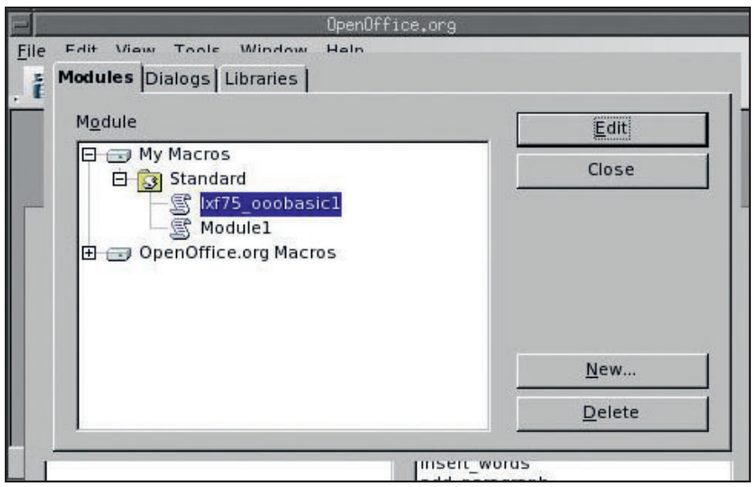
OOo Basic
Use macros in Writer

Take your office skills to the next level by learning how to use OpenOffice.org Basic and scripting your own macros. Awesome!

There is a powerful programming language built in to *OpenOffice.org* – and in this new series we, the lucky ones, are going to learn how to use it. The language is OpenOffice.org Basic, and in this first section we'll apply it to the word processing part of the suite, *Writer*, an later on to *Calc*, then finish with some tips and tricks.

You may well ask why a user of *OOo Writer* should be interested in programming. The answer is simple: automation. Imagine that you have to produce a report every day, and in the report you need to include disk space usage, or a list of logged-on users. Not a difficult job by any means (you could just use *who* and *df*, then copy and paste the results into your *Writer* document). However, it's boring and time-consuming – it might not take long on a daily basis, but five minutes here, five minutes there all add up. Wouldn't it be better to have *Writer* do the work for you, then get down the pub a bit quicker?



Access macros, dialog boxes and libraries through the Macro Organiser.

As you'll find in this tutorial, using OpenOffice.org Basic we can write macros to automate all sorts of tasks, from opening Writer documents and inserting external data to creating a dialog box, working with dynamic data and beyond. For the tutorial I've used *OpenOffice.org 2.0* (1.9.79) on Linux and version 1.1.4 on Windows (sorry – I just wanted to see how well it would work: most things should be the same for the version on the DVD).

There are many similarities between OpenOffice.org Basic and every other 'Basic' out there. I first used Basic on a Sinclair ZX81 in the early eighties. Now there are other implementations of Basic all over the place – Visual Basic and Gambas to name but two. All of them have the same command structure; it's really just a matter of learning each one's peculiarities. But remember that when I talk about Basic from now on I only mean OpenOffice.org Basic. Don't expect to be able to take code from the examples and have it work in every other implementation of Basic!

You will naturally want me to discuss functions and subroutines, variables and objects before doing anything else. No? You just want to get in there and get something to work? OK, here's a simple piece of code that opens a new blank *Writer* document. Use the Macro Organizer to create a new module *(see The Macro Organizer box, below)*, then type in the following code:

```
Sub Main
   loadNewFile
End Sub
Sub LoadNewFile
   dim doc as object
   dim desk as object
   dim url as string
   dim args()
   desk = CreateUnoService("com.sun.star.frame.Desktop")
   url = "private:factory/swriter"
   doc = desk.loadComponentFromUrl(url, "_blank", 0, args())
End Sub
```

You can now use the Run BASIC button on the toolbar to see the end result – which is a new document, as promised.

From the code above you might be able to deduce how to load any file that you want. The secret is in the URL, such as

```
   url ="file:///home/bainm/test.odt"
```

Even better, you can build a subroutine that will open either a named file or, failing that, an empty one:

```
Sub LoadNewFile (optional myFile as string)
   dim doc as object
   dim desk as object
   dim url as string
   dim Dummy()
   if isMissing(myFile) then
      myFile = "private:factory/swriter"
   end if
   desk = CreateUnoService("com.sun.star.frame.Desktop")
   url = myFile
   doc = desk.loadComponentfromurl(url,"_blank",0,Dummy())
End Sub
```

If you've used Basic at all you'll recognise the general structure – we've created two subroutines. The first (**Main**) is used to control the operation of the macro. The second (**loadNewFile**) does the actual work. It defines some variables to use (**doc**, **desk**, **url** and **args**), then creates a UNO (Universal Network Object), which gives you access to the methods and properties of the *Writer* objects.

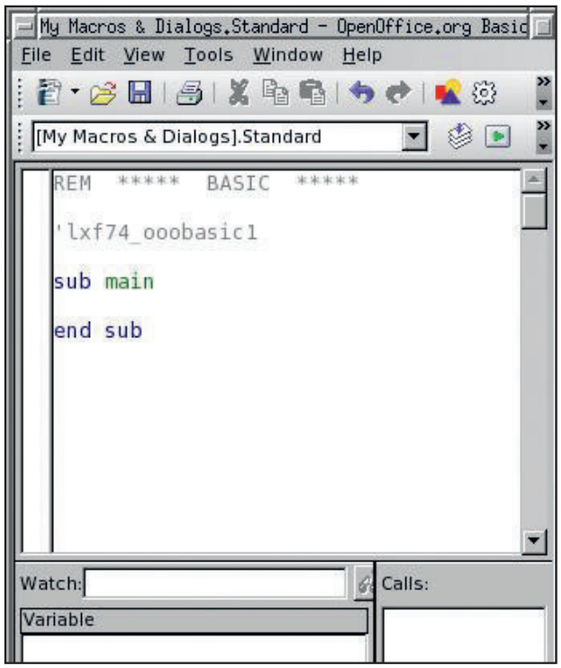
Macro Organiser

The OOo Macro Organizer is your way of accessing and maintaining macros, dialogs and even libraries. You can create new items, edit them and delete old ones if you need. If you are very brave (or maybe that's foolhardy) you can modify OOo's built-in macros. The way that you access the Organizer will depend on the version of OpenOffice.org that you're using. In version 1.1.4 you need to click on Tools, then Macros > Macro... > Organizer. If you're using version 2.0 you'll need Tools > Macros > Organize Macros, OpenOffice.org Basic > Organizer.

LXFS11.scripting Sec2:76 22/8/07 10:47:41

Write to a document

A blank file is not particularly useful by itself, and you can create one pretty easily yourself with Ctrl+N. So let's add a subroutine that will write to the document:



The *OpenOffice.org* desktop contains a Basic code editor.

```
Sub Insert_words
   dim doc as object
   dim cursor as object
   doc=thisComponent
   cursor=doc.text.createTextCursor
   cursor.string="Hello World"
End Sub
```

You'll also need to modify the **Main** subroutine:

```
Sub Main
   loadNewFile
   insert_words
End Sub
```

This shows you how easy it is to use code to control the *Writer* app. To make most use of this, you can write a subroutine that accepts an input and writes it to the document as a paragraph:

```
Sub Add_paragraph (myText as String)
   dim doc as object
   dim cursor as object
   doc=thisComponent
   cursor=doc.text.createTextCursor
   cursor.gotoEnd(False)
   doc.text.insertControlCharacter(cursor, _
   com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,
   False)
   doc.text.insertControlCharacter(cursor, _
   com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,
   False)
   cursor.string = myText
End Sub
```

This time the code moves the cursor to the end of the document, creates a new paragraph and inserts any text that you sent it. For instance:

```
Sub Main
   loadNewFile
   add_paragraph("This is my first paragraph.")
   add_paragraph("This is my second paragraph.")
End Sub
```

In this example it would actually be quicker to type the details into a *Writer* document. But, you're probably already thinking of potential uses – especially if we now combine this functionality with the ability to load information from external files. If you've used any Basic flavour at all, this next bit is going to be quite familiar to you:

```
Sub Load_report_file(myFile as String)
   dim filenumber As Integer
   dim iLine As String
   dim pText As String
   filenumber = Freefile
   open myFile For Input As filenumber
   while not EOF(filenumber)
      Line Input #filenumber, iLine
      if (iLine <> "") then
         pText = PText & iLine
      else
         add_paragraph(pText)
         pText=""
      end if
   wend
   if (PText<>"") then
      add_paragraph(pText)
   end if
   close #filenumber
End Sub
```

This subroutine takes a filename as an input. It then scans through the file looking for complete paragraphs. If it finds a complete paragraph (it identifies this by an empty line) it'll send it to our new document. If not, it goes on looking until it gets to the end of the file.

There are just a couple of things that may need clarifying in the code. The first of these is the use of **Freefile**. The **open** statement expects you to assign a unique integer to the opened file as a reference number. You could give it your own number... but then have to remember any that you've already used (this becomes important if you have more than one file open at a time). Or you could use **Freefile**, which simply assigns the next from a sequence of numbers. The second thing that you may ask is why there is a second **add_paragraph** statement outside of the **while... wend** loop. This is simply to catch any paragraph at the end of a file that is not terminated by an empty line.

