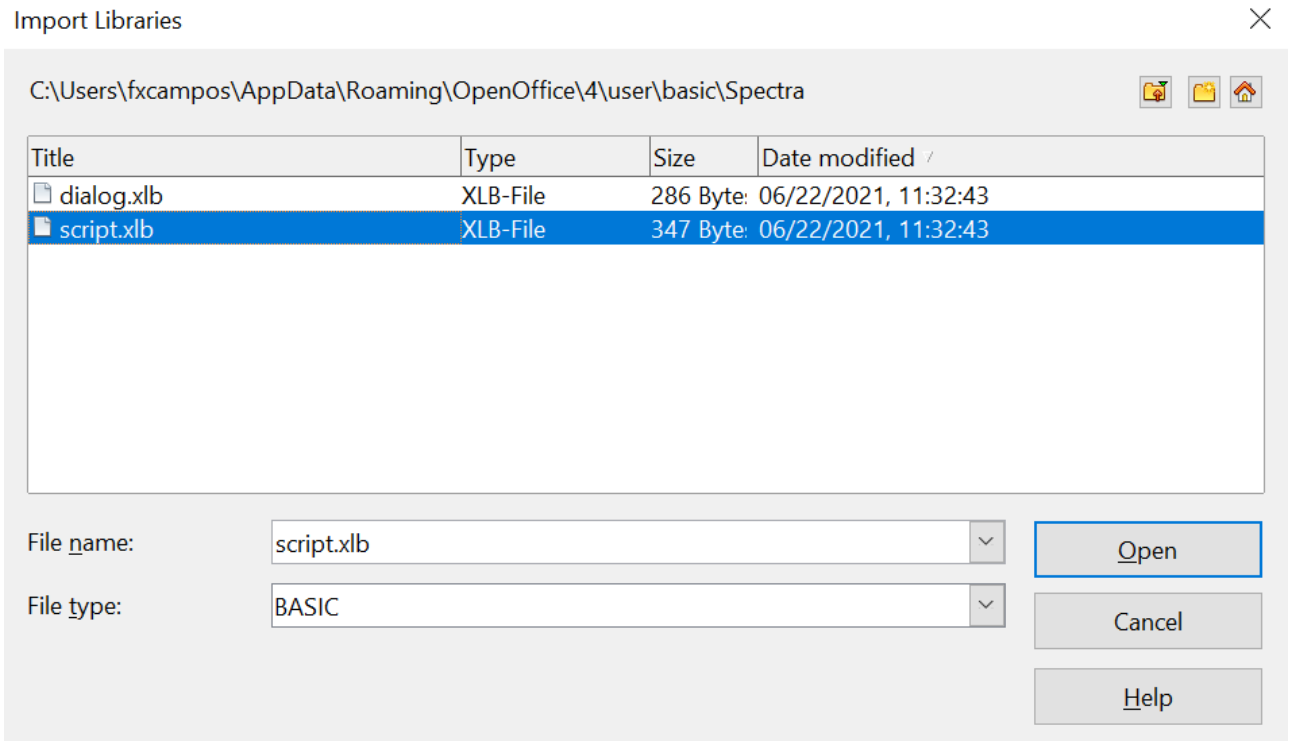


Figure 9: Select a macro library to import

Navigate to the directory containing the library to import. There are usually two files from which to choose, dialog.xlb and script.xlb. It does not matter which of these two files you select; both will be imported. Select a file and click **Open** to continue.

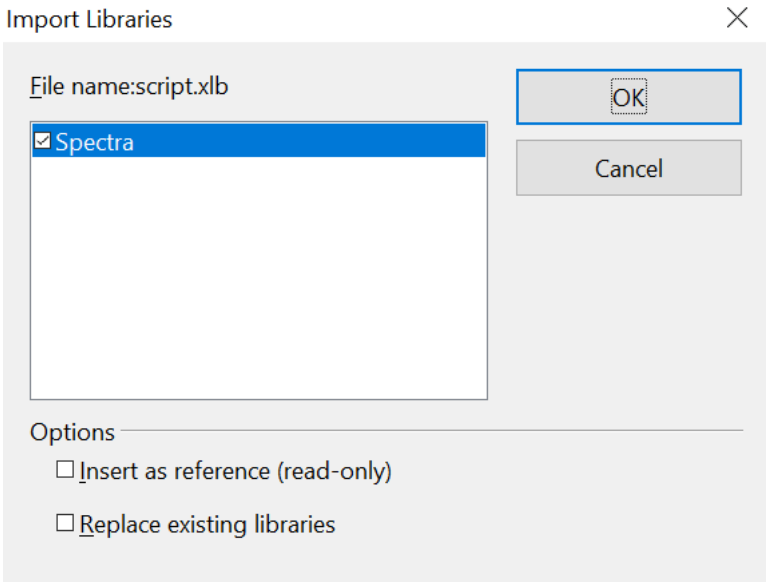


Figure 10: Choose library import options

If the library already exists, it will not be replaced unless **Replace existing libraries** is checked. If **Insert as reference** is checked, the library is referenced in its current location, but you cannot edit the library. If **Insert as reference** is not checked, however, the library is copied to the user's macro directory.

Macros can be stored in libraries inside OpenOffice documents. Select a document rather than a directory on disk (as shown in Figure 9) to import libraries contained in a document.

