

Introduction

The macro capability in Ability is based on VBScript. To be a little more exact, Ability is an ActiveX Scripting Host. Other examples of Scripting Hosts include Microsoft's Internet Explorer, Internet Information Server and the Windows Scripting Host that ships with Windows 98 and Windows 2000.

VBScript can be thought of as a lightweight version of Visual Basic for Applications. Ability itself is programmable through an OLE automation interface.

What it all means

You don't really need to be familiar with all the above terminology. It's what you can do with it that counts. The "macros" in Ability will allow you to control the Ability applications for such tasks as adding standard text to documents and looking up a user's name and address for a letter. In addition, because it is based on industry standards, you can easily communicate with other software packages that use the same standards (which includes nearly all new application software from Microsoft).

For example, you can create a macro that takes an email address out of a database and uses Microsoft Outlook to send the email. The same is true in the opposite direction - you can use an external environment to program Ability (for example, Windows Scripting Host, Visual Basic, Delphi and any other application that supports OLE automation).

See:

[What you need](#)

[Getting started](#)

[How the macros are created, stored and executed](#)

What you need

To run the Ability macros, you'll need to have the Microsoft script engine installed. This will already be installed if you have Windows 98 (or later) or if you've installed Microsoft Internet Explorer 4.0 (or later).

If you haven't got the script engine, or are not sure if you have it, or you want to get the latest version, you can download it from the Microsoft web site:

<http://www.microsoft.com/msdownload/vbscript/scripting.asp>

Note the link is above is liable to change – if it is out of date as you read this, try searching on the Microsoft web site for "vbscript" or try the scripting home page:

<http://msdn.microsoft.com/scripting/>

Getting started


To create and edit macros, use the Tools/Macro command from the menu.

Use the Macro dialog (Tools/Macro/Macros) to organize your macros and create new ones and the Macro Editor (Tools/Macro/Macro Editor) to edit your existing macro. For an introduction to this process see the [Quick tutorial](#) for a Write macro.

To program Ability, you'll need to know two things:

1. The commands to drive Ability

See how to program each module: [Write](#) , [Database](#) and [Spreadsheet](#) .

Use the Class Browser to examine the object model of Ability – to see this, select Tools/Macro/Macro Editor and use the Class Browser button  .

See the [Example macros](#) for real life, fully working macros.

2. The programming language (VBScript)

See the [VBScript introduction](#).

How the macros are created, stored and executed

To create a macro in Ability, use the Tools/Macro command. Macros can either be stored in a particular file (i.e. a Write document, spreadsheet or database) or in a globally accessible file. Global macros are available whenever you run that particular application (Write, Spreadsheet, Database) and can be found, for back up purposes, in the following locations:

c:\progam files\ability office\abwrite.abd
c:\progam files\ability office\abspread.abd
c:\progam files\ability office\abdata.abd

The above assumes you installed Ability into the default location.

Some macros will be a series of steps requiring no user interaction - you just run it and it does the job in the same way each time. With others, you will need to get feed back from the operator. Ability allows sophisticated forms to be created that can deal with user input. The forms are activated by macros and can themselves contain scripting instructions to intelligently process the user selections.

Once the macro has been created, it can be run in one of five ways:

- Directly from the menus using Tools/Macros
- From the shortcut menu (set the "Quick Macro" attribute in the Tools/Macros dialog)
- From a keyboard shortcut (set this under Tools/Customize)
- From a button on the toolbar (also set using Tools/Customize)
- Externally (as you start an application or from an external program)

Write macros

The most common need for macros in Write is to manipulate text - add text, change fonts, copy text and so on. There are two main objects that can be used for dealing with the text of a document:

- [The Text collection](#) - gives access to the entire text of a document
- [The Selection object](#) - gives access to text, fonts, paragraph attributes, tab settings and styles of the current selection

For some tasks (inserting text, deleting text) you could use either the text or the selection object. If you want to specify any font or paragraph settings, the Selection object must be used. If you want to check the entire text of a document for a particular word then the Text object is the one to use.

There are also objects and collections to allow you to manipulate (create, open, print) documents - see [Working with documents](#).

Working with the text collection

The text collection contains the details of every character in a document. This includes special characters such as the Enter key and the Tab key. The easiest way to picture the text collection is as a single long, continuous stream of characters. You can reference, insert, and delete characters anywhere in this text stream.

See:

[Counting the characters](#)

[Examining each character in a document](#)

[Dealing with document chunks](#)

[Inserting Text](#)

[Deleting Text](#)

Counting the characters

Consider a new document. The operator types "Hello" on line 1 and then presses the Enter key and types "?" on line 2 so the document looks like this:

```
Hello  
?
```

How many characters does the document contain? Easy, just use the Count property of the text box. For example:

```
MsgBox ActiveDocument.Text.Count
```

displays the answer 7 (that is, 5 for hello + 1 for the question mark + 1 for the Enter key). Note that ActiveDocument is needed before the Text object can be referenced. It doesn't need to be the current document - any document object will do.

Examining each character in a document

Any character in a document can be referenced directly. Consider a document that contains nothing more than the following text on lines 1 and 2:

```
Hello  
?
```

This code will access individual letters within the document:

```
MsgBox ActiveDocument.Text.Item(0) ' returns 72  
MsgBox ActiveDocument.Text.Item(1) ' returns 101  
MsgBox ActiveDocument.Text.Item(2) ' returns 108
```

What's going on?

The Text object is actually returning a number code. So "H" = 72, "e" = 101 and "l" = 108. Anyone familiar with ASCII codes may recognize these numbers. Fortunately, you don't need to remember what the codes translate to - there's a built-in function to VBScript function, Chr(), that does this for you. For example:

```
MsgBox Chr(ActiveDocument.Text(0)) ' returns H
```

Note that the "Item" has been dropped - it's not necessary since Item is the default property of the Text object.

So let's examine the entire document:

```
For i = 0 to ActiveDocument.Text.Count - 1  
    MsgBox "Character number: " & i & ", code: " & ActiveDocument.Text(i) & ", text: " &  
    Chr(ActiveDocument.Text(i))  
Next
```

This will produce the output:

```
Character number: 0, code: 72, Text: H  
Character number: 1, code: 101, Text: e  
Character number: 2, code: 108, Text: l  
Character number: 3, code: 108, Text: l  
Character number: 4, code: 111, Text: o  
Character number: 5, code: 13, Text:  
Character number: 6, code: 63, Text: ?
```

Note there is no actual character to represent the Enter key (the output is blank).

A better way of coding the above is to make use of the With....End With VBScript statement so you don't have to keep writing *ActiveDocument.Text* repeatedly.

```
With ActiveDocument.Text  
    For i = 0 to .Count - 1  
        x = .Item(i)  
        MsgBox "Character number: " & i & ", code: " & x & ", text: " & Chr(x)  
    Next  
End With
```


Dealing with document chunks

If you know the start and end position of the text you want, you can simply get at it using the Mid function, which has the general form:

`Mid(start, length)`

For example:

`MsgBox ActiveDocument.Text.Mid(0, 10)` ' returns the first 10 characters of the document

`i = ActiveDocument.Text.Count`

`MsgBox ActiveDocument.Text.Mid(0, i)` ' returns all the characters of the document

`MsgBox ActiveDocument.Text.Mid(10, 20)` ' returns 20 characters starting at position 10

Inserting Text

To insert text in a document, just specify a starting position. For example:

```
ActiveDocument.Text.Insert(0, "The start") ' puts the text at the beginning
```

```
i = ActiveDocument.Text.Count
```

```
ActiveDocument.Text.Insert(i, "The end") ' puts the text at the end of the document
```

Deleting Text

To delete text from a document, specify a starting point and a length. For example:

`ActiveDocument.Text.Delete(10, 3)` ' deletes 3 characters, starting at position 10

`i = ActiveDocument.Text.Count`

`ActiveDocument.Text.Delete(0, i)` ' deletes the entire document contents

Working with the Selection object

The selection object refers to the currently selected text, that is the text that is highlighted. If no selection is current, the selection object still functions and works with the current cursor position.

The selection object is an application level object so can be used without any qualifiers (like ActiveDocument).