OK, now you can use this functionality by loading information from any files that you need, *ie*:

```
Sub Load_report_simple
   dim rep_dir as String
   rep_dir = "~/articles/lxf75_ooobasic1/demo/"
   load_report_file(rep_dir & "manager_header.txt")
   load_report_file(rep_dir & "body.txt")
End Sub
```

There is still little advantage in building up your document in this way – you could just as easily type the info directly in to your

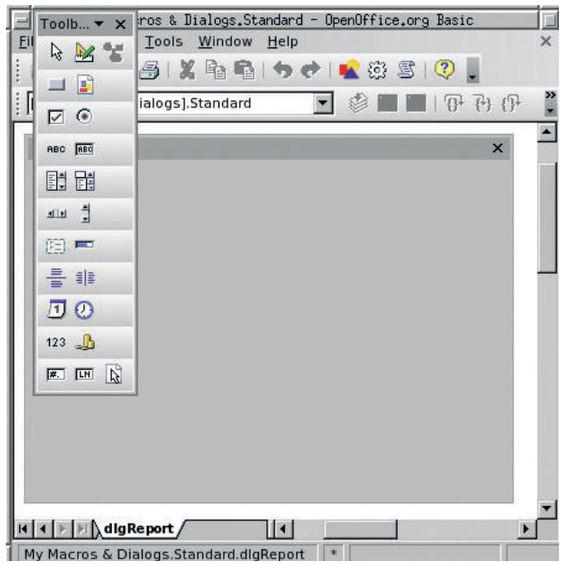
Associate Code & Objects

You may be used to languages that automatically associate code and object – Delphi, Kylix or Gambas for instance (or even Visual Basic). However, when using OpenOffice.org Basic you must build the code and the objects separately, then manually associate the code with the object (such as a button). You can do this through the Events tab on the object's property form (use the right mouse button to click on the object, then select Properties). Easy, yes: but not that intuitive to start with.

LXFS11.scripting Sec2:77 22/8/07 10:47:42

Writer document, then save or print it. However, we can start making the code very useful if we introduce an element of choice:

```
Sub Load_report (optional reportType as integer)
    const rep_dir as string = "~/articles/lxf75_ooobasic1/
    demo/"
    if isMissing(reportType) then
        reportType = 1
```



You can use the Organizer to create new dialog boxes – with a selection of buttons – as well as code.

```
    end if
    select case reportType
    case 1
        load_report_file(rep_dir & "manager_header.txt")
    case 2
    load_report_file(rep_dir & "contractor_header.txt")
    end select
    load_report_file(rep_dir & "body.txt")
End Sub
```

This time there are two possible files that can be loaded (**manager_header.txt** or **contractor_header.txt**) depending on the value of the **reportType** variable supplied to the subroutine. However, both cases end with the same file being inserted (**body.txt**). All you need to do is to change the **Main** subroutine, with

```
Sub Main
    loadNewFile
    load_report(1)
End Sub
```

or

```
Sub Main
    loadNewFile
    load_report(2)
End Sub
```

I'm sure that you'll immediately see a drawback here – you have to modify the code every time that the different reports need to be created. We need some neat way of being able to call the particular files that we need. To do this we are going to build a nice little dialog box to allow us to control the output.

Build a dialog box

You need to use the Macro Organizer to create a dialog box. This time, go to the Dialogs tab before clicking on the New button. If you name the dialog **dlgReport**, you can call it with the following code:

```
dim dlgReport as object
Sub DlgReport_show
    basicLibraries.loadLibrary("Tools")
    dlgReport = loadDialog("Standard","dlgReport")
    dlgReport.execute()
End Sub
```

You'll need to change the main subroutine as well:

```
Sub Main
    dlgReport_show
End Sub
```

The dialog box won't actually do anything yet (when you've run the code, just press the Escape key to close the box). However, we can now create buttons and list boxes in the dialog box, and then write some code for the necessary functionality that we want.

We're going to use this dialog box to control the type of report that *Writer* opens. To do this we'll need a list box and a button. For the button, select the object that you want from the toolbox, then draw it on to the dialog box. Use the Properties editor to set their names to lstReport and btnReport respectively, and put some useful text on the button (I'd suggest the words 'Create Report').

Next we'll load the list box with details using one of *OOo*'s built-in methods – *addItem*. You might immediately jump in and try the following (I know I did):

```
Sub DlgReport_show
    basicLibraries.loadLibrary("Tools")
    dlgReport = loadDialog("Standard","dlgReport")
    dlgReport.lstReport.AddItem("Managers",0)
    dlgReport.lstReport.AddItem("Contractors",1)
    dlgReport.execute()
End Sub
```

Seems logical enough, but that's not the way that OpenOffice.org Basic does it. Instead you need

```
dim lstReport as object
sub DlgReport_show
    basicLibraries.loadLibrary("Tools")
    dlgReport = loadDialog("Standard","dlgReport")
    lstReport = dlgReport.getControl("lstReport")
    lstReport.AddItem("Managers",0)
    lstReport.AddItem("Contractors",1)
    dlgReport.execute()
End Sub
```

Notice that you need to define the list box as a unique object. You can only access it once you've used the dialog box's **getControl** method. We've also defined **lstReport** as a global parameter – this means that once it has been initiated, it can be used in any subroutine that we write (as we'll see).

Activate the button

Next we write the code that will run when the Create Report button is clicked. Add this subroutine:

```
Sub BtnReport_Click
    loadNewFile
    load_report(lstReport.selectedItemPos)
End Sub
```

No doubt you've now run the main macro and found that you've got a working combo box, but when you click on the button nothing happens. You probably haven't associated any code with it yet. Go to the button's Properties screen, click on the Events tab and select the subroutine that you want to run when the button is clicked (use the Escape key to close the screen when you've finished). With that done you will have a fully functioning form to control the creation of the two different documents.

Going further

I'm sure you'll agree that it's always important to save your work. In all our examples so far we've compiled the documents but then needed to save them manually. Since this tutorial is all about

Quick tip

OpenOffice.org Basic is case-insensitive. Therefore it will recognise myVariable as being the same as myvariable or even MYVARIABLE. It doesn't matter which one you choose. Just choose one naming convention and stick to it.

To close a dialog box just press the Escape key.

When you've created a button, and written the code for it, don't forget that you will have to associate the code with the button through the Properties screen.

LXFS11.scripting Sec2:78 22/8/07 10:47:43

automation, we'd better look at using code to save the files. Try:

```
Sub SaveMyFile (fileUrl as string)
   dim params()
   doc.saveAsUrl("file:/" & fileUrl, params())
   doc.close(true)
End Sub
```

We may, however, just want to print each document and not save it. In this case we could create a subroutine to do that:

```
Sub PrintMyFile
   dim params()
   doc.print(params())
   doc.close(true)
End Sub
```

OK, we've looked at handling *Writer* documents and how to extract data from external files. We've looked at static files and how to read from them. Excitingly, it is also possible to generate dynamic data by making use of *OpenOffice.org's SystemShellExecute* method, thus:

```
Sub RunCommand (command as string)
   dim svc as object
   svc = createUnoService("com.sun.star.system.
   SystemShellExecute")
   svc.execute(command, "", 0)
End Sub
Sub BtnReport_Click
   const tmpfile as string = "/tmp/myfile.tmp"
   loadNewFile
   load_report(lstReport.getselectedItemPos())
   runCommand("df > " & tmpfile)
   load_report_file(tmpfile)
End Sub
```

Here, the **df** command is sent as a Linux shell command with its output being saved to a file (in this case **/tmp/myfile.tmp**). The content of the file is then loaded in to the new *Writer* document with a result that looks something like

'Filesystem 1K-blocks Used Available Use% Mounted on
/dev/hda3 3470204 3089264 201816 94% /
/dev/hda4 1510060 1064572 368780 75% /opt
/dev/hda1 4593600 3732708 860892 82% /WINDOWS.'

Useful though this is, it isn't that attractive or easy to read; the output would look much better in a table. In the **www.linuxformat.co.uk/special/ooo** directory on our website, you'll find the complete code for loading a table with the contents of a file; download all the file name at this location. The code is much too long for us to show you here, but if you look through it carefully, you'll certainly find a wealth of useful Basic functionality such as *Chr* (which returns an ASCII code for a given integer), *Array* (which creates an array from a series of strings) and *ubound* (which returns the maximum index number for an array). Your homework is to look at the *btnReport_Click* subroutine. Examine the way that the command variable is built up and then sent to the shell.

OOo Basic
Use macros in Calc

Keep spreadsheets at arm's length and work with data from the console – just follow our lead and you'll get to grips with it right away.