How to Run a Macro

Running a macro can be done easily, as follows:

1) Use **Tools > Macros > Run Macro** to open the Macro Selector dialog (see Figure 11).
2) Select the library and module in the Library list (left side).
3) Select the macro in the Macro name list (right side).
4) Click **Run** to run the macro.

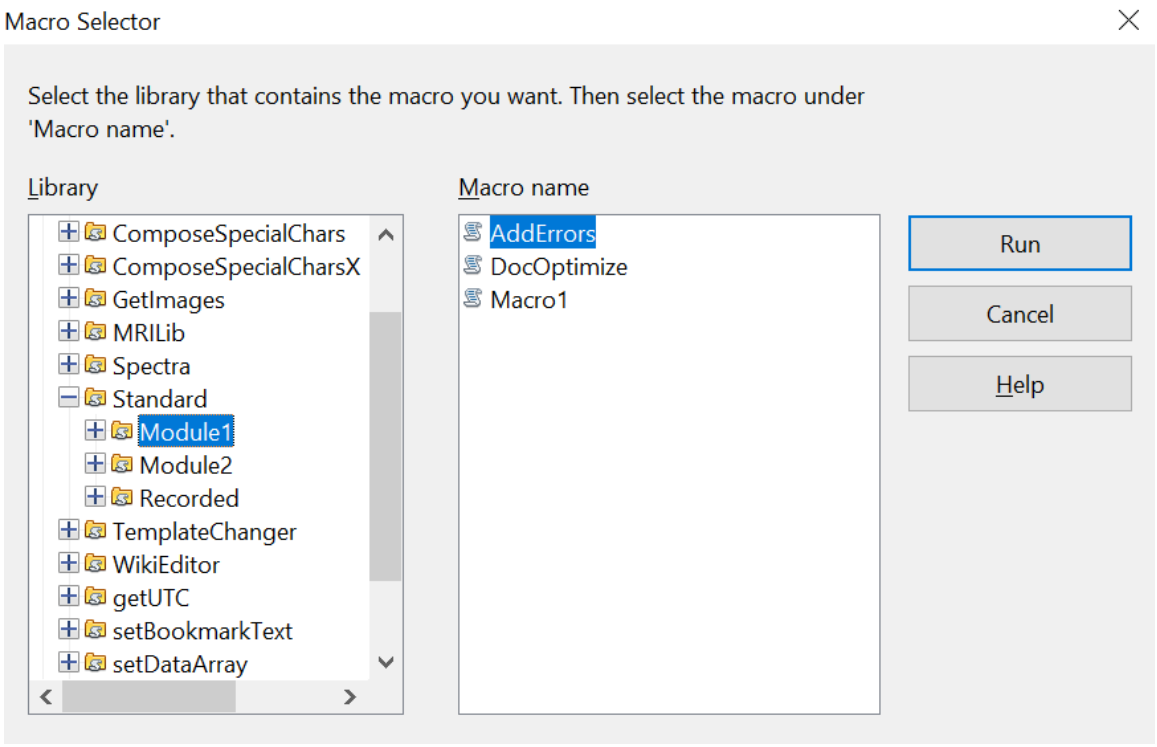


Figure 11: Use the Macro Selector dialog to run macros

Although you can use **Tools > Macros > Run Macro** in any situation, it's not efficient for frequently run macros. A more common technique is to assign a macro to a toolbar button, menu item, keyboard shortcut, or a button embedded in a document. While choosing a method, it is also good to ask questions such as:

- Should the macro be available for only one document, or globally for all documents?
- Does the macro pertain to a specific document type, such as a Calc document?
- How frequently will the macro be used?

The answers will determine where to store the macro, and how to make it available. For example, you probably don't need to add a rarely used macro to a toolbar. Since keyboard shortcuts can be assigned to a single application, that might be the better choice for running a macro that applies to only one kind of document.

To add a menu item, keyboard shortcut, or toolbar icon that calls a macro, use the Customize dialog (see Figure 12). Open this dialog in either of these ways:

- Choose **Tools > Customize** from the main menu bar.
- Each toolbar has an icon that opens a menu; choose the **Customize Toolbar** option.

