# Vision Analyzer Macro Cookbook

## Version 1.05

**Contents**

# 1. Introduction

This short macro cookbook is intended to help you start creating macros for the Vision Analyzer without going too deeply into theory.

You can use macros in the Analyzer for (almost) anything. You can automate processing steps, and implement your own procedures and algorithms for recalculating data. You can also add or remove markers to or from data sets, import channel positions from all kinds of files into EEG data sets, export data and markers in your own formats, create reports and much more besides.

Although macros are written in the Basic programming language you do not have to be a programmer to create them. As you will see in the "Practical examples" chapter later, quite simple macros are capable of increasing functionality considerably.

The easiest way to create macros is to look at the examples, take the one that comes closest to your problem and modify it.

The chapters that precede the examples are intended to put you in a position to manipulate the existing macros easily.

You will find the macro examples in the Examples subfolder of the Vision folder. If you want to use or modify an example, you should copy it to the Workfiles subfolder of the Vision folder and then work with the copy.

# 2. Creating a simple macro

In this chapter we will create a simple macro which closes all open history files and the associated windows, i.e. clears up the desktop. So that you can follow this example, and all others, you should have installed a functional version of the Analyzer. There should also be some history files in your current workspace.

Now proceed as follows.

Launch the Analyzer. Then select the **Macro > New** menu item. This opens a window titled "Macro1 (macro)...".This window contains two lines – Sub Main and End Sub. These lines enclose the actual macro, i.e. you place your code between these two lines. Type in the macro as follows:

```
Sub Main
    for each hf in historyfiles
        hf.close
    next
End Sub
```

The editor will change upper/lower case automatically sometimes. Basically, no distinction is drawn between upper and lowercase except when texts need to be compared.

Select the **File > Save** menu item and store your macro under the name Close All. Now press the F5 key to execute the macro. Nothing should happen as long as you have not made any typing mistake. Otherwise the program will take you to the line containing the typing mistake.

Now open some history files by pressing on the (+) character next to the book icons. Execute the macro again. All history files will be closed as if by magic.

You can now close the macro window. To run the macro again, select it under **Macro > Run.**

As an alternative, you can make the macros appear as items on the macro menu bar. To do this, select **Macro > Options**. Here you can select up to 10 macros. When you have completed your choice and selected the **Macro** menu again, you will find your macro on the menu. You can now call the selected macros via keyboard commands (Alt-M, 1, 2 , 3...). Note that these macros must always be located in the current work folder. You define the current work folder in the Analyzer under **Configuration > Select Folder for Workfiles**.

In the sections that follow, we will deal with the meaning of the lines that you just typed in.

# 3. Quick tour of Basic

This chapter gives you a brief – and therefore incomplete – overview of some aspects of macro creation with the Basic language. Use the online Help to find out more information. You can do this while creating a macro by means of **Help > Language Help**.

If you find terms in the examples that you do not understand, you can also use context-sensitive help. To do this, move the cursor to a term and then press the F1 key. The online Help facility will then give you an explanation of the term if it is registered in the Help file. That does not apply to Analyzer objects which are explained in the following chapter.

The Basic dialect that is used is compatible with Microsoft's Visual Basic so that you can refer to Visual Basic documentation if you have any other questions.

## 3.1. Variables

Variables can be regarded more or less as small containers or holders in which certain data, e.g. numbers or texts, can be stored so that it can be used in the macro again at any point.

Example:

```
i = 0
i = i + 1
```

In the first line, the value 0 was assigned to variable i. In the second line, i was incremented by 1.

Variables can also be declared explicitly (dimensioned) in a macro, as shown in the following example:

```
Dim f as Single
```

Here, we have a variable named f as a holder for single precision floating point numbers. Declaration of a variable is optional. You can also use variables without declaring them.

Declaration is unimportant as long as macros are short but if your macros exceed 30 to 50 lines it is advisable to declare variables in order to keep track of the situation. Read the "Tips for advanced users" chapter at the end of this book for more information on this subject.

If you want to store not just one value in a variable but several, then we refer to an array. The following declaration generates an array with 20 single-precision floating point numbers. In the second line, the second entry of this array is set to a value. In the third line, the value of the second entry in the array is assigned to variable x.

```
Dim fArray(1 To 20) as Single
fArray(2) = 12
x = fArray(2)
```

If you need an array whose size can change while the macro is running, then you declare it as follows:

```
Dim fArray() as Single
Redim fArray(1 to 20)
```

Here, the array named fArray was declared (dimensioned) and then redimensioned for twenty entries.

```
Redim Preserve fArray(1 to 20)
```

The above statement also redimensions an array but preserves any existing content.

## 3.2. Procedures and functions

The Basic interpreter features a wide range of built-in functions and procedures that you can use in your macros.

Functions perform operations and return a result. The *sin* function is an example. It returns the sine of a number in radians, and is used as follows:

```
x = sin(0.3)
```

In this case, the sine of 0.3 was assigned to variable x.

Procedures also perform one or more operations but do not return any value. Example:

```
beep
```

The Beep procedure generates a short beep.

A complete description of all built-in functions and procedures of the Basic interpreter is given in the online Help.

You can use your own procedures and functions in your macros. This may make large macros easier to read. The "Tips for advanced users" chapter gives more information on this.

## 3.3. Objects

The term *object* is used a great deal so we will explain here what it means in this manual.

An object is a function unit that you can manipulate with a program. It has methods and properties. In their form, methods correspond to functions or procedures. Properties can be either simple variables or, in turn, objects.

The most important object in the Analyzer is called Application. It represents the Analyzer. The following macro uses the Application object to terminate the program.

```
Sub Main
    Application.Quit
End Sub
```

Here, the Application object's Quit method was used to terminate the Analyzer.

The Application object is the Analyzer's default object. This means that it can also be omitted from the macro code. The following macro is identical to the preceding one in terms of its functioning:

```
Sub Main
    Quit
End Sub
```

In turn, the Application object has other objects, e.g. the HistoryExplorer object which represents the History Explorer. HistoryExplorer has the Visible property which is set to 1 when the Explorer is visible and to 0 when it is invisible.

The line

```
    Application.HistoryExplorer.Visible = 0
```

or

```
    HistoryExplorer.Visible = 0
```

make the History Explorer invisible.

You will learn more about the Analyzer's object model in the next chapter.

You can use variables that reference objects. In this case, you use the keyword Set in addition to the assignment character (=) to assign the value. Example:

```
    set he = HistoryExplorer
    he.Visible = 0
```

Objects can contain default elements which do not have to be mentioned explicitly in the macro code. In the following example, a reference to the first history file in the workspace is assigned to the hf variable.

```
    set hf = HistoryFiles.Item(1)
```

Since Item is the default element of Application.HistoryFiles, the expression can also read as follows:

```
    set hf = HistoryFiles(1)
```

A *collection* is a special type of object. These are objects which, in turn, contain multiple objects of one type. The HistoryFiles object, which contains multiple objects of the HistoryFile type is an example of this. Such collections can be recognized in the Analyzer, and also in most other OLE Automation servers, in that they are written as an English plural. For example, HistoryFiles contains objects of the HistoryFile type, HistoryNodes contains objects of the HistoryNode type, etc. Elements of collections can be indexed in the same way as arrays.

```
    Dim hf as HistoryFile
    set hf = HistoryFiles(1)
```

## 3.4. User interaction

While a macro is running, you can output messages to the user, or prompt for entry of parameters.

The InputBox and MsgBox functions are available for input and output respectively. The following macro receives a user input and outputs it again as a message.

```
Sub Main
    x = InputBox("Enter Text")
    MsgBox x
End Sub
```

The GetFilePath function is available to select files. Read the description in the online Help for more details of this.

Finally, you can also design your own dialogs and use them in the macro. You can find out more about this in the "Tips for advanced users" chapter as well as in the online Help.

### 3.5. Comments

You can insert comments in macros to make them clearer and easier to understand. Comments begin with the **'** character. The text after this character up to the end of the line is then ignored by the Basic interpreter.

Example:

```
' This macro receives a user input and shows the result in a message box.
Sub Main
    x = InputBox("Enter Text")       ' Get user input and store it in x.
    MsgBox x                         ' Show user input in a message box.
End Sub
```

You should not be sparing with comments. Otherwise it can happen, especially in larger macros, that you lose track of the situation and no longer know what individual statements are actually supposed to do.

### 3.6. Control structures

Control structures belong to every powerful macro language. They permit conditional execution of code or repeat one or more operations several times on the basis of a condition.

The Basic interpreter uses the *If ... then ... else ... end if* -construct for conditional branching. Example:

```
Sub Main
    Dim x as long
    x = InputBox("Enter a Number:")
    if x > 20 then
        MsgBox "X is greater than 20."
    else
        MsgBox "X is not greater than 20."
    end if
End Sub
```

If a number greater than 20 is input here, then the first message is output. Otherwise the second message is output.