From Charles Babbage's Difference Engine to *OOo's* spreadsheet *Calc*, number crunching was always meant to be automated. Taking some of the pain and monotony out working with columns of data is one of the great reasons for using a spreadsheet in the first place. Thanks to a combination of OOo Basic and *Calc*, it's possible not only to automate of the most arduous tasks but, as I'll show you, to manipulate interesting data directly from the command line. As with the previous section, the first step to macro nirvana is creating a document. The code to open a new, blank *Writer* document is:

```
sub main
  loadNewFile
end sub
sub loadNewFile
  dim doc as object, desk as object, myFile as string, Dummy()
  myFile = "private:factory/swriter"
  desk = CreateUnoService("com.sun.star.frame.Desktop")
  doc = desk.loadComponentFromUrl(myFile,"_blank",
  0,Dummy())
end sub
```

If you look through the code you will see that the type of file to be opened is defined by the line

```
myFile = "private:factory/swriter"
```

It's then just a matter of knowing what to put in instead of **swriter**. To open a spreadsheet we need to change it to **scalc**:

```
myFile = "private:factory/scalc"
```

Remember to be lazy

Now, I know what you're thinking – you don't want a separate subroutine for each file type; you just want a single subroutine to do all the work. Well that's exactly how a good programmer should think, and here's how you'd do it:

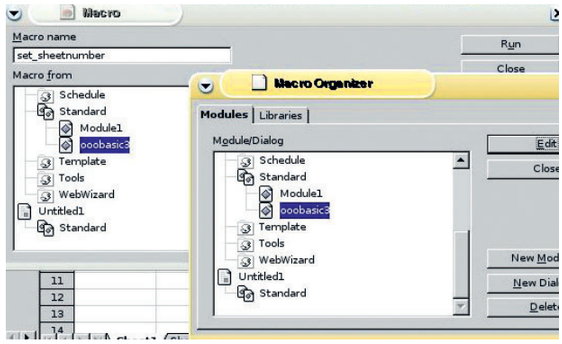
```
sub main
  loadNewFile("scalc")
end sub
sub loadNewFile (filetype as string)
  dim doc as object, desk as object, myFile as string, Dummy()
  myFile = "private:factory/" & filetype
  desk = CreateUnoService("com.sun.star.frame.Desktop")
  doc = desk.loadComponentfromurl(myFile,"_blank",0,Dummy())
end sub
```

Notice how we now send the file type to the subroutine, giving us much more flexibility. Most important of all, we only need one subroutine regardless of whether we want to open a *Writer* document or a *Calc* spreadsheet. If you want, you can even give it a default file type, by making use of the Optional parameter and the **isMissing** method:

```
sub loadNewFile (optional filetype as string)
  if isMissing(filetype) then
    filetype = "scalc"
  end if
```

OK – now we can open up a blank spreadsheet – but what about writing to one of the cells? This next subroutine will do just that:

LXFS11.scripting Sec2:79 22/8/07 10:47:44



***Don't forget – you create, edit and run your code through the OpenOffice.org* Macro Organizer.**

```
sub writeToCell
  dim sheet as object, cell as object
  sheet=thisComponent.sheets(0)
  cell=sheet.getCellByPosition(0,0)
  cell.string="Hello World"
end sub
```

You'll have to remember to run this from the Main subroutine. It may be worth looking through the **writeToCell** subroutine just to understand the basics fully. We've seen **thisComponent** before (when we looked at OOo Basic and the *Writer* document) and it simply refers to the current document (in this case, the spreadsheet) that we're in. Next we select the sheet that we're dealing with, which is sheet(0) – the first sheet (or Sheet1) in *Calc*. Sheet(1) would refer to the second sheet, and so on. Finally we select the cell that we want by using the **getCellByPosition** method, which requires the column number and row number to be input. **Position (0,0)** refers to A1, (1,0) to B1, (0,1) to A2 etc.

This is great, but the order of your sheets may change; what if you want to refer to them by name?. No problem – instead of the sheets statement use the **getByName** method:

```
  sheet=thisComponent.sheets.getByName("Sheet1")
```

We've seen how easy it is to write text to the document (even easier than it is in *Writer*), so let's start using the spreadsheet to do something useful:

```
sub simple_maths
  dim sheet as object, cell as object
  sheet=thisComponent.sheets.getByName("Sheet1")
  cell=sheet.getCellByPosition(0,0)
  cell.value=10
  cell=sheet.getCellByPosition(0,1)
  cell.value=10
  cell=sheet.getCellByPosition(0,2)
  cell.formula="=A1+A2"
end sub
```

Admittedly that's not particularly useful, but it does show you how simple it is to load the spreadsheet with data, and then to manipulate that data. It can be made more useful by allowing the numbers to be input to the subroutine:

```
sub simple_maths(numbA as double, numbB as double)
  dim sheet as object, cell as object
  sheet=thisComponent.sheets.getByName("Sheet1")
  cell=sheet.getCellByPosition(0,0)
  cell.value=numbA
  cell=sheet.getCellByPosition(0,1)
  cell.value=numbB
  cell=sheet.getCellByPosition(0,2)
  cell.formula="=A1+A2"
end sub
```

Now you just need to amend the Main subroutine:

```
  simple_maths(12.5,35.7)
```

This is a very simple example, and it would be quicker to type the details directly into the spreadsheet. However, it's only meant to be a starting point, and you can start making the operation as complicated as you require. You may also think that only passing two numbers into the subroutine is just too limiting – you may want to pass 10 figures, or 100 or 1,000. Fortunately, it's very easy to pass an array to a subroutine:

```
sub main
  loadNewFile
  simple_maths_array(array(45,67,89,34))
end sub
sub simple_maths_array(numbers)
  dim sheet as object, cell as object, r as integer, sum as double
  sheet = thisComponent.sheets.getByName("Sheet1")
  sum = 0
  for r = 0 to ubound(numbers)
    sum = sum + numbers(r)
    cell = sheet.getCellByPosition(0,r)
    cell.value = numbers(r)
  next
  cell = sheet.getCellByPosition(0,r+1)
  cell.value = sum
end sub
```

The **simple_maths_array** subroutine populates the first column of Sheet1 with the contents of an array of numbers, then inserts the sum of all of the numbers at the bottom.

Having written information to a spreadsheet, you may well be asking if it is possible to use data in an existing one. Of course it is – I wouldn't have mentioned it if it wasn't. This next subroutine opens an existing spreadsheet (**~/test.ods**) and displays the contents of cell A1 of Sheet1:

```
sub dataFromExistingFile
  dim doc as object, desk as object, sheet as object, cell as object
  dim url as string, contents as double, Dummy()
  desk = CreateUnoService("com.sun.star.frame.Desktop")
  url="file://~/test.ods"
  doc=desk.loadComponentfromurl(url,"_blank",0,Dummy())
  sheet = thisComponent.sheets.getByName("Sheet1")
  cell = sheet.getCellByPosition(0,0)
  contents = cell.value
  msgbox(contents)
end sub
```

A thought may occur to you at this point – what happens if the cell contains text instead of a number? Surely the command **contents = cell.value** will cause the subroutine to crash. Actually it doesn't: if the cell contains text, the value parameter is set to zero, thus preventing any such problems.

Adding maths to the equation

Everything we've looked at so far is very simple – just reading and writing to cells. How about doing something a little more interesting? How about using the mathematical formulae that are built into *OpenOffice.org Calc*? Let's say that instead of just writing an array of numbers to the spreadsheet, we want the total, or the average, or even the standard deviation. We can do this by using the FunctionAccess service:

```
sub usingOOoFunctions(iArray)
  dim service as object, sheet as object, cell as object
  service = createUnoService( "com.sun.star.sheet.
FunctionAccess" )
  sheet = thisComponent.sheets.getByName("Sheet1")
  cell = sheet.getCellByPosition(0,0)
  cell.value = service.callFunction( "STDEV", iArray )
end sub
```

As always remember to change Main so that the new subroutine can be run:

Quick tip

Use CreateUno Service to access *OpenOffice.org*'s various interfaces (or Universal Network Objects)

If you find it a bit unwieldy referring to thisComponent all the time you could use a simpler alias:

```
dim doc as object
doc = thisComponent
```

Remember the difference between a function and a subroutine – a function runs some code and returns a result. A subroutine runs some code but doesn't return a result.

LXFS11.scripting Sec2:80 22/8/07 10:47:44

```
  usingOOoFunctions(array(45,67,89,34))
```

I'm sure that you can immediately see a couple of disadvantages with **usingOOoFunctions –** at the moment it would only be able to calculate standard deviation, it will only use Sheet1, and it will only write to cell A1. However, by the suitable use of input parameters we can make this a very adaptable subroutine:

```
sub usingOOoFunctions(  fType as string, sName as string, _
          c as integer, r as integer, iArray    )
  dim service as object, sheet as object, cell as object
  service = createUnoService( "com.sun.star.sheet.
FunctionAccess" )
  sheet = thisComponent.sheets.getByName(sName)
  cell = sheet.getCellByPosition(c,r)
  cell.value = service.callFunction( fType, iArray )
end sub
```

Modify Main so that it contains:

```
  usingOOoFunctions("STDEV","Sheet1", 1, 1, array(45,67,89,34))
```

This raises rather an important question – how can you handle results that might cause the program to crash? For instance, you could try the following:

```
  usingOOoFunctions("SQRT","Sheet1", 1, 1, array(-1))
```

It's probably obvious to you that this won't work, because it calls for the square root of –1, always a no-no! You can try capturing all error-creating situations by writing code such as

```
  if (fType <> "SQRT" and iArray(0) <> -1 ) then
```

but this means you'll have to know every possible combination of function and number that could cause you a problem.