Here are some of the more important properties and methods:

Properties

<code>Document</code>	Read only; the document to which the selection object is associated.
<code>End</code>	Set or read the ending position of selection.
<code>Start</code>	Set or read the starting position of selection.
<code>Font</code>	Merged font of the selection, represented by Font object
<code>Paragraphs</code>	Collection of paragraphs in the selection.

Methods

<code>Collapse(Direction)</code>	Turn selection back into a cursor. Possible directions are <code>abForw</code> & <code>abBack</code> .
<code>Copy()</code>	Copies it to the Clipboard.
<code>Cut()</code>	Moves it to the Clipboard.
<code>Delete()</code>	Delete the selection.
<code>InsertBefore(String)</code>	Insert text before the selection and adds it to the selection.
<code>InsertAfter(String)</code>	Insert text after the selection and adds it to the selection.

The selection object makes it easy to insert text at the current cursor position. For example:

```
Selection.InsertAfter("Inserted text")
```

After text has been inserted it will itself be selected. This makes it easy to adjust the font properties. For example:

```
Selection.InsertAfter("Inserted text")
```

```
With Selection.Font
```

```
    .Bold = True
```

```
    .Italic = True
```

```
    .Underline = True
```

```
    .Size = 18
```

```
    .Name = "Times New Roman"
```

```
End With
```

```
Selection.Collapse(abForw)
```

The last line returns back to a normal cursor.

If you want to work with a specific part of a document, use the `Start` and `End` properties. For example, the following copies the first 10 characters of a document to the clipboard and then tells the user how many characters he has copied:

```
Selection.Start = 0
```

```
Selection.End = 9
```

```
Selection.Copy()
```

```
MsgBox "You copied " & _  
        Selection.End - Selection.Start + 1 & _  
        " characters"
```

Or if you want to insert text at the end of a document, use the selection object in conjunction with the

text object:

```
Selection.End = ActiveDocument.Text.Count  
Selection.InsertAfter "End of document"  
Selection.Collapse(abForw)
```

Working with documents

Macros can be used open, close, create and print documents. Documents can be manipulated on two levels:

See:

- [Documents collection](#) – a container for all the open documents (note the plural).
- [Document object](#) – an individual document.

Documents collection

With the documents collection you can open and create documents and refer to particular documents by name or number.

For example:

```
Documents.Add("NORMAL")
```

The above line creates a new document based on the normal template and makes it the current document. This is exactly the same as selecting the New button on the toolbar.

```
Set mydoc = Documents.Add("NORMAL")
```

This has the same effect but also save a reference to the new document in a variable called *mydoc*. Mydoc is a Document object (see [Document object](#) for details) and you can use it, for example, to add text and print:

To open an existing document, use the Open method. As with the Add method, you can choose to save a reference to the document or not (which would depend on the other requirements of your macro):

```
Documents.Open("c:\my documents\myfile.aww")
```

or

```
Set mydoc = Documents.Open("c:\my documents\myfile.aww")
```

If you are opening several files in the same directory (or just want to change the current directory) you can use the following:

```
ChangeFileOpenDirectory("c:\my special folder")
```

The documents collection contains useful information about open documents as the following macro demonstrates:

```
MsgBox "There are " & Documents.Count & " documents, names to follow."
```

```
For each doc in Documents
```

```
    MsgBox doc.GetTitle
```

```
Next
```

Individual documents can be referred to by name or number. Suppose you had opened two documents, Write1.aww and Write2.aww, in that order then:

```
Documents(0).Activate
```

```
Documents("Write1.aww").Activate
```

are equivalent, as are:

```
Documents(1).Print
```

```
Documents("Write2.aww").Print
```

Document object

Here is a summary of the most important properties and methods of the document:

Properties

Selection	The Selection Object
Paragraphs	The Paragraphs collection.
Text	The Text collection
Styles	Styles collection.
Frames	Frames collection.
Fields	Fields collection.

Methods

Activate	Makes the document the current one.
GetTitle	Returns title of the document.
Copy	Copies selection to the Clipboard.
Cut	Moves selection to the Clipboard.
Print star, end, copies, collate	Prints the document.
Paste()	Copies Clipboard to the current position of the Document; Deletes selection if any.

The easiest way to "get hold of" a document is to use `ActiveDocument` to refer to the current document. For example:

```
MsgBox ActiveDocument.GetTitle
```

would show the name of the current document. `ActiveDocument` can be used with any of the above properties and methods.

You can also refer to the active document when creating or opening documents. See [Documents collection](#) for more details. For example, to create and print a document:

```
Set mydoc = Documents.Add("NORMAL")  
mydoc.Text.Insert 0, "hello from a macro"  
mydoc.Print 1, 9999, 1, False
```


A Write macro tutorial

This tutorial will show how to build a macro for a standard way of closing a letter (the "yours sincerely" bit).

1. In Write, create a new macro: select **Tools/Macro/Macros**
2. Type in "LetterClosing" in the "Macro Name:" box.
3. Click on the **Create** button.

You are now presented with the following:

```
Sub LetterClosing()  
' Put macro code here  
End Sub
```

Sub is short for subroutine and the name you entered follows it. This is the standard way of declaring a macro. The very last line of your subroutine is "End Sub". An error is generated if this is missing.

The line in-between begins with an apostrophe. You can put as much descriptive text into the macro as you want - just begin the line with an apostrophe. For now, you can delete the "Put macro code here"

Let's start with the simple problem of inserting some text, somewhere.

The macro command for inserting text is simply "Insert". But this is not enough on its own. You need to know that "Insert" *belongs* to something called "Text", which is the entire text of your document. How about "which document" - you could have several documents open so you need to specify which one. You could do this by name, but this would make the macro rather too specific. What you really want is to have it work with the currently active document. The way to refer to this is "ActiveDocument". So a command to insert text is as follows:

```
ActiveDocument.Text.Insert
```

What we've been describing is part of the "object model" for Ability.

To finish off, you need to specify what to insert and where to insert it, in the format:

```
ActiveDocument.Text.Insert position, text
```

where position is the number of characters from the beginning of the document and text is the actual text to insert. So:

```
ActiveDocument.Text.Insert 0, "Hello from Ability"
```

would insert "Hello from Ability" at the start of your document (the reference to position is zero based) and your complete macro will now look like this:

```
Sub LetterClosing  
  ActiveDocument.Text.Insert 0, "Hello from Ability"  
End Sub
```

Lets run the macro to see what happens:

1. Select a Write document (use the Windows menu).
2. Select **Tools/Macro/Macros**.
3. You'll see "LetterClosing " as a named macro - click on it to select it.
4. Click on the "Quick macro" checkbox so a tick appears (we'll see why in a minute).
5. Click the Run button to execute the macro.

The text should appear at the start of your document. In step 4. above, we asked Ability to remember the macro as a "Quick macro" - this means it will appear on the shortcut menu as follows:

1. Put the mouse pointer over the document and right-click.
2. Select Macros.
3. You'll see the LetterClosing macro available - select it.

The text will be inserted again at the beginning of the document.

Refining the macro to make it more useful

The text we want to insert should be at the end of the document rather than the beginning. So we need a way of working out where the end of the document is. One way is to ask Ability for the total number of characters in a document as follows:

```
ActiveDocument.Text.Count
```

So the following macro would always insert the text at the end of the document:

```
Sub LetterClosing
    pos = ActiveDocument.Text.Count
    ActiveDocument.Text.Insert pos, "Hello from Ability"
End Sub
```

Notice that we assigned a variable to count the number of characters in the document. Strictly speaking, it is not really necessary and we could have written it as a single line instead:

```
ActiveDocument.Text.Insert ActiveDocument.Text.Count, "Hello from Ability"
```

But the first way is easier to read (which is a good rule of thumb to apply when creating macros).

Supposing you had lots more similar commands you wanted to issue. Do you have to keep writing "ActiveDocument.Text" all the time? The answer is No. You can ask Ability to perform a series of tasks using the same "thing" (or you can call it object if you want). For example, the following macro gives identical results to the previous one:

```
Sub LetterClosing
    With ActiveDocument.Text
        pos = .Count
        .Insert pos, "Hello from Ability"
    End With
End Sub
```

There are some important points here. When using the With statement, the . (dot or period) must precede any items that refer back to the object being referred to. So it reads: "pos equals dot count" and "dot insert equals....". Also note that it is good practice to indent macro commands within "block" statements. This makes it easier to read.