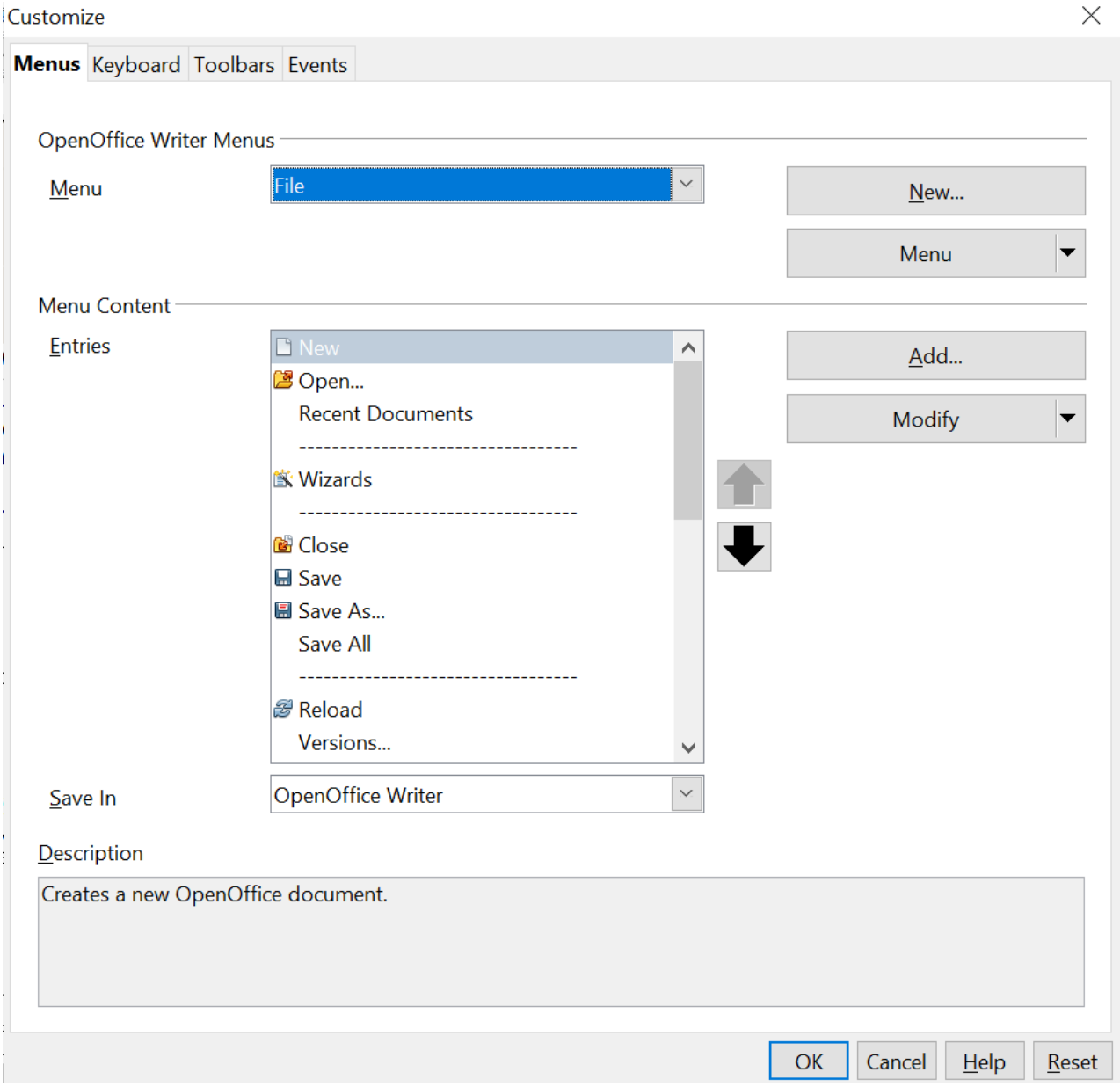


Figure 12: OpenOffice Customize dialog

Tip	Complete coverage of the Customize dialog is beyond the scope of this chapter. See Chapter 11 of this guide for more information about customizing menus, toolbars and the keyboard shortcuts.

The Customize dialog contains tabs to configure menus, keyboard bindings, toolbars, and events.

Toolbar

Macros can be added to toolbars. For more about modifying toolbars, see Chapter 11 (Customizing OpenOffice).

Menu Item

Use **Tools > Customize** to open the Customize dialog, and select the Menus tab. You can modify an existing menu, or create new menus that call macros. For more about modifying menus, see Chapter 11.

Keyboard Shortcuts

Use **Tools > Customize** to open the Customize dialog, and select the Keyboard tab. Assigning keyboard shortcuts is discussed in Chapter 11.

Event

In OpenOffice, when something happens, we say that an event occurred. For example, a document was opened, a key was pressed, or the mouse moved; any of these can be called an event. OpenOffice allows events to cause a macro to be called; Under those circumstances, the macro itself is called an event handler. Full coverage of event handlers is well beyond the scope of this document, but we can at least summarize here.

Caution

Be careful when you configure an event handler. For example, assume that you write an event handler that is called every time a specific key is pressed. If you make a mistake, the event may not be properly handled. One possible result of such a scenario is that your event handler will recognize and consume all key presses, forcing you to terminate OpenOffice.

Use **Tools > Customize** to open the Customize dialog, and select the Events tab (see Figure 13). The events in the Customize dialog are related to the entire application and specific documents. Use the Save In box to choose OpenOffice, or a specific document.

A common use is to tell the Open Document event to call a specific macro. The macro then performs certain setup tasks for the document. Select the desired event and click the **Macro** button to open the Macro Selector dialog (see Figure 14).

Select the desired macro and click **OK** to assign the macro to the event. The Events tab shows that the event has been assigned to a macro (see Figure 15). When the document opens, the PrintHello macro is run.

Many objects in a document can be set to call macros when events occur. The most common usage is to add a control, such as a button, to a document.