The most efficient solution would be to write an error handler. Let's look at an example (that will crash):

```
function dummy as double
  dim service as object
  service = createUnoService( "com.sun.star.sheet.
FunctionAccess" )
  dummy = service.callFunction( "SQRT", array(-1) )
end function
```

Run it with

```
  msgbox (dummy)
```

Handling an error nicely after our intervention.

This will complain as soon as it gets to the return line, but we can stop that happening by introducing an 'on error resume next' statement at the start of the function. If an error occurs this time, the function will just move straight on to the next line of code. However you may (quite rightly) say that you don't want the code to continue – you want it just to exit neatly. If so, you need to add some code to handle the error appropriately:

```
function dummy as double
  dim service as object
  on error goto errorFound
  service = createUnoService( "com.sun.star.sheet.
FunctionAccess" )
  dummy = service.callFunction( "SQRT", array(-1) )
  exit function
  errorFound:
  msgbox("Invalid input. Result set to -1")
  dummy=-1
end function
```

Rather than just continuing, the function will jump to the point in the code marked **errorFound:** – the colon (**:**) defines it as being a jump destination. Notice that the code contains a line stating

An error message that you don't want to see.

exit function just before the error-handling portion. Without this, the error-handling code will always be run even if there is no error – we, of course, only want the error handling to operate if there actually been an error.

Functions are not subroutines

In the examples above we've used functions and subroutines. You may be wondering what the difference is between the two. A function and a subroutine are basically the same, except that the function will return a result. This means that when you define a function you must state which data type it is going to return. Here's a simple example to give you the idea.

First we'll set a variable by using a subroutine:

```
dim sheet as object, cell as object
sub main
  loadNewFile
  sheet=thisComponent.sheets(0)
  cell=sheet.getCellByPosition(0,0)
  simple_sub
end sub
sub simple_sub
          cell.value = 1
end sub
```

Next we'll do the same again, but this time by using a function:

```
dim sheet as object, cell as object
sub main
  loadNewFile
  sheet=thisComponent.sheets(0)
  cell=sheet.getCellByPosition(0,0)
  cell.value = simple_function
end sub
function simple_function as integer
  simple_function = 1
end function
```

Notice that the subroutine writes to the cell directly, whereas the function supplies an output that is then used to write to the cell. A second thing to take note of is that some of the variables (sheet and cell) have been made global. This means that they are made available to all of the functions and subroutines. If a variable is defined within a procedure, it only exists for the time that the subroutine or function is running (this is often referred to as the scope of the variable). This is very useful, but it does mean that you have to be very careful when it comes to the naming of variables:

```
dim sheet_number as integer
dim sheet as object, cell as object
sub main
          loadNewFile
```

Unos to use

You can access *OpenOffice.org*'s Universal Network Objects through the **CreateUnoService** method. These objects are referred to as "Services".

Quick tip

If you find that you're repeating the same piece of code, consider putting it into a subroutine or a function.

If you are building code to send to the shell, test it by viewing it in a msgbox.

LXFS11.scripting Sec2:81 22/8/07 10:47:45

```
        set_sheetnumber
        sheet= _
        thisComponent.sheets(sheet_number)
        cell=sheet.getCellByPosition(0,0)
        cell.value = sheet_number
end sub
sub set_sheetnumber
        sheet_number = 1
end sub
```

The number 1 is written to A1 in Sheet2.

If we were to insert **dim sheet_number as integer** into the subroutine **set_sheetnumber** in the example above, a new variable called **sheet_number** would be created. This new variable would only be accessible within the **set_sheetnumber** subroutine. Despite having the same name as the variable in the main subroutine, both variables are different, and can hold different values.

Now we can happily read and write to any cell that we want, in any of the sheets within the spreadsheet. This means that we can have a look at the sheet names next. They're a bit boring as they stand – Sheet1, Sheet2, Sheet3 – and not very informative. And there are only three of them anyway.

```
sub changeSheetNames
  dim sheet as object
  sheet = thisComponent.createInstance("com.sun.star.sheet.
Spreadsheet")
  thisComponent.Sheets.insertByName("MySheet", Sheet)
  thisComponent.sheets.removebyname("Sheet1")
  thisComponent.sheets.removebyname("Sheet2")
  thisComponent.sheets.removebyname("Sheet3")
end sub
```

Nice and easy – but unfortunately still a bit limiting. It can be made really useful by passing in a array containing the sheet names to be created – watch:

```
  dim i as integer
  for i = 0 to ubound(sheetNames)
    sheet = thisComponent.createInstance("com.sun.star.sheet.
Spreadsheet")
  thisComponent.Sheets.insertByName(sheetNames(i), Sheet)
  next
```

On my command...

Finally, we can bring together everything that we've looked at in this tutorial (plus some bits from last month). The following code will run shell commands (in this case *df* and *du*), save the results to file and load the data into a spreadsheet.

Here goes:

```
const tmpFile as string = "/tmp/myfile.tmp"
const bshFile as string = "/tmp/runme.bsh"
sub main
  theFullWorks
end sub
function buildCommand (ipCommand as string) as string
  buildCommand = "rm -f " & tmpFile & ";" _
  & ipCommand & "  | sed s/'\t'/' '/g >" & tmpFile & ";" _
  & "while [ ""$(grep '  ' " & tmpFile & ")"" != """" ];" _
  & "do cat " & tmpFile & " | sed s/'  '/' '/g > " & tmpFile &
"1;" _
  & "mv " & tmpFile & "1 " & tmpFile & ";" & "done"
end function
sub theFullWorks
  dim command as string
  loadNewFile
  changeSheetNames (array("Disk Space Usage","File Usage"))
  command = buildCommand("df|grep -v Filesystem")
  reportSheet(command,"Disk Space Usage")
  command = buildCommand("du /| sort -nr")
  reportSheet(command,"File Usage")
end sub
sub reportSheet (command as string, sheetName as string)
  dim sheet as object, cell as object
  dim iNumber As Integer, oNumber As Integer, iLine As String
  dim i as integer, c as integer
  iNumber = Freefile
  oNumber = Freefile
  Open bshFile For output As #oNumber
  print #oNumber,command
  close #oNumber
  shell("bash -c """ & bshFile & """",,,true)
  i = 1
  sheet=thisComponent.sheets.getByName(sheetName)
  Open tmpFile For Input As #iNumber
  While not EOF(iNumber)
    dim cArray
    Line Input #iNumber, iLine
    cArray = split(iLine)
    for c=0 to ubound(cArray)
      cell=sheet.getCellByPosition(c,i)
      cell.string=cArray(c)
    next
    i = i + 1
  wend
  Close #iNumber
end sub
```

Most of the code here is quite straightforward, but there are a few places that may look a bit intimidating. For instance, that bit with all the **&**s. What's that all about? This is just building up the command that will be sent to the Linux shell. If you want to see what is actually going to be sent, just add a msgbox thus:

```
Sub main
        dim command as string
        command = buildCommand("df|grep -v Filesystem")
        msgbox(command)
end sub
```

(see the example below). The code that we've looked at is all fairly simple, but I'm sure that you'll agree that we can start to do some very powerful things with it. There's plenty more example code for this section at **www.linuxformat.co.uk/special/ooo/calc**.

Use msgbox to view code to be sent to the shell.

LXFS11.scripting Sec2:82 22/8/07 10:47:46

OOo Basic
Query databases

More time-saving tricks: macro and database tips will help you run queries, create reports in an OOo app and keep track of your stuff!

Our aim here is to be as lazy as possible. Imagine, for instance, that you're just about to prepare an invoice for a popular Linux magazine that you write for. Why spend time retyping stuff that you've already got stored? This tutorial will give you the tools to shirk many more time-leaking jobs!.

First, you'll need a database – after all, this is all about using macros to get information out of one. But, I'm not going to go into setting up a database – that's outside the scope of the tutorial. Of course, if you *were* to say that you hadn't got a database and didn't know where to start, I'd then suggest that you wanted a database server. You could use just any old PC that you've got, connect it to your network and then install Debian (if you didn't have a second PC, you could run the server on your own machine). You might create yourself a minimal boot installation CD from **www.debian.org**, stick it in your CD drive, reboot and follow the instructions. You wouldn't bother installing any of the extras (desktop, file server, web server *etc*) as you'd just want a bare bones setup. Now I would tell you to turn this into a database server by typing **apt-get install mysql-server**, and then to edit the **/etc/mysql/my.cnf** file to hash out the line 'bind-address = 127.0.0.1' (so that it looked like '#bind-address = 127.0.0.1'). This would allow you to access the server from any other PC on your network. As you'd want a database and a user to access it, I'd probably tell you to do the following:

```
mysql -uroot mysql
set password for 'root'@'localhost' = password('put your own password here');
create database accounts;
grant all privileges on accounts.* to 'your user'@'%'
identified by 'your user password';
exit;
```

Finally, I would suggest you gave your new server a static IP address by editing the **/etc/network/interfaces** file so that the end of it looked something like

```
#iface eth0 inet dhcp
iface eth0 inet static
        address 192.168.1.3
        netmask 255.255.255.0
        gateway 192.168.1.1
```

At that point, I'd say, you could reboot and log on to the PC on which you'd be running *OpenOffice.org*. But since this tutorial is all about using OOo Basic and not creating databases, I won't !

