BRAIN PRODUCTS
Solutions for neurophysiological research

BrainVision Analyzer
Automation Reference Manual

Manual Version 003 as of Analyzer Software Version 2.0.2*

valid as of September 30, 2013*

BRAINVISION
professional ANALYZER2

* Valid until publication of a new version of the manual

Blank page

Imprint

Blank page

Contents

List of figures

Chapter 1 Underlying concepts

List of tables

About this manual

Structure and content of the new Automation Reference Manual

The new Analyzer Automation Reference Manual now includes an extensive theoretical chapter that uses short examples to familiarize you with important basic concepts of Analyzer Automation and which are intended to facilitate your first steps in creating your own macros and programs.

The Reference Manual has six chapters:

- Chapter 1 explains important fundamental concepts of Analyzer Automation and provides simple programming examples.
- Chapter 2 describes all the object classes of the Analyzer in detail.
- Chapter 3 describes the transforms which you can currently call using Analyzer Automation together with the parameters used.
- Chapter 4 describes the enumerator types used in Analyzer Automation.
- Chapter 5 contains a list of all error codes returned by Analyzer Automation methods.
- Chapter 6 provides an overview of Analyzer Automation for .NET.

Who is the manual intended for?

The Reference Manual is aimed at users from the fields of psychophysiological and neurological research who have a knowledge of programming in BASIC or a comparable programming language.

Conventions used in the manual

The manual uses the following typographical conventions:

`monospaced`	A monospaced font is used to indicate text or characters to be entered at the keyboard, such as source code and programming examples.
italic	Italic text is used to identify menus, menu commands, dialog boxes, options, and the names of files and folders.

underscore	Underscored text indicates a cross-reference or a web address.
●	The blue dot indicates the end of a chapter.

The manual also uses the following symbols to help you find your way around:

📘	A *cross-reference* refers to a section of the manual or an external document that has a bearing on the running text at this point.
NEW	The *New* symbol indicates that new material has been added at this point.

Revision history

Page	**Status**	**Subject**
Title page	new	Remove of CE mark

Reporting errors and support

You can search for updates of this manual on our Web site using the following link: http://www.brainproducts.com/downloads.php?kid=5&tab=2.

If you require support or if you discover a mistake in the manual, the software or during operation, please contact:

Brain Products GmbH
Zeppelinstraße 7
D-82205 Gilching
Phone: +49 8105 73384 – 0
Fax: +49 8105 73384 – 505
Web site: http://www.brainproducts.com
Email: support@brainproducts.com

●

Preface

This Reference Manual describes how to address and control the BrainVision Analyzer application from your own macros or programs. In order to achieve this, the Analyzer defines a hierarchy of object classes that represent its components and contents, such as history nodes or EEG views.

You can use the OLE Automation technology integrated in Windows® to access these object classes and thus interact with the Analyzer. This provides a simple way of implementing a broad spectrum of applications ranging from simple scripts up to complex calculations.

The SAX BASIC interpreter integrated in the Analyzer makes access to Analyzer Automation extremely simple. We recommend that you first become familiar with simple automation applications in the interpreter.

Throughout this manual, we assume that you are familiar with the BASIC programming language and are confident in using constructs such as method calls, loops and conditional statements. All object definitions and programming examples are given in BASIC syntax. In principle, however, Analyzer Automation can be addressed using other programming languages.

This Reference Manual refers to Version 2.0 of the Analyzer. The object classes in this version of the Analyzer are extensions of the object classes in Analyzer 1.0, and existing macros or scripts should continue to run without errors.

●

Chapter 1 Underlying concepts

When you control the Analyzer using Analyzer Automation, you are working with object classes that represent the contents of the Analyzer application that is currently running. If you have already worked with the Analyzer, you will be familiar with the majority of such content, such as history files or markers.

In this chapter, we shall use simple examples to describe how to access the running Analyzer application from the integrated SAX BASIC interpreter. In further sections, we shall provide an overview of the object hierarchy in Analyzer Automation and explain some of the most important object classes in detail.

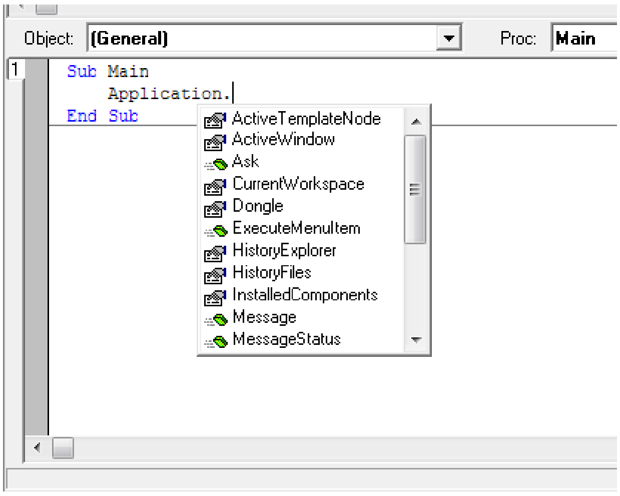
1.1 First steps and simple examples

The Analyzer application object `Application` has been predefined in the SAX BASIC interpreter and can be used directly. To do this, open the interpreter by choosing *Macros › Macros › New* from the Analyzer ribbon. An editing window opens.

The interpreter provides all the usual functions of an integrated development environment (IDE), including a debugger and syntax assistance. Controls for operating the interpreter are located at the top of the editing window. You can, for example, run the macro by clicking the *Start/Resume* button.

If you now type the text `Application` followed by a period in the `Main` method, the drop-down list providing syntax assistance opens (see Figure 1-1). Starting with the Application object, you can use the syntax assistance to easily build the calls needed for a simple macro.

Figure 1-1. Syntax assistance for the Application object



Object: (General)

Proc: Main

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

This is a simple sample program:

```
Sub Main

    HistoryFiles(1).Open

    Dim Node As HistoryNode

    Set Node = Application.HistoryFiles(1).HistoryNodes(1)

    FileName = Node.Name

    ChannelName = Node.Dataset.Channels(1).Name

    Application.HistoryFiles(1).Close()

    MsgBox("Node: " & FileName & " Channel: " & ChannelName)

End Sub
```

When you run the macro, the name of the raw data node of the first history file in the workspace is displayed together with the name of the first EEG channel. In this context, it is important that the history file is opened using `Open` and closed using `Close` after use.

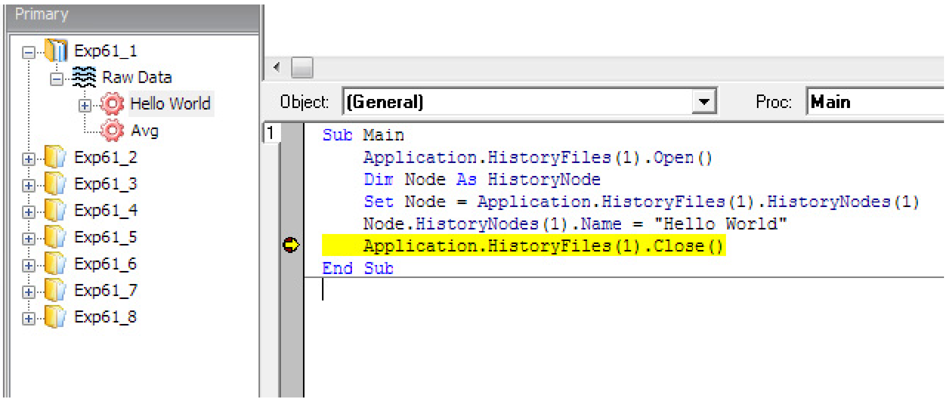
This process reflects the fact that when you are working with the Analyzer normally, you have to open history files before you can use their contents. The access modalities in Analyzer Automation reflect those that apply when working normally with the Analyzer and are subject to the same constraints.

The programming example below assumes that a further history node exists below the raw data node of the first history file in the workspace. The macro renames this node as *Hello World*:

```
Sub Main

    HistoryFiles(1).Open

    Dim Node As HistoryNode

    Set Node = Application.HistoryFiles(1).HistoryNodes(1)

    Node.HistoryNodes(1).Name = "Hello World"

    Application.HistoryFiles(1).Close()

End Sub
```

If you click the gray bar in the editing window to set a breakpoint before you call the macro, you can explicitly stop the program before `Close` is called. This allows you to check whether the node has been renamed before the history tree is collapsed again (see Figure 1-2).

Figure 1-2. Don't just take it on trust: The debugger in action



Object: (General)

Proc: Main

```
Sub Main
    Application.HistoryFiles(1).Open()
    Dim Node As HistoryNode
    Set Node = Application.HistoryFiles(1).HistoryNodes(1)
    Node.HistoryNodes(1).Name = "Hello World"
    Application.HistoryFiles(1).Close()
End Sub
```

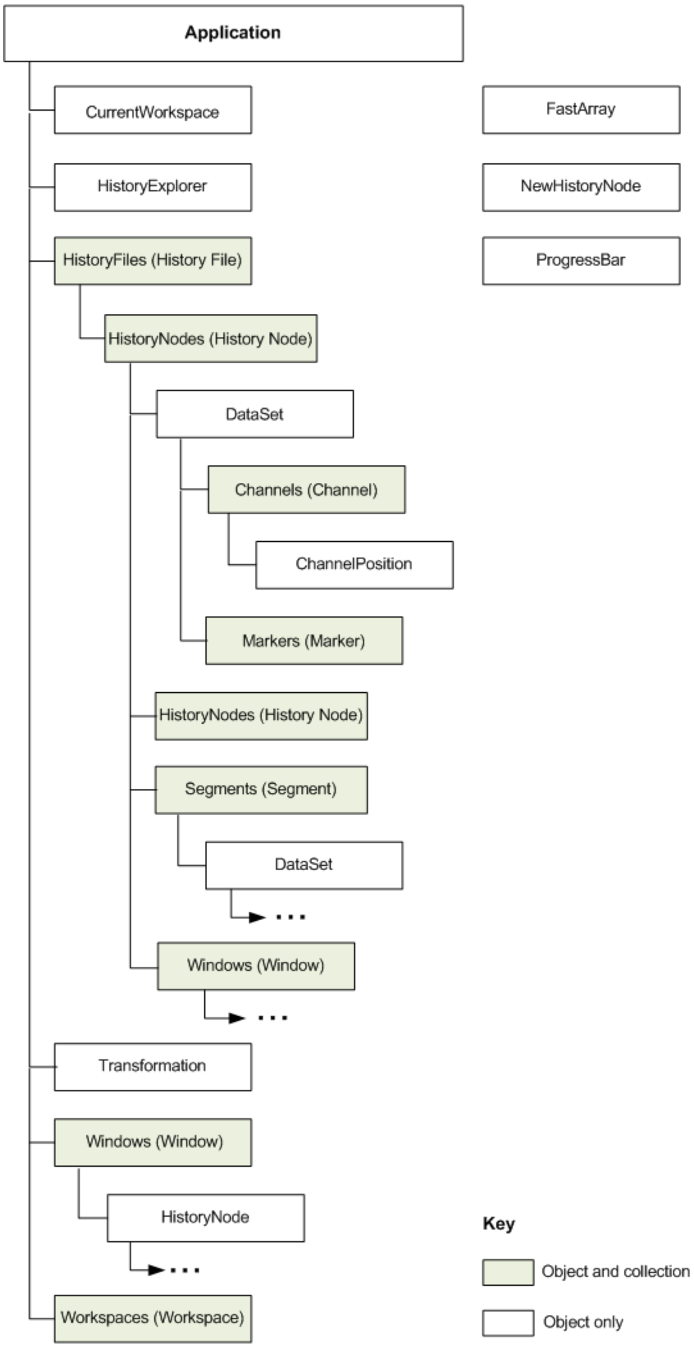
In the Analyzer, it is not possible to make changes to existing history nodes and rename channels, for example. You can only create new nodes with the required properties. There are only a few exceptions to this rule, such as the ability to rename nodes. Accordingly, virtually all the properties of objects that you can access using `Application` are read-only.