Now let's make the macro put some real text in.

```
Sub LetterClosing
    With ActiveDocument.Text
        pos = .Count
        .Insert pos, vbCr & "Yours sincerely" & vbCr & vbCr & vbCr & vbCr & "Philip Roach" & vbCr &
"Technical Manager"
    End With
End Sub
```

What's happening! Imaging you were typing the text in real life - after the "Yours sincerely", you'd press the Enter key. You'd probably press it once (or twice) before you started "Yours sincerely" as well. This is what the vbCR does. It's short for "visual basic carriage return" and is a constant built-in to the VBScript language.

Also, the above example shows how to combine text strings using the "&" operator.

Having long strings of text combined in a single line as in the above example is not good practice (it's hard to read later). Here is a neater way of laying it out:

```
Sub LetterClosing
    With ActiveDocument.Text
        pos = .Count
        .Insert pos, vbCr & "Yours sincerely" & String(4, vbCr) & _
"Philip Roach" & vbCr & _
"Technical Manager"
    End With
End Sub
```

Instead of repeatedly chaining vbCr's together, we can use a built-in function (built-in to the VBScript language that is) called "String" that takes a character and repeats it any number of times. Also notice that the long line is split into 3 with an underscore used as a "continuation" character at the end of lines 1 and 2.

Spreadsheet macros

Most of the macros you will be writing for Spreadsheet will involve fetching and setting the values, formulas and formats for cells within a spreadsheet. You'll also need to know how to open and create workbooks and deal with worksheets.

See:

[Working with cells](#)

[Working with Ranges](#)

[Working with worksheets and workbooks](#)

Working with cells

When working with cells in a worksheet, you'll usually need to work with a known specific cell (e.g. A1) or work with a cell relative to the current cell (e.g. the cell to the right of the current cell). We'll begin with the current cell itself (a special case of the second type).

See:

[Using the active cell](#)

[Cut, copy and paste](#)

[Working with a specific cell](#)

[Working with cells relative to the active cell](#)

Using the active cell

The active cell is just the current cell - wherever you move to, or click on, in the worksheet. The way to address the active cell is, not surprisingly:

`ActiveCell`

A cell has lots of properties - a font, value, a formula, background color and so on. Here are some of the more important properties:

`ActiveCell.Value` - The simple text or number or date of a cell (as it is displayed in the sheet). If the cell contains a formula, then `Value` displays the results of the formula. For example:

```
If ActiveCell.Value = "0" Then
    ActiveCell.Value = "Zero"
Else
    MsgBox "Cell has value: " & ActiveCell.Value
End If
```

This converts a current cell containing "0" to "zero" and displays the value of current cell if it does not contain zero.

In a similar way, the current cells formula can be determined or set using:

`ActiveCell.Formula`

The `Formula` property returns either the formula (for example "`=a1 + a2`") or, if the cell does not contain a formula, the value.

Here are some more properties:

`ActiveCell.NumberFormat`

`ActiveCell.Font`

(for example)

```
ActiveCell.Font.Bold
ActiveCell.Font.Italic
ActiveCell.Font.Name
ActiveCell.Font.Size
```

Cut, copy and paste etc

As well as setting the properties of the cell, there are methods to perform cut, copy and paste, just like the Edit menu:

`ActiveCell.Clear ()`

`ActiveCell.ClearContents ()`

`ActiveCell.ClearFormats ()`

`ActiveCell.Copy ()`

`ActiveCell.Cut ()`

`ActiveCell.Paste ()`

Note that you need to include the brackets to execute these methods.

Working with a specific cell

To refer to a specific cell within the spreadsheet, use the Range object as follows:

`Range("a1")`

This is equivalent to the ActiveCell in terms of its properties and methods, so all of the following are valid:

`Range("a1").Value`

`Range("a1").Formula`

`Range("a1").Cut()`

Working with cells relative to the active cell

Suppose you wanted to take the contents of the current cell, do something with it and then put the result in the cell to the right of the current cell. The target cell's address is neither set in advance nor the active cell. So how do you refer to it? The answer is that you use a Cells property that treats the active cell as 1, 1 and refer to other cells by it's relative co-ordinates.

This is best seen by example. Suppose the current cell was C3, then:

```
ActiveCell.Cells(1, 1)  refers to C3 (i.e. itself)
ActiveCell.Cells(1, 3)  refers to E3
ActiveCell.Cells(3, 1)  refers to C5
ActiveCell.Cells(0, 0)  refers to B2
ActiveCell.Cells(-1, 0) refers to B1
```

Here's an example routine that takes the current cell and adds 1 to it and puts the result in the cell immediately to the right:

```
ActiveCell.Cells(1, 2).Value = ActiveCell.Value + 1
```

Of course, this code assumes that the active cell contains a number and if contains text instead, a run time error will be generated. Later we'll see a general technique for dealing with such errors.

Because the ActiveCell.Cells(row, col) notation returns an equivalent object to ActiveCell, all the same properties and methods can be applied. These are all valid:

```
ActiveCell.Cells(2, 2).Cut()
ActiveCell.Cells(2, 2).Formula = "=a1 + a2"
ActiveCell.Cells(2, 2).Font.Bold = True
ActiveCell.Cells(2, 3).Paste()
```

Ranges

Quite simply, a range is one or more cells. In fact there's no such thing as a "cell" object and all the previous examples for dealing with a single cell are really manipulating the properties and methods of a "range" object. The only difference in your treatment of a cell and a range is that you can read and write a cell value and formula but you can only write values and formulas with a range.

See:

[Specifying a range](#)

[Working with a range to the left and/or above the active cell](#)

[Working with individual cells in a range](#)

Specifying a range

Here are the most important ways of identifying ranges (there are other ways!)

Selection	The current selection (highlighted area)
Range("a1..a10")	A specific range, defined by cell references
Range("myrange")	A specific range, defined a named range
ActiveCell.Range("a1..a10")	The first 10 cells below and including the active cell. In this case, the "a1..a10" reference is relative to the current cell.
Cells	The entire worksheet

Since all the above return a range, they all support the same methods and properties. For example:

```
Sub TestRange
    Selection.Font.Bold = True
    Range("a1..a10").Font.Underline = True
    Cells.Font.Size = 10    ' the whole sheet!
    Range("b1..b10").Value = "test"
    Range("c1..c10").Formula = "=a1 * 10"
End Sub
```

Note the last line of the example specifies a formula. This does not increment cell references down the column (e.g. cell c2 will contain "=a1 * 10" just like cell c1). If you want to increment the references, do the following:

```
Range("c1").Formula = "=a1 * 10"
Range("c1").Copy()
Range("c2..c10").Paste()
```

This results in cell c2 containing "=a2 * 10".

Working with a range to the left and/or above the active cell

This has to be done in two stages, first to change the active cell and then second specify the range relative to the new active cell. For example:

```
ActiveCell.Cells(-1, -1).Activate()  
ActiveCell.Range("a1..a5").Select()
```

This is necessary because there is no way to specify a range like Range("-a1, -a10").

Working with individual cells in a range

Once you've got hold of a range, you might want to do something with each cell individually. For example, to check the contents of each cell.