Accessing your database

Don't fire up *OOo* just yet. To make life even easier for ourselves (which is what macros are all about) we'll be using UnixODBC, an API for accessing data sources. This will save us the nitty-gritty of making connections to server and databases – the protocols, signals sent, *etc*. Instead, the hardest thing that you'll have to do is install UnixODBC and its *MySQL* libraries on to the machine where you're going to be using *OOo*. On Debian this is as easy as

```
apt-get install unixodbc
apt-get install libmyodbc
```

Obviously, if you're using another distro you'll have to check the process for that – have a look at the UnixODBC website at **www.unixODBC.org**. However you get UnixODBC installed, you'll just need to finish off by doing two things. First, edit **/etc/hosts** so that it includes a reference to your database server, eg 192.168.1.3 acamas. Then edit **/etc/odbc.ini** to include something like:

```
[accounts]
Description             = MySQL db test
Driver      = MySQL
Server      = acamas
Database= accounts
Port        = 3306
```

We're done: no more command line stuff – for a few lines, anyway!

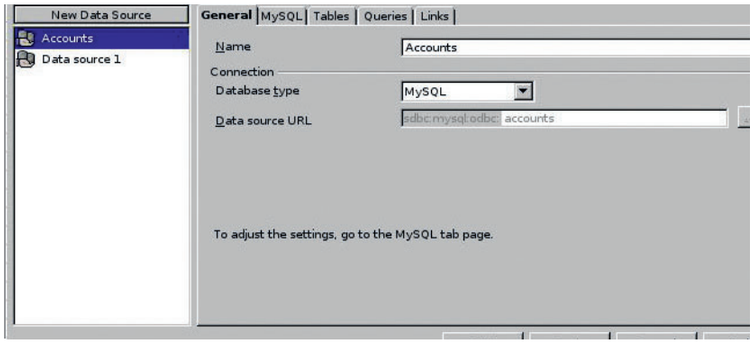
Sort out your database

Open *OpenOffice.org*. It doesn't matter which type of document, but let's say *Writer* for now. In Tools, you'll see that one of the sub-menus says Data Sources. Click on that and you'll notice the Data Source Administration form.

Click on New Data Source and set the database type to *MySQL* in the General tab. Then go to the *MySQL* tab, add the database name to Data Source URL and put in your username (remember to create the empty database and a user before trying to access it from *OOo*). Next click on the Tables tab. There won't be anything there (because you haven't created any tables yet). Guess what we're going to do now? That's right – it's time to create the data. If you're a command-line freak (and I must admit I am), go back to your database server, log on to the database and create the tables. Don't forget that you can log on directly from your current server.

```
bainm@hector:~/ooobasic3$ mysql -hacamas -ubainm -pmypassword accounts
mysql > create database accounts;
create table accounts.customer (id int auto_increment,
surname varchar(50), firstname varchar(50),
address1 varchar(50), address2 varchar(50), city varchar(50),
county varchar(50),
country varchar(50), postcode varchar(50),primary key (id));
create table accounts.invoice (id int auto_increment,customer_id int,
sent_date date,paid_date date,primary key (id));
create table accounts.item (id int auto_increment,customer_id int,
invoice_id int,title varchar(50),details varchar(255),value double,
primary key (id));
insert into  accounts.customer
(surname,firstname,address1, address2,city,county,country,postcode) values
('Smith','John','The Big House','1 The Street','Thistown','Thisshire','UK','TH1 1HT');
insert into  accounts.customer
(surname,firstname,address1, address2,city,county,country,postcode) values
('Jones','Mary','Building A','Industrial Est.','Hereton','Herehire','UK','HE1 1EH');
insert into accounts.item (customer_id,title,value) values (1,'A fine piece of work',500);
```


LXFS11.scripting Sec2:83 22/8/07 10:47:47



You'll need to know which database you want to access from *OpenOffice.org*.

```
insert into accounts.item (customer_id,title,value) values (1,'A great job',500);
insert into accounts.item (customer_id,title,value) values (2,'Day 1',1500);
insert into accounts.item (customer_id,title,value) values (2,'Day 2',1600);
```

If that doesn't appeal, just use the Data Source Administration form, go to the Tables tab and click on the New Table Design button. You can use the Table Design form to easily create any tables that you need.

Using your new tables

We've spent quite a bit of time on the database side of things – simply because that has to be right before you start. Everything else will just fall into place. Now we can take a look at our first database-using macro. If you were paying attention in the last section, you'll be used to the OOo *CreateUnoService* function (you have been practising, haven't you?). We're going to be using it again here, this time to get access to *OOo*'s data **RowSet**. This is an *OOo* name for the record set, and it allows you to run queries on the database and retrieve information from it.

```
RowSet = createUnoService("com.sun.star.sdb.RowSet")
```

Now, all you have to do is to tell the **RowSet** about the database that you want to connect to (*ie* the one you set up in Data Source Administration), give it your username, password and the query that you want to run. The **RowSet** then obtains the result of the query and makes it available for you to use.

Therefore you probably want to do something like

```
sub main
   sql1
end sub
Sub sql1
   Dim RowSet
   RowSet = createUnoService("com.sun.star.sdb.RowSet")
   RowSet.DataSourceName = "Accounts"
   RowSet.User="bainm"
   RowSet.Password = "password"
   RowSet.Command = "SELECT count(*) c FROM item"
   RowSet.execute()
   RowSet.next()
   MsgBox "There are " + rowSet.getString(1) + " items"
End Sub
```

This is fine, but have a look at the next example:

```
Dim RowSet
Sub Main
        connectToDatabase ("Accounts", "bainm", "kawasaki")
        sql1
End Sub
Sub connectToDatabase(database as string, username as string, password as string)
   RowSet = createUnoService("com.sun.star.sdb.RowSet")
   RowSet.DataSourceName = database
   RowSet.User = username
   RowSet.Password = password
End Sub
Sub updateRowSet(sql as string)
   RowSet.Command = sql
   RowSet.execute()
 End Sub
Sub sql1
  updateRowSet("SELECT count(*) c FROM item")
  RowSet.next()
  MsgBox "There are " + rowSet.getString(1) + " items"
End Sub
```

With the second example, it becomes very easy to extend the functionality of the macro. Take a look:

```
Sub sql2
        updateRowSet("SELECT id, surname, firstname  FROM customer")
        while RowSet.Next()
                MsgBox "Customer No. " + rowSet.getString(1) + " " + rowSet.getString(2) + _
                " " + rowSet.getString(3)
        wend
End Sub
```

Writer reports

So far we've seen just how easy it is to access a database from a macro and to display the results. But we haven't really seen anything that you couldn't do as easily directly from the command line. If you cast your mind back to the first section, you'll recall that we were writing directly to *OOo Writer* documents. That seems the sensible thing to do now, using information from our database.

The great thing is that we can start doing very impressive things with very little new code. We've already got the **loadNewFile** subroutine (we set off with that in the first section, and modified it in the second section of this mega-tutorial) for creating a new *Writer* document, and we have the **add_paragraph** subroutine for writing to the document (and don't worry – the code that you need is on the coverdisc). All we have to do is add simple subroutines to create reports from the information in the database. Here's a simple way to create a document containing a list of all of the customers in the **Accounts** database:

```
Dim RowSet
Sub Main
        connectToDatabase ("Accounts", "bainm", "kawasaki")
        loadNewFile
        createCustomerReport
End Sub
Sub createCustomerReport
        updateRowSet("SELECT id, surname, firstname  FROM customer")
        while RowSet.Next()
                add_paragraph("Customer No. " + _
                rowSet.getString(1) + " " + rowSet.getString(2) + " " + rowSet.getString(3))
        wend
End Sub
```

Really, that's all there is to it. The process is as simple as that: send your query to the database, then display the result in a document. End of story. Well, not quite. As we identified in the first section of this tutorial, you don't really want to have to change the Main subroutine every time you want to run a new report unless you're a masochist. Again, the key thing here is to build yourself a dialog box to control the jobs that need to be done.

This time you won't hard code the contents of elements such as list boxes. No, this time you'll load them directly from the database. Let's say that you've added a list box and called it

LXFS11.scripting Sec2:84 22/8/07 10:47:48

lstCustomers in a dialog called dlgAccounts. What to load it with? You're there ahead of me: we can just send a query to the database requesting a list of customers:

```
        updateRowSet("SELECT surname, firstname  FROM customer")
```

Then you can loop through the record set, loading the list box with info as you go:

```
        lstCustomers.AddItem(rowSet.getString(2) + " " + rowSet.getString(1), i)
```

Have a look at the subroutine **loadlstCustomers** that's at **www.linuxformat.co.uk/special/ooo/qdb** to see how this works.