You can use Analyzer Automation to create new history nodes. This functionality is represented by separate object classes that are not accessed via the application object `Application`.

1.2 Overview of the object hierarchy

Figure 1-3 shows the hierarchy of the object classes of Analyzer Automation. The chart does not contain all the defined objects and is only intended as an overview. The individual object classes are described in detail in Chapter 2 as of page 33.

Figure 1-3. Object hierarchy



The left-hand side of the chart shows objects that you can address directly or indirectly via the `Application` object. These objects represent the current state of the Analyzer application.

As a rule, the name of the subordinate object class corresponds to the name of the property via which the corresponding object can be addressed. For example, the object `Application` has a property `CurrentWorkspace`, that you can use to access an object of the class `CurrentWorkspace`.

The right-hand side of the chart shows objects that are independent of the Application object. These objects are used to create new content for the Analyzer. Thus, for example, you can use the class `NewHistoryNode` to create a new history node.

Some object classes in Analyzer Automation are *collections* and act as containers for objects. Collection objects are highlighted in color in the chart. The class name of the objects in the collection is shown in parentheses.

The objects in a collection can be indexed with a number. The first index is always 1 rather than 0. Some collections also permit indexing via the name or title of the objects they contain. This method of indexing is only sensible, however, when the name occurs just once in the collection.

If *arrays* are used, it is assumed that the first index is 1.

Some of the object classes in Analyzer Automation have what is known as a "default element". This is a method or property that can be addressed without naming it explicitly in the code. For example, the default element of the `Channel` object is the `DataPoint` property. Consequently the two calls below are equivalent:

```
fValue = Channels(1).DataPoint(1)
```

```
fValue = Channels(1)(1)
```

1.3 Creating new data sets with "NewHistoryNode"

You can use an object of the class `NewHistoryNode` to create either a new history file or a new history node within an existing history file. You do this by first creating the `NewHistoryNode` object and specifying its basic properties. You then write the data values to be contained in the new data set to the `NewHistoryNode` object and optionally specify other properties of the data set.

There are two ways of creating an object of the `NewHistoryNode` class. If you have included the type library for Analyzer Automation, you can write the following:

```
Dim nhn as New NewHistoryNode
```

The type library is automatically included in the integrated BASIC, and you can use this statement. If you have not included the type library, you must instead write the following:

```
Set pb = CreateObject("VisionAnalyzer.NewHistoryNode")
```

1.3.1 Specifying basic properties of data sets

Some properties of a data set are of crucial importance and must be defined before data is written to the data set. This includes the specification as to whether the data set is to form a new history file or a new history node within an existing history file. The data type and the length of the data set must also be defined.

The basic properties of the data set are defined by calling the methods `Create`, `CreateEx` or `CreateWithChannelMap`. This call is issued immediately after the `NewHistoryNode` object has been created. The specific application scenario will determine which of the methods you should use and what individual specifications you need to make.

The `CreateEx` method is an extension of the `Create` method that is used if the new data set is to contain multiple frequency levels (e.g. wavelet data). In all other respects, it is identical to the `Create` method, which we shall describe in detail below.

Because the `Create` method has several different application scenarios, it is difficult to identify which of the parameters must be specified in which cases simply on the basis of the parameter list. The list of application scenarios below is intended to help you identify which parameters you should use in order to create the required data set:

- If you wish to create a new history file, you must pass the file name you wish to assign to the `Create` method. You must also specify the length and type of the data. The new history file will be listed in the *Secondary* tab in the history tree of the Analyzer. Example:

  ```
  nhn.Create("Raw Data", Nothing, "New File Name", False,
     viDtTimeDomain, 4, 1500, 4000)
  ```

  The raw data node of a new history file with the name *New File Name* is created in the example. The node contains four channels of time data with a sampling rate of 250 Hz.

- If you wish to create a new child node containing the same data as its parent node, you must pass the parent node to the `Create` method. In this application scenario, you can, for instance, modify markers and channel properties in the new node. It is not necessary to make any further specifications regarding the length and type of the data. Example:

  ```
  nhn.Create("New Child Node", ActiveNode, "", True)
  ```

  A new child node of the node that is active in the Analyzer main window is created in the example. The name of the child node is *New Child Node* and the value `True` of the argument `InheritData` specifies that the data of the parent node is to be taken over unchanged.

- If you wish to create a new child node and define the data of this node yourself, you must pass the parent node to the `Create` method and also specify the length and type of the data. Example:

  ```
  nhn.Create("New Child Node", ActiveNode, "", False, _
     viDtTimeDomain, 4, 1500, 4000)
  ```

  A new child node of the node that is active in the Analyzer main window is created in the example. The name of the child node is *New Child Node*. It contains four channels of time data with a sampling rate of 250 Hz.

- If you want to create a child node that both inherits the data of some of the channels of its parent node and also modifies or rearranges channels, you must use the `CreateWithChannelMap` method. This method is described in detail in Section 1.3.4 as of page 26. Example:

  ```
  Dim channels(1 To 3) As Long
  channels(1)=5
  channels(2)=0
  channels(3)=1
  nhn.CreateWithChannelMap("New Child Node", ActiveNode, channels)
  ```

  A new child node of the node that is active in the Analyzer main window is created in the example. The name of the child node is *New Child Node*. It contains channels 5 and 1 of the parent node and a newly defined channel.

1.3.2 Defining the contents of data sets

After you have used `Create`, `CreateEx` or `CreateWithChannelMap` to define the basic properties of the data set you wish to create, you can use the other properties and methods of the `NewHistoryNode` object.

At this time, some of the contents of the data set such as any data inherited from the parent node have already been defined. Default values are used for all other contents of the `NewHistoryNode` object. You can overwrite these as necessary to suit your requirements.

The example below shows how to define the contents of the `NewHistoryNode` object:

```
Sub Main
    ' Creates the object and defines basic properties.
    Set nhn = New NewHistoryNode
    nhn.Create "BasicTest", ActiveNode, "", False, _
         viDtTimeDomain, 32, 250, ActiveNode.Dataset.SamplingInterval

    ' Defines channel properties. All other channel properties
    ' retain the properties inherited from the parent node as
    ' defaults.
    nhn.SetChannelName 1, "Channel B"
    nhn.SetChannelName 2, "Channel A"
    nhn.SetChannelPosition 4, 1, 0, 90

    ' Sets an interval marker
    nhn.AddMarker 0, 201, 20, "Bad Interval", "", False

    ' Specifies data: Reads 250 points from channel 3 of the parent
    ' node and writes them to channel 1. All other data points
    ' retain the default value 0.0.
    Dim Data() As Single
    ActiveNode.Dataset(3).GetData 1, 250, Data
    nhn.WriteData 1, 1, 250, Data

    ' Writes a sample text for "Operation Infos"
    nhn.Description = "Line1" & vbCrLf & "Line2"

    ' Concludes the operation and creates the node
    nhn.Finish
End Sub
```

As you can see from the example, the `Finish` method is used to complete creation of the data set. The data set is only displayed in the Analyzer after `Finish` has been called. If the data set should no longer be created, for instance as a result of user input, you should use `Cancel` instead of the `Finish` method.

Note the following convention: If you use the integrated BASIC interpreter to create a data set, execution of your macro always ends with the call to `Finish`. This means that any lines that follow `Finish` will no longer be executed. You can, however, jump to these lines using the `GoTo` command.

1.3.3 Creating data sets suitable for history templates

If you create a new data set with `NewHistoryNode`, you can use it in history templates provided that certain prerequisites are met:

- You must have created a child node within a history file using the `NewHistoryNode` object, because history files as such cannot be used in history templates.
- The data set must have been created using the integrated BASIC interpreter.
- The new node must be created as a child node of the node that is active in the Analyzer main window. To do this, use the predefined variable `ActiveNode`.

The macro below shows a simple example of a node that can be used in a template. It inherits the data from its parent node and sets an additional marker at data point 200:

```
Sub Main

    Set nhn = New NewHistoryNode

    nhn.Create "Added Marker", ActiveNode

    nhn.AddMarker 0, 201, 1, "Stimulus", "S1", False

    nhn.Finish

End Sub
```

The example program in Section 1.3.2 on page 23 also creates a node that can be used in a template. Note that in this example the data of the parent node is read using the variable `ActiveNode`. This approach ensures that the new node is only dependent on the parent node and can be created unproblematically from within a history template.

When the new node is generated, the entire macro code is copied to the node. If you right-click the node and choose *Operation Infos...* from the context menu, the macro code is shown in addition to the Operation Infos.

You can use the new node that has been created in this way as a regular transform in history templates or drag it onto other nodes.

1.3.4 Efficient handling of data from the parent node

If you initialize a new history node using `Create` and take over the data from the parent node without making any changes (`InheritData = True` argument), no copy is made of the data in the new node. If, on the other hand, you write custom data to the node (`InheritData = False` argument), all the data of the new node is saved in the history file. Depending on the application scenario, the quantity of data involved can be very large.