There are two notations, a (row, col) coordinate or the nth item in the range. The row, col notation is convenient if you are working with a set range. For example:

```
MsgBox Range("a1..b5")(1, 1).Value 'displays contents of cell a1  
MsgBox Range("a1..b5")(2, 2).Value 'displays contents of cell b2.
```

The alternative notation takes a range from left to right, top to bottom and numbers it from 1 to n. So the exact equivalent of the above is:

```
MsgBox Range("a1..b5")(1).Value 'displays contents of cell a1  
MsgBox Range("a1..b5")(2).Value 'displays contents of cell b2.
```

But really this notation is more useful when you don't know in advance the extent of a range (for example, dealing with a selection). You can be sure to see every item in a range using the following:

```
For i = 1 to Selection.Count  
    MsgBox Selection(i).Value  
Next
```

A brief note on the Item property of a range

To give you a fuller picture, the previous methods of working with cells within a range use a shorthand notation. All ranges have an "Item" property that allows you to reference an individual cell. Since "Item" is the default property, it is not necessary to actually write it out. For example, the following are identical:

`Range("a1..b5")(1, 1).Value`

`Range("a1..b5").Item(1, 1).Value`

and if you happened to start a selection on a1 then these are identical with each other as well as the above:

`Selection(i).Value`

`Selection.Item(i).Value`

Working with worksheets and workbooks

The spreadsheet application can have none, one or many workbooks open at one time and each workbook can have one or more worksheets. All of these are part of a straightforward hierarchy:

`Workbooks` ' all the workbooks in an application
`Workbooks.Count` ' how many workbooks do we have open
`Workbooks(0)` ' the first open workbook
`Workbooks("myspread")` ' the workbook called `myspread`

Once you've got hold of a workbook, you can examine it's name, count it's worksheets and refer to individual worksheets:

```
Dim wb, ws
Set wb = Workbooks(0)
MsgBox "Name is " & wb.Name & ", and has " & wb.Worksheets.Count & "worksheets."
Set ws = wb.Worksheets(0) ' or alternatively Set ws = wb.Worksheets("mysheet")
Msg "First worksheet is called: " & ws.Name
```

There can be only one active (or current) workbook however. Also, there can be only one active sheet and one active cell.

<code>ActiveCell</code>	Returns the current cell in the current worksheet in the current workbook.
<code>ActiveSheet</code>	Current worksheet.
<code>ActiveWorkbook</code>	Current workbook.

Database macros

Database macros can be stored in one of two ways in: global or local. The following table explains the differences:

	Global macros	Database specific macros
Stored in	Global database macro file (abdata.abd). The file is in the program directory ("c:\program files\ability office\" by default)	The database it was created in.
Create using	Tools/Macros menu.	File/New/Macro from menu (a database must be open).
Run by	Tools/Macros menu – can also be run from a button on the toolbar or by keyboard shortcut.	Double-click directly from Database Manager.
Notes	Identical operation to global macros in Spreadsheet and Write.	Can contain one or more subroutines or functions – the first one in the file will be the default (i.e. double-click will run the top most macro).

Database shares the same database engine as Microsoft Access and some users may already be familiar with automation using DAO. Ability allows access to two of the fundamental DAO objects. See [Using DAO automation](#) for more details.

See also:

[Working with Databases](#)

[Working with tables](#)

[Working with fields](#)

Working with Databases

Naturally enough, there is a collection called Databases that opens and creates databases and a Database object that contains the tables of information you'll want to work with. The following summarizes the most important properties and methods:

Databases collection

[Open\(path to file, Exclusive\)](#) Open an existing database.
[Create\(path to file\)](#) Create a new database.

Database object

[Close](#) Close the database.
[Admin](#) Access to [DAO Database](#) property
[Tables](#) The collection of tables belonging to the database
[Forms](#) The collection of forms belonging to the database
[Reports](#) The collection of reports belonging to the database
[Queries](#) The collection of queries belonging to the database.

Since the database can only use one database at a time, you can use the Databases collection to create and open a database and then use ActiveDatabase, which is itself just a database object, to do the rest.

For example to open and use a table from a database:

```
Databases.Open("c:\my documents\mydata.adb", False)  
ActiveDatabase.Tables.Open("mytable")
```

Note the second parameter of the open command is False, which means open the database in shared mode. True would be exclusive (i.e. lock every other user out for that session).

See: [Working with tables](#) for details on what to do with tables.

Working with tables

To open a table in the current database, use the following command:

```
Set mytb = ActiveDatabase.Open("mytable")
```

The variable *mytb* now allows access to the table object. Here are the most important methods and properties of the table object:

Activate	Make the table the current one (bring it to the front).
Fields	The collection of fields – field information and data.
MoveFirst	Go to the first record.
MoveNext	Go to the next record.
MovePrev	Go to the previous record.
MoveLast	Go to the last record.
Close	Close the table.
EditRecord	Allow changes to be written back to the current record. Puts the table into an "Edit" state.
AddNewRecord	Allow changes to be saved as a new record. Puts the table into an "Append" state.
UpdateRecord	Write any pending changes to the table and turn off Edit or Append state.

Here's how to move open and move around the table:

```
Set mytb = ActiveDatabase.Open("mytable")
```

```
mytb.MoveFirst
```

```
MsgBox "First Record"
```

```
mytb.MoveNext
```

```
MsgBox "Second Record"
```

.....and so on.

Instead of supplying a specific name for the table, it's sometimes useful to work with the current table (which could be any table). Here's some code that does the same as above:

```
Set mytb = ActiveDataObject
```

```
mytb.MoveFirst
```

etc.

See: [Working with fields](#) for details on how to write changes to a table.

Working with fields

The Fields collection contains the all the fields for a table. An individual field has the following properties:

Value	Contents of the field.
Name	Name of the field.

For example, for quick look at the contents of record 1 in a table:

```
Set mytb = ActiveDatabase.Open("mytable")
mytb.MoveFirst
s = ""
For Each fld In mytb.Fields
    s = s & fld.Name & ": " & fld.Value & vbCr
Next
MsgBox "The first record looks like this: " & vbCr & vbCr & s
```

To update a record, you should really know a little about the type of fields you are using. For example, updating two fields, one called email and the other tel:

```
With mytb
    .EditRecord
    .Fields("email").Value = "test@ability.com"
    .Fields("tel").Value = "01234 5678"
    .UpdateRecord
End With
```

Note that you always have to set the table into an "edit" state with the EditRecord command and complete the edit with UpdateRecord.

Adding a new record would be very similar:

```
With mytb
    .AddNewRecord
    .Fields("email").Value = "test@ability.com"
    .Fields("tel").Value = "01234 5678"
    .UpdateRecord
End With
```

Using DAO automation

Database provides direct access to two important DAO objects:

- [DAO Database object](#)
- [DAO DBEngine object](#)

These are made available for experienced users. Full documentation on how to use them is beyond the scope of this help guide.

DAO.DBEngine

DBEngine is the top level object in the DAO object model – all DAO objects and collections are available through DBEngine. For example, from outside Ability Database (e.g. in Spreadsheet or Write):

```
Set dbapp = CreateObject("AbilityDatabase.Application")  
MsgBox dbapp.DBEngine.Version
```

Inside an Ability Database macro, DBEngine is accessible directly:

```
MsgBox DBEngine.Version
```

The Version property simply shows the version of DAO in use.

Another example of using DBEngine is to compact a database. Although this is available from the Tools menu, one of the options in compacting a database is to set a version number.

```
DBEngine.CompactDatabase "c:\my documents\mydata2.adb", _  
                        "c:\my documents\mydata3.adb", _  
                        32
```

The code above creates the file mydata3.adb from mydata2.adb and at the same time, compacts the file and converts the version number from 2 to 3. It is the last parameter (32 is equal to the DAO constant **dbVersion30**) that tells DAO to convert the version number.

Note that Ability creates JET databases using version 2. When creating databases in Microsoft Access 97, the JET version is 3. As far as Ability is concerned, there is no difficulty using either version whereas Microsoft Access will always want to "update" the version if opening version 2.

DAO Database

The DAO Database object is accessible through the Admin property of an Ability Database. For example:

```
Set dbapp = CreateObject("AbilityDatabase.Application")
dbapp.Databases.Open("c:\my documents\mydata.adb", False)
Set db = ActiveDatabase
Set dbdao = db.Admin
```

Note that the first two lines would not be necessary if you were creating a macro inside Ability Database and already had a database open.

The above code creates a variable *dbdao* which is equivalent to the standard Database object in the DAO object model.

Automating Ability from other programs

Any program that supports OLE automation can run the Ability application "from outside". For example, you may have a custom database application and want to press a button that launches Write with a letter and insert the address block from the details in the current record.

The exact syntax to use depends on your development environment. For an application that supports VBScript (for example, all the Ability applications and WSCRIPT.EXE and CSCSCRIPT.EXE that ship with Windows 98) you can use the following commands to start an Ability application:

```
Set app = CreateObject("AbilityDatabase.Application")  
Set app = CreateObject("AbilitySpreadsheet.Application")  
Set app = CreateObject("AbilityWrite.Application")
```

Note: the Set statement is part of the VBScript language and allows you to create a *reference* to an object. In the above case, CreateObject creates an instance of the named application and the Set statement makes the variable *app* point to the application object. If you used this syntax instead:

```
app = CreateObject("AbilityDatabase.Application")
```

the implication would be that you are trying to make a copy of an application (i.e. your computer memory would be holding two separate running applications) and this is not allowed.