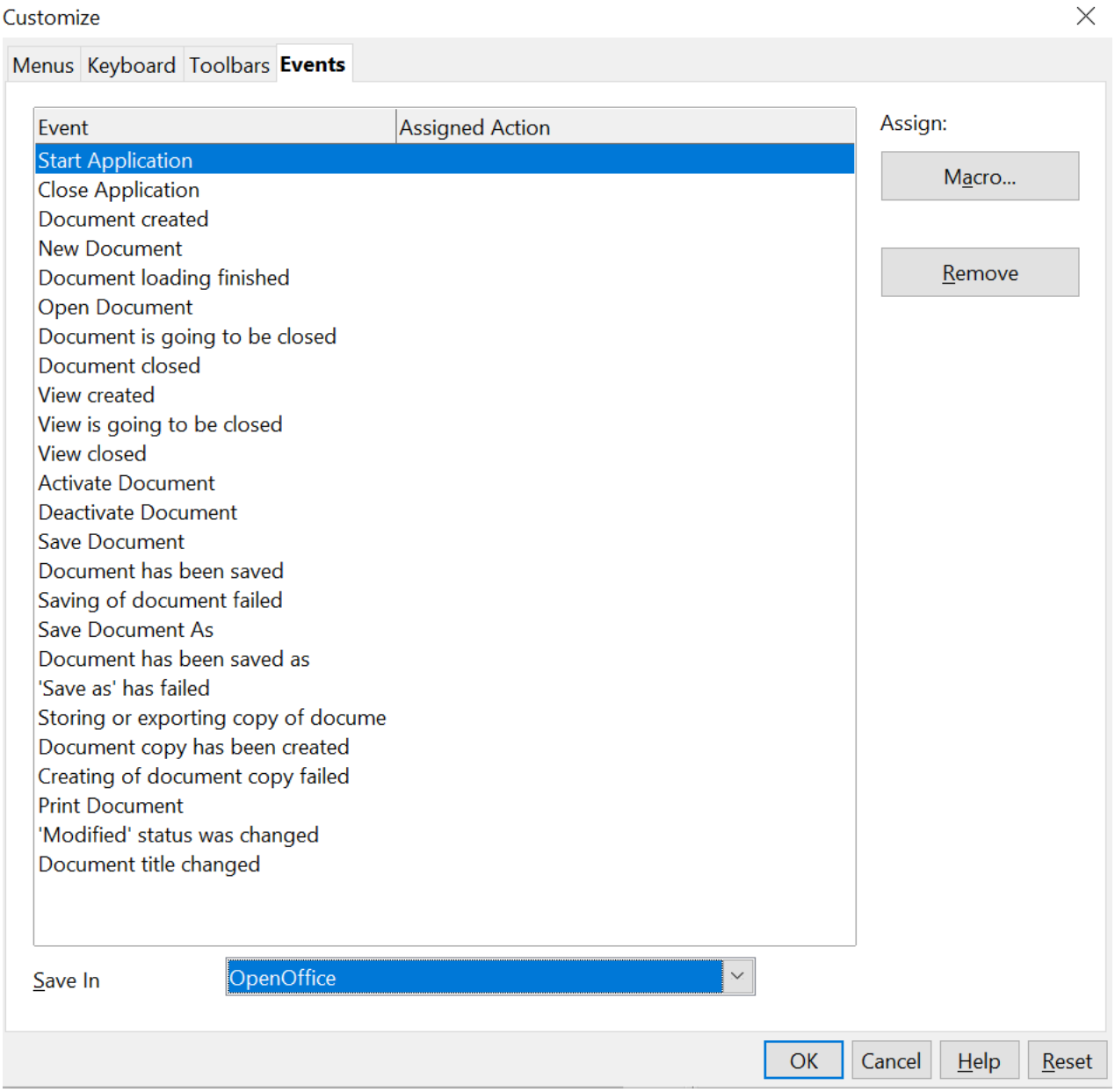


Figure 13: Assign a macro to an application level event

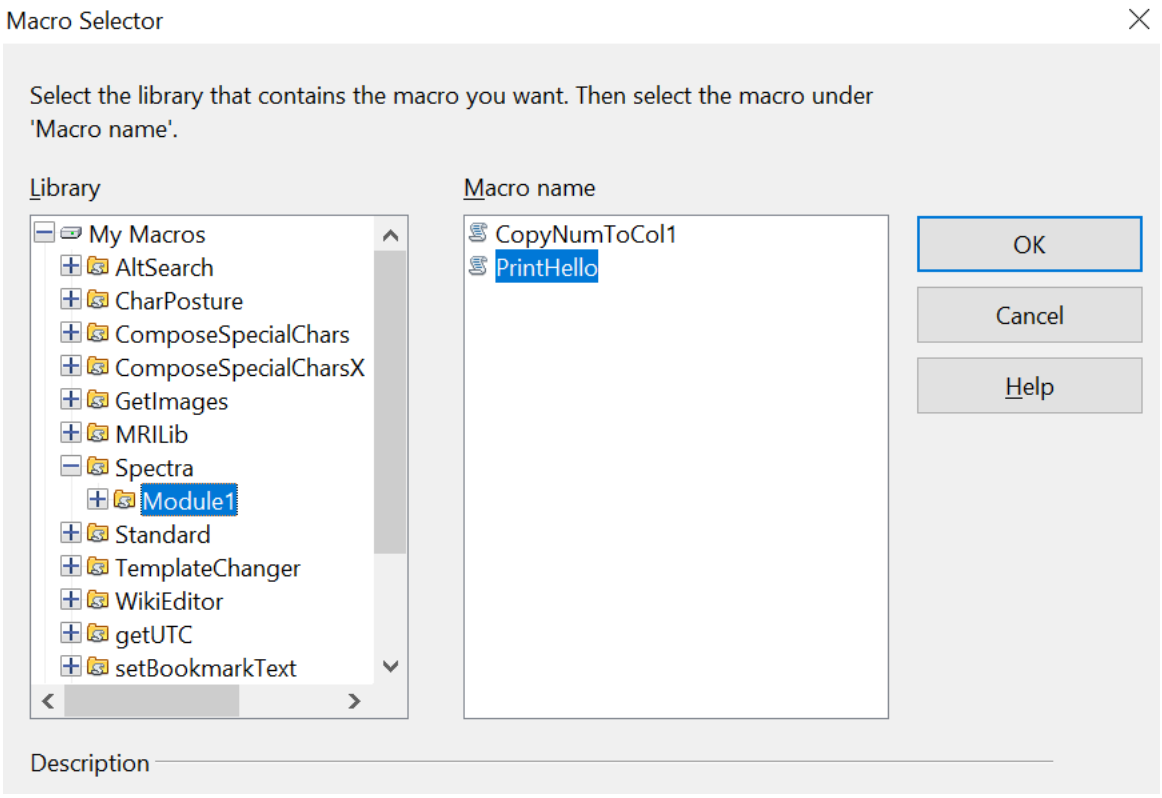


Figure 14: Assign a macro to the document open event

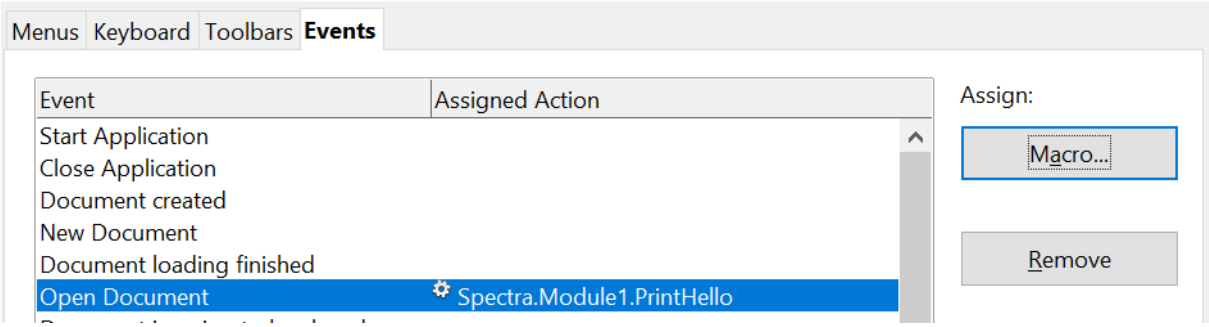


Figure 15: PrintHello is assigned to the Open Document event

Writing Macros Without the Recorder

The examples covered in this chapter were created using the macro recorder and the dispatcher. You can also write macros that directly access the objects that comprise OpenOffice. In other words, you can directly manipulate a document. Directly manipulating OpenOffice's internal objects is a very big topic. Within the limits of this chapter, we can cover only basic topics, and provide references to more detailed explanations.

Procedures and Functions

The building blocks of macros are blocks of codes called procedures and functions. These blocks of code are marked by a line naming the block and a line stating either *End Sub* or *End Function*. A procedure's structure can be sketched like this:

```
Sub ProcedureName()
```

```
    ' Your code goes here
    MsgBox "Hello World"
End Sub
```

- `Sub`: Marks the start of the procedure.
- `End Sub`: Marks the mandatory ending of the procedure.

A function is slightly more complicated.

```
Function FunctionName(Parameter)
    ' Your code goes here and it produces a value named Result
    FunctionName = Result
End Function
```

A function receives a value, named Parameter in the example and it sends a value back to the code that called it. The value is returned by setting the function name to the value. In the example above, this is done with the line

```