The *else* branch can be omitted. In this case, simply nothing happens if the specified number is less than or equal to 20:

```
Sub Main
    Dim x as long
    x = InputBox("Enter a Number:")
    if x > 20 then
        MsgBox "X is greater than 20."
    end if
End Sub
```

Lines are indented to indicate levels in the macro code. This is not mandatory but, in large macros, it makes the code much easier to read.

If one or more operations need to be repeated (loop) then the *for ... next* construct is a good approach. Example:

```
    Dim fArray(1 to 20) as single
    for i = 1 to 20
        fArray(i) = 2
    next
```

All elements in the array named fArray are set to 2.

Another construct for repetition deals specifically with collection objects – namely the *for each ...in .. next* construct:

```
     Dim hf as HistoryFile
     for each hf in HistoryFiles
          hf.Close
     next
```

Here, all HistoryFile objects in the HistoryFiles collection are referenced by hf. Then the history files are closed in the loop.

It is now time to point out a special characteristic of loops – endless loops. If you program a macro as follows, then i is incremented internally and then decremented again in the loop. The program does not leave the loop until i exceeds the value 1000, which never happens here.

```
Sub Main
     For i = 1 To 1000
          i = i - 1
     Next
End Sub
```

The macro will run forever. However, you can abort it by pressing the **Ctrl-Break** key combination.

Please refer to the online Help for information on other control structures.

## 3.7. Error handling

An error may occur when a macro is executing. Let's assume your workspace contains 1000 history files and the macro contains a statement to open the 1001st file:

```
hf = HistoryFiles(1001)
hf.Open
```

An error message is output in the status bar and the macro is aborted. To give you the chance of taking some action before the macro is stopped, there is the *On Error* statement. Example:

```
Sub Main
     On Error Goto CheckError
     hf = HistoryFiles(1001)
     hf.Open
     exit sub
CheckError:
     MsgBox "Could not open history file"
End Sub
```

If an error occurs here, the macro continues to execute at the CheckError label. Note the exit sub statement before the CheckError label. It terminates the macro at this point, thus preventing the code after the CheckError label from being executed if no error has occurred.

This construct does not offer many advantages for the problem given in the example above, but it can be useful in certain situations.

Please refer to the online Help for more information on error handling routines.

## 3.8. Optimizing the speed of macros

Macros can be accelerated considerably by avoiding unnecessary object references. Cases with acceleration up to a factor of 100 have occurred in practice.

The most important place in which optimization can take effect is inside a loop. Here, every loss of time that is programmed is multiplied by the number of passes through the loop.

Below you see an example of a non-optimized loop. The underline character ("_") is used as the last character of a line when a statement extends over multiple lines.

```
Sub Main
    Set nhn = New NewHistoryNode
    nhn.Create "New", ActiveNode
    For i = 1 To 100
          nhn.RemoveMarker 0, hn.Dataset.Markers(i).Position, 1, _
          ActiveNode.Dataset.Markers(i).Type, ActiveNode.Dataset.Markers(i).Description
          nhn.AddMarker 0, ActiveNode.Dataset.Markers(i).Position, 1, "Stimulus", _
          ActiveNode.Dataset.Markers(i).Description + "A"
    Next
    nhn.Finish
End Sub
```

Below, there is a macro with the same functionality which runs about 100 times faster and is also easier to read. Note that the chain of objects ActiveNode.Dataset.Markers is only referenced once outside the *For ... Next* loop.

```
Sub Main
    Set nhn = New NewHistoryNode
    nhn.Create "New", ActiveNode
    Set Mks = ActiveNode.Dataset.Markers    ' Use variable as object reference.
    For i = 1 To 100
          Set mk = Mks(i)
          nhn.RemoveMarker 0, mk.Position, 1, mk.Type, mk.Description
          nhn.AddMarker 0, mk.Position, 1, "Stimulus", mk.Description + "A"
    Next
    nhn.Finish
End Sub
```

A simple rule of thumb for acceleration is:

**Remove as many periods (".") as possible from the loop if these describe object references.**

Object references often make the creation of objects within the Analyzer complex. They are continually deleted in the loop and recreated if they are not assigned once to a variable as in:

```
    Set Mks = ActiveNode.Dataset.Markers    ' Use variable as object reference.
```

Such an approach can sometimes impair speed dramatically.

Some periods in the loop relate to properties which represent variables in the object, e.g. mk.Position. Here, the anticipated gain in speed as a result of assignment to a variable is so low that the disadvantages of less clarity prevail.

# 4. The Analyzer's object model

Here we will run through the Analyzer's object model briefly. For details, please refer to the OLE Automation Reference Manual which is part of the package.

The figure below shows the Analyzer's object hierarchy as a guide. The Application object is at the very top of the hierarchy.

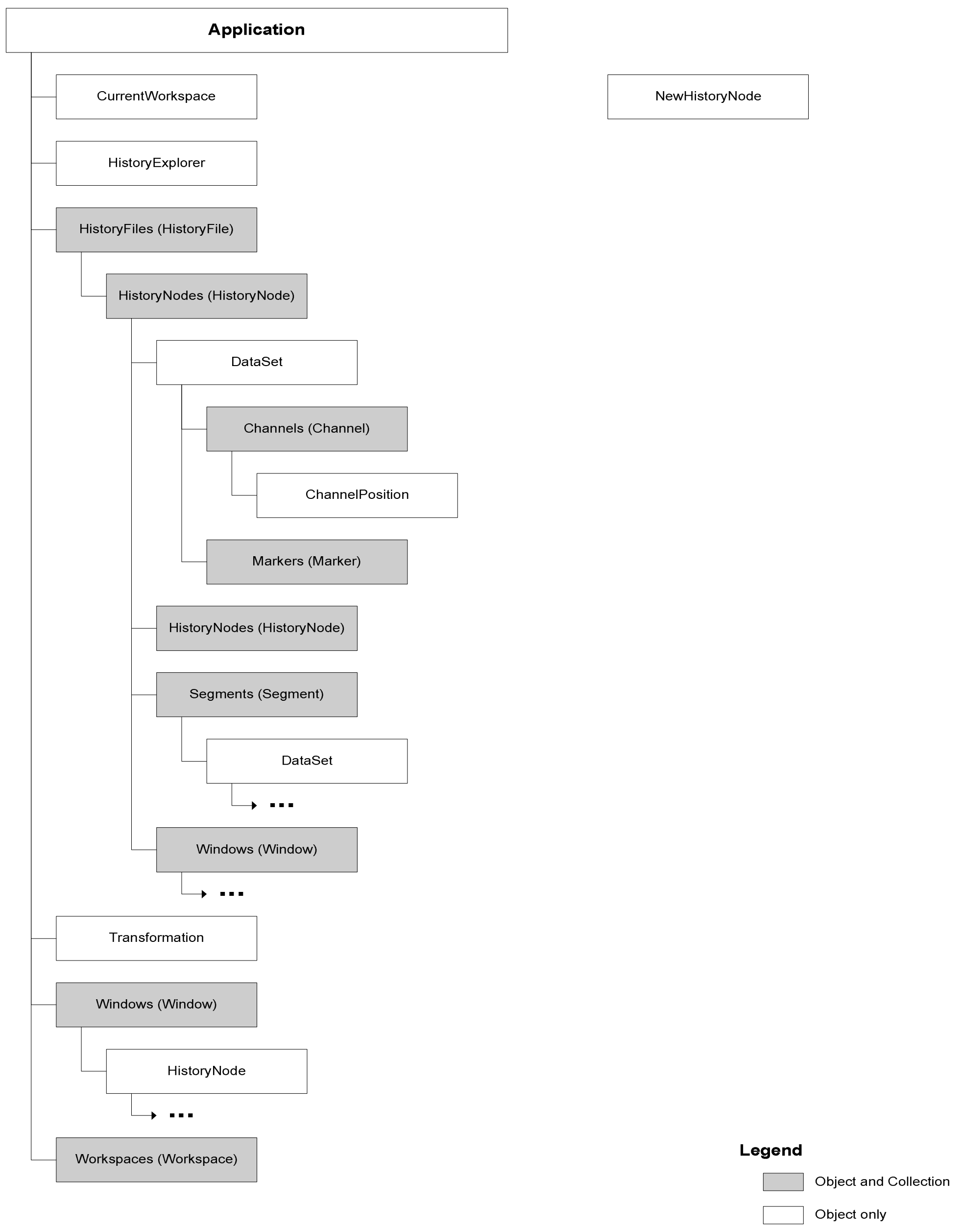

**Fig. 4-1: The Analyzer's object hierarchy**

Of the Application class, there is just one object which represents the program as a whole. It is the default object which means that its methods and properties can be accessed directly, i.e. HistoryFiles is equivalent to Application.HistoryFiles, for example.

The HistoryFiles object collection represents all history files in the current workspace. A single history file is represented as HistoryFile.