To prevent large quantities of data from being copied unnecessarily, you should check whether the new node contains any unchanged channels from the parent node. If this is the case, you can initialize the new node using `CreateWithChannelMap`. This method represents a compromise between the two application scenarios for `Create` described here.

You can use the `CreateWithChannelMap` method to implement the following operations in the new node:

- Delete a channel of the parent node
- Add a new channel
- Change the sequence of the channels

The behavior of the `CreateWithChannelMap` method is determined by the `ChannelMap` argument. This argument is an array of integer values. A new channel with data is created for each item in the array. The value of each item specifies the data the channel is to contain:

- The value 0 means that the channel is to contain new data that you then have to write to the new node in the macro using `WriteData`.
- A value greater than 0 means that the channel is to contain the data from the corresponding channel in the parent node. Note that this data is **not** stored in the new node. It is instead read directly from the parent node.

Any channels of the parent node that are not contained in the `ChannelMap` array are not inherited by the new node. You can rearrange channels from the parent node by changing the sequence in which you insert them in the `ChannelMap` array.

When you use `WriteData` to write the data for new channels to the new node, the number of channels to be written is equal to the number of times that you have used the value 0 in the array `ChannelMap`. As far as `WriteData` is concerned, the new channels form a "reduced" data set made up of only those channels to which you have assigned the value 0 in the `ChannelMap`.

For example, if you have used the array (0, 1, 0, 2), you must write the data for two new channels. These channels have the numbers 1 and 2 when you use `WriteData` to write the individual channels. In the new node, however, these channels appear at the first and third positions in the channel list. Example:

```
Sub Main

    Dim a(1 to 4) as Long

    Channels(1)=0

    Channels(2)=1

    Channels(3)=0

    Channels(4)=2


    Set hn = ActiveNode

    Set nhn = New NewHistoryNode

    nhn.CreateWithChannelMap "BasicTest", ActiveNode, Channels


    ' Constant value of 50 for channel 1

    Dim Data(1 To 10000) As Single

    For i = 1 To 10000

         Data(i) = 50

    Next i

    ' Note: Index is 1 because this is the first 0 in ChannelMap

    nhn.WriteData 1, 1, 10000, Data


    ' Constant value of 100 for channel 3

    For i = 1 To 10000

         Data(i) = 100

    Next i

    ' Note: Index is 2 because this is the second 0 in ChannelMap

    nhn.WriteData 2, 1, 10000, Data
```

```
    ' Channel properties are inherited from the parent node,
    ' new channels are assigned default values
    nhn.SetChannelName 1, "New Channel 1"
    nhn.SetChannelName 3, "New Channel 2"
    nhn.SetChannelName 4, "Renamed Channel"

    nhn.Finish
End Sub
```

1.4 Processing arrays with "FastArray"

If you are using the integrated BASIC interpreter, you can use the auxiliary class `FastArray` to access arrays efficiently. This class provides a number of simple arithmetic operations (addition, multiplication, etc.) that allow you to manipulate the elements of the array. The calculations are then performed significantly faster than if you had implemented the operations directly in the BASIC interpreter.

As a rule, the arithmetic operations use a source array (`SourceData` parameter) and a target array (`TargetData` parameter). The target array is at the same time the left-hand operand. In operations with only one operand, the target array is simultaneously the source array. The source array remains unchanged in operations with two operands.

All indices used in the operations refer to the start of an array. This means that an array declared with

`Dim Data(12 to 24) As Single`

has its first item at `Data(12)`. An index parameter of 1 refers to this entry. To avoid confusion, we therefore recommend that you declare an array as

`Dim Data(1 to …) As Single`

All arrays that are used in the operations must be one-dimensional and must already have a defined field length.

The methods of the `FastArray` class use parameters with the same names. The parameters `StartIndex`, `Step` and `Count` are used to describe a subset of the elements in an array. Depending on how they are used, these parameters are prefixed with `Source` or `Target` (for the source or target array respectively). The resulting subset is determined as follows:

`StartIndex` determines the first item of the subset. The next item is offset by `Step` and each subsequent one also. `Count` is the maximum number of items. If the array is too small to accommodate `Count` elements described in this fashion, the quantity of data is limited by the end of the array. If `Count` is set to -1, the subset is only limited by the end of the array.

Examples (`StartIndex`, `Step`, `Count`):

- 1, 1, -1: Entire array (default setting)
- 1, 2, -1: All odd-numbered elements of an array
- 200, -1, -1: All elements as of position 200 in reverse order
- 7, 32, -1: The 7th, 39th, 71st item

If, for example, you have requested multiplexed data using `Dataset.GetData()`, and the number of channels is 32, then 7, 32, -1 corresponds to the data of the seventh channel.

Channels in arrays can only be defined in this manner if the data is not complex and only has one frequency level. You can use the following method to define subsets that describe channels whose data points are, for instance, complex (two values per data point):

```
RepeatNextOperation(Count As Long, [TargetIndexIncrement As Long],
   [SourceIndexIncrement As Long])
```

This method instructs the subsequent method to repeat `Count` times. For every repetition, the default indices are incremented by `TargetIndexIncrement` and `SourceIndexIncrement`.

For example, if you have complex data with 32 channels, the seventh channel can be copied to a separate array as follows:

```
Dim fa As New FastArray

Dim ChannelData() As Single

...

SourceData = ds.GetData(...)

fa.RepeatNextCommand(-1,,64)

ChannelData = fa.GetSelectedElements(SourceData,13,,2)
```

The first data point of the seventh channel starts at position 13. This is the start index. Two points are copied. Following this, the start index of the source field is increased by 64 and the operation is repeated up to the limits of the array.

1.5 Dynamic parameterization

You can create new history nodes by calling primary transforms available in the Analyzer in Analyzer Automation. To do this, the `Transformation.Do` method is used to pass parameters to the transform in the form of a string (see the description of the method on page 84). This functionality is currently only available for a selection of transforms. A list of supported transforms and their parameters is given in Chapter 3 as of page 91.

The advantage of this approach compared with calling a transform via the *Transformations* tab (see the `Application.ExecuteMenuItem` method on page 33) is that it enables dynamic parameterization. In other words, the parameters for the transform can be determined at runtime. This means, for example, that results of previously completed operations can be taken into account.

If you create a new history node with `Transformation.Do`, you can use this node in history templates provided that certain prerequisites are met:

- The history node must have been created using the integrated BASIC interpreter.
- The new node must be created as a child node of the node that is active in the Analyzer main window. To do this, use the predefined variable `ActiveNode`.

Note the following convention: If you use the integrated BASIC interpreter to create a new data set, execution of your macro always ends with the call to `Do`. This means that any lines that follow `Do` will no longer be executed. You can, however, jump to these lines using the `GoTo` command.

When the new node is generated, the entire macro code is copied to the node. If you right-click the node and choose *Operation Infos...* from the context menu, the macro code is shown in addition to the Operation Infos.

You can use the new node that has been created in this way as a regular transform in history templates or drag it onto other nodes.

Example program that performs a filter operation:

```
Sub Main

    Transformation.Do "Filters", "HighCutoff=30,48;Notch=50", ActiveNode, "FilterTest"

End Sub
```

1.6 Alternatives to the integrated BASIC interpreter

The integrated BASIC interpreter allows you to create your own macros without the need to install a separate development environment in addition to the Analyzer. You can, however, also address Analyzer Automation from an external development environment using the OLE Automation technology integrated in Windows®. This allows you, for instance, to use a programming language other than BASIC.

If you wish to use Analyzer Automation, it is an advantage if the development environment is able to include type libraries for OLE Automation. This is the case with Microsoft Visual Studio, for instance. The terms used vary somewhat between development environments, which means that the functionality you need may go under the name of *Add COM reference* or *Include ActiveX library*.

If you include a type library in a development environment, a list of the type libraries available on the system will usually be displayed. The type library used by Analyzer Automation appears in the list as *Vision Analyzer 1.0 Type Library*.

If you do not wish to include the type library for Analyzer Automation in your development environment, you may be able to use appropriate constructs provided by your programming language to directly access the OLE Automation objects. Example in BASIC syntax:

```
Set analyzer = CreateObject("VisionAnalyzer.Application")

analyzer.HistoryFiles("odd_phob_2").Open()
```

When you access Analyzer Automation from an external program, data that is requested from Analyzer Automation by the program is copied to the memory of the external program. This causes a certain loss of processing speed. On the other hand, it is possible that the external program can perform its own calculations significantly faster than the integrated BASIC interpreter.

The object definitions and programming examples given in this manual use BASIC syntax. If you are using a different programming language, the syntax of this language will provide similar constructs (e.g. `NULL` or `NIL` in place of `Nothing`) and you can translate the objects accordingly using a uniform pattern.

●

Chapter 2 Object classes

2.1 Application

2.1.1 Description

The `Application` class contains only one object, which represents the entire program. This is the default object. The methods and properties of this object can be addressed directly in SAX BASIC. Thus, for example, `Visible` corresponds to `Application.Visible`.

2.1.2 Methods

Ask

Prompts the user for the response Yes or No. This function should always be used in place of the integrated `MsgBox` function if this line of code could potentially be executed inside a history template.
If the BASIC script is run inside a history template, and messages are only output as a log, execution is not interrupted by `Ask`, and the response is always taken to be Yes. You can use this setting when you run the history template using the function *Apply to History File(s)*.

Definition

`Function Ask(Text as String) as Long`

Parameters

`Text` Text of the prompt displayed

Return value

Returns `vbYes` (numeric 6) or `vbNo` (numeric 7) depending on the response

ExecuteMenuItem

Executes an item in the ribbon. This is entered as text. The parameter is not case-sensitive, and spaces/full stops in the menu text are ignored. \ is used to distinguish between the various levels in the ribbon. For reasons of compatibility, it is still possible to address the names of the menu items used in Analyzer Version 1.0.
Note that some menu items can only be used when certain prerequisites have been met. For instance, a data set must be active in the Analyzer main window before a transform can be applied to it.

Definition

`Sub ExecuteMenuItem(MenuItem as String)`

Parameters

`MenuItem` Name of the item

Example

The IIR Filters transform can be executed using either
`ExecuteMenuItem "Transformations\Artifact Rejection/Reduction\Data Filtering\IIR Filters"` or
`ExecuteMenuItem "Transformations\Filters"`.