Now the application has been started, you'll need to do something useful, like create a new document or open a database.

Automating Write

As an example, let's assume Write needs to be started from another application (e.g. Spreadsheet – but it could be any application that supports OLE automation) and you want to create and print some text. Here's the code:

```
Set app = CreateObject("AbilityWrite.Application")
app.Activate
Set mydoc = app.Documents.Add("NORMAL")
myDoc.Text.Insert 0, "This is a simple document"
myDoc.Print 0, 99, 1, False
```

The second line "app.Activate" causes the application to become visible to the user (otherwise the application would run in the background and not be visible). Next, a new document is created, based on the normal template, and a reference to the document is saved in a variable *mydoc*.

The rest of the macro adds text to a document and prints it.

Note: The **Print** function takes four parameters: start page, end page, number of copies and "collate" (or not).

Automating Spreadsheet

As an example, let's assume Spreadsheet needs to be started from another application (e.g. Write – but it could be any application that supports OLE automation) and a you want to create a new worksheet. Here's the code:

```
Set app = CreateObject("AbilitySpreadsheet.Application")
app.Visible
Set mywb = app.Workbooks.Add
Set myws = mywb.Worksheets(0)
myws.Cells(1, 1).Value = "A big hello to Spreadsheet"
```

The second line "app.Visible" causes the application to become visible to the user (otherwise the application would run in the background and not be visible). Next a new workbook is created and a reference to the first worksheet in it is saved as *myws*. Finally, some text is written into the cell A1.

Automating Database

As an example, consider opening a table within a database. The following code would work in a macro inside Write or Spreadsheet (or any application supporting OLE).

```
Set app = CreateObject("AbilityDatabase.Application")
app.Visible = True
app.Databases.Open "c:\my documents\mydata.mdb", False
Set mydb = app.ActiveDatabase
Set mytb = mydb.Tables.Open("PhilData")
MsgBox mytb.Fields(0).Value
```

The second line "app.Visible" causes the application to become visible to the user (otherwise the application would run in the background and not be visible). The next three lines open a named database and creates a reference to a specified table in *mytb*. The last line displays the data from the first field in record 1.

Macros – examples

Write macros

[Printing multiple copies](#)

[Printing the selection](#)

[Inserting a database address into Write](#)

[Sending email from Write](#)

Database macros

[Making a letter from the current record](#)

Miscellaneous macros

[Opening a web page inside Ability](#)

[Getting information on the script engine](#)

Printing multiple copies

Not all printer drivers have the capability to print multiple copies. Here is a very simple Write macro that provides a work around for such printers.

```
Sub MultipleCopies
```

```
    Dim ncopies, startpage, endpage, collate
```

```
    startpage = 1
```

```
    endpage = 9999
```

```
    collate = False
```

```
    ncopies = InputBox("Number of copies required?", "Print current document", 2)
```

```
    If ncopies = "" Then
```

```
        Exit Sub
```

```
    End If
```

```
    ActiveDocument.Print startpage, endpage, ncopies, collate
```

```
End Sub
```

You can specify whatever you want for startpage, endpage and collate (or even prompt the user). Setting endpage = 9999 will print every page for most documents.

Printing the selection

The following macro prints out the current selection (this is not supported in the Write print dialog – sometimes macros can be used as work-arounds for non-existent product features!)

```
Sub PrintSelection
  Set currdoc = ActiveDocument
  Selection.Copy()
  Set tempdoc = Documents.Add("NORMAL")
  tempdoc.Paste()
  tempdoc.Print 1, 9999, 1, False
  currdoc.Activate
End Sub
```

The macro first sets a reference to the current document (so it can re-display it at the end). Next, it copies the current selection to the clipboard.

The third line of the macro creates a new document, based on the normal template. This will be used as a temporary store for the selected text, which is pasted and printed.

Lastly, the macro returns to the original document using the Activate command.

Inserting a database address into Write

This example assumes you are writing a letter and you want to insert an address held in a database. Although you could use mail merge to perform this task, it can be convenient at times to do this with a macro.

Each section of the macro will be broken down and explained. The full source listing can be found [here](#).

1. Start the macro, declare and initialize some variables

```
Sub InsertDBAddress
    Dim app, mydb, mytb, srchtxt, dbpath, tbname, srchfld, qrytxt

    dbpath = "c:\my documents\mydata.adb"
    tbname = "mytable"
    srchfld = "Lastname"
```

All the variables used in the macro are declared locally – this is good practice as it safes any possible confusion over whether a variable is local or global.

Insert your own details for database name (including path), the table name and the field you want to search on. In this case, we're allowing records to be found by lastname – not always practical with large tables.

2. Open the database

```
Set app = CreateObject("AbilityDatabase.Application")
app.Databases.Open dbpath, False
Set mydb = app.ActiveDatabase
```

Creates the database application object, opens a database and sets a reference, *mydb*, to it.

3. Get the input from the user

```
srchtxt = InputBox("Enter " & srchfld & ":", "Ability Message")
If Len(srchtxt) = 0 Then
    Exit Sub
End If
```

This prompts the user to enter some data to search for. As a refinement, if the user enters nothing (or selects the "Cancel" button) then the macro will terminate with the "Exit Sub" statement.

4. Open a query

```
qrytxt = "SELECT * FROM " & tbname & " WHERE " & srchfld & "=" & srchtxt & ""
Set mytb = mydb.Admin.OpenRecordset(qrytxt)
```

The first line creates a query, based on the whatever the user entered. For example if the table was called "contacts" and your were searching on a company name field for "Ability Software" the variable *qrytxt* would contain:

```
SELECT * FROM contacts WHERE company ='Ability Software'
```

Which is slightly more readable. Note the use of single apostrophes surrounding Ability Software – these are in the code but only just visible since they are next to double quotes. Without these single apostrophes, the macro would fail!

The second line uses a property of the database object called Admin – for users familiar with DAO, Admin is exactly equivalent to a DBEngine.Database object. The end result is that we've opened a query that contains only the record(s) matching the search text.

5. See if there are any records found

```
If mytb.EOF Then
    MsgBox "No matching records"
    Exit Sub
End If
```

BOF stands for beginning of file (EOF being end of file). When the query was opened, if it contains no records, BOF will be true, and if so, a message is displayed and the macro terminates with "Exit Sub"
If there are records BOF will be false, and we will be positioned on record 1.

6. One or more records?

```
mytb.MoveNext
If mytb.EOF Then
    mytb.MoveFirst
    Selection.InsertAfter BuildAddBlock(mytb.Fields)
    Selection.Collapse(1)
End If
```

Assuming we're on record 1, a call to move to the next record will cause EOF to be true only if there is only one record. That is, the user entered some search text and there is only one matching record, and we've found it. No ambiguity, so we can paste the data into the Write.

Inside the If Then statement, the first thing to do is head back to record 1. Next, the Selection.InsertAfter command pastes the text into the current Write document, where the text is generated by the BuildAddBlock function (described below). Selection.Collapse(1) turns of selected text.

7. More than one record, so browse

Logic tells us that if there is at least one record (part 5 above) and not exactly 1 record (part 6. above) then there must be two or more matching records. This could easily happen searching for the lastname of Smith for example. In this case, we can browse through the records until the user picks one.

```
mytb.MoveFirst
Do
    ret = MsgBox("More than one record - Select this record " & _
                "With the Yes button or choose No For Next " & _
                "record" & vbCr & vbCr & BuildAddBlock(mytb.Fields), _
                vbYesNoCancel, _
                "Ability Message")
    If ret = vbCancel Then
        Exit Sub
    ElseIf ret = vbYes Then
        Exit Do
    End If
    mytb.MoveNext

    If mytb.EOF Then
        ret = MsgBox("End of records!" & vbCr & vbCr & _
                    "Review records again?", vbYesNo, _
                    "Ability Message")
        If ret = vbNo Then
            Exit Sub
        End If
        mytb.MoveFirst
    End If
Loop

Selection.InsertAfter BuildAddBlock(mytb.Fields)
Selection.Collapse(1)
End Sub
```

The basic idea repeat for ever (Do....Loop) until the user chooses to select a record, browse to the next record or start the browse over again if he reaches the end. The command Exit Do jumps out of the loop at the users bidding.