Every history file contains a HistoryNodes collection which normally contains an object of the HistoryNode type. As far as primary history files are concerned, that is the data node named Raw Data. In turn, every object in the HistoryNode class contains a HistoryNodes collection with references to derived data sets. You can navigate through an entire history file with all its branches in this way.

The HistoryNode object also has another object. This is the Dataset object which contains a collection of all markers in the data set (the Markers object) and a collection of channels (the Channels object). Finally, every Channel object gives you access to every data point. Access to the first data point of the first channel can therefore be described as follows:

```
Dataset.Channels(1).DataPoint(1)
```

Since Channels is the default element of Dataset, and DataPoint is the default element of Channels, the description can be shortened as follows:

```
Dataset(1)(1)
```

We will deal with other aspects of HistoryNode, Dataset and Channel objects in the "Practical examples" chapter.

The NewHistoryNode object stands somewhat apart from the other objects. It is used to create new history nodes. It is shown separately in the object hierarchy because it is regenerated when needed. Its use is described in detail in the next chapter.

# 5. Data manipulation with macros – the basics

In this chapter, data manipulation means changing of the actual data as well as removal and addition of markers, and changing of properties such as channel names or positions. In fact, no physical change is made to the data. Instead, a newly derived data set is generated in the way we are familiar with in transformations in the Analyzer. You can also delete this data set in exactly the same way as other data sets. Here, too, there are no limits to your scope for experimentation.

Before you use macros to manipulate data, however, you should check whether your problem can be solved by one of the existing transformation modules.

To generate a new data set we require a NewHistoryNode object. The easiest way to generate it is with the following line:

```
        Dim nhn as New NewHistoryNode
```

Basically, two procedures are responsible for creating and completing the node: Create and Finish.

Channel names, markers and data can be set between these two procedures. If Create was called successfully and the data was not inherited from the parent, a data set is created with standard settings. This can now be manipulated. Finish completes the creation process.

The newly generated history nodes can also be used in history templates to a limited extent.

To do this, the new node must be generated as a child node of the predefined variable named **ActiveNode**. This variable is always defined when the Basic interpreter is running. It is defined as follows:

```
        Dim ActiveNode As HistoryNode
```

The node represents the data window that is currently open. If no data window is open, this node does not contain any data.

To generate a data set that can be used in a template, you should close all open data windows apart from the window that is to act as the parent for the new child node. Then the code of the Basic macro is executed. Note that execution of the macro is terminated automatically when the NewHistoryNode.Finish() method is executed.

When the new node is created, the entire macro code is copied into it. Now you can drag the node to another node just like any normal transformation in order to repeat the operation. The node can also be included in a history template.

If you right-click the node to see the Operation Infos, then the code that generated the node is also displayed.

Below you will see some sample code which simply renames the first channel as xxx but otherwise leaves everything else unchanged. The new data is stored in the Basic Test node under the currently open history node. This code is for demonstration purposes only here. If you actually want to rename channels, you can do this more easily with the Edit Channels transformation.

```
Sub Main
```

```
    Dim nhn as New NewHistoryNode
    nhn.Create "BasicTest", ActiveNode
    nhn.SetChannelName 1, "xxx"
    nhn.Finish
End Sub
```

As you see, it only takes minimal effort to create a new data set or history node.

We just generated a data set which takes all its data from its predecessor. In this case, the macro only has very little work to do so it runs very fast. Furthermore, the new data set takes up very little space because, in fact, only a reference to the data is stored.

If, on the other hand, you generate really new data, this is stored in the history file. We therefore advise you to manipulate data only after averaging, if possible, because there is considerably less data than with a raw EEG. This also has a positive effect on the speed because the macro language is relatively slow. If you want to perform operations on raw data, you should therefore check whether a transformation module can do this job (e.g. Formula Evaluator). A mixed form of macros and transforms is also possible to a limited extent, as described in the "Dynamic parameterization" section of the following chapter.

It is a different matter if you only want to set, delete or rename markers. Since markers are stored internally in a table, a macro can act quite fast on raw EEGs, and the resultant data set only requires very little space. The same applies to the changing of properties such as channel names or positions.

The various types of new history nodes are explained in the "Practical examples" chapter. Please refer to the "NewHistoryNode" section of the "Object classes" chapter in the OLE Automation Reference Manual for the exact syntax that is required to create the nodes.

# 6. Practical examples

## 6.1. Automation

### 6.1.1. Compressing all history files

If you experiment a great deal with your data, create nodes in history files and delete them again, then this will normally give rise to fairly large gaps in the files. This may make the history files unnecessarily large. The following macro runs through all history files in the current workspace and compresses them, i.e. removes all gaps.

Compress All.vabs file:

```
' Compress all history files
Sub Main
    For each hf in HistoryFiles
        hf.Compress
    Next
End Sub
```

### 6.1.2. Printer batch run

The following macro searches for the Average data set in all history files. When one is found, it is printed out. If there are several data sets with the same name in the history file, only the first data set that is found is printed here.

PrintAverages.vabs file:

```
' Search in each history file for a node named "Average". If found, print it.
Sub Main
    For Each hf In HistoryFiles
        hf.Open
        Set hn = hf.FindNode("Average")
        If Not hn Is Nothing Then    ' "Average" node found?
            hn.Show
            ' When the node is shown, at least one window is attached.
            hn.Windows(1).Print
            Wait 2
        End If
        hf.Close
    Next
End Sub
```

### 6.1.3. Renaming a history node in all history files

Let's assume you have carried out segmentation on the basis of two criteria, i.e. you have one node named Segmentation and another named Segmentation2, for example. Then you performed some operations until you came to the average. Your history file now contains two different nodes with the same name – Average. You carried out the operation with 162 history files, and would now like a grand average for the average derived from Segmentation 2. You will find that this doesn't work! The Grand Average module only accepts one node name, for example Average, and then uses the first node that it finds in the history file. Now you can either create a history template from one of the files and rename the second Average as Average 2 and then run the template, or rename the second average node in every existing history file, or run the following macro:

RenameToAverage2.vabs file:

```
' Search for "Average" below "Segmentation 2" and rename it as "Average 2".
' This macro assumes that there is no branch below "Segmentation 2".
Sub Main
    Dim hf As HistoryFile
    Dim hn As HistoryNode
    For Each hf In HistoryFiles
        hf.Open
        Set hn = hf.FindNode("Segmentation 2")
        If Not hn Is Nothing Then   ' "Segmentation 2" found?
            Dim nChildren As Long   ' Check for children of node.
            Dim hn2 As HistoryNode
            Set hn2 = hn
            nChildren = hn2.HistoryNodes.Count
            If nChildren > 1 Then   ' branch found.
                MsgBox "Branch found in " & hf.DisplayName & "!", "Warning"
            End If
            Do While nChildren > 0
                Set hn2 = hn2.HistoryNodes(1)
                If StrComp(hn2.Name, "Average", 1) = 0 Then  ' Case-insensitive comparison.
                    ' Rename it.
                    hn2.Name = "Average 2"
                End If
                nChildren = hn2.HistoryNodes.Count
                If nChildren > 1 Then      ' branch found.
                    MsgBox "Branch found in " & hf.DisplayName & "!", "Warning"
                End If
            Loop
        End If
        hf.Close
    Next
End Sub
```

The macro shown above has one drawback. There must be no branches after Segmentation 2, i.e. Segmentation 2 and every subsequent node is only allowed to have one child node. If that is not the case, a warning is output and an Average node may be overlooked.

In order to take all branches into consideration, you can use the following macro which has a somewhat more complicated structure. It operates with recursion, i.e. a function is defined which calls itself. Information on the structure of functions is given in the "Tips for advanced users" chapter. The FindSubNode function searches for a node with the specified name beneath a specified node. It may also be of interest for other applications.

RenameToAverage2Rec.vabs file:

```
' Search for "Average" below "Segmentation 2" and rename it as "Average 2".
Sub Main
    Dim hf As HistoryFile
    Dim hn As HistoryNode
    For Each hf In HistoryFiles
        hf.Open
        Set hn = hf.FindNode("Segmentation 2")
        If Not hn Is Nothing Then   ' "Segmentation 2" node found?
            Dim hnAverage As HistoryNode
            Set hnAverage = FindSubNode(hn, "Average")
            If Not hnAverage Is Nothing Then
              hnAverage.Name = "Average 2"
            End If
        End If
        hf.Close
    Next
End Sub

' This function searches recursively for a history node with the given name below the given
' node.
Function FindSubNode(hn As HistoryNode, sName As String) As HistoryNode
    Dim hnChild As HistoryNode
    For Each hnChild In hn.HistoryNodes
        If StrComp(hnChild.Name, sName, 1) = 0 Then       ' Case-insensitive comparison.
```

```
            Set FindSubNode = hnChild
            Exit Function
        End If
        Set FindSubNode = FindSubNode(hnChild, sName)       ' Recursive call
        If Not FindSubNode Is Nothing Then         ' Found?
            Exit Function
        End If
    Next
End Function
```

### 6.1.4. Exporting graphics to Winword with a report

The following macro copies the content of the current data window to the clipboard, launches Microsoft Word 2000, copies the data into it and then leaves it up to you to carry out any further manipulation.