Message

Prompts the user for the response *OK* or *Cancel*. This function should always be used in place of the integrated `MsgBox` function if this line of code could potentially be executed inside a history template.

If the BASIC script is run inside a history template, and messages are only output as a log, execution is not interrupted by `Message`, and the response is always taken to be *OK*. You can use this setting when you run the history template using the function *Apply to History File(s)*.

Definition `Function Message(Text as String) as Long`

Parameters `Text` Text of the message displayed

Return value Returns `vbOk` (numeric 1) or `vbCancel` (numeric 2) depending on the response

MessageStatus Outputs a text in the status bar.

Definition `Sub MessageStatus(Text as String)`

Parameters `Text` Text displayed in the status bar

Msg Outputs a text to a message box. The user can only respond with *OK*. This function should always be used in place of the integrated `MsgBox` function if this line of code could potentially be executed inside a history template.

If the BASIC script is run inside a history template, and messages are only output as a log, execution is not interrupted by `Msg`, and the response is always taken to be *OK*. You can use this setting when you run the history template using the function *Apply to History File(s)*.

Definition `Function Msg(Text as String) as Long`

Parameters `Text` Text of the message displayed

Return value Always returns `vbOk` (numeric 1)

Quit Terminates the program.

Definition `Sub Quit()`

2.1.3 Properties

ActiveTemplateNode Write-protected

This object describes the template node that is currently being executed if the Analyzer is executing a history template. If not, the value is `Nothing`.

Definition `ActiveTemplateNode As HistoryTemplateNode`

ActiveWindow Write-protected

The active tab in the Analyzer main window. This value is `Nothing` if no tab is open.

Definition `ActiveWindow As Window`

CurrentWorkspace Write-protected

The currently open workspace.

Definition
`CurrentWorkspace As CurrentWorkspace`

Dongle

Write-protected
The dongle currently in use.

Definition
`Dongle as Dongle`

HistoryExplorer

Write-protected
The History Explorer.

Definition
`HistoryExplorer As HistoryExplorer`

HistoryFiles

Write-protected
Collection containing all the history files in the currently open workspace.

Definition
`HistoryFiles As HistoryFiles`

InstalledComponents

Write-protected
List of installed program components as text.

Definition
`InstalledComponents as String`

Sublicenses

Write-protected
Collection containing the currently available licenses for optional program components of the Analyzer.

Definition
`Sublicenses as Licenses`

TempFileFolder

Write-protected
Folder for temporary files.

Definition
`TempFileFolder as String`

TemplateMode

If the value of this flag is `True` (-1), the Analyzer is currently executing a history template.

Definition
`TemplateMode As Boolean`

Version

Write-protected
Specifies the current program version.

Definition
`Version As Double`

Visible

This flag specifies whether the Analyzer main window is visible (`True` or -1) or not (`False` or 0).

Definition
`Visible As Boolean`

Windows

Write-protected

Collection containing all the tabs in the Analyzer main window.

Definition `Windows As Windows`

WorkFileFolder

Write-protected

Folder for the work files (workspace files, macros and history templates).

Definition `WorkFileFolder As String`

Workspaces

Write-protected

Collection containing all the workspaces in the Workfiles folder.

Definition `Workspaces As Workspaces`

2.2 Channel

2.2.1 *Description*

The `Channel` object describes a channel in a history node. Since `DataPoint` is the default element, it is easy to access an individual data point.

2.2.2 *Example*

```
Dim fValue As Single
Dim hn As HistoryNode
Dim hf As HistoryFile
Dim ch As Channel
' First history file
Set hf = HistoryFiles(1)
hf.Open
' First data set
Set hn = hf(1)
' First channel
Set ch = hn.Dataset(1)
```

```
' First data point
f Value = ch(1)
' Alternative access using the channel name
Set ch = hn.Dataset("FP1")
' First data point
f Value = ch(1)
hf.Close
```

Alternative short version:

```
Dim fValue As Single
HistoryFiles(1).Open
fValue = HistoryFiles(1)(1).Dataset(1)(1)
HistoryFiles(1).Close
```

You can read large quantities of data significantly faster if you read in a vector using the `Channel.GetData()` method. If you wish to read data from multiple channels, you should use the `Dataset.GetData()` method. For detailed information, refer to Section 2.6 as of page 43.

2.2.3 Methods

GetData

Reads a number of data points into an existing vector.
If the data set contains complex data, each data point has two values: The first value is the real part of the number and the second value is the imaginary part. This means, for instance, that `Data(1)` is the real part of the first data point and `Data(2)` is the imaginary part. The second data point is thus assigned to `Data(3)` and `Data(4)`, etc.

Definition

```
Sub GetData(Position As Long, Points As Long, Data() As Single)
```

Parameters

`Position` Position of the data points to be read in the data set (1 – ...)
`Points` Number of data points to be read
`Data` Vector that receives the data points that have been read

Example

```
Dim channel As Channel
Dim node as HistoryNode
Dim fVector() As Single
HistoryFiles(1).Open
```

```
' Raw data node of the first history file
Set node = HistoryFiles(1)(1)
' First channel in the node
Set channel = node.Dataset(1)
channel.GetData(1, 1000, fVector)
HistoryFiles(1).Close
```

PropertyName

Returns the name of a channel property.

This function can be used to list all channel properties including user-defined channel properties.

Definition

`Function PropertyName (Number As Long) As String`

Parameters

`Number` Number of the channel property (1 – ...)

Return value

Name of the channel property with the number specified or an empty string if this property does not exist

Example

This example lists the names of all channel properties including any user-defined channel properties that may be present.

```
Dim channel As Channel
HistoryFiles(1).Open
Set channel = HistoryFiles(1)(1).Dataset(1)
s = ""
For i = 1 To channel.PropertyCount
    s = s & channel.PropertyName(i) & vbCrLf
Next i
' Output to message window
Application.Msg(x)
HistoryFiles(1).Close
```

PropertyValue

Returns the value of a channel property.

Definition

`Function PropertyValue(Name As String) As Variant`

Parameters

`Name` Name of the channel property to be read

Return value

Returns the value of the specified channel property or `Nothing` if the property does not exist

2.2.4 Properties

DataPoint

Default element, write-protected

Reads the value of a data point. If the data set involves complex data, this variable specifies the real part of the complex number.

Definition

`DataPoint(Index As Long) As Single`

Parameters

`Index` Specifies the position in the data set (1 – ...)

DataPointLayered

Write-protected

Reads the value of a data point in a data set comprising multiple frequency levels ("layers"), such as a data set comprising continuous wavelets. If the data set involves complex data, this variable specifies the real part of the complex number. This property corresponds to `DataPoint`.

Definition

`DataPointLayered(Index As Long, Layer As Long) As Single`

Parameters

`Index` Specifies the position in the data set (1 – ...)
`Layer` Specifies the frequency level

ImgPoint

Write-protected

Reads the value of the imaginary part of a data point if the data set contains complex data.

Definition

`ImgPoint(Index As Long) As Single`

Parameters

`Index` Specifies the position in the data set (1 – ...)

ImgPointLayered

Write-protected

Reads the value of the imaginary part of a data point in a data set comprising multiple frequency levels ("layers"), such as a data set comprising continuous wavelets. This property corresponds to `ImgPoint`.

Definition

`ImgPointLayered(Index As Long, Layer As Long) As Single`

Parameters

`Index` Specifies the position in the data set (1 – ...)
`Layer` Specifies the frequency level

Name

Write-protected

Name of channel.

Definition

`Name As String`

Position

Write-protected

Position of the channel on the surface of the head.

Definition

`Position As ChannelPosition`

PropertyCount

Write-protected

Number of property values of the channel. This number includes all channel properties including any user-defined channel properties.

Definition `PropertyCount As Long`

ReferenceChannel

Write-protected

Name of reference channel.

Definition `ReferenceChannel As String`

SecondPosition

Write-protected

The second position is used when an additional position on the surface of the head is assigned to a channel in addition to its own coordinates (for instance with the Coherence transform).

Definition `SecondPosition As ChannelPosition`

Unit

Write-protected

Unit for the data on this channel, such as µV, µV^2, etc. (📖 see also Section 4.2 as of page 98).

Definition `Unit As VisionDataUnit`

UnitString

Write-protected

The unit for the channel as a text string. This specification is used if the unit is not one of the predefined units. In this event, the convention is to use the value `viDuUnitless` for the `Unit` property.

Definition `UnitString As String`

2.3 ChannelPosition

2.3.1 Description

The `ChannelPosition` object describes the position of a channel.

2.3.2 Properties

Phi

Write-protected

Phi in degrees.

Definition

`Phi As Single`

Radius

Write-protected

Radius in millimeters. A value of 0 indicates an invalid position specification. The value of 1 assumes the head to be a perfect sphere with a uniform radius.

Definition

`Radius As Single`

Theta

Write-protected

Theta in degrees.

Definition

`Theta As Single`

2.4 Channels

2.4.1 *Description*

The `Channels` object is a collection of `Channel` objects.

2.4.2 *Properties*

Count

Write-protected

Number of channels in the collection.

Definition

`Count As Long`

Item

Default element, write-protected

Returns a `Channel` object from the collection. You can use either the channel number or the channel name to specify the channel.

Definition

`Item(NameOrIndex As Variant) As Channel`

Parameters

`NameOrIndex` Specifies the channel number (1 – ...) or the channel name

2.5 CurrentWorkspace

2.5.1 *Description*

The `CurrentWorkspace` object represents the currently open workspace.

2.5.2 *Methods*

Load

Loads the specified workspace file `FileName`.

Definition

`Sub Load(FileName As String, [SingleHistoryFile As String])`

Parameters

`FileName` Name of the workspace file or fully-qualified path of the workspace file if it is not located in the Workfiles folder

`SingleHistoryFile` (optional) Allows you to load a single history file

Save

Saves the currently open workspace file.

Definition

`Sub Save()`

SaveAs

Saves the currently open workspace file under a new name.

Definition

`Sub SaveAs(FileName As String)`

Parameters

`FileName` Specifies the name of the workspace file

2.5.3 *Properties*

ExportFileFolder

Write-protected

Default folder for exported files.

Definition

`ExportFileFolder As String`

FullName

Write-protected

Name of the workspace file including fully-qualified path.

Definition

`FullName As String`

HistoryFileFolder

Write-protected

Folder for history files.

Definition

`HistoryFileFolder As String`

Name

Write-protected
Base name of the workspace file without folder and file name extension.