The macro is finished of by inserting the text as described in part 6. above.

A quick note on the use of MsgBox. MsgBox is a built in function that takes one or more parameters. In the examples above, MsgBox is called with three parameters:

Prompt
Buttons
Title

Prompt is the text displayed inside the dialog. *Title* is the text displayed on the title bar of the dialog. *Buttons* controls what buttons to display on the dialog. vbYesNoCancel builds the dialog with three buttons: Yes, No and Cancel. Which button the user selects can be tested for by assigning the return value of MsgBox to a variable (called *ret* above). So `ret = vbYes` is true if the user selected the Yes button and so on.

8. BuildAddBlock and AddLine functions

There are several calls to this function from the main code above. The idea is to take named fields and construct a single string that is suitable for an address block (i.e. no gaps).

```
Function BuildAddBlock(Flds)
```

```
    Dim s  
    s = Flds("CustTitle") & " " & Flds("FirstName") & " " & Flds("Lastname") & vbCrLf  
    s = s & AddLine(Flds("Company"))  
    s = s & AddLine(Flds("Add1"))  
    s = s & AddLine(Flds("Add2"))  
    s = s & AddLine(Flds("Add3"))  
    s = s & AddLine(Flds("Add4"))  
    s = s & AddLine(Flds("Postcode"))  
    BuildAddBlock = s
```

```
End Function
```

The function is declared with a single parameter. In the calling code the following line is used to invoke the function:

```
BuildAddBlock(mytb.Fields)
```

The result of this is that `mytb.Fields` (i.e. the fields for the current record) is mapped into a variable called `Flds` in the function definition.

A variable `s` is built-up by calling the function `AddLine` which checks to make sure the line is not blank.

```
Function AddLine(x)
```

```
    AddLine = ""  
    If Len(x) > 0 Then  
        AddLine = x & vbCrLf  
    End If
```

```
End Function
```

Inserting a database address - code listing

This is a complete listing of the code referred to in the previous example.

If you want to make use of this macro, copy and paste into the macro editor and then replace the dbpath, tname and srchfld text on lines 4 to 6 with your own data. Also adjust the function BuildAddBlock to match your own field names.

```
Sub InsertDBAddress
    Dim app, mydb, mytb, srchtxt, dbpath, tname, srchfld, qrytxt

    dbpath = "c:\my documents\mydata.adb"
    tname = "mytable"
    srchfld = "Lastname"

    Set app = CreateObject("AbilityDatabase.Application")
    app.Databases.Open dbpath, False
    Set mydb = app.ActiveDatabase

    srchtxt = InputBox("Enter " & srchfld & ":", "Ability Message")
    If Len(srchtxt) = 0 Then
        Exit Sub
    End If

    qrytxt = "SELECT * FROM " & tname & " WHERE " & srchfld & "=" & srchtxt & ""
    Set mytb = mydb.Admin.OpenRecordset(qrytxt)

    If mytb.EOF Then
        MsgBox "No matching records"
        Exit Sub
    End If

    mytb.MoveNext
    If mytb.EOF Then
        mytb.MoveFirst
        Selection.InsertAfter BuildAddBlock(mytb.Fields)
        Selection.Collapse(1)
    End If

    mytb.MoveFirst
    Do
        ret = MsgBox("More than one record - Select this record " & _
            "With the Yes button or choose No For Next " & _
            "record" & vbCr & vbCr & BuildAddBlock(mytb.Fields), _
            vbYesNoCancel, _
            "Ability Message")
        If ret = vbCancel Then
            Exit Sub
        ElseIf ret = vbYes Then
            Exit Do
        End If
        mytb.MoveNext
    Loop

    If mytb.EOF Then
        ret = MsgBox("End of records!" & vbCr & vbCr & _
            "Review records again?", vbYesNo, _
            "Ability Message")
        If ret = vbNo Then
            Exit Sub
        End If
    End If
End Sub
```

```
        mytb.MoveFirst
    End If
Loop

Selection.InsertAfter BuildAddBlock(mytb.Fields)
Selection.Collapse(1)
End Sub

Function BuildAddBlock(Flds)
    Dim s
    s = FlDs("CustTitle") & " " & FlDs("FirstName") & " " & FlDs("Lastname") & vbCr
    s = s & AddLine(FlDs("Company"))
    s = s & AddLine(FlDs("Add1"))
    s = s & AddLine(FlDs("Add2"))
    s = s & AddLine(FlDs("Add3"))
    s = s & AddLine(FlDs("Add4"))
    s = s & AddLine(FlDs("Postcode"))
    BuildAddBlock = s
End Function

Function AddLine(x)
    AddLine = ""
    If Len(x) > 0 Then
        AddLine = x & vbCr
    End If
End Function
```

Sending email from Write

The following example relies on Active Messaging - an OLE wrapper for MAPI. You may or may not have this installed on your system!

1. Start the macro and declare the variables

```
Sub SendEmail
```

```
    Dim objSession, objMessage, objRecip, strEmsg, strTo, strSubject
```

2. Get the entire text of the document

```
    strEmsg = ActiveDocument.Text.Mid(0, ActiveDocument.Text.Count)
```

This is the quickest way to get the text of the document – no consideration is given to font changes!

3. Set the fields for Send To and Subject

```
    strTo = "support@ability.com"
```

```
    strSubject = "Send from Ability Write"
```

These details are in the code at the moment – it would be much better to extract them from the actual document. See the suggestions at the bottom of this page for more on how to do this.

4. Start a MAPI session

```
    Set objSession = CreateObject("MAPI.SESSION")
```

```
    objSession.Logon "MS Exchange Settings"
```

Start a session and logon. This is a bit like starting Microsoft Outlook. Note the logon is dependent on the current profiles on the user machine – in other words, you may have to change "MS Exchange Settings" to whatever your logon details are.

5. Create a message

```
    Set objMessage = objSession.Outbox.Messages.Add
```

```
    objMessage.Subject = strSubject
```

```
    objMessage.Text = strEmsg
```

```
    Set objRecip = objMessage.Recipients.Add
```

```
    objRecip.Name = strTo
```

```
    objRecip.Type = 1 '
```

```
    objRecip.Resolve
```

The first block above creates a new message and copies in the text from the document.

The second block adds the recipient details. Note the *Type* is set to 1 – this is the value of *mapiTo* constant (defined as part of OLE Messaging).

6. Send the message

```
    objMessage.Update
```

```
    objMessage.Send
```

```
    objSession.Logoff
```

```
End Sub
```

Save the message, send it and logoff.

See: [Refining the send email macro](#) for suggestions on how to refine this macro.

For the complete code listing plus refinements, see [Sending email from Write - code listing](#).

Refining the send email macro

As a refinement to the [send_email_macro](#) we can extract the "To" and "Subject" fields from the text of a document using the following function:

```
Function GetEmailHeader(msg, str)
    Dim spos, fpos, abText, slen

    slen = Len(str)
    ' Find for example, To: somewhere in the document text
    spos = InStr(1, msg, str, 1)
    If spos Then
        ' If found, find the end of the line
        fpos = InStr(spos, msg, vbCr) - 1
        ' Set the function to return the desired text
        GetEmailHeader = Trim(Mid(msg, spos + slen, fpos - spos - slen + 1))

        ' Delete the line from the email and tidy-up excess linefeeds
        msg = Left(msg, spos - 1) & Mid(msg, fpos+2)
        While Left(msg, 1) = vbCr
            msg = Mid(msg, 2)
        Wend
    Else
        ' Not found, so prompt user for the text
        GetEmailHeader = InputBox("Please enter text for the " & _
            str & " part of the email header")
    End If
End Function
```

The function is called by replacing the lines:

```
strTo = "support@ability.com"
strSubject = "Send from Ability Write"
```

in the SendEmail subroutine with the following

```
strTo = GetEmailHeader(abTtxt, "To:")
' Check to see if the user Cancelled the function..
If Len(strTo) = 0 Then Exit Sub
strSubject = GetEmailHeader(abTtxt, "Subject:")
```

See the [complete code listing](#) .