CopyToWord.vabs file:

```
' Copy content of a window to the clipboard.
' and then paste it into a new Word document.
Sub Main
    If Not ActiveNode.DataAvailable Then
       MsgBox "This macro needs an open data window."
       Exit Sub
    End If
    Set hn = ActiveNode
    hn.Show   ' Now it should be the active window.
    hn.Windows(1).Copy
    ' Word 97 must be on the machine.
    ' The following commands are Word commands.
    Set Word = CreateObject("Word.Application")
    Word.Visible = True
    Word.Documents.Add
    Word.Selection.TypeText hn.HistoryFile.DisplayName & "-" & hn.Name
    Word.Selection.TypeParagraph
    Word.Selection.Paste
End Sub
```

The smaller part of the macro comprises Analyzer commands, whereas the larger part comprises Word Automation commands which we will not explain here. Please refer to Microsoft's Word documentation for more details of that.

The CreateObject function is worth mentioning. It is used to start external applications and control them remotely by means of OLE Automation. This applies to the Microsoft-Office programs (Word, Excel, Powerpoint, Access, Outlook) and many more (Visio, SPSS etc.).

If you need a simple report for a data set, you can use the following macro. It also works with Word 2000. Again, the Word application is controlled remotely from the macro. The macro generates a new file which has the name of the current history file and the current data set. The new file is stored in the Export folder of the current work space. You define the Export folder in the Analyzer under **File > Edit Workspace**.

WordReport.vabs file:

```
' Copy content of the active window to the clipboard.
' Create a Word document.
' Write title.
' Paste clipboard content.
Sub Main
    On Error GoTo CheckError
    If Not ActiveNode.DataAvailable Then
        MsgBox "This macro needs an open data window."
        Exit Sub
    End If
    Set hn = ActiveNode
    hn.Windows(1).Copy       ' When the active node has valid data it also has at
                             ' least one attached window.
    ' Save new document in export folder.
    Dim sOutput As String
    ' Build output file name.
    sOutput = CurrentWorkspace.ExportFileFolder & "\" & hn.HistoryFile.DisplayName & _
              "-" & hn.Name & ".doc"

    ' MS Word 97 must be on the machine.
    ' The following commands are Word commands.
    ' Look at the Office documentation for the Word object model .
    Set WordDoc = CreateObject("Word.Document")
    WordDoc.Select
    Dim Sel As Object
    Set Sel = WordDoc.Application.Selection ' Reference Word selection object.
    ' Save current font.
    With Sel.Font
        OldBold = .Bold : OldName = .Name
        .Bold = True : .Name = "Arial"
    End With
    ' Write caption.
    Sel.TypeText hn.HistoryFile.DisplayName & "-" & hn.Name
    Sel.TypeParagraph
    ' Restore font
    With Sel.Font
        .Bold = OldBold : .Name = OldName
    End With
    Sel.Paste
    Sel.MoveEnd
    Sel.TypeParagraph
    WordDoc.SaveAs sOutput                                ' Save document.
    WordDoc.Close

    Exit Sub

CheckError:
    MsgBox Err.Description, vbExclamation, "Error"
End Sub
```

## 6.2. Data manipulation

### 6.2.1. Removing, setting and renaming markers

Macros can remove markers and set new ones. A marker is renamed by removing a marker and setting a new one at the same position.

The main use for marker manipulation is to prepare for special segmentation. Although the Analyzer's segmentation module contains a very powerful method for intelligent segmentation in the shape of *Advanced Boolean Expression (ABE)*, it cannot allow for all conceivable segmentation algorithms. Nevertheless you should check whether ABE is capable of solving your problem before using a macro for this purpose.

Here is our first example. The first five stimuli in a data set are to be ignored, and then the next 500 stimuli are to be included in averaging. To do this, the following macro simply deletes all unrequired stimuli from the data set. An error message is output if there are not enough stimuli in the data set.

500Stimuli.vabs file:

```
' Remove all stimulus markers from 1 to 5 and > 505.
' -> keep exactly 500 stimuli.
Sub Main
     Dim nhn As New NewHistoryNode
     nhn.Create "500 Stim", ActiveNode
     Dim Mks As Markers
     Dim mk As Marker
     Dim i As Long
     Set Mks = ActiveNode.Dataset.Markers
     For Each mk In Mks
          If mk.Type = "Stimulus" Then
               i = i + 1
               If i < 6 Or i > 505 Then
                    nhn.RemoveMarker mk.ChannelNumber, mk.Position, mk.Points, mk.Type, _
                         mk.Description
               End If
          End If
     Next
     If i < 505 Then   ' Not enough markers?
          MsgBox i & " Markers in Dataset!", "Macro 500Stimuli"
          Exit Sub
     End If
     nhn.Finish
End Sub
```

If you only want to include every third stimulus marker named S  1 in averaging, you have two ways of doing this.

1. Delete all other stimuli markers named S..1 from the data set, and then carry out segmentation on the basis of the remaining stimuli.
2. Rename every third S..1 stimulus marker, and then carry out segmentation on the basis of stimuli with the new name.

The ThirdS1a.vabs macro applies the first method:

```
' Search for "S  1" stimulus markers and erase them if they are not
' divisible by 3, i.e. keep every third stimulus marker "S  1".
Sub Main
     On Error GoTo CheckError
     Dim sDescription As String
     sDescription = "S  1"    ' Change the string for a different stimulus, be careful with
                              ' spaces in the name, "S  1" contains two spaces.
```

```
    Dim nhn As New NewHistoryNode
    nhn.Create "Third S1a", ActiveNode
    Dim Mks As Markers
    Set Mks = ActiveNode.Dataset.Markers
    Dim mk As Marker
    Dim i As Long
    For Each mk In Mks
         If mk.Type = "Stimulus" And mk.Description = sDescription Then
              i = i + 1
              If i Mod 3 Then ' Not divisible by 3?
                   nhn.RemoveMarker 0, mk.Position, mk.Points, mk.Type, mk.Description
              End If
         End If
    Next
    nhn.Finish  ' Finish creation.
    Exit Sub

CheckError:      ' Error
    MsgBox Err.Description
End Sub
```

If you are confronted with a similar problem, you can change the sixth line to use a different stimulus ("sDescription = ..."), and line number 17 ("if i Mod 3 then") to set a different devisor.

The ThirdS1b.vabs macro applies the second method, i.e. it renames every third S  1 marker.

```
' Search for "S  1" stimulus markers and rename them if they are
' divisible by 3, i.e. rename every third stimulus marker "S  1" to "1000Hz".
Sub Main
    On Error GoTo CheckError
    Dim sDescription As String
    sDescription = "S  1"    ' Change the string for a different stimulus, be careful with
                             ' spaces in the name, "S  1" contains two spaces.
    Dim nhn As New NewHistoryNode
    nhn.Create "Third S1b", ActiveNode
    Dim Mks As Markers
    Set Mks = ActiveNode.Dataset.Markers
    Dim mk As Marker
    Dim i As Long
    For Each mk In Mks
          If mk.Type = "Stimulus" And mk.Description = sDescription Then
               i = i + 1
               If i Mod 3 = 0 Then   ' Divisible by 3?
                       ' Rename marker: remove / add
                       nhn.RemoveMarker 0, mk.Position, mk.Points, mk.Type, mk.Description
                       nhn.AddMarker 0, mk.Position, mk.Points, mk.Type, "1000Hz"
               End If
          End If
    Next
    nhn.Finish' Finish creation.
    Exit Sub

CheckError:           ' Error
    MsgBox Err.Description
End Sub
```

### 6.2.2. Generating new data

In the following macro, a new data set is generated as a child node of an existing one. This contains the rectified data of the original data set. Since the data is not inherited, properties and markers have to be set explicitly. They are copied from the original data set by means of the CopyProperties and CopyMarkers procedures. Owing to their encapsulation, these procedures can be transferred to other macros very easily.