Definition

```
Name As String
```

RawFileFolder

Write-protected
Folder for raw data.

Definition

```
RawFileFolder As String
```

2.6 Dataset

2.6.1 *Description*

The `Dataset` object represents a data set. This data set is either the entire data of a history node or the data of an individual segment within a node.

Properties prefixed with `Layer` are only used with data containing multiple frequency levels (layers) such as continuous wavelets.

2.6.2 *Methods*

GetData

Reads a number of data points and returns them as a vector.
If the `ChannelList` parameter is not used, the data of all channels is returned.
The data is returned in multiplexed format. This means that all the data of the first sampling point is returned first, followed by that of the second sampling point and so on. This format corresponds to the internal data management format in the history nodes. This results in a significantly higher processing speed compared with `Channel.GetData()`.
If the data set contains complex data, each data point has two values: The first value is the real part of the number and the second value is the imaginary part. This means, for instance, that `Data(1)` is the real part of the first data point and `Data(2)` is the imaginary part.

Definition

```
Function GetData(Position As Long, Points As Long, 
    [ChannelList as Variant]) as Single()
```

Parameters

`Position` Position of the data points to be read in the data set (1 – ...)
`Points` Number of data points to be read
`ChannelList` (optional) Contains either an array of channels or a single channel. Channels can be specified by their number (1 – ...) or their name.

Return value

Vector containing the data

Example

Each of the following examples reads the first 2000 data points from the raw data set of the first history file. The first `GetData` call reads all the channels. The remaining calls show how individual or multiple channels can be addressed.

```
Dim ds As Dataset

HistoryFiles(1).Open

Set ds = HistoryFiles(1)(1).Dataset

Dim Data() as Single

Data = ds.GetData(1, 2000)

Data = ds.GetData(1, 2000, Array("FP1", "Fp2"))

Data = ds.GetData(1, 2000, Array(1, 12))

Data = ds.GetData(1, 2000, "F3")

Dim a(1 to 2) as Long

a(1) = 1

a(2) = 12

Data = ds.GetData(1, 2000, a)
```

PropertyName

Returns the name of a property.
This function can be used to list all properties including user-defined properties.

Definition

```
Function PropertyName (Number As Long) As String
```

Parameters

`Number` Number of the property (1 – ...)

Return value

Name of the property with the number specified or an empty string if this property does not exist

PropertyValue

Returns the value of the specified property.

Definition

```
Function PropertyValue(Name As String) As Variant
```

Parameters

`Name` Name of the property to be read

Return value

Value of the specified property or `Nothing` if the property does not exist

2.6.3 Properties

Averaged

Write-protected
This flag specifies whether the data set has been produced directly or indirectly from an averaging operation (`True` or -1) or has not been averaged (`False` or 0).

Definition

`Averaged As Boolean`

AverageCount

Write-protected

Number of segments included in averaging. Only valid if the value of the `Averaged` flag is `True` (-1).

Definition

`AverageCount As Long`

Channels

Default element, write-protected

Collection of channel objects in the data set. You can use the channel objects to read properties of a channel or to query data of this channel.

Definition

`Channels As Channels`

LayerFunction

Write-protected

Increment function between the frequency levels (layers) of a data set. Frequency levels of this type occur with continuous wavelets, for instance. 📘 You will find the possible values of this property in Chapter 4 as of page 97.

Definition

`LayerFunction As VisionLayerIncFunction`

LayerLowerLimit

Write-protected

The value of the lowest frequency level (layer) of a data set with multiple frequency levels.

Definition

`LayerLowerLimit As Double`

Layers

Write-protected

Number of frequency levels (layers) in the data set.

Definition

`Layers As Long`

LayerUpperLimit

Write-protected

The value of the highest frequency level (layer) of a data set with multiple frequency levels.

Definition

`LayerUpperLimit As Double`

Length

Write-protected

Length of the data set in data points.

Definition

`Length As Long`

Markers

Write-protected

Collection of markers in the data set.

Definition

`Markers As Markers`

PropertyCount

Write-protected

Number of property values of the data set. This number includes all properties including any user-defined properties.

Definition `PropertyCount As Long`

SamplingInterval

Write-protected

Sampling interval of the data set. Specified in microseconds for data in the time domain and in hertz for data in the frequency domain.

The following formula converts the sampling interval for data in the time domain to the sampling rate: `Frequency = 1000000.0 / SamplingInterval`

Definition `SamplingInterval As Double`

SegmentationType

Write-protected

Specifies the segmentation type of the data set (📘 see also Section 4.3 on page 99).

Definition `SegmentationType as VisionSegType`

Type

Write-protected

Type of the data in the data set (📘 see also Section 4.1 on page 97).

Definition `Type As VisionDataType`

2.7 DeletedHistoryNode

2.7.1 *Description*

The `DeletedHistoryNode` object represents a deleted history node. This node is stored for a time in its former parent node and can be restored if needed.

2.7.2 *Methods*

Undelete

Restores a deleted node together with its child nodes.

Definition `Sub Undelete()`

2.7.3 Properties

Name

Write-protected

Name of the deleted node.

Definition

`Name as String`

2.8 DeletedHistoryNodes

2.8.1 Description

The `DeletedHistoryNodes` object lists the deleted child nodes of a `HistoryNode`.

2.8.2 Properties

Count

Write-protected

Number of deleted nodes in the collection.

Definition

`Count As Long`

Item

Default element, write-protected

Returns a `DeletedHistoryNode` object from the collection. You can use the position of the node in the collection or its name to specify the deleted node.

Definition

`Item(NameOrIndex As Variant) As DeletedHistoryNode`

Parameters

`NameOrIndex` Specifies the position of the node in the collection (1 – ...) or its name

2.9 Dongle

2.9.1 Description

The `Dongle` object describes the dongle currently in use.

2.9.2 Properties

NetworkDongle

Write-protected
This flag is set if the Analyzer is being used with a network dongle.

Definition `NetworkDongle As Boolean`

InternalSerialNumber

Write-protected
Specifies the internal serial number of the dongle.

Definition `InternalSerialNumber As Long`

2.10 FastArray

2.10.1 Description

The `FastArray` object is an auxiliary class for accelerating access to arrays. It only makes sense to use this class in the integrated BASIC interpreter. 📘 For further information on how to use the `FastArray` object, refer to Section 1.4 as of page 28.

2.10.2 Example

You have to create an object of the type `FastArray` in order to be able to use its methods. You can use the following line of code in the integrated BASIC interpreter:

```
Dim Fa as New FastArray
```

You can then use the object in order to perform calculations:

```
Data = node.Dataset.GetData(1,2000)
```

```
Fa.AddValue(Data, 5.0)
```

2.10.3 Methods

AddArray

Adds a subset of `SourceData` to a subset of `TargetData`.
The operation is limited by the smaller subset of the two arrays.

Definition `Sub AddArray(TargetData() As Single, SourceData() As Single, _`

```
    [TargetStartIndex As Long = 1], [TargetStep As Long = 1],
    [SourceStartIndex As Long = 1], [SourceStep As Long = 1],

    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

AddValue

Adds the fixed value `Value` to a subset of `TargetData`.
`Value` is incremented by `ValueIncrement` after each operation.

Definition

```
Sub AddValue(TargetData() As Single, Value As Single,

    [StartIndex As Long = 1], [Step As Long = 1],

    [Count As Long = -1], [ValueIncrement As Single = 0])
```

Parameters

`TargetData` Array containing output data

`Value` Value to be added

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

`ValueIncrement` (optional) The value of `Value` is incremented by this value after each addition operation.

Atan2Array

Calculates the arc tangent of a subset of `TargetData` and `SourceData`:
`TargetData` = Atan2(`TargetData`/`SourceData`)
The result is stored in `TargetData` in radian values (range +/- µ).

Definition

```
Sub Atan2Array(TargetData() As Single, SourceData() As Single,
    [TargetStartIndex As Long = 1], [TargetStep As Long = 1],
    [SourceStartIndex As Long = 1], [SourceStep As Long = 1],

    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` (optional) Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

CopyArray

Copies a subset of `SourceData` into a subset of `TargetData`.
The operation is limited by the smaller subset of the two arrays.

Definition

```
Sub CopyArray(TargetData() As Single, SourceData() As Single, [TargetStartIndex As Long = 1], [TargetStep As Long = 1], [SourceStartIndex As Long = 1], [SourceStep As Long = 1],
[Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` (optional) Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

CopyValue

Copies the fixed value `Value` into a subset of `TargetData`.
`Value` is incremented by `ValueIncrement` after each operation.

Definition

```
Sub CopyValue(TargetData() As Single, Value As Single,
[StartIndex As Long = 1], [Step As Long = 1],
[Count As Long = -1], [ValueIncrement As Single = 0])
```

Parameters

`TargetData` Array containing output data

`Value` Value to be added

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

`ValueIncrement` (optional) The value of `Value` is incremented by this value after each assignment.

DivideArray

Divides a subset of `TargetData` by a subset of `SourceData`:

`TargetData = TargetData / SourceData`

The operation is limited by the smaller subset of the two arrays.

Definition

```
Sub DivideArray(TargetData() As Single, SourceData() As Single,
    [TargetStartIndex As Long = 1], [TargetStep As Long = 1],
    [SourceStartIndex As Long = 1], [SourceStep As Long = 1],

    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` (optional) Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

DivideValue

Divides a subset of `TargetData` by the fixed value `Value`:

`TargetData = TargetData / Value`

`Value` is incremented by `ValueIncrement` after each operation.

Definition

```
Sub DivideValue(TargetData() As Single, Value As Single, [StartIn-
    dex As Long = 1], [Step As Long = 1],

    [Count As Long = -1], [ValueIncrement As Single = 0])
```

Parameters

`TargetData` Array containing output data

`Value` Value to be added

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

`ValueIncrement` (optional) The value of `Value` is incremented by this value after each division operation.

GetMaximumValue

Returns the largest value of a subset of `SourceData`.

Definition

```
Function GetMaximumValue(SourceData() As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1]) As Single
```

Parameters

`SourceData` Array containing input data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

Return value

The largest value of the subset

GetMeanValue

Returns the mean value of a subset of `SourceData`.

Definition

```
Function GetMeanValue(SourceData() As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1]) As Single
```

Parameters

`SourceData` Array containing input data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

Return value

The mean value of the subset

GetMinimumValue

Returns the smallest value of a subset of `SourceData`.