    FunctionName = Result
```

As an example of using a function, if the code for FunctionName was in a Calc spreadsheet, you could write

```
   =FunctionName(2)
```

in a cell and the cell would then have the value that Result has when Parameter has the value 2. The ability to return a value is the most important difference between procedures and functions. Both types of code can receive values, as FunctionName did.

Variables

OpenOffice Basic supports four kinds of variables: numbers, strings (text), Boolean (True or False), and objects. There are several types of numbers that are explained in the Help. On the Help Contents tab, look under **Macros and Programming. > Guides > Using Variables**. For most purposes, the only important distinctions are between integer and decimal values.

Declaring Variables

OpenOffice Basic does not generally require that variables be declared. If a variable has not been declared, it will be created when a value is first assigned to it. However, declaring variables is a good practice for writing high quality code. In some cases, it's necessary. To require that every variable be declared, write *Option Explicit* at the top of the module, before any other code. This guards against the common problem of a variable name being mistyped.

DIM Statement

A DIM statement declares a variable. At it simplest, the statement includes only the keyword DIM and the variable's name.

```
Dim oDoc
```

Optionally, you can in set the variable's type by following it's name with "as" and the type.

```
Dim x_count as integer
```

If the type is not stated, the variable is assumed to be a Variant, that is, it can hold any data type.