Rectify Data.vabs file:

```
' Rectify data of the active node.
Sub Main
    If Not ActiveNode.DataAvailable Then
```

```
          MsgBox "This macro needs an open data window."
          Exit Sub
     End If
     Dim ds As Dataset
     Set ds = ActiveNode.Dataset
     ' Limit operation to small data sets, i.e. Averages etc.
     If ds.Length > 10000 Then
          MsgBox "Data set contains " & ds.Length & _
                    " data points (macro is limited to 10000 data points)."
          Exit Sub
     End If
     Dim nhn As New NewHistoryNode
     ' Create new data set.
     nhn.Create "Rectify", ActiveNode, "", False, ds.Type, ds.Channels.Count, ds.Length, _
                    ds.SamplingInterval

     ' Description of operation (operation info)
     nhn.Description = "Rectify all channels"
     ' Copy properties.
     CopyProperties ds, nhn
     ' Copy markers.
     CopyMarkers ds, nhn

     ' Read / modify / write data
     Dim fData() As Single
     Dim Chs As Channels
     Set Chs = ds.Channels
     Dim ch As Channel
     For i = 1 To Chs.Count
          Set ch = Chs(i)
          ' Read
          ch.GetData 1, ds.Length, fData
          ' Modify
          For j = 1 To ds.Length
               fData(j) = Abs(fData(j))
          Next
          ' Write
          nhn.WriteData i, 1, ds.Length, fData
     Next
     nhn.Finish
End Sub

' Copy properties from source node to target node.
Sub CopyProperties(dsSrc As Dataset, nhnTarget As NewHistoryNode)
     Dim i As Long
     Dim Chs As Channels
     Set Chs = dsSrc.Channels
     Dim ch As Channel
     For i = 1 To Chs.Count
          Set ch = Chs(i)
          nhnTarget.SetChannelName i, ch.Name
          nhnTarget.SetRefChannelName i, ch.ReferenceChannel
          nhnTarget.SetChannelUnit i, ch.Unit
          Dim pos As ChannelPosition
          Set pos = ch.Position
          nhnTarget.SetChannelPosition i, pos.Radius, pos.Theta, pos.Phi
     Next
     nhnTarget.SegmentationType = dsSrc.SegmentationType
     nhnTarget.Averaged = dsSrc.Averaged
End Sub

' Copy markers from source node to target node.
Sub CopyMarkers(dsSrc As Dataset, nhnTarget As NewHistoryNode)
     Dim mk As Marker
     Dim Mks As Markers
     Set Mks = dsSrc.Markers
     For Each mk In Mks
          nhnTarget.AddMarker mk.ChannelNumber, mk.Position, mk.Points, _
                    mk.Type, mk.Description, mk.Invisible
     Next
End Sub
```

The following macro creates a new secondary history file containing all FP1 channels of all Average nodes of the primary history files in the current workspace.

Collect FP1.vabs file:

```
' Look in each primary history file for history node "Average" with channel "Fp1".
' If the node and the channel exist, add the channel to a new secondary
' history file called "Collect Fp1", history node "Fp1".
Option Explicit
Sub Main
    Dim sFiles() As String     ' Container for valid history file names.
    Dim hf As HistoryFile
    Dim hn As HistoryNode
    Dim nCount As Long, nLength As Long, nType As Long
    Dim fSamplingInterval As Double
    ' First count number of files that match the criteria.
    For Each hf In HistoryFiles
        If hf.LinkedData Then ' Primary history file?
            hf.Open
            Set hn = hf.FindNode("Average")
            If Not hn Is Nothing Then
                If Not hn.Dataset("Fp1") Is Nothing Then
                    If nCount = 0 Then           ' Use first data set length as reference
                        nLength = hn.Dataset.Length
                        nType = hn.Dataset.Type
                        fSamplingInterval = hn.Dataset.SamplingInterval
                    End If
                    ' Only data sets with the same length.
                    If nLength = hn.Dataset.Length Then
                        nCount = nCount + 1
                        ReDim Preserve sFiles(1 To nCount)      ' Resize container of names.
                        sFiles(nCount) = hf.DisplayName
                    End If
                End If
            End If
            hf.Close
        End If
    Next
    ' Now we know the number of channels for the new history node.
    Dim nhn As New NewHistoryNode
    HistoryFiles.KillFile "Collect Fp1"     ' Kill secondary history file if it exists.
    ' Create a new history file called "Collect Fp1" with the node "Fp1"
    nhn.Create "Fp1", Nothing, "Collect Fp1", False, nType, nCount, nLength, fSamplingInterval
    Dim i As Long
    Dim fData() As Single
    For i = 1 To nCount
        nhn.SetChannelName i, sFiles(i) & "-Fp1"   ' Set name of new channel
        ' Copy data.
        Set hf = HistoryFiles(sFiles(i))
        hf.Open
        Set hn = hf.FindNode("Average")
        Dim ch As Channel
        Set ch = hn.Dataset("Fp1")
        ch.GetData 1, nLength, fData
        nhn.WriteData i, 1, nLength, fData
        hf.Close
    Next
    nhn.Finish
End Sub
```

### 6.2.3. Reading in stimulator data from external files

The following macro reads stimulus/response data from a text file. It is assumed that the associated stimulus markers exist in the EEG. The correctness of the response serves as information for renaming the stimulus markers. A "-c" or "-i" is added to the marker descriptions, depending on whether the response was correct or incorrect.

The text file contains five columns of numbers separated by blanks. The first column contains the ordinal number of the stimuli, and the fourth column contains information on the correctness of the response – where 0 stands for incorrect and 1 for correct. The other columns are ignored. If a line begins with a non-numeric character (except space), it is skipped. Leading spaces in the lines are ignored.

Example of a line:

```
12 0 0 1 0
```

Here we have the 12th stimulus (column 1) and the response is correct (column 4). Stimulus files with this structure are relatively frequent. If your files have a different structure, you can adjust the macro easily.

The stimulus info file must be located in the raw data folder of the current workspace. Its base name corresponds to the base name of the associated EEG file. The file name extension is ".stm". Example:
Raw EEG: E0000001.eeg, associated stimulus info file: E0000001.stm.

This naming convention makes it possible to automate the reading in of response data into templates.

ReadResponses.vabs file:

```
' This macro reads stimulus / response information from a text file.
' It is assumed that the stimuli are also recorded in the EEG.
' The correctness of the response is used to rename the stimulus markers.
' A marker's description is expanded with "-c" if the response is correct and with
' "-i" if it is incorrect.
' The text file contains five columns of numbers:
' The first column contains the stimulus number, the fourth column the correctness where
' 1 indicates "correct" and 0 indicates "incorrect". The other columns are ignored. If
' a line starts with a non-digit character, the line is skipped. Leading spaces are ignored.
' The file must be in the raw data folder and have the extension ".stm". Its base name must
' be the same as the base name of the raw EEG file. For example when the raw file is called
' "E0000001.eeg" the name of the corresponding stimulus information file is "E0000001.stm".

Const sExtension As String = ".stm" ' Extension of stimulus information file.
                                    ' Change if your info files have a different extension.
Sub Main
     If Not ActiveNode.DataAvailable Then
          MsgBox "This macro needs an open data window."
          Exit Sub
     End If

     Dim sFolder As String ' Folder / directory of stimulus information file.
     ' Set folder (directory) of the stimulus information file to the raw file folder.
     ' Change the following line if your info files are located in a different folder.
     sFolder = CurrentWorkspace.RawFileFolder
     ' Build the full file name
     Dim sStimFile As String
     sStimFile = sFolder & "\" & ActiveNode.HistoryFile.Name & sExtension

     Dim nFile As Integer
     nFile = FreeFile      ' Get a free file handle number.
     Open sStimFile For Input As nFile   ' Open stimulus info file
     Dim tblCorrectness() As Long        ' Array of correctness flags
     Dim nStimCount As Long         ' Number of items of stimulus info in the stim. info file
     ' Read in lines and fill tblCorrectness array.
     Do Until EOF(nFile)
          Dim sLine As String
          Line Input #nFile, sLine
          sLine = Trim(sLine)   ' Remove leading spaces.
          ' Check whether the first character is a number.
          If IsNumeric(Left(sLine, 1)) Then
               ' Retrieve the different numbers.
               Dim cols(1 To 5) As Single ' Array of numbers for each column.
               Dim i As Long, nPos As Long, nStim As Long, nCorrectness As Long
               For i = 1 To 5
                    nPos = InStr(sLine, " ")
```

```
                If nPos > 0 Then
                    cols(i) = CSng(Left(sLine, nPos - 1))
                    sLine = LTrim(Mid(sLine, nPos))
                else
                    cols(i) = CSng(sLine)
                End If
            Next
            nStim = CLng(cols(1))      ' Stimulus number is in column 1.
            ' Handle ascending, descending and no order of stimuli.
            If nStim > nStimCount Then
                ReDim Preserve tblCorrectness(1 To nStim)
                nStimCount = nStim
            End If
            tblCorrectness(nStim) = CLng(cols(4)) ' Correctness flag is in column 4.
        End If
    Loop
    Close nFile

    ' Now we have all the information we need to set the stimulus markers.
    Dim nhn As New NewHistoryNode
    nhn.Create "Correctness", ActiveNode
    Dim Mks As Markers
    Dim mk As Marker
    Set Mks = ActiveNode.Dataset.Markers
    Dim nStimuliFound As Long, nCorrectResponses As Long, nIncorrectResponses As Long

    For Each mk In Mks
        If mk.Type = "Stimulus" Then
            nStimuliFound = nStimuliFound + 1
            ' Leave loop if no more entries are in tblCorrectness.
            If nStimuliFound = nStimCount Then Exit For
            Dim sDescription As String
            sDescription = mk.Description
            If tblCorrectness(nStimuliFound) > 0 Then             ' Correct response?
                sDescription = sDescription & "-c"
                nCorrectResponses = nCorrectResponses + 1
            Else
                sDescription = sDescription & "-i"
                nIncorrectResponses = nIncorrectResponses + 1
            End If
            ' Call procedure to rename the marker.
            RenameMarkerDescription mk, nhn, sDescription
        End If
    Next

    ' Write descriptions for operation info.
    ' Operation, inherited by templates
    nhn.Description = "Checked responses for correctness and coded stimulus markers with " & _
                      "'-c' (correct) or '-i' (incorrect)" & vbCrLf & vbCrLf
    ' Operation results, not inherited by templates
    nhn.Description2 = "Correct responses found:   " & nCorrectResponses & vbCrLf & _
                        "Incorrect responses found: " & nIncorrectResponses
    nhn.Finish
End Sub

' It is not possible to rename a marker description directly. This procedure does
' this by removing a marker, and then adding a new one.
Sub RenameMarkerDescription(mk As Marker, nhn As NewHistoryNode, sNewDescription As String)
    With mk
        nhn.RemoveMarker .ChannelNumber, .Position, .Points, .Type, .Description
        nhn.AddMarker .ChannelNumber, .Position, .Points, .Type, sNewDescription
    End With
End Sub
```