Definition

```
Function GetMinimumValue(SourceData() As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1]) As Single
```

Parameters

`SourceData` Array containing input data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

Return value

The smallest value of the subset

GetSelectedElements

Returns a subset of `SourceData`.

Definition

```
Function GetSelectedElements(SourceData() As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1]) As Single()
```

Parameters

`SourceData` Array containing input data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

Return value

Array containing the subset

MultiplyArray

Multiplies a subset of `TargetData` by a subset of `SourceData`:

`TargetData = TargetData * SourceData`

The operation is limited by the smaller subset of the two arrays.

Definition

```
Sub MultiplyArray(TargetData() As Single, SourceData() As Single,
    [TargetStartIndex As Long = 1], [TargetStep As Long = 1],
    [SourceStartIndex As Long = 1], [SourceStep As Long = 1],
    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` (optional) Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

MultiplyValue

Multiplies the fixed value `Value` by a subset of `TargetData`:

`TargetData = TargetData * Value`

`Value` is incremented by `ValueIncrement` after each operation.

Definition

```
Sub MultiplyValue(TargetData() As Single, Value As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1], [ValueIncrement As Single = 0])
```

Parameters

`TargetData` Array containing output data

`Value` Value to be added

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

`ValueIncrement` (optional) The value of `Value` is incremented by this value after each division operation.

RectifyArray

Converts all elements of a selected subset into their absolute value (rectification). `TargetData` is both the source and the target.

Definition

```
Sub RectifyArray(TargetData() As Single, [StartIndex As Long = 1],
    [Step As Long = 1], [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

RepeatNextOperation

Repeats the next method call in the `FastArray` class the number of times specified by `Count` and increments the method parameter `SourceIndex` or `TargetIndex` on each call. This allows complex operations on subsets of the array to be formulated simply.

Irrespective of the parameters used, every operation is limited by the end of the array. If the next operation does not have these parameters, the relevant parameter is ignored by `RepeatNextOperation`.

The `TargetIndexIncrement` and `SourceIndexIncrement` parameters must not contain negative values. The `Step` parameter of the subsequent operation is also not permitted to be negative.

Definition

```
Sub RepeatNextOperation([Count As Long = -1],
    [TargetIndexIncrement As Long = 1],
    [SourceIndexIncrement As Long = 1])
```

Parameters

`Count` Specifies the number of times the operation is repeated

`TargetIndexIncrement` Specifies the value by which the `TargetIndex` parameter of the method call is incremented on each repetition

`SourceIndexIncrement` Specifies the value by which the `SourceIndex` parameter of the method call is incremented on each repetition

RootArray

Calculates the square roots of all elements of a selected subset based on their absolute value. This means negative values are also allowed to be present in the output data set. `TargetData` is both the source and the target.

Definition

```
Sub RootArray(TargetData() As Single, [StartIndex As Long = 1],
    [Step As Long = 1], [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

RotateLeftArray

Rotates all elements of a selected subset to the left. The first element gets the value of the second, the second the value of the third, and so on. The value that was originally first becomes the last value.

Definition

```
Sub RotateLeftArray(TargetData() As Single, 

    [StartIndex As Long = 1], [Step As Long = 1], 

    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

RotateRightArray

Rotates all elements of a selected subset to the right. The second element gets the value of the first, the third the value of the second, and so on. The value that was originally last becomes the first value.

Definition

```
Sub RotateRightArray(TargetData() As Single, 

    [StartIndex As Long = 1], [Step As Long = 1], 

    [Count As Long = -1])
```

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

SortArray

Sorts all elements of a selected subset in ascending order. `TargetData` is both the source and the target.

Definition

`Sub SortArray(TargetData() As Single, [StartIndex As Long = 1], [Step As Long = 1], [Count As Long = -1])`

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

SquareArray

Squares all elements of a selected subset. `TargetData` is both the source and the target.

Definition

`Sub SquareArray(TargetData() As Single, [StartIndex As Long = 1], [Step As Long = 1], [Count As Long = -1])`

Parameters

`TargetData` Array containing output data

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

SubtractArray

Subtracts a subset of `SourceData` from a subset of `TargetData`:

`TargetData = TargetData - SourceData`

The operation is limited by the smaller subset of the two arrays.

Definition

`Sub SubtractArray(TargetData() As Single, SourceData() As Single, [TargetStartIndex As Long = 1], [TargetStep As Long = 1], [SourceStartIndex As Long = 1], [SourceStep As Long = 1], [Count As Long = -1])`

Parameters

`TargetData` Array containing output data

`SourceData` Array containing input data

`TargetStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing output data

`TargetStep` (optional) Offset between two items to be processed in the array containing output data

`SourceStartIndex` (optional) Position (1 – ...) of the first item to be processed in the array containing input data

`SourceStep` (optional) Offset between two items to be processed in the array containing input data

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

SubtractValue

Subtracts the fixed value `Value` from a subset of `TargetData`:

`TargetData = TargetData - Value`

`Value` is incremented by `ValueIncrement` after each operation.

Definition

```
Sub SubtractValue(TargetData() As Single, Value As Single,
    [StartIndex As Long = 1], [Step As Long = 1],
    [Count As Long = -1], [ValueIncrement As Single = 0])
```

Parameters

`TargetData` Array containing output data

`Value` Value to be added

`StartIndex` (optional) Position (1 – ...) of the first item to be processed in the array

`Step` (optional) Offset between two items to be processed in the array

`Count` (optional) Maximum number of items to be processed. This specification is ignored if the end of the array is reached before this number of elements has been processed. Specify the value -1 to continue processing up to the end of the array.

`ValueIncrement` (optional) The value of `Value` is incremented by this value after each subtraction operation.

2.11 HistoryExplorer

2.11.1 Description

The `HistoryExplorer` object represents the History Explorer.

2.11.2 Properties

Visible

This flag specifies whether the History Explorer is visible (`True` or -1) or not (`False` or 0).

Definition

```
Visible As Boolean
```

2.12 HistoryFile

2.12.1 Description

The `HistoryFile` object represents a history file.

2.12.2 Methods

AppendFile	Appends the raw data of the specified history file to this history file. 📘 For detailed information on this function, refer to the Analyzer User Manual.
Definition	`Sub AppendFile(FileName As String)`
Parameters	`FileName` Name of the history file to be appended

Example	The example appends the history file *P300b* to the history file *P300a*.
	`Dim hf as HistoryFile`
	`Set hf = HistoryFiles("p300a")`
	`hf.AppendFile "p300b"`

ApplyTemplate	Applies the specified history template to the history file. The file name can be specified in full or in part and, if possible, is completed automatically.
Definition	`Sub ApplyTemplate(FileName As String)`
Parameters	`FileName` Name of the history template to be used

Close	Closes the history file. This should be done as soon as the file is no longer needed because an open file may take up a considerable amount of memory in certain circumstances.
Definition	`Sub Close()`

Compress	Compresses the history file. If you frequently delete larger files (e.g. FFT) then empty spaces are often left in the history file due to performance reasons. These areas can be removed with the aid of compression.
Definition	`Sub Compress()`

FindNextNode	Searches for the next history node with the name specified in `FindNode`.
Definition	`Function FindNextNode() As HistoryNode`
Return value	Matching history node or `Nothing` if there is no further history node

FindNode	Searches for the first history node which has the specified name.

Definition

```
Function FindNode(Name As String) As HistoryNode
```

Parameters

`Name` Name of the history node

Return value

Matching history node or `Nothing` if there is no further history node

Example

The example uses `FindNode` and `FindNextNode` in a loop to rename all nodes with the name *Average* to *Avg*.

```
Dim hn As HistoryNode

Dim hf As HistoryFile

Set hf = HistoryFiles(1)

Set hn = hf.FindNode("Average")

Do While Not hn Is Nothing

    hn.Name = "Avg"

    Set hn = hf.FindNextNode()

Loop
```

Open

Opens the history file. It is only possible to access the history nodes contained in a history file after the history file has been opened.

Definition

```
Sub Open()
```

PurgeDeletedNodes

Irrevocably deletes all history nodes in the history file that have been deleted but can still be restored.

Definition

```
Sub PurgeDeletedNodes()
```

2.12.3 Properties

DisplayName

Alias of the history file as used by the Analyzer for display purposes. When a workspace has been loaded, this name is initially identical to the `Name` property. A database program which controls the Analyzer can use this property to display test subject's names, for example.

Definition

```
DisplayName As String
```

FullName

Write-protected
Name of the history file including the fully-qualified path.

Definition

```
FullName As String
```

HistoryNodes

Default element, write-protected

Collection containing the raw data nodes in this history file. These raw data nodes generally represent a raw EEG file. Most history files have just one raw data node because the raw EEG file contains one data set only.

As well as raw files, the raw data nodes can also represent secondary history files such as grand averages.

Definition `HistoryNodes As HistoryNodes`

IsOpen

Write-protected

The value of this flag indicates whether the history file is open.

Definition `IsOpen As Boolean`

LinkedData

Write-protected

The value of this flag indicates whether the history file represents a raw data set (primary history file).

Definition `LinkedData As Boolean`

Name

Write-protected

Base name of the history file without the folder and file name extension.

Definition `Name As String`

SubjectInfos

Write-protected

Information about the test subject. This information is not available with all file formats.

Definition `SubjectInfos As String`

2.13 HistoryFiles

2.13.1 Description

The `HistoryFiles` object is a collection of `HistoryFile` objects.

2.13.2 Methods

FindFile

Searches for a history file by its base name without the folder and filename extension.

Definition `Function FindFile(Name As String) As HistoryFile`

Parameters `Name` Name of the history file

KillFile

Deletes the specified history file.

Definition

`Sub KillFile(DisplayName as String)`

Parameters

`DisplayName` Alias of the history file

Refresh

Generates or, as appropriate, loads new history files if new raw data or history files were generated after the workspace was loaded.

Definition

`Sub Refresh()`

Sort

Sorts the history files in the collection and in the History Explorer display.

Definition

`Sub Sort()`

2.13.3 Properties

Count

Write-protected
Number of history files in the collection.

Definition

`Count As Long`

Item

Default element, write-protected
Returns a `HistoryFile` object from the collection. You can use the position of the file in the collection or its name to specify the history file.

Definition

`Item(DisplayNameOrIndex As Variant) As HistoryFile`

Parameters

`DisplayNameOrIndex` Specifies the position of the history file in the collection (1 – ...) or its alias

2.14 HistoryNode

2.14.1 Description

The `HistoryNode` object describes a node in the tree structure of a history file (history tree). Such a node represents a data set. The data set is a raw file, or was created by a transform. Because the `HistoryNodes` property has been defined as the default element, it is very easy to access child nodes.