Key Aspects of DIM in BASIC

- **Purpose:** Declares variables (`DIM x As Integer`) and defines array sizes (`Dim Matrix(10)`)
- **Location:** Usually placed at the beginning of a program
- **Initialization:** `Numeric variables are typically initialized to 0, and string variables to empty strings`

- **Common DIM Examples**
  - `DIM Total As Integer 'The Integer type can store values between -32,768 and 32,767.`
  - `DIM Pop As Long 'Long Integers can store values between about -2 billion and +2 billion`
  - `DIM X As Single 'Single variables store decimals from about 3.4 x 10^38 to 1.4 x 10^-45`
  - `DIM Y as Double 'Double variables store decimal numbers from about 1.8 x 10^308 to 4 x 10^-324.`
  - `DIM Name as String 'Strings store text`
  - `DIM T_F as Boolean  'Booleans are either True or False`
  - `DIM X, Y, Z`: Declares multiple variables at once.

There are some circumstances when declaring a variable is, if not absolutely required, at least far more convenient than the alternatives. One of these is when you want a variable to be of a particular object type and another is when you want to declare an array.

An array is declared by following the variable name with a number or numbers in parentheses. The numbers determine how many elements are in the array. Each array element is numbered, starting at zero. An array named *vals* declared with

```
DIM vals(2) as Integer
```

would have three integer elements numbered 0, 1,and 2. To get the value of the second element of the array, you would write vals(1). An array can have more than one dimension. Declaring

```
DIM vals(2, 5) as Integer
```

makes a variable with three sets (numbered 0, 1, 2) of six values (numbered 0 - 5).

We saw an example of declaring an object variable in Listing 4 where a PropertyValue variable was declared. The DIM statement actually made an array of PropertyValues

```
dim args1(0) as new com.sun.star.beans.PropertyValue
```

Notice that the *args1* array has only one element, element 0. That is different from a simple variable that is a Property Value. The *args1* variable was used when calling the *executeDispatch* function. That function requires an array for the argument where *args1* was used, even if the array has one element.

Flow Control

It's common to want code to adapt its actions to values in the data or to repeat an action many times. Basic includes several statements that help you do this.

If .. Then Statement

An *If* statement determines whether a block of code will be run, depending on a logical test. As a simple example, let's say you want to print one of three messages depending on if the variable *score* is less than 10, equal to 10, or greater than 10.

```
If score < 10 Then
  MsgBox "Your score is low."
Elseif score = 10 Then
  MsgBox "Your score is on target."
Else
  MsgBox "You exceeded the target."
End If
```

If the comparison *score < 10* is True, the first message is printed and the rest of the code down to the *End If* is skipped. If the comparison is False, the next test, *score = 10* is done. If it is True, the second message is printed and the code jumps to the point after *End If*. If neither of the comparisons is True, the third message is printed.

For Statement

A *For* statement allows you to run a block of code a number of times. It runs with a variable set to a starting value, and repeats with the variable increased in value until it reaches an ending value. Let's say you have two arrays storing the lengths and widths of 100 rectangles and you want to store the areas of the rectangles. Assuming you have prepared an array named *area* with 100 elements, you can do this with three lines of code.

```
For i = 0 to 99
  area(i) = width(i) * length(i)
Next i
```

The *For* loop will run first with the variable *i* set to zero, then again with *i* set to one, continuing until *i* equals 99. Notice that the *For* loop ends with the keyword *Next*.

Do...Loop

The Do loop is a very flexible structure that can test a condition before a block of code is run or after it is run. The loop will keep running until a condition is True or while a condition is True. The beginning of the code block is marked with *Do* and the end with *Loop*. After either keyword a *While* or *Until* test controls when execution of the code stops. Two examples can illustrate the options.

```
'First version
x = 0
Do While x < 10
    MsgBox "x is equal to " + x
    x = x + 1
Loop

'Second version
x = 0
Do
    MsgBox "x is equal to " + x
    x = x + 1
Loop Until x > 9
```

The While or Until test can be at the top of the loop or the bottom. Their positions in the two examples are not definitive. The two examples produce the same output, printing values of x from zero to nine, but the logic is not identical. With the While or Until test is at the top of the loop, the loop may not be executed at all. With the test at the bottom of the loop, the loop will always be executed once.

Exiting Code Blocks

It's often useful to stop executing procedures, functions, *If* statements, *For* loops, or *Do* loops without finishing the code block. This may be because data input from a file or user has ended or because the values being calculated indicate that something is wrong. For example, if you are calculating the length of physical objects and the results become negative, you may want to exit the code block and print a message to the user. Basic has a set of Exit statements for each kind of code block: *Exit Do*, *Exit For*, *Exit Function*, *Exit Sub*.

The API and Object Oriented Code

To this point, we have covered aspects of OpenOffice Basic. To work with OpenOffice documents, Basic must use OpenOffice's Application Programming Interface (API). The API provides an object oriented framework for interacting with OpenOffice documents. The idea of object oriented code is simple. Each thing that the code interacts with is an *object*. An object contains two kinds of things, data, called *properties*, and functions that act on the data, called *methods*. Taking a Writer document as an example, the top level object represents the entire document. The document object has many properties and methods and many of its properties are themselves objects with their own properties and methods. Among the properties of the document are a *TextTables* property, which is an object that provides access to any tables, a *DrawPage*, that provides access to shapes drawn in the document, and a *CharacterCount* property that is simply the number of characters in the document. Among the methods are *getTextFields*, which returns the object that gives access to text fields like page numbers in footers, and *findAll*, which provides searching for text. The idea of object oriented code is that the information associated with a particular kind of object and the functions that provide or change that information are grouped together. The usefulness of this encapsulation of properties and methods is very clear if the code handles multiple documents. To change the text tables in one document, you start with the object representing that document and you get its TextTable property. The methods in that TextTable object affect only its document.