### 6.2.4. Reading in channel positions from external files

The following macro reads in electrode positions from a position file and sets them in a new data set.

The position file has the following line format:

<Electrode Name>, <Radius>, <Theta>, <Phi>

Example:

```
Fp1,1,-92,-72
Fp2,1,92,72
```

Lines starting with the comment character (#) are skipped.

Please refer to the user manual for the exact definition of electrode positions as used in the Analyzer.

The position file must be located in the raw data folder of the current workspace. Its base name corresponds to the base name of the associated EEG file. The file name extension is ".pos". Example:
Raw EEG: E0000001.eeg, associated position file: E0000001.pos.

ReadPositions.vabs file:

```
' This macro reads electrode positions from a text file.
' The text file has the following line format
' (of course without the leading comment character "'"):
' <Electrode Name>,<Radius>,<Theta>,<Phi>
' Example:
'
' Fp1,1,-92,-72
' Fp2,1,92,72
'
' The file must be in the raw data folder and have the extension ".pos". Its base name must
' be the same as the base name of the raw EEG file. For example, when the raw file is named
' "E0000001.eeg" the name of the corresponding position file is "E0000001.pos".
Option Explicit

Const sExtension As String = ".pos" ' Extension of electrode position file.
                                    ' Change if your info files have a different extension.
Sub Main
    If Not ActiveNode.DataAvailable Then
        MsgBox "This macro needs an open data window."
        Exit Sub
    End If

    Dim sFolder As String ' Folder / directory of stimulus information file.
    ' Set folder (directory) of the position file to the raw file folder.
    ' Change the following line if your position files are located in a different folder.
    sFolder = CurrentWorkspace.RawFileFolder
    ' Build the full file name
    Dim sPosFile As String
    sPosFile = sFolder & "\" & ActiveNode.HistoryFile.Name & sExtension

    Dim nFile As Integer
    nFile = FreeFile      ' Get a free file handle number.
    Open sPosFile For Input As nFile ' Open position info file

    ' Create new node.
    Dim nhn As New NewHistoryNode
    nhn.Create "Read Pos", ActiveNode

    ' Read in lines.
    Do Until EOF(nFile)
        Dim sLine As String
        Line Input #nFile, sLine
        sLine = Trim(sLine)   ' Remove leading spaces.
        ' Check whether the first character is a comment character.
        If Not Left(sLine, 1) = "#" Then   ' No comment?
            ' Retrieve the different columns.
            Dim cols(1 To 4) As String     ' Array of strings for each column.
            Dim i As Long, nChannel As Long, nPos As Long
            For i = 1 To 4
                nPos = InStr(sLine, ",")
                If nPos > 0 Then
                    cols(i) = Left(sLine, nPos - 1)
                    sLine = LTrim(Mid(sLine, nPos + 1))
                Else
                    cols(i) = sLine
                End If
            Next
            ' Do we have a corresponding channel in the data set?
            nChannel = GetChannelIndex(ActiveNode, cols(1))
            If nChannel > 0 Then
                nhn.SetChannelPosition nChannel, CLng(cols(2)),  CLng(cols(3)), _
                     CLng(cols(4))
            End If
        End If
    Loop
    Close nFile

    ' Write descriptions for operation info.
    ' Operation, inherited by templates
    nhn.Description = "Read electrode positions from external file." & vbCrLf & vbCrLf
    ' Operation results, not inherited by templates
    nhn.Description2 = "Position info file: " & sPosFile & vbCrLf
    nhn.Finish
End Sub

' Get the index of the channel that matches the given label in the given history node.
Function GetChannelIndex(hn As HistoryNode, sLabel As String) As Long
    Dim Chs As Channels
    Dim ch As Channel
    Set Chs = hn.Dataset.Channels
    Dim i As Long
    For i = 1 To Chs.Count
        If StrComp(Chs(i).Name, sLabel, 1) = 0 Then  ' Found?
            GetChannelIndex = i
            Exit Function
        End If
    Next
End Function
```

A modified version of this macro – named  ReadPosXYZ.vabs – reads coordinates in XYZ format and converts them to the internal coordinate system.

### 6.2.5. Exporting frequency data to an ASCII file

The following macro exports the alpha band, which is defined here as ranging from 7.5 to 12.5 hertz, from the currently displayed frequency data set. All values in this range are exported. It will not take much effort to export just the average instead. The macro automatically checks whether there is any complex frequency data. In this case, the absolute values are exported.

The macro generates a new file based on the name of the current history file and the current data set. The new file is stored in the Export folder of the current workspace. You define the Export folder in the Analyzer under **File > Edit Workspace**.

ExportAlpha.vabs file:

```
' Export a frequency interval to an ASCII file.
Option Explicit
' Define band:
Const fIntervalStart = 7.5     ' Start in hertz
Const fIntervalLength = 5      ' Length in hertz

Sub Main
     If Not ActiveNode.DataAvailable Then
          MsgBox "This macro needs an open data window."
          Exit Sub
     End If
     Dim ds As Dataset
     Set ds = ActiveNode.Dataset
     If ds.Type <> viDtFrequencyDomain And ds.Type <> viDtFrequencyDomainComplex Then
          MsgBox "This macro expects data in the frequency domain."
          Exit Sub
     End If

     ' Build file name based on the export folder, the history file and the history node.
     Dim sFilename As String
     sFilename = CurrentWorkspace.ExportFileFolder & "\" & ActiveNode.HistoryFile.DisplayName _
                 & "_" & ActiveNode.Name & "_Alpha.txt"

     ' Check for the interval.
     Dim nFirstPoint As Long, nPoints As Long
     ' First data point
     nFirstPoint = fIntervalStart / ds.SamplingInterval + 1
     nPoints = fIntervalLength / ds.SamplingInterval
     If nFirstPoint + nPoints - 1 > ds.Length Then  ' Out of range?
          MsgBox "The requested interval is out of range."
          Exit Sub
     End If
     Dim nFile As Long ' File handle
     nFile = FreeFile
     ' Create output file.
     Open sFilename For Output As nFile
     Dim ch As Channel
     For Each ch In ds.Channels
          ' Write channel names first.
          Print #nFile, ch.Name;
          Dim fData() As Single
          ch.GetData nFirstPoint, nPoints, fData
          Dim i As Long
          Dim fValue As Single
          For i = 1 To nPoints
               ' Complex data ? -> convert.
               If ds.Type = viDtFrequencyDomainComplex Then
                    ' fData() contains nPoints * 2 values if complex
                    fValue = Sqr(fData((i - 1 ) * 2 + 1)^2 + fData((i - 1 ) * 2 + 2)^2)
               Else
                    fValue = fData(i)
               End If
               Print #nFile, " " & fValue;
          Next
          Print #nFile ' CrLf
```

```
    Next
    Close nFile
End Sub
```

If you want to use this macro in a history file in order to export the alpha band automatically again and again, you can apply the following trick. Insert the statements to create a new history node between the last two lines. This node merely acts as a home for the macro but does not change anything regarding the data.

ExportAlpa2.vabs file:

```
    Close nFile
    ' Build a new data set as home of the macro. This allows the macro to be used
    ' in a history template.
    Dim nhn As New NewHistoryNode
    nhn.Create "Export Alpha", ActiveNode
    nhn.Description = "Export Alpha Band" & vbCrLf
    nhn.Description2 = "Exported to '" & sFilename & "'"
    nhn.Finish
End Sub
```

## 6.3. Dynamic parameterization

Some transforms can be called via the *Transformation* object. We talk about dynamic parameterization because the macro can calculate and pass the parameters for transforms, e.g. based on the result of an FFT and the like, at run time.