2.14.2 Example

You can access child nodes of the current node using either their name or their index in the collection of all child nodes. Because `HistoryNodes` has been specified as the default element, you can omit the property. This means that the lines of code

```
Set node2 = node.HistoryNodes("Segmentation")
```

and

```
Set node2 = node("Segmentation")
```

are identical in meaning. Multiple calls of this type can be easily chained:

```
Set hn = HistoryFiles("File1")("Segmentation")(1)("Average")
```

2.14.3 Methods

ApplyTemplate

Applies the specified history template to the history node.

Definition

```
Sub ApplyTemplate(FileName As String)
```

Parameters

`FileName` Name of the history template

Delete

Removes the history node and all its child nodes from the history file.

Definition

```
Sub Delete()
```

GetHeadSurface

Fills the `Data` array with the coordinates of a point cloud that describes the head surface.

Definition

```
Function GetHeadSurface(Data() As Single) as Boolean
```

Parameters

`Data` The data consists of a continuous sequence of coordinates that describe the points in the order radius, theta and phi.

Return value

If there are no coordinates, the function returns the value `False`.

Hide

Closes all open data windows that belong to this history node.

Definition

```
Sub Hide()
```

PurgeDeletedNodes

Irrevocably deletes all child nodes of this history node that have been deleted but can still be restored.

Definition

```
Sub PurgeDeletedNodes()
```

Show

Shows the data of the node as the active tab in the Analyzer main window. If a tab containing the data of the node is already open, a second tab is not opened.

Definition

`Sub Show()`

ShowDescription

Shows the description that is stored in `Description` and `Description2` in a dialog box.

Definition

`Sub ShowDescription()`

2.14.4 Properties

Class

Write-protected

Name of the program component that created the node.

Definition

`Class As String`

Comment

User comment.

Definition

`Comment As String`

ContainsData

Write-protected

The value of this flag specifies whether the node contains data. The node can, for example, also represent an export component. In this case it does not contain any data.

Definition

`ContainsData As Boolean`

DataAvailable

Write-protected

The value of this flag specifies whether the data of this node is available.

If the associated raw file is not in the raw data folder, then most data sets of a history file will not be available.

Definition

`DataAvailable As Boolean`

Dataset

Describes the data that belongs to the history node.

This object allows you to access the entire data set in order to query properties or data. In the case of segmented data, additional `Dataset` objects are available representing individual segments. These objects can be accessed using the `Segments` collection.

Definition

`Dataset As Dataset`

DeletedNodes

Write-protected

Collection containing the deleted child nodes that can still be restored.

Definition

`DeletedNodes As DeletedHistoryNodes`

Description

Write-protected

	Description of the history node. This information describes the operation used to create the node, together with its associated parameters.
Definition	`Description As String`
Description2	Write-protected Detailed information on the operation used to create the node. This information can differ between input data sets even though the operation and parameters are the same. This information is also used for calculated operation results such as the signal-to-noise ratio (SNR) during averaging.
Definition	`Description2 As String`
FullPath	Write-protected Fully qualified path of the node. The path contains the name of the history file to which the node belongs and the name of all predecessor nodes separated by a `/`.
Definition	`FullPath As String`
HistoryFile	Write-protected History file containing the history node.
Definition	`HistoryFile As HistoryFile`
HistoryNodes	Default element, write-protected Collection of child nodes of the node.
Definition	`HistoryNodes As HistoryNodes`
Landmarks	Write-protected Collection of landmarks of the node.
Definition	`Landmarks As Landmarks`
Name	Name of the node as shown in the History Explorer.
Definition	`Name As String`
ParentNode	Write-protected Parent node of the history node.
Definition	`ParentNode as HistoryNode`
RecordingInfos	Write-protected Information that was input during acquisition, e.g. comments or error messages. This information is normally only available for nodes which represent raw data. Whether information

is actually available or not also depends on the acquisition system and the file format being used.

Definition

`RecordingInfos As String`

Segments

Write-protected
Collection of all segment objects in the node.

Definition

`Segments As Segments`

Version

Write-protected
Version of the program component that created the node.

Definition

`Version As String`

Windows

Write-protected
Collection of all data windows in the node.

Definition

`Windows As Windows`

2.15 HistoryNodes

2.15.1 Description

The `HistoryNodes` object is a collection of `HistoryNode` objects.

2.15.2 Properties

Count

Write-protected
Number of history nodes in the collection.

Definition

`Count As Long`

Item

Default element, write-protected
Returns a `HistoryNode` object from the collection. You can use the position of the node in the collection or its name to specify the history node.

Definition

`Item(NameOrIndex As Variant) As HistoryNode`

Parameters

`NameOrIndex` Specifies the position of the node in the collection (1 – ...) or its name

2.16 HistoryTemplateNode

2.16.1 Description

The `HistoyTemplateNode` object describes an individual node in a history template. It is used in the `Application.ActiveTemplateNode` property.

2.16.2 Properties

Description

Write-protected
Description of the node. This information describes the operation used to create the node, together with its associated parameters.

Definition `Description As String`

2.17 Landmark

2.17.1 Description

The `Landmark` object allows significant head positions to be indicated.

2.17.2 Properties

Name

Write-protected
Name of the landmark.

Definition `Name As String`

Phi

Write-protected
Phi in degrees.

Definition `Phi As Single`

Radius

Write-protected
Radius in millimeters. A value of 0 indicates an invalid position specification. The value of 1 assumes the head to be a perfect sphere with a uniform radius.

Definition

`Radius As Single`

Theta

Write-protected
Theta in degrees.

Definition

`Theta As Single`

2.18 Landmarks

2.18.1 Description

The `Landmarks` object is a collection of `Landmark` objects.

2.18.2 Properties

Count

Write-protected
Number of markers in the collection.

Definition

`Count As Long`

Item

Default element, write-protected
Returns a `Landmark` object from the collection. You can use the position of the landmark in the collection or its name to specify the landmark.

Definition

`Item(NameOrIndex As Variant) As Landmark`

Parameters

`NameOrIndex` Specifies the position of the landmark in the collection (1 – ...) or its name

2.19 License

2.19.1 Description

The `License` object describes a license for an optional program component of the Analyzer.

2.19.2 Properties

ID

Write-protected
The unique identification number of the license.

Definition `ID as Long`

Description

Write-protected
Description of the licensed program component.

Definition `Description as String`

2.20 Licenses

2.20.1 Description

The `Licenses` object is a collection of `License` objects.

2.20.2 Properties

Count

Write-protected
Number of licenses in the collection.

Definition `Count As Long`

Item

Default element, write-protected
Returns a `License` object from the collection.

Definition `Item(Index As Long) As License`

Parameters `Index` Specifies the position of the license in the collection (1 – ...)

2.21 Marker

2.21.1 Description

The `Marker` object represents a single marker in a data set.

2.21.2 Methods

PropertyName

Returns the name of a property of the marker.
This function can be used to list all properties of a marker including user-defined marker properties.

Definition
`Function PropertyName(Number As Long) As String`

Parameters
`Number` Number of the marker property (1 – ...)

Return value
Name of the marker property with the number specified or an empty string if this property does not exist

PropertyValue

Returns the value of a marker property.

Definition
`Function PropertyValue(Name As String) As Variant`

Parameters
`Name` Name of the marker property to be read

Return value
Value of the specified marker property or `Nothing` if the property does not exist

2.21.3 Properties

ChannelNumber

Write-protected
Channel to which the marker refers (1 – ...). The value 0 means that the marker relates to all channels.

Definition
`ChannelNumber As Long`

DateTime

Write-protected
Date and time that the marker is representing. This value is only valid for "New Segment" markers.

Definition
`DateTime As Date`

Description

Write-protected
Description of the marker.

Definition
`Description As String`

Invisible

Write-protected
The value of this flag specifies whether the marker is invisible or not when the EEG is displayed.

Definition
`Invisible As Boolean`

Points

Write-protected
Length or duration of the marker in data points.

Definition `Points As Long`

Position

Write-protected
Position of the marker in data points (1 – ...).

Definition `Position As Long`

PropertyCount

Write-protected
Number of property values of the marker. This number includes all marker properties including any user-defined marker properties.

Definition `PropertyCount As Long`

Type

Write-protected
Type of marker.

Definition `Type As String`

2.22 Markers

2.22.1 Description

The `Markers` object is a collection of `Marker` objects.

2.22.2 Properties

Count

Write-protected
Number of markers in the collection.

Definition `Count As Long`

Item

Default element, write-protected
Returns a `Marker` object from the collection.

Definition `Item(Index As Long) As Marker`

Parameters `Index` Specifies the position of the marker in the collection (1 – ...)

2.23 NewHistoryNode

2.23.1 Description

The `NewHistoryNode` object allows you to create new history nodes. You can create either new child nodes in an existing history file or the raw data node of a new secondary history file. For further information on how to use the `NewHistoryNode` object, refer to Section 1.3 as of page 21.

2.23.2 Methods

AddMarker

Inserts a marker in the new data set.

Definition

```
Sub AddMarker(ChannelNumber As Long, Position As Long,
    Points As Long, Type As String, Description As String,
    [Invisible As Boolean = False])
```

Parameters

`ChannelNumber` Number of the channel in which the marker is located. The value 0 means that the marker relates to all channels.

`Position` Position of the marker in the data set in data points (1 – ...)

`Points` Length of the marker in data points

`Type` Marker type as freely definable text. Markers of the types "New Segment" and "Time 0" are special cases.

`Description` Description of the marker. This appears in the EEG view.

`Invisible` (optional) If the value of this flag is set to `True`, the marker is not shown in the EEG.

Cancel

Cancels creation of the new node. Once you have called the `Create` or `CreateEx` method, you should call either `Cancel` or `Finish` before the end of your BASIC script.

Definition

```
Sub Cancel()
```

Create

Specifies important properties of the new data set. This includes the position of the data set in the history tree and the number of channels.

After you have created a new `NewHistoryNode` object, you must initialize it using `Create` or `CreateEx` before you can specify additional properties.

Definition

```
Sub Create(NodeName As String, ParentNode As HistoryNode,
    [FileName As String = ""], [InheritData As Boolean = True],
    [Type As VisionDataType = viDtTimeDomain],
```

```
    [NumOfChannels As Long = 0], [Length As Long = 0], [SamplingInterval As Double = 0])
```

Parameters

`NodeName` Name of the new node

`ParentNode` The parent node of the new node. If the value of this parameter has been set to `Nothing`, `FileName` must be set.