To see how this all works in an actual macro, we'll discuss the code in Listing 7 that inserts the text “Hello” at the end of a Writer document.

Listing 7: Append the text “Hello” to the current document.

```
Sub AppendHello
  Dim sTextService as String
  Dim oCurs as Object

  REM Verify that this is a text document
  sTextService = "com.sun.star.text.TextDocument"
  REM ThisComponent refers to the currently active document.
  If ThisComponent.supportsService(sTextService) Then
    REM Get the view cursor from the current controller.
    oCurs = ThisComponent.currentController.getViewCursor()
```

```
    REM Move the cursor to the end of the document
    oCurs.gotoEnd(False)

    REM Insert text "Hello" at the end of the document
    oCurs.Text.insertString(oCurs, "Hello", False)
  Else
    MsgBox "This macro only works with a text document"
  End If

End Sub
```

The macro first declares the variables sTextService and *oCurs* in DIM statements. The names of these variables follow a common convention that the first letter tells the reader the type of the variable. The prefix “s” means a string and “o” means an object. These prefixes don't have any effect in the code; they are hints for the person reading it. The variable type is set by the *as String* and *as Object* parts of the DIM statements.

The code now checks if the current document is a Writer document. This part of the code is not necessary, but it results in more user-friendly code. First, the value of *sTextService* is set.

```
  sTextService = "com.sun.star.text.TextDocument"
```

The value of the variable will be explained shortly. For the moment, it is just some plain text.

The next line of code is an *If* that we'll explain in steps.

```
  If ThisComponent.supportsService(sTextService) Then
```

The variable ThisComponent is built in to OpenOffice Basic and refers to the currently active document. OpenOffice documents of all types have the method *supportsService*. To explain what it does, we need to understand what a “service” is. A service defines a set of properties and methods for an object. For an object to be of a certain type, it must include certain services. For an object to be a Writer document, it must include the service *com.sun.star.text.TextDocument*. The method *supportsService* allows an object to ask itself whether it includes the given service. If it does, the method returns True, otherwise it returns False. If the macro is run from a Writer document, *supportsService* will return True and the macro will continue running the code following the *Then*.

The next line of code gets the c*urrentController* of the Writer document and its *ViewCursor*.

```
    oCurs = ThisComponent.currentController.getViewCursor()
```

The *currentController* of a document is the object that controls what the user sees. In a Writer document that's such things as the page displayed, what text, if any, is selected, and the active cursor. The active cursor is called the *ViewCursor* and it is that sub object that is stored in the variable *oCurs*.

The *ViewCursor* has a method *gotoEnd* that moves the cursor to the end of the document.

```
oCurs.gotoEnd(False)
```

The False argument that is passed to *gotoEnd* sets whether the cursor selects all the text between the cursor's original and final positions. We don't want it to do that, so we set it to False.

The last action of this block of code is to insert the text “Hello”. This is done with the *insertString* method of the *Text* property of the cursor.

```
oCurs.Text.insertString(oCurs, "Hello", False)
```

The *insertString* method takes three arguments: the location where to insert the text, which is set to oCurs, the current cursor location, the text to insert, and a True/False value that sets whether the text under the cursor should be overwritten (True) or displaced (False).

If the macro had been run on a document other than a Writer document, all of the steps to insert text would have been skipped and a message box would have told the user that the macro only works on text documents.

In a certain sense, the macro is very simple. It tests if the document is a Writer document. If it is, it moves the view cursor to the end and inserts the text, if it isn't, it prints a message. However, the details of each step are not obvious. How do you learn about the *supportsService* method and what exact text should be passed to it? How do you find that the *insertString* method belongs to the *Text* property, of the *ViewCursor* of the *currentController*? That is the fundamental difficulty of writing macros.

The MRI Object Inspection Tool

Having a tool that allows you to see the properties and methods of each object is essential for writing macros. The MRI extension provides this information and more. See chapter 11 for information about installing an extension. Once MRI is installed, there will be a new item for it on the Contents tab of OpenOffice Help and new menu items under **Tools > Add-Ons**, one for launching MRI for the current document and one for launching MRI for whatever is currently selected. We will briefly discuss the first case. Using MRI for the current selection is very similar.

If you select **Tools > Add-Ons > MRI** while viewing a Writer document, the window shown in Figure 16 will appear.