Sending email from Write - code listing

This is complete code listing for the previous example. You can open the Tools/Macros/Macro Editor screen and paste the entire listing below into your macros file.

```
Sub SendEmail
    Dim objSession, objMessage, objRecip, strEmsg, strTo, strSubject
    strEmsg = ActiveDocument.Text.Mid(0, ActiveDocument.Text.Count)

    strTo = GetEmailHeader(abTxt, "To:")
    If Len(strTo) = 0 Then Exit Sub
    strSubject = GetEmailHeader(abTxt, "Subject:")
    Set objSession = CreateObject("MAPI.SESSION")
    objSession.Logon "MS Exchange Settings"

    Set objMessage = objSession.Outbox.Messages.Add
    objMessage.Subject = strSubject
    objMessage.Text = strEmsg

    Set objRecip = objMessage.Recipients.Add
    objRecip.Name = strTo
    objRecip.Type = 1
    objRecip.Resolve

    objMessage.Update
    objMessage.Send
    objSession.Logoff
End Sub

Function GetEmailHeader(msg, str)
    Dim spos, fpos, abText, slen

    slen = Len(str)

    spos = InStr(1, msg, str, 1)
    If spos Then
        fpos = InStr(spos, msg, vbCr) - 1

        GetEmailHeader = Trim(Mid(msg, spos + slen, fpos - spos - slen + 1))

        msg = Left(msg, spos - 1) & Mid(msg, fpos+2)
        While Left(msg, 1) = vbCr
            msg = Mid(msg, 2)
        Wend
    Else
        GetEmailHeader = InputBox("Please enter text for the " & _
            str & " part of the email header")
    End If
End Function
```

Making a letter from the current record

This macro shows how to take the data from the current record in a Database and generate a letter in Write. The macro is broken into manageable chunks in the description below. To see the macro in full, with tips on how to use it with your own data, see [Create letter from database - code listing](#).

1. Initialize the macro

```
Sub MakeLetterFromDB
    Dim myflds(9), lfld, s, tb, appWrite, mydoc, i

    lastfld = 8
    myflds(0) = "CustTitle"
    myflds(1) = "Firstname"
    myflds(2) = "Lastname"
    myflds(3) = "Company"
    myflds(4) = "Add1"
    myflds(5) = "Add2"
    myflds(6) = "Add3"
    myflds(7) = "Add4"
    myflds(8) = "Pcode"
```

The first line declares in advance all the variables that are going to be used (good practice). Note the declaration of an array *myflds* with 9 elements, that holds the names of the fields to be used for the address block – obviously, this will need adjustment for your own database.

The variable *lastfld* is just a reminder that any loops addressing the *myflds* array will need to stop at 8 (arrays are zero based).

2. Get "hold off" the current table

```
Set tb = ActiveDataObject
```

`ActiveDataObject` returns the current table or query or relation and the line above saves a reference to this in the *tb* variable. This makes the macro as general as possible – no table name or type of object (table/query/relation) need be specified.

You could replace this with a specific table if necessary. For example

```
Set tb = ActiveDatabase.Tables("mytable")
```

would also do the job, providing *mytable* was open at the time (and pointing to the actual record you want to use).

3. Make an address block

```
s = ""
For i = 0 To lastfld
    txt = tb.Fields(myflds(i)).Value
    If Len(txt) > 0 Then
        s = s & txt
        If i < 2 Then
            s = s & " "
        Else
            s = s & vbCr
        End If
    End If
Next
```

The code above loops through the fields listed in part 1 and adds them to a variable *s*. It checks to see if the field is blank (in which case it's skipped) and also takes account of the first line of the address block, the code: `If i < 2 then` tests for this. All other address lines are added to *s* along with a new line (`vbCr`).

4. Ask the user if he wants to continue

```
i = MsgBox("Make a letter with this record?" & _
```

```
        vbCr & vbCr & s, vbOKCancel, _  
        "Ability Database Macro Message")  
If i = vbCancel Then  
    Exit Sub  
End If
```

This shows a message box with the address block and asks the user to continue or not. If the user selects the cancel button, the macro is terminated.

5. Open a new write document and drop in the text

```
Set appWrite = CreateObject("AbilityWrite.Application")  
Set mydoc = appWrite.Documents.Add("NORMAL")  
mydoc.Text.Insert 0, s & vbCr & vbCr  
mydoc.Selection.End = mydoc.Text.Count  
mydoc.Selection.Start = Selection.End  
  
appWrite.Activate
```

End Sub

The first two lines start Write and create a new document. The third line drops in the text.

The fourth and fifth line makes sure the cursor is at the end of the document and finally the Write application is made visible by the Activate command so the user can finish off the letter.

Create letter from database - code listing

This is a complete listing of the code referred to in the previous example.

If you want to make use of this macro, copy and paste into the macro editor of Database. It need to be a global macro so use Tools/Macros. Since it works with the current table, you'll need to execute the macro while actually viewing a table. For example, you can add a button to the toolbar that allows you to do this. (Select Tools/Customize and then the Macro Shortcuts tab. Drag the macro "MakeLetterFromDB", by its button, onto one of the toolbars at the top of the screen).

To tailor it to your own data, replace the field names starting at line 5 with your own.

```
Sub MakeLetterFromDB
    Dim myflds(9), lfld, s, tb, appWrite, mydoc, i

    lastfld = 8
    myflds(0) = "CustTitle"
    myflds(1) = "Firstname"
    myflds(2) = "Lastname"
    myflds(3) = "Company"
    myflds(4) = "Add1"
    myflds(5) = "Add2"
    myflds(6) = "Add3"
    myflds(7) = "Add4"
    myflds(8) = "Pcode"

    Set tb = ActiveDataObject
    s = ""
    For i = 0 To lastfld
        txt = tb.Fields(myflds(i)).Value
        If Len(txt) > 0 Then
            s = s & txt
            If i < 2 Then
                s = s & " "
            Else
                s = s & vbCr
            End If
        End If
    Next

    i = MsgBox("Make a letter with this record?" & _
        vbCr & vbCr & s, vbOKCancel, _
        "Ability Database Macro Message")
    If i = vbCancel Then
        Exit Sub
    End If

    Set appWrite = CreateObject("AbilityWrite.Application")
    Set mydoc = appWrite.Documents.Add("NORMAL")
    mydoc.Text.Insert 0, s & vbCr & vbCr
    mydoc.Selection.End = mydoc.Text.Count
    mydoc.Selection.Start = Selection.End

    appWrite.Activate
End Sub
```

Opening a web page inside Ability

Of course, this requires Microsoft's Internet Explorer. The only trick here is knowing how to refer to Explorer (which is why we've included this as an example).

```
Sub OpenWebPage()  
    Dim mIE  
    Set mIE = CreateObject("InternetExplorer.Application")  
    mIE.Visible = True  
    mIE.Navigate "http://www.ability.com"  
End Sub
```

The first line declares mIE as a variable to be used in this sub-routine. This is not strictly necessary but is good practice (so as not to confuse it with a globally defined variable of the same name were you to have a complex sequence of macros). The second line creates an instance of Explorer, the third line makes it visible to the user (rather than run in the background) and the last line navigates to the Ability web site.

Instead of "hard coding" the web site address, we can instead extract the URL from the current text. For example, in Spreadsheet, using the following line:

```
mIE.Navigate ActiveCell.Value
```

causes the macro to take the text from the current cell, so that, if cell A1 had the text "http://www.ability.com" and was also the current cell, the macro would again navigate to the Ability web site.

Ability has a built-in function that performs a similar task – see [HYPERLINK](#) for details.

Getting information on the script engine

Sometimes it is useful to get information about the script engine being used. For example, certain features of the VBScript language are only available in version 5.0 of the script engine. Here's how to do it:

```
Sub GetScriptVersionInfo
  Dim s
  s = ScriptEngine & " Version "
  s = s & ScriptEngineMajorVersion & "."
  s = s & ScriptEngineMinorVersion & "."
  s = s & ScriptEngineBuildVersion
  MsgBox s
End Sub
```

"ScriptEngine" etc are built-in VBScript functions.

An introduction to the VBScript language

VBScript is a derivative of Microsoft's Visual Basic designed to be used for web server scripting and adding intelligence to web pages in Internet Explorer. Because it is lightweight and very easy to use it makes an ideal language for application Macros.