You can use this new list box as the filter for any reports that you wish to great. For example, if you wanted to see all of the items bought by a particular customer you would use the **selectedItem** property of the list box to obtain the text that's been selected. You could then use that to build an SQL statement, like this:

```
sql = " select title,value from customer, item " + _
        " where customer.id = item.customer_id " + _
        " and concat(customer.firstname,concat(' ',customer.surname))  = '" + _
        lstCustomers.selectedItem +"'"
```

Even better – build the SQL into a function. Why? This way you can use the query in any of the subroutines that you write without having to rewrite any code. Now you can add a button to the dialog box, associate a subroutine to the button and start making use of this. For a start, get the subroutine to output a message box so that you can see the SQL statement that you've built. When you're happy with that, use the SQL to load a new record set, then write this all to a *Writer* document. Scope out **cmdItemReport** on the disc to see this in action.

I'm sure that you can see just how easy this all is (and that's the key thing to remember – this is easy), and that automating the extraction from a database into an *OOo Writer* document is pretty simple. It probably won't surprise you to learn that it's just as easy with the spreadsheet program *Calc*. The interactions with the database are just the same. The only difference is that you have to write to individual cells rather than paragraphs – if anything this gives you even more flexibility in the way that you can lay out your information.

So I'll leave it to you to work out what to do now – we've already discussed all that you need in the previous couple of sections. And if you really still don't know what to do, just look on the URL at the end of this section – it's all there waiting for you to use.

A media library

To finish today, we'll just look at a simple application – one in which you can store and view a library of all of your CDs, DVDs, LPs or books.

Start by creating the tables in your database. You'll have to ask yourself a question: do I put all of my tables in my original database or do I create a new database for each project that I'm working on? I'd recommend the latter – you'll find it much easier to manage all of your information this way. However, if you do choose this method don't forget to add a reference for your new database to your **/etc/odbc.ini** file, then add it as a new data source into *OpenOffice.org*. You'll also need to instruct your macro to use the new database by changing **connectToDatabase ("Accounts", "bainm", "kawasaki")** to **connectToDatabase ("library", "bainm", "kawasaki")**.

Next thing: don't be tempted to try to shove everything into a single table – you're just asking for problems if you do. What kind of problems? Well, let's look at a simple example – a field containing a name. You know that 'Bill Gates', 'William Gates', 'B Gates', and 'Evil Overlord of Darkness' all refer to the same person, but your computer doesn't. This can make querying the data very difficult. Look at this table:

Table: item

Title	Author
Cat's Cradle	Kurt Vonnegut
Slaughterhouse 5	K Vonnegut

Instead of this, you could use two tables – one with the item details, the second with the Author details:

Table: item

Title	Author ID
Bagombo Snuff Box	1
The Sirens of Titan	1

Table: author

ID	Name
1	Kurt Vonnegut Jr.

This way, instead of having to remember every possible spelling of the author's name, all you need is the author's ID number.

Similarly, you don't want to store the words 'cd' or 'lp' or 'book' in the table containing the title. Instead use something like:

Table: item

Title	Media ID
Mind Bomb	2
Zen and the Art of Motorcycle Maintenance	1

Table: media

ID	Type
1	Book
2	CD

Now, all it takes is a little crafty SQL so you can get useful information out of the database:

```
select item.title, author.name, media.type
from item, author, media
where item.author_id = author.id
and item.media_id = media.id;
```

Next use this SQL in a subroutine to fill a spreadsheet with the results from the query – look at the files available from the URL overpage to see just how this works (you'll also find the SQL to create your new database as well as an example **/etc/odbc.ini**). If you examine the code you'll find that the macro does not include

"By their own follies…"

If you're wondering about my choice of host names – they're all from Homer's *The Iliad*. Much as I love *The Lord of the Rings* (the source for most host names), I find it amazing that a story from the Bronze Age has as much relevance today as it ever did, and that humanity hasn't really changed that much over all that time.

LXFS11.scripting Sec2:85 22/8/07 10:47:49

hard-coded column numbers when writing to the spreadsheet; rather, the **RowSet.Columns.Count** property is used to create a loop. So what? Well, this means that it doesn't matter if you change the number of records obtained from a query – the macro will automatically insert the correct number of columns into the spreadsheet.Now *that's* handy!

Filtering data

"But I don't want to see everything in the database!" I hear you cry, "I want to be able to see only CDs or only books, or just a single artist's work." Calm down, it's easily done – if you create a new form you can add list boxes and populate them from the **author** and **media** tables (just like we've already done in the accounting example). These list boxes can now be used build filters for the query. The files at **www.linuxformat.co.uk/special/ooo/qdb**, find **showFilteredLibrary** show you how to use optional inputs to build such a filter and then to display the results in the spreadsheet. To add new items, authors or media types to the database you need an insert statement, such as

```
insert into  library.author (name) values ('Hawkwind');
insert into  library.item (title,author_id,media_id) values ("The Ambient Anarchists',4,1);
```

This can easily be done on the command line, but with the knowledge that you've picked up in this in-depth tutorial, you should be able to build forms to do the job for you.

A little homework

Have a look at your normal day-to-day tasks and see which ones could be automated in the ways that we've been looking at. That's nothing to do with 'improved productivity' – this just comes down to pure unadulterated laziness. I'd also recommend you look into update queries – there's no reason that you can't write to your database as well as reading from it. There's more example code for this section at **www.linuxformat.co.uk/special/ooo/qdb**.

OOo Basic Tips and tricks

Take your macro knowledge further with our pearls of wisdom...

Over the past three sections, we've automated tasks and reports, written to documents from the command line, sped up data interrogation and more. This section should build on that know-how that you've picked up: there's 14 OOo Basic tips and tricks that will help you get more from your macros, whichever program you use them with.

1 Get your data structure right

Strictly this isn't exactly an OOo Basic tip, it's just a good programming tip: think about your data before you even think about doing any coding – it'll save you a heap of problems later on.

Let's look at a common example that you're going to come across using *OOo* macros: a database containing a list of names and addresses. If you're new to databases you might be tempted to put these in a single table, say one containing five fields – name, address, telephone number, mobile phone number and email address. That seems sensible enough. But let's think: imagine that you're dealing with some large organisations. You might end up with details for several people at the same address, and straight away you've got massive duplication of data. Not a good idea – you'll be using more disk space than you need and there's a much greater likelihood of data being incorrect. Look at this example:

Table - Customer	
Name	**Address**
Joe Thwaites,	Unthank, Cumbria, CA11 9TG
Joseph Thwaites,	Unthank, Cumbria, CA10 9TG

Given that there is an Unthank in CA11 and one in CA10, and that Thwaites is a common surname around here, we don't know if this is two men by the name of Joe Thwaites, or one Joe Thwaites with a spelling mistake in the postcode. By designing the database well you can minimise this type of problem.

I'm not going to get into database design here, but there is a simple rule that you can use: make sure that you only enter any one piece of information once. So instead of creating one big table, we could split the data into two tables:

Table - Customer			
ID	First_Name	Surname	Address_id
1	Joe	Thwaites	1
2	Joseph	Thwaites	2

Table - Address			
ID	Town	County	Postcode
1	Unthank	Cumbria	CA11 9TG
2	Unthank	Cumbria	CA10 9TG

Now you can see that there are two customers with the same name, but with different addresses. Actually, Joe (who calls at our house every Tuesday with his mobile video shop) actually lives in Langwathby, but that doesn't work as well for the example.

2 Make the most of the database

With your database constructed correctly you can avoid one of the biggest clangers: hard coding values into your macros. Let's say you've got a form for adding addresses to a spreadsheet, and that one of the columns is to contain the county. One option is to type all of the details directly into the sheet – but we don't want to do that, do we? No, far better to use a dialog box and macros to do all of the hard work.

So back to that data entry – obviously the best thing to do is to have a list box containing all of the county names. More choices for you: choose A to write a macro with all of the county names in an array then load a list box from this, or B to store the county names in a table on the database and load the list box from that. A big, loud "Baaaarp" if you chose A, and a nice warm, squelchy feeling if you chose B.

Why is B the better solution? Because things change ("When I wur a lad there wurnt none of this Cumbria rubbish, it wuz Westmulund and Cumbalund, my lad – proper names they wur"). You don't want to have to edit your code every time a little bit of data is added, so store your data in the database and let the macro do all the work for you. And that takes us neatly on to:

3 Automate with dialog boxes

If you're happy maintaining your data from the command line, that's great. But if you don't, or if you want to give the job to

LXFS11.scripting Sec2:86 22/8/07 10:47:50

someone else, consider using dialog boxes to do the updating for you. This way, it's possible to add data by writing a macro that builds an **insert** statement from the contents of a text box, thus:

```
SQL = "insert into county (name) values ('" + textbox.value +
"')"
```

You can also change data by using a list box, a text box and an insert statement:

```
SQL = "update county set name = '" + textbox.value + "'" _
        + " where name ='" + listbox.value + "'"
```

4 Work with OOo 2.0

Before you start creating applications, make sure you're using *OpenOffice.org 2.0*. You may find that your favourite distro doesn't have the most up-to-date version of the suite installed by default – if you use Debian Sarge, for instance, the choice is *OOo 1.1.3*. You should install version 2.0 before you get down to coding. Why? Because OOo Basic has been extended for that version; in particular, you can now use code to add menu items.