The list of transforms that you can call with a macro, together with their parameter syntax, is given in the "Callable transforms" chapter of the OLE Automation Reference Manual.

The advantages of dynamic parameterization using existing transforms compared with complete implementation of algorithms in the macro are:

- Algorithms do not have to be developed.
- Data is calculated much faster. Raw data can also be transformed fast.
- The space required in the history file is normally slight because most transforms do not calculate their data until requested and do not store it in the history file.

The combination of macros and the Formula Evaluator transform provides some very interesting options. You could, for instance, calculate new channels from existing ones and use the actual measured channel positions instead of standard positions.

The following example uses the Filter transform. Here, raw data based on an FFT evaluation is to be filtered in the alpha band with a bandpass. First, the following transform sequence, which is also based on raw data, is carried out: Segmentation, Artifact Rejection, FFT, Average.

Now the frequency in the alpha band of the O1 channel that has the strongest amplitude is to be used for bandpass filtering (+/-3 Hz) of all channels in the raw data set. The following figure shows the relationship between nodes.

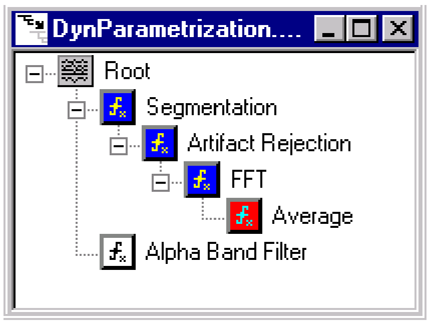

**Fig. 6-1: Relationship between nodes**

The macro generates the *Alpha Band Filter* node based on an analysis of the *Average Node*. You will find the *Transformation.TryLater* call in the macro. This should be used when data is needed from a side-branch of the history tree. It is possible that the *Alpha Band Filter* branch is calculated first when this tree is used in a history template. In this case the FFT data is missing, though. *TryLater* causes the template processor to carry on with the next branch and to try to calculate the *Alpha Band Filter* node again later. If this fails again although no new nodes can be calculated, the template processor gives up.

*DynParameterization.vabs* file:

```
' Example for Dynamic Parameterization
' This macro looks for a history node 'Average' that contains an averaged FFT.
' If found, it looks in the Alpha band for the maximum amplitude in channel "O1".
' Then it uses the frequency at the maximum amplitude as input parameter for
' a bandpass (+/-3Hz) to filter the active data set with the 'Filters' transformation.
Option Explicit
' Define band:
Const fIntervalStart = 7.5     ' Start in Hertz
Const fIntervalLength = 5      ' Length in Hertz
Const fBandwidth = 6           ' Bandwidth (+/-3Hz)
Const sTestChannel = "O1"      ' Test channel, i.e. channel where the Alpha band is checked.

Sub Main
     If Not ActiveNode.DataAvailable Then
          MsgBox "This macro needs an open data window."
          Exit Sub
     End If
     On Error Resume Next

     ' Look for node 'Average' that is in frequency domain
     Dim FFTAverageNode As HistoryNode
     Set FFTAverageNode = ActiveNode.HistoryFile.FindNode("Average")
     Do
          ' No more history node? -> try later if in template mode.
          If FFTAverageNode Is Nothing Then
               Transformation.TryLater
               ' If the macro has not terminated here, we are not in template mode.
               MsgBox "Missing history node 'Average' (FFT average)"
               Exit Sub
          End If
          Dim ds As Dataset
          Set ds = FFTAverageNode.Dataset
          ' Frequency domain?
          If ds.Type = viDtFrequencyDomain Or ds.Type = viDtFrequencyDomainComplex Then
               Exit Do
          End If
          ' Not frequency domain? -> Look for the next history node with the same name.
          Set FFTAverageNode = ActiveNode.HistoryFile.FindNextNode
     Loop

     ' Now we have our averaged FFT history node.
     ' Lets get the data from the test channel.
     Dim ch As Channel
     Set ch = ds(sTestChannel)
     If ch Is Nothing Then
          Message "Missing test channel '" & sTestChannel & "'"
          Exit Sub
     End If

     ' Check for the interval.
     Dim nFirstPoint As Long, nPoints As Long
     nFirstPoint = fIntervalStart / ds.SamplingInterval + 1
     nPoints = fIntervalLength / ds.SamplingInterval
     If nFirstPoint + nPoints - 1 > ds.Length Then  ' Out of range?
          Message "The requested interval is out of range."
          Exit Sub
     End If

     ' Look for maximum value in the defined interval.
     Dim fData() As Single
     ch.GetData nFirstPoint, nPoints, fData
     Dim i As Long
     Dim fValue As Single, fMax As Single
     Dim nMaxPosition As Long
     fMax = -1
     nMaxPosition = 0
     For i = 1 To nPoints
          ' Complex data ? -> convert.
          If ds.Type = viDtFrequencyDomainComplex Then
               ' fData() contains nPoints * 2 values if complex
               fValue = Sqr(fData((i - 1 ) * 2 + 1)^2 + fData((i - 1 ) * 2 + 2)^2)
          Else
               fValue = fData(i)
```

```
        End If
        If fValue > fMax Then ' New maximum found?
            fMax = fValue
            nMaxPosition = i - 1
        End If
    Next

    ' Convert position to frequency
    Dim fFrequency As Single
    fFrequency = (nFirstPoint + nMaxPosition - 1) * ds.SamplingInterval

    ' Build parameter string
    Dim sParameters As String
    sParameters = "Lowcutoff=" & SingleToString(fFrequency - fBandWidth / 2) _
        & ",24;highcutoff=" & SingleToString(fFrequency + fBandWidth / 2) & ",24"

    ' Perform the transformation
    Transformation.Do "Filters", sParameters, ActiveNode, "Alpha Band Filter"

    Exit Sub

CheckError:
    Resume Next
End Sub

' The function converts a single value to a string. In opposite to 'Str' the decimal delimeter
' is always a dot ('.'), even in European countries.
Function SingleToString(fValue As Single) As String
    SingleToString = Replace(Str(fValue), ",", ".")
End Function
```

# 7. Tips for advanced users

## 7.1. Declaring variables

As mentioned in the "Quick tour of Basic" chapter, you have the option of declaring variables explicitly:

```
Dim fData as Single
```

You can make it compulsory to declare variables, though. You do this by means of the following statement:

```
Option Explicit
```

You have to insert this statement as the first line of the macro, above Sub Main. In this case you get an error message when you try to run the macro and it uses undeclared variables. The advantage of this is that it is no longer possible to make typing mistakes in variable names which otherwise would have caused variables to be created implicitly.

Example:

```
Dim fValue as Single
fValue = 1
fValue = fVolue + 1
```

Here we wrote fVolue + 1 in the last line instead of fValue + 1. If Option Explicit is not specified, no error message is output. There would simply be the wrong result as fVolue is created implicitly and set to 0.

Such errors are extremely hard to find in larger macros so it is generally worth using Option Explicit there.

## 7.2. User-defined dialog boxes

In addition to the input and output facilities provided by InputBox and MsgBox, you can also create more complex dialog boxes. To do this, create a new macro with **Macro > New,** and then select **Edit > User Dialog**. A dialog appears enabling you to create a dialog template.

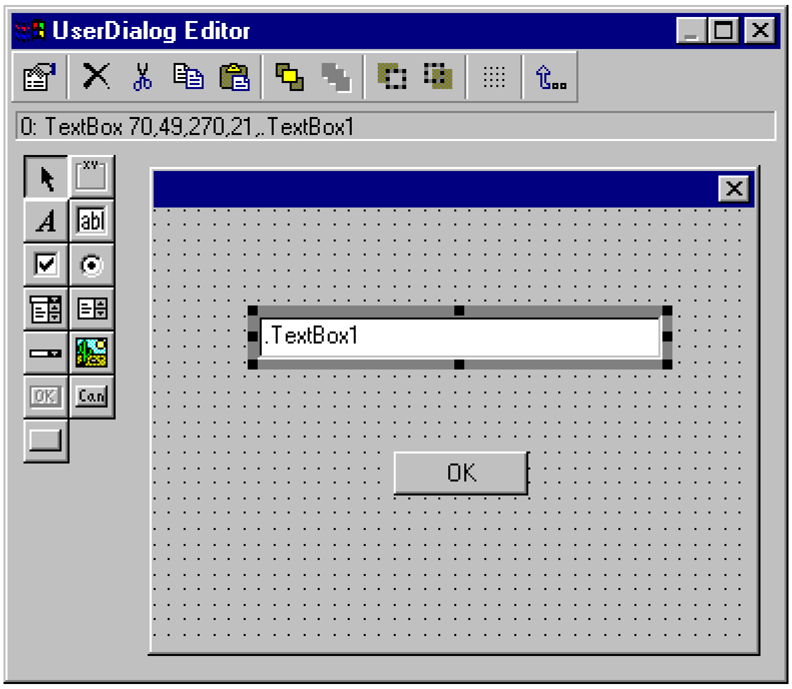

**Fig. 7-1: Dialog template editor**

Click on the following icon.