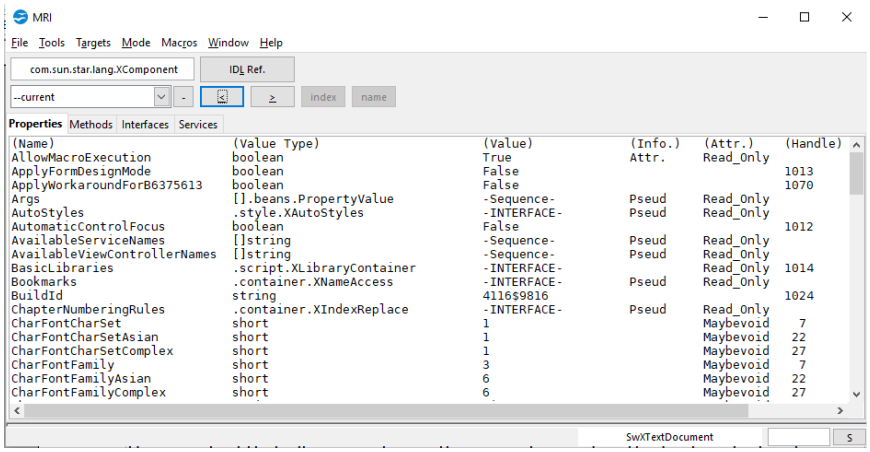


Figure 16: The MRI window for a Writer document

There are many buttons and menus available in that window that, in the interest of brevity, we will not discuss. The Help menu in the window has a lot of information. The most important thing to notice is the four tabs named Properties, Methods, Interfaces, and Services just above the main display. The properties are displayed in alphabetical order and the scroll bar on the right allows you to go through the long list. If you double click on any item, the display will drill down to the properties and methods of that item. After choosing the currentController, its View Cursor, its Text property and switching to the Methods tab, we can see the information in Figure 17. The insertString method is listed there and has been highlighted along with its parameters. These are the text range of the insertion location, the text to insert, and the boolean that controls whether the existing text is absorbed.

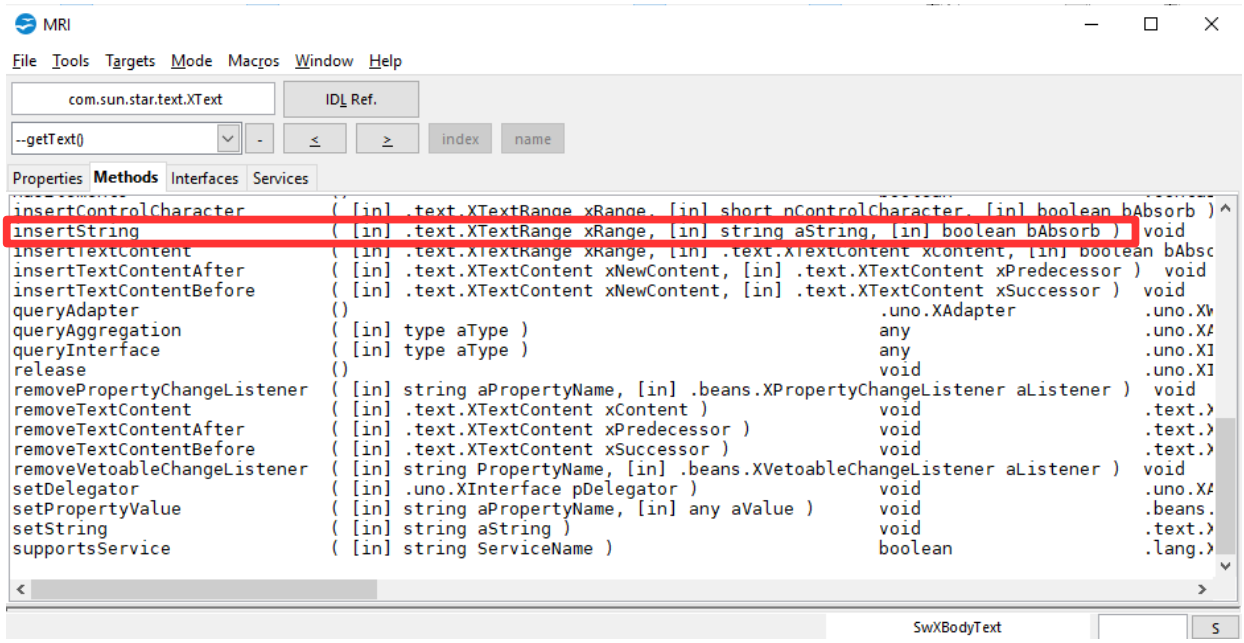


Figure 17: Methods of the Text property of the ViewCursor

Working with MRI as in the last example is efficient if you have a rough idea of what you are looking for. If you remember that the ViewCursor does what you want and it can be reached through the currentController, you will be able to find what you need. If you don't know where to start, MRI will probably not help you find the property or method that you need.

Finding More Information

Numerous resources are available that provide help with writing macros. Use **Help > OpenOffice Help** to open the help pages. The upper left corner of the help system contains a drop-down list that determines which help set is displayed. To view the help for Basic, choose *OpenOffice Basic* from this list.

Included Material

Many excellent macros are included with OpenOffice. Use **Tools > Macros > Organize Macros > OpenOffice Basic** to open the Macro dialog. Expand the Tools library in the OpenOffice library container. Inspect the Debug module—some good examples include WritedbgInfo(document) and printdbgInfo(sheet).

Online Resources

The following links and references contain information regarding macro programming:

OpenOffice User Forum

The OpenOffice user forum has a section dedicated to macros and several volunteers experienced with macros. You must register in the forum to ask questions, but anyone can browse the content.

https://forum.openoffice.org/

Basic User Guide

The Basic Guide gives detailed information about the Basic language and how to work with different kinds of documents. There are many examples of macros.

Basic Guide

Pitonyak.org

Andrew Pitonyak published a book about OpenOffice macros and maintains a web page with several additional very useful documents. The book *OpenOffice.org Macros Explained* and the *English Macro Document* are particularly useful. These contain hundreds of macro examples.

pitonyak.org