If you've created macros using *OOo 1.x.x* and then upgrade to *OOo 2.x.x*, you will find that your modules have disappeared – all of that blood, sweat and tears gone, for nothing. Don't despair – all is not lost. Start whichever terminal that you like to work in, go to your home directory, and run **ls -la**. In among all your other files and directories you should see something like

```
drwxr-xr-x  3 bainm bainm  4096 1999-10-20 04:08
.openoffice
drwx------ 3 bainm bainm  4096 2006-05-31 20:30 .openoffice.org2
```

Of course those lovely *OpenOffice.org* developers haven't zapped your old macros – they're just stored in a different directory. If you hunt through the two directories you'll come across something like **.openoffice/1.1.3/user/basic/Standard** and **.openoffice.org2/user/basic/Standard**. These are the directories where your modules and dialogs are stored, so before you get coding make sure that *OOo 2.0* has access to your old macros with

```
mv .openoffice.org2/user/basic/Standard .openoffice.org2/user/
basic/Standard_old
ln -s .openoffice/1.1.3/user/basic/Standard  openoffice.org2/user/
basic/Standard
```

Restart *OpenOffice.org*, and it should pick up your old macros.

5 Run macros from the menu bar

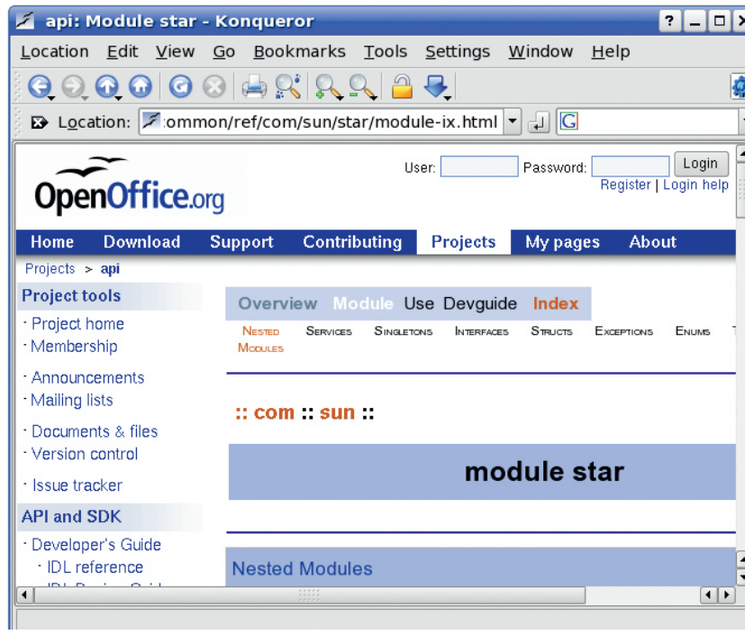
You're already used to running macros by using the Run button in the Basic window (the one where you write the code). You also know how to run a macro by going to the Menu and clicking on Tools > Macros > Macro... before selecting the macro that you want to run. That's fine, but it's a bit long-winded – especially if there's a macro that you use a lot. Why not just add a link to useful macros into the *OOo* menu? Good idea!

Click on Tools in the *OOo* menus then Configure.... *OOo* will display its Configuration form – make sure that you've got the Menu tab selected. Use the bottom half of the screen to select the macro that you want to add to the menu. Use the top half of the screen to find the position for your new menu item, and then click on New to create it.

Now you will be able to call up your macro easily and whenever you want. If you're more of a Ctrl+Shift+G or F5 type, it's possible to assign macros to keystrokes instead. Again, use *OOo's* Configuration form, but this time use the Keyboard tab instead. To run a macro when a document opens, yes, you've guessed it – we just use the Configuration form. If you select the Events tab you'll see all of the possible events that you can assign macros to – just select Open Document.

6 Add menus automatically

If you use or have upgraded to *OOo 2.0* you'll be able to create additional menu items from within a macro. To do this, make use



Go online to get more information on the modules built into *OpenOffice.org*.

of *OOo*'s *com.sun.star.beans.PropertyValue* object property. For example if you want to add the subroutine **showMessage** from a module named **ooobasic4** you could use:

```
MenuItem(0).Name = "CommandURL"
MenuItem(0).Value = "macro:///Standard.ooobasic4.
showMessage()"
MenuItem(1).Name = "Label"
MenuItem(1).Value = "Do whatever you want direct from the
menu"
Menu.insertByIndex( 0,MenuItem)
```

To see this actually working, have a look at **setUpMenu** in the files at **www.linuxformat.co.uk/special/ooo/tips**. And before you ask, you can't delete any of the default's and you can only add your own items as sub-items on to the ordinary *OOo* menus.

7 Split lines of code

This tip will improve the readability of your code. Readability. Is that actually a word? Anyway, you can improve it by splitting long lines of code with an underscore to spread them over multiple lines. We can turn

```
MenuItem = CreateMenuItem( "macro:///Standard.ooobasic4.
showMessage()", "Show a test message" )
```

into

```
MenuItem = _
        CreateMenuItem( "macro:///Standard.ooobasic4.
showMessage()", _
        "Show a test message" )
```

This has no effect on the way that the code works, it just looks nicer – sorry, I mean it improves readability.

Using an underscore won't work, though, if you're trying to split a string, such as

```
SQL = "select customer.firstname, customer.surname, address.
town, address.county, address.postcode where customer.address_
id = address.id"
```

To split this into a more visually appealing format you also need to use a plus sign:

```
SQL = "select customer.firstname, customer.surname" _
        + ", address.town, address.county, address.postcode" _
        + " where customer.address_id = address.id"
```

Again, it won't affect how the macro runs, but it does make it much easier for you to see what's going on.

LXFS11.scripting Sec2:87 22/8/07 10:47:50

8 Remember modules and libraries

Rather than save all your work into one massive module, aim to organise your macros into modules according to their functions, then give the module a name that reflects its purpose – macro1, macro2, macro3 *etc* is very boring and doesn't tell you much. If you give some thought to the names (you might use 'accountingMacros' or 'customerCareMacros', say) you'll thank me for it. Don't go too far – try not to end up with dozens of modules each with just one or two macros.

As you write more modules it might be worth grouping them into libraries. Again, try to give them useful names.

9 Use comments liberally

At the time when you're writing a macro, you tend to think that you're coding it in a sensible and logical manner. More often than not, when you come back to it six months or a year later when it is no longer fresh in your mind, things don't seem so clear. I'd recommend you leave notes to yourself as you're going along – just to explain what you're trying to achieve.

How do you add a comment? Start the line with **REM** or a single quote, like this:

```
REM This is a comment
' So is this
But this line would cause you a problem
```

You can also add comments at the end of a line:

```
msgbox "This is a message" 'but this is comment
```

On the other hand, don't put in so many comments that you lose sight of the code – you don't have to explain *everything:*

```
REM The aim of the next piece of code is to feed visual information
REM back to the user. It will display text in a simple form, and will
REM expect the user to confirm that they have read the screen. The
REM program will not continue until they have carried out this
REM confirmation.
msgbox "Hello user"
```

10 Try built-in dialog boxes

There's no reason why you have to do all of the work yourself when it comes to building dialog boxes – there are already some useful ones built in to *OpenOffice.org* that you can make use of. Let's say that you want the user of your macro to be able to manually select a folder, perhaps to save an output. All you need to do is use the *FolderPicker* dialog:

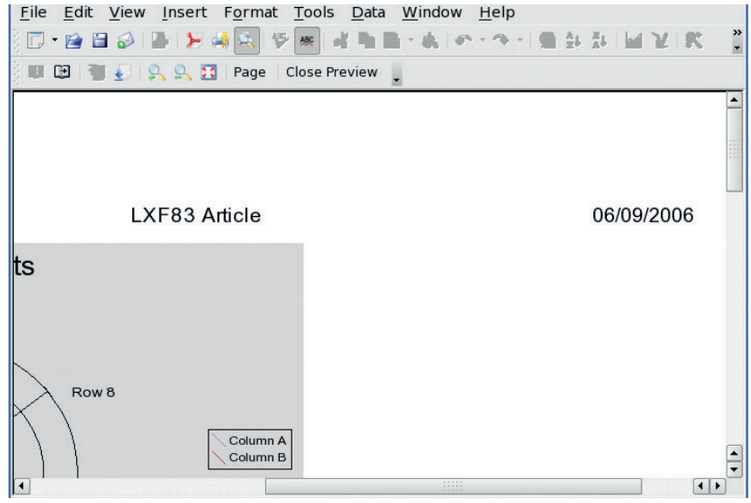
```
Dim FolderDialog, UserFolder
FolderDialog = CreateUnoService("com.sun.star.ui.dialogs.FolderPicker")
UserFolder = FolderDialog.Execute()
```

To find out what other dialogs you can use, consult the *OpenOffice.org* website – in particular, **http://snipurl.com/rh5z.** On this page you'll find the details of each dialog box. There aren't many, but you should find them useful and time-saving.