Now click on the dotted area to the right of that icon. This creates a text box with the text ".TextBox1".

Repeat the action with the following icon.

Your dialog should now contain a text box and an OK button. You can change the position and size of the elements, including the dialog box itself, with the mouse. Click the following button:

This takes you back to the text editor. The program has automatically inserted the code for the dialog. It looks something like this:

```
Sub Main
    Begin Dialog UserDialog 400,203 ' %GRID:10,7,1,1
       TextBox 70,49,270,21,.TextBox1
       OKButton 110,112,90,21
    End Dialog
    Dim dlg As UserDialog
    Dialog dlg
End Sub
```

If you now run the macro, a dialog appears with the text box and the OK button. You can input text and press the OK button.

But how do you get to the text that has been input? Very easily. The automatically declared variable dlg is an object which has the TextBox1 property, so dlg.TextBox1 provides the text as the following extended macro shows:

```
Sub Main
    Begin Dialog UserDialog 400,203 ' %GRID:10,7,1,1
       TextBox 70,49,270,21,.TextBox1
       OKButton 110,112,90,21
    End Dialog
    Dim dlg As UserDialog
    Dialog dlg              ' Start dialog.
    MsgBox dlg.TextBox1     ' Show user input.
End Sub
```

The MsgBox dlg.TextBox1 line outputs the text that has been input.

You can also predefine a text which the user can overwrite if necessary. To do this, assign a text to dlg.TextBox1 before the dialog is displayed:

```
Sub Main
    Begin Dialog UserDialog 400,203 ' %GRID:10,7,1,1
       TextBox 70,49,270,21,.TextBox1
       OKButton 110,112,90,21
    End Dialog
    Dim dlg As UserDialog
    dlg.TextBox1 = "Hello world"     ' Assign default text.
    Dialog dlg                       ' Start dialog.
    MsgBox dlg.TextBox1              ' Show user input.
End Sub
```

If you want to extend or modify the dialog, move the cursor between Begin Dialog ... and End Dialog, and then select **Edit > User Dialog** again. Then you can insert other elements in the dialog, such as pictures, radio buttons, list boxes and much more besides. Press the F1 key to look up information on the various elements of the dialog editor.

After closing the dialog editor you can look up further information on the options of the user-defined dialog in the online Help.

## 7.3. Functions / procedures

You can define your own functions and procedures to make a macro clearer or shorten it.

The prestimulus interval is output in the following example. To do this, the FindTimeZero function searches for a marker of the Time 0 type and returns its position.

```
' Print time 0 of the active node.
Sub Main
     If Not ActiveNode.ContainsData Then
          MsgBox "No active node found."
          Exit Sub
     End If
     Dim ds As Dataset
     Set ds = ActiveNode.Dataset
     ' Sampling interval is in microseconds -> convert to ms.
     MsgBox "Prestimulus: " & (FindTimeZero(ds) - 1) * ds.SamplingInterval / 1e3 & "ms"
End Sub

' Find position of time 0 marker (prestimulus interval length)
Function FindTimeZero(ds As Dataset) As Long
     Dim mk As Marker
     FindTimeZero = 1
     For Each mk In ds.Markers
          ' Case insensitive comparison.
          If StrComp(mk.Type, "Time 0", 1) = 0 Then
               FindTimeZero = mk.Position
               Exit Function
          End If
     Next
End Function
```

The following example defines the RenameMarkerDescription procedure which changes the description of a marker.

```
' Rename all stimulus markers with the description "S  1" as "Hand".
Sub Main
     Dim nhn As New NewHistoryNode
     nhn.Create "S1->Hand", ActiveNode
     Dim Markers As Markers
     Dim mk As Marker
     Set Markers  = ActiveNode.Dataset.Markers
     For Each mk In Markers
          If mk.Type = "Stimulus" And mk.Description = "S  1" Then
               RenameMarkerDescription mk, nhn, "Hand"
          End If
     Next Mk
     nhn.Finish
End Sub

' It is not possible to rename a marker description directly. This procedure does
' this by removing a marker, and then adding a new one.
Sub RenameMarkerDescription(mk As Marker, nhn As NewHistoryNode, NewDescription As String)
     nhn.RemoveMarker mk.ChannelNumber, mk.Position, mk.Points, mk.Type, mk.Description
     nhn.AddMarker mk.ChannelNumber, mk.Position, mk.Points, mk.Type, NewDescription
End Sub
```

## 7.4. Suppression of dialogs in history templates

If you want to accept input parameters from the user, in a similar way that the Analyzer does in most transforms, when he or she calls the macro directly but not when the macro is stored in a history template, you can use the new property named *NewHistoryNode.Description* to store the parameters. If a new *NewHistoryNode* object is generated when a macro is called, *Description* is empty but it is filled in the template context with the text that you defined when running the macro. You can thus save user inputs in *Description*. After creating the new node with *NewHistoryNode.Create()*, check whether *Description* is empty. If so, get the user to make an input. Otherwise you are in the template context and evaluate *Description*. The following example illustrates the procedure.

File *RenameMarkersInteractive.vabs*

```
' Rename markers
' User input for old/new name, skipped input in template processing
Sub Main
    If Not ActiveNode.DataAvailable Then
        MsgBox "This macro needs an open data window."
        Exit Sub
    End If
    Dim nhn As New NewHistoryNode
    Dim sOldName As String, sNewName As String
    nhn.Create "Renamed Markers", ActiveNode
    If nhn.Description = "" Then     ' Interactive mode
        sOldName = InputBox("Enter markers name", "Rename Markers")
        sNewName = InputBox("Enter new name", "Rename Markers")
        nhn.Description = "Rename Markers" & vbCrLf & "Old name: " & sOldName & _
            vbCrLf & "New name: " & sNewName
    Else ' Template mode
        ' Retrieve names from description text
        Dim nPos As Integer, sTemp As String
        sTemp = nhn.Description
        nPos = InStr(sTemp, "Old name: ")
        If nPos > 0 Then
            sTemp = Mid(sTemp, nPos + Len("Old name: "))
            nPos = InStr(sTemp, vbCrLf)
            sOldName = Left(sTemp, nPos - 1)
            sNewName = Mid(sTemp, nPos + 2 + Len("New name: "))
        End If
    End If
    If sOldName = "" Then
        MsgBox "Missing old name"
        Exit Sub
    ElseIf sNewName = "" Then
        MsgBox "Missing new name"
        Exit Sub
    End If
    Dim Markers As Markers
    Dim mk As Marker
    Set Markers  = ActiveNode.Dataset.Markers
    For Each mk In Markers
        If mk.Description = sOldName Then
            RenameMarkerDescription mk, nhn, sNewName
        End If
    Next Mk
    nhn.Finish
End Sub
```

...

## 7.5. Debugging

Debugging means searching for errors in a macro and eliminating them. In addition to inspecting the macro code visually, you can also run through the macro line by line, look up values of variables at any time, and set break points at which the macro will pause.

We will demonstrate these steps on the basis of the following example:

```
Sub Main
    i = 1
    i = i + 1
    MsgBox i
End Sub
```

Type in the macro and then press the F8 key. The window is split into an upper and lower area. At the top there are four tabs with the following titles: Immediate, Watch, Stack and Loaded. In this demonstration we are only interested in the Immediate tab. The first line of the macro is marked yellow. This mark indicates the current macro line. Now press the F8 key two more times. The mark has moved to the third line (i = i  + 1). Now click in the Immediate window and input the following:

```
? i
```

Follow that by pressing the Enter key. 1% appears. % stands for an integer. The most important thing is that you can read out the content of variables. Now press the F8 key and look up the value of i again. The value 2% appears now. If you do not need to know more, press the F5 key and the program will run through to the end.

To set a break point, move the cursor to the fourth line, for example, and press the F9 key. The line is highlighted in dark red. Now start the macro with F5. It stops in the fourth line. Pressing F5 again causes execution of the macro to resume.

We just went through the debugging session using function keys. You can use the **Macro** and **Debug** menus and their submenus instead. The toolbar also provides these commands, but the fastest way is to use function keys.

Incidentally, it is not only the values of variables that you can look up. You can also search through the Analyzer and send commands. To do this, select **View > Always Split**. Now the window is split even when no macro is running. Click in the Immediate window and type

```
? HistoryFiles(1).Name
```

and then press the Enter key. The name of your first history file appears.

Entering HistoryFiles(1).Open will open the file, and HistoryFiles(1).Close will close it. This means you can test the effect of the various automation constructions interactively here.

More information on debugging is given in the Online Help.