Please note that this guide is designed as a very brief introduction to the language. If you want a more detailed introduction, there are plenty of third party books - just go to <http://www.amazon.com/> and search for "VBScript". There is also lots of material at the Microsoft web site.

See:

[Creating and assigning variables](#)

[Simple interaction with the user](#)

[Conditional statements](#)

[Repeating commands with a For...Next loop](#)

[Using functions and subroutines](#)

Creating and assigning variables

VBScript allows you to create variables on the fly, as simple as this:

```
x = 24
```

which creates a variable, x, and gives it the value 24.

With VBScript you don't need to worry about data types. All of this is managed for you. So all of this is a valid, if pointless, sequence:

```
Sub MyTest
  x = 24
  x = "Hello"
  y = 12
  x = x & " " & y
End Sub
```

The last line shows how to concatenate strings in VBScript - it also demonstrates that the language is smart enough to know that, in this instance, you want a number to be treated as text. The end value of x is "Hello 12".

Explicit declaration of variables

If you are going to create a complex sequence of macros, it is a good idea to declare your variables using a **Dim** statement in each subroutine. This prevents any confusion over whether a variable is local to a subroutine or a global one (variables declared outside of subroutines are global). For example:

```
Sub MyTest
  Dim x, y
  x = 24
  x = "Hello"
  y = 12
  x = x & " " & y
End Sub
```

Simple interaction with the user

There are two functions built into the VBScript language that allow you to get some input value from the user and to display a message:

```
x = InputBox "Please enter your height in meters"  
x = x * 3.279  
MsgBox "Your height is " & x & " feet."
```

will ask the user to type a value in meters and assign it to the variable x, convert it to feet and then display the result.

Conditional statements

You can conditionally execute commands using if...then...else statements. For example:

```
If 1 = 2 Then
  MsgBox "It's true"
End If
```

More generally, you can evaluate multiple conditions in one go:

```
If [condition1] Then
  [commands]
Elseif [condition2] Then
  [more commands]
Else
  [if all else fails, some default commands]
End If
```

A condition is a statement that logically evaluates as true or false. Here are some examples:

$a = b$	True only if a and b have the same value
$a > b$	True if a is greater than b
$a \leq b$	True if a is less than or equal to b
$a \neq b$	True if a and b do not have the same value
$(a = b) \text{ And } (c < d)$	True if a and b are equal and at the same time, c is less than d
$((a = b) \text{ And } (a < c)) \text{ Or } (a < b)$	True if some complex relationship holds

Note in the last two examples that brackets must be used to surround individual conditions.

Here's another example that uses a function built-in to VBScript, Len(), that gets the number of characters in a text string.

```
str = InputBox "Please enter your name."
If Len(str) = 0 Then
  MsgBox "You didn't enter anything!"
Else
  MsgBox "The name you entered is: " & str
End If
```


Repeating commands with a For...Next loop

If you want to repeat commands a set number of time, use a For...Next loop. For example:

```
For i = 1 to 10
  If i < 9 Then
    MsgBox "Reminder " & i
  Else
    MsgBox "Reminder " & i & ", almost there"
  End If
Next
```

VBScript starts by setting a variable i to 1 and then executing the commands sandwiched between the For and Next statements. Because i can be referenced within this block, the first time the block is executed, the message box will display:

"Reminder 1"

When the Next line is reached, VBScript adds one to i and returns execution to the top of the block. And this process is repeated until i reaches 10. So the last message to be displayed is:

"Reminder 10, almost there"

Repeating commands with a While...Wend loop

If you want to repeat a command while a particular condition holds true, use a While...Wend loop. For example:

```
str = InputBox "Please enter your name"  
While Len(str) = 0  
    str = InputBox "You must enter at least one character, please try again"  
Wend
```

This rather unfriendly bit of code forces the user to type something into the InputBox.

Using functions and subroutines

What is the difference between a function and a subroutine? Actually, very little. There is the notional idea that a function usually returns a value whilst a subroutine never does. Everything else is the same – both subroutines and functions can be passed parameters and both return to the caller on completion.

If you intend to write a lot of script, you should plan to make good use of functions and subroutines and make an effort to keep each one short – this will aid in maintaining the code and should also make you a more efficient programmer. For example, if you have a For...Next or Do While loop that contains 30 lines of script – consider putting this script into a separate subroutine. Or if you find yourself re-writing some lines of script – make a general subroutine or function out of it instead.

The use of functions and subroutines are best explained by example.

See:

[Subroutines - an example](#)

[Functions - an example](#)

Subroutines - an example

To call a subroutine just type it's name in your main script. As an example, consider a replacement for MsgBox that replaces the dialog title with your own fixed message. Here's the entire code:

```
Sub MainScript
  ShowMsg("Hello")
End Sub
```

```
Sub ShowMsg(msgstr)
  MsgBox msgstr, 0, "My custom msgbox"
End Sub
```

Note: MsgBox, the built-in VBScript function, takes up to 5 parameters, three of which are used in the command above: *prompt*, *buttons* (0 makes it display only an "OK" button) and *title*. (the text in the title bar of the MsgBox dialog).

What's the point? This is obviously a trivial example but even so, if you wanted all your message boxes to display with your own message in the dialog title, repeatedly writing out the longer form with "my custom msgbox" every time you wanted to use this function would become tedious. To push the point home, which is quicker to write and easier to maintain:

Example A (using a subroutine)

```
Sub MainScript
  ShowMsg("Hello1")
  ShowMsg("Hello2")
  ShowMsg("Hello3")
  ShowMsg("Hello4")
  ShowMsg("Hello5")
  ShowMsg("Hello6")
End Sub
```

```
Sub ShowMsg(msgstr)
  MsgBox msgstr, 0, "My custom msgbox"
End Sub
```

Example B (without using a subroutine)

```
Sub MainScript
  MsgBox "Hello1", 0, "My custom msgbox"
  MsgBox "Hello2", 0, "My custom msgbox"
  MsgBox "Hello3", 0, "My custom msgbox"
  MsgBox "Hello4", 0, "My custom msgbox"
  MsgBox "Hello5", 0, "My custom msgbox"
  MsgBox "Hello6", 0, "My custom msgbox"
  MsgBox "Hello7", 0, "My custom msgbox"
End Sub
```

Functions - an example

Let's take some very simple macro and show how it can be converted into a function that returns a value. Here's the basic macro:

```
Sub MainMacro()  
    y = InputBox "Enter a value in centigrade"  
    y = 32 + (y * 9) / 5  
    MsgBox "The answer is " & y & " Fahrenheit"  
End Sub
```

Splitting this into a main subroutine and a function, we get the following:

```
Sub MainMacro()  
    y = InputBox "Enter a value in centigrade"  
    x = FtoC(y)  
    MsgBox "The answer is " & x & " Fahrenheit"  
End Sub
```

```
Function FtoC (y)  
    FtoC = 32 + (y * 9) / 5  
End Function
```

The important part is that the function returns a value by assigning a value to the function name (as though the function name was itself a variable).

As with the subroutine example, the advantage of splitting the code will only come if you have to perform the centigrade to Fahrenheit conversion many times in your code. For example, suppose your code had ten separate lines, in different places that did a conversion of centigrade to Fahrenheit:

```
x = 32 + (y * 9) / 5
```

and you now found out that you needed to check that a sensible value was created. In this instance, you'd have to go and modify ten lines of code. If you'd split the code into a function instead, you could adjust the function once. For example:

```
Function FtoC (y)  
    If y < -273.15 Then  
        MsgBox "Colder than absolute zero!"  
        Exit Function  
    End If  
    FtoC = 32 + (y * 9) / 5  
End Function
```

Dealing with run-time errors

There are many situations that can generate run-time errors: careless programming, system errors and so on. Any run-time error will have stop the script - not usually desirable.

Here's a general way of dealing with this problem:

```
On Error Resume Next
[Some statement or call to another procedure]
If Err.Number Then
    MsgBox Err.Description
    Err.Clear
End If
```

It works by making use of a global object called Err that is part of VBScript. By stating that "On Error Resume Next" VBScript knows that you want to deal with any errors rather than letting VBScript deal with them in it's own way (i.e. to halt the script).