While I'm on the subject of **com.sun.star**... These are *OOo*'s set of built-in modules. You've no doubt noticed in these tutorials that whenever we make use of one of the *OOo* objects we always use it in the format of

```
MyObject = CreateUnoService("com.sun.star.something.theother.someobject")
```

That's straightforward enough, but then you've got the problem of finding out which other objects you can make use of. Again, you'll be able to find the information that you need on the *OpenOffice.org* website at **http://snipurl.com/rh61,** where you'll find a listing of all of the modules contained in **com.sun.star**.



Use Print Preview to see the header and footer that your macro has created.

11 Format your reports

Over this whole in-depth tutorial, we've looked at how to write to *Writer*, calculate in *Calc* and use data in databases. You can also use macros to make your work look professional with the minimum of effort. Imagine you've extracted your information from your database into a spreadsheet, you've carried out all of the work that you need to do (by using a macro, of course), and you're now ready to print your report for your managing director/degree supervisor (who's obviously going to give you a pay rise/good grades because of your fantastic work). What are the normal things that you'd have to add prior to printing? I'd say that typically this would be a title, a page number and a page count, oh, and the date. This is easily done:

```
oDoc = ThisComponent
oPageNumber = oDoc.createInstance( "com.sun.star.text.TextField.PageNumber" )
oPageCount  = oDoc.createInstance( "com.sun.star.text.TextField.PageCount" )
oDateTime   = oDoc.createInstance( "com.sun.star.text.TextField.DateTime" )
Now you need to get access to the header and footer:
oStyles = oDoc.getStyleFamilies().getByName( "PageStyles" )
oPStyle = oStyles.getByName( "Default" )
oHeader = oPStyle.RightPageHeaderContent
oFooter = oPStyle.RightPageFooterContent
```

You can write text to any of the panels in the header:

```
oHeader.getCenterText().setString( "LXF83 Article" )
```

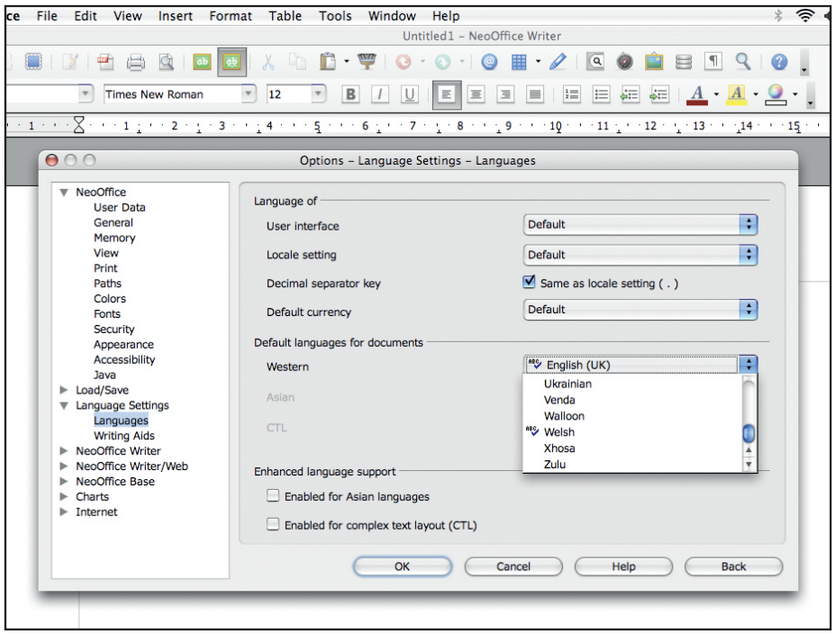
However, the functions (such as **DateTime**) need to be handled slightly differently, by making use of a cursor:

```
oCursor = oHeader.getRightText().createTextCursor()
oHeader.getRightText().insertTextContent( oCursor, oDateTime, True )
```

You can go on to combine both of these techniques to create more involved inserts:

```
oFooter.getRightText().setString( "Page " )
oCursor = oFooter.getRightText().createTextCursor()
oCursor.gotoEnd( False )
oFooter.getRightText().insertTextContent( oCursor, oPageNumber, True )
oCursor.gotoEnd( False )
oCursor.setString( " of " )
oCursor.gotoEnd( False )
oFooter.getRightText().insertTextContent( oCursor, oPageCount, True )
Finally, you need to update the page with your new information:
```


LXFS11.scripting Sec2:88 22/8/07 10:47:51



You can always use all the tips in this tutorial once you've localised your install of *OpenOffice.org* into another tongue – see p96 to find out how.

```
oPStyle.RightPageHeaderContent = oHeader
oPStyle.RightPageFooterContent = oFooter
```

12 Merge documents

Don't just create new ones – if you've got multiple documents you need to combine (*eg* when there's more than one person working on a project) use a cursor to combine these in a single document:

```
oCursor.gotoEnd(false)
oCursor.BreakType = xom.sun.star.style.BreakType.PAGE_
BEFORE
oCursor.insertDocumentFromUrl(SrcFile, argsInsert())
```

Obviously, you could pick these files up from a directory, or read a list from a table on a database, then loop through them adding one file at a time into the master. With all of your documents merged, a table of contents would be useful. Again, you could do this automatically:

```
oDoc = ThisComponent
oCurs = oDoc.getText().createTextCursor()
oCurs.gotoStart(False)
oDoc.getText().insertTextContent(oCUrs, oIndex, False)
```

13 Find everything you need

Maybe not *absolutely* everything, but if all you want is an alphabetical list of all the built-in *OpenOffice.org* functionality that's available to you, go to **http://snipurl.com/rh67**. You'll be able to view every function, property and service and the object that they belong to.

14 Learn from others

A good way to learn OOo Basic is to look at how other people write macros – you may not agree with them, but you'll get useful ideas. You could Google for 'openoffice.org macros', but start at the official *OOo* site. You'll find the examples at **http://codesnippets.services.openoffice.org**.and there's more example code for this section at **www.linuxformat.co.uk/special/ooo/tips**.

In his marathon in-depth tutorial, we've looked at a few examples, but I hope you can see just how easy it is to start building powerful and efficient macros. Thanks to this *Linux Format* special edition, you've got all of *OOo*'s resources at your fingertips, so enjoy – the world's your *OOoO* (that's *OpenOffice.org* oyster). LXF

LXF is the Answer!

Linux Format magazine features an *Answers* section every issue where our panel of experts solve all your knottiest hardware and software problems. If you want to contact the magazine with your problem, and also have the chance to win a prize, submit your question to **lxf.answers@futurenet.co.uk**. Also, why not check out the forums at **www.linuxformat.co.uk**? One of our expert readers may be able to help out.

Joe's magical mystery tour

Remember I told you about Joe Thwaites in tip 1? Well, just for a moment, let's think about Joe driving about in the Lakes, doing his rounds. Whenever he stops at a customer's house he opens up the back of his van to reveal the latest stock of videos, DVDs and games. Imagine, if you will, a wooden panel built into the van. In the middle of the wall is a 17-inch Flatron screen, and on it *The Lion, The Witch and The Wardrobe* is playing. If you look carefully you can see the front of a PC poking out at the bottom of the panel, along with a keyboard and a mouse.

Obviously you're wondering about the PC – it's running Debian 3.1 (thanks to *Linux Format* issue 70), with *Kaffeine* installed to play the DVDs. Both the PC and the Flatron monitor run off a Belkin DC-AC converter.

Interesting, but what does all this have to do with *OpenOffice.org*? Good question. Before Joe sets off on his daily run he'll start an *OOo Calc* document with a macro assigned to the 'Open Document' activity. This macro is almost the same as the **connectToDatabase** subroutine that we've already developed – the difference is that this time it has error handling built into it. Why? Because this time the main *MySQL* database is not on the PC in the van – it's on a PC in Joe's house. Joe's main PC runs a wireless network, and the van PC has a Wi-Fi antenna. Now, if Joe's at home the macro will connect to his database; if he's not the *Calc* document will just open

Once the macro detects that the database is present (that is, a connection can be made) it looks on one of the sheets in *Calc* (called Daily data). Cell A1 contains a date reference. If that date is less than today's, the macro will go to a second sheet and run a subroutine: **uploadYesterdayRoute**. When that's done, another macro, **downloadTodayRoute**, will query the database and extract the details for customers to be visited today – this data includes the videos, DVDs, and games that are currently being rented at each address.

As Joe travels around the county he updates the *Calc* spreadsheet, recording who rents what from him. If anyone is out when he calls, another subroutine called **printReminder** creates a *Writer* document containing the rental details and when Joe's next due in the area. The subroutine also prints off a copy (to leave for the customer) on a printer in Joe's cab. Next morning the process all starts again. However, for Joe this means that he has a complete history of his rentals. Also, he can start to analyse his routes – seeing what's most profitable for him, Cumbrian trends in films and so on. It even means that he can cater for the audience he's visiting on any given day.

And why is Joe relevant to you? Bear in mind that many large companies each spend hundreds of thousands of pounds developing such systems as the one we've dreamed up for him. You can do it for free with Linux and a little bit of knowledge.

LXFS11.scripting Sec2:89 22/8/07 10:47:52