`FileName` (optional) File name without path or file name extension. Only use this parameter if you wish to create the new node as a raw data node of a new secondary history file. If this parameter has a value other than `""` or `vbNullChar`, a new history file with this name will be created in the currently open workspace.

`InheritData` (optional) If you create the node as a child node (see `ParentNode` parameter) and the value of this flag is set, the data, properties and markers of the parent node will be inherited. This allows you to add and delete markers, for instance. In this case, the `Type`, `NumOfChannels`, `Length` and `SamplingInterval` parameters are ignored.

`Type` (optional) Type of the new data set. The four data types below are allowed:

- `viDtTimeDomain`
- `viDtTimeDomainComplex`
- `viDtFrequencyDomain`
- `viDtFrequencyDomainComplex` (📘 see also Section 4.1 as of page 97)

`NumOfChannels` (optional) Number of channels in the new data set

`Length` (optional) Length of the new data set in data points

`SamplingInterval` (optional) Sampling interval in microseconds

CreateEx

Specifies important properties of the new data set. This includes the position of the data set in the history tree and the number of channels.

After you have created a new `NewHistoryNode` object, you must initialize it using `Create` or `CreateEx` before you can specify additional properties.

This function is an enhanced version of `Create`. It permits the creation of data sets with several frequency levels (layers) such as occur with continuous wavelets. The `Type` parameter therefore also supports the data types `viDtTimeFrequencyDomain` and `viDTimeFrequencyDomainComplex`.

Definition

```
Sub CreateEx(NodeName As String, ParentNode As HistoryNode, [FileName As String = ""], [InheritData As Boolean = True], 

    [Type As VisionDataType = viDtTimeDomain], 

    [NumOfChannels As Long = 0], [Length As Long = 0], [SamplingInterval As Double=0], [Layers As Long = 1], [LayerLowerLimit As Double=0], [LayerUpperLimit As Double=0], [LayerFunction As VisionLayerIncFunction= viLifLinear]))
```

Parameters

`NodeName` Name of the new node

`ParentNode` Parent node of the new node. If the value of this parameter has been set to `Nothing`, `FileName` must be set.

`FileName` (optional) File name without path or file name extension. Only use this parameter if you wish to create the new node as a raw data node of a new secondary history file. If this parameter has a value other than `""` or `vbNullChar`, a new history file with this name will be created in the currently open workspace.

`InheritData` (optional) If you create the node as a child node (see `ParentNode` parameter) and the value of this flag is set, the data, properties and markers of the parent node will be inherited. This allows you to add and delete markers, for instance. In this case, the `Type`, `NumOfChannels`, `Length` and `SamplingInterval` parameters are ignored.

`Type` (optional) Type of the new data set. The four data types below are allowed:

`viDtTimeDomain`
`viDtTimeDomainComplex`
`viDtFrequencyDomain`
`viDtFrequencyDomainComplex` (📖 see also Section 4.1 as of page 97)

`NumOfChannels` (optional) Number of channels in the new data set

`Length` (optional) Length of the new data set in data points

`SamplingInterval` (optional) Sampling interval in microseconds

`Layers` (optional) Number of frequency levels in the data set

`LayerLowerLimit` (optional) Frequency assigned to the lowest frequency level

`LayerUpperLimit` (optional) Frequency assigned to the highest frequency level

`LayerFunction` (optional) Increment function between the frequency levels of a data set. 📖 You will find the possible values of this property in Chapter 4 as of page 97.

CreateWithChannelMap

Specifies important properties of the data set. This method replaces the `Create` or `CreateEx` call and can be used if you wish to take over some of the channels of the parent node into the new data set and optionally wish to add further channels (📖 see also Section 1.3.4 as of page 26).

In contrast to `Create` and `CreateEx`, you cannot create a raw data node of a new history file. You can only create data sets that have a parent node. Specifications such as the length of the data set or the data type are determined by the parent node.

The `ChannelMap` array allows you to specify what data is to appear in the channels of the new data set. In this context, you can specify for each channel the number of a channel in the parent node in order to take over the data of this channel. You can also specify 0 in order to define the contents of the channel using `WriteData`. In this way, you can rearrange the channels in any way you wish or delete them by omitting them from the specification.

If, for example, you use an array with the values (1, 0, 0, 2) as `ChannelMap`, you take over the first channel of the parent node as the first channel of the new node and the second channel of the parent node as the fourth channel of the new node. The values of the second and third channels of the new node are defined using `WriteData`. If the parent node contains a third channel, this is not taken over.

Definition

```
Sub CreateWithChannelMap(NodeName As String,
    ParentNode As HistoryNode, ChannelMap() as Long)
```

Parameters

`NodeName` Name of the new node

`ParentNode` Parent node of the new node

`ChannelMap` This array specifies the source of the data for each channel of the new node. Specify 0 in order to write new values to the channel in the node or specify a channel number of the parent node to take over this channel. The length of the array determines the number of channels in the new node.

Finish

Completes creation of the new node. Once you have called the `Create` or `CreateEx` method, you should call either `Cancel` or `Finish` before the end of your BASIC script.

If you call `Finish` from the integrated BASIC interpreter, execution of your script is automatically terminated after the method has been processed. This means that any lines that follow `Finish` will no longer be executed.

Definition

```
Sub Finish()
```

RemoveMarker

Removes the marker corresponding to the description. Uppercase/lowercase and spaces in `Type` and `Description` are ignored in the comparison.

Definition

```
Sub RemoveMarker(ChannelNumber As Long, Position As Long,
    Points As Long, Type As String, Description As String)
```

Parameters

`ChannelNumber` Number of the channel in which the marker is located. The value 0 means that the marker relates to all channels.

`Position` Position of the marker in the data set in data points (1 – ...)

`Points` Length of the marker in data points

`Type` Marker type as freely definable text. Markers of the types "New Segment" and "Time 0" are special cases.

`Description` Description of the marker. This appears in the EEG view.

SetChannelName

Sets the name of a channel.

Definition

```
Sub SetChannelName(ChannelNumber As Long, NewName As String)
```

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`NewName` Name of the channel

SetChannelPosition

Sets the position of a channel.

Definition

```
Sub SetChannelPosition(ChannelNumber As Long, Radius As Single,
    Theta As Single, Phi As Single)
```

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`Radius` Radius in millimeters. Set the value 0 if the channel does not have any valid head coordinates. Set the value 1 if you assume that the head is an ideal sphere with a uniform radius.

`Theta` Theta in degrees

`Phi` Phi in degrees

SetChannelUnit

Sets the unit for the data of a channel, e.g. µV, μV^2 (📘 see also Section 4.2 on page 98). If the unit is not set for a channel, the channel is assigned the unit specified for the entire data set in the `Unit` property.

Definition

```
Sub SetChannelUnit(ChannelNumber As Long, Unit As VisionDataUnit)
```

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`Unit` Unit for the data of the channel

SetChannelUserProperty

Sets a user-defined property for a channel. The name of the property should comprise a namespace (e.g. institute name), a period and then the actual name of the property. The namespace "*BrainVision.*" is reserved.

Definition

```
Sub SetChannelUserProperty(ChannelNumber as Long, 
    PropertyName as String, PropertyValue as Variant)
```

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`PropertyName` Name of the property to be set

`PropertyValue` Value to be set

SetDatasetUserProperty

Sets a user-defined property for a data set. The name of the property should comprise a namespace (e.g. institute name), a period and then the actual name of the property. The namespace "*BrainVision.*" is reserved.

Definition

```
Sub SetDatasetUserProperty(PropertyName as String, 
    PropertyValue as Variant)
```

Parameters

`PropertyName` Name of the property to be set

`PropertyValue` Value to be set

SetHeadLandmark

Sets a named landmark for the purposes of orientation.

Definition

```
Sub SetHeadLandmark(Name As String, Radius As Single, 
    Theta As Single, Phi As Single)
```

Parameters

`Name` Name of the landmark

`Radius` Radius in millimeters. Set the value 1 if you assume that the head is an ideal sphere with a uniform radius.

`Theta` Theta in degrees

`Phi` Phi in degrees

SetHeadSurface

Describes the head surface as a point cloud. `Data` stands for a continuous sequence of coordinates that describe the points in the order radius, theta and phi.

Definition

`Sub SetHeadSurface(Data() As Single)`

Parameters

`Data` Coordinates of the points

SetMarkerUserProperty

Sets a user-defined property for a marker. The first parameters identify the marker in which the property is to be set. The last two parameters specify the name and value of the property. The name of the property should comprise a namespace (e.g. institute name), a period and then the actual name of the property. The namespace "*BrainVision.*" is reserved.

Definition

`Sub SetMarkerUserProperty(ChannelNumber as Long, Position as Long, Points as Long, Type as String, Description as String, PropertyName as String, PropertyValue as Variant)`

Parameters

`ChannelNumber` Number of the channel in which the marker is located. The value 0 means that the marker relates to all channels.

`Position` Position of the marker in the data set in data points (1 – ...)

`Points` Length of the marker in data points

`Type` Marker type as freely definable text. Markers of the types "New Segment" and "Time 0" are special cases.

`Description` Description of the marker. This appears in the EEG view.

`PropertyName` Name of the property to be set

`PropertyValue` Value to be set

SetRefChannelName

Sets the name of the channel that is to be used as the reference channel for the specified channel.

Definition

`Sub SetRefChannelName(ChannelNumber As Long, NewName As String)`

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`NewName` Name of the referenced channel

SetSecondChannelPosition

Sets an additional position that describes the channel on the head surface. This specification is used when an additional position is assigned to a channel in addition to its own coordinates (for instance with the Coherence transform).

Definition

`Sub SetSecondChannelPosition(ChannelNumber As Long, Radius As Single, Theta As Single, Phi As Single)`

Parameters

`ChannelNumber` Number of the channel (1 – ...)

`Radius` Radius in millimeters. Set the value 0 if the channel does not have any valid head coordinates. Set the value 1 if you assume that the head is an ideal sphere with a uniform radius.

`Theta` Theta in degrees

`Phi` Phi in degrees

TryLater

This function is only needed if a macro is used in a history template. If information from a history node which is neither the current node nor the parent of the current node is needed to calculate the new data set in this case, then this node may not have been calculated as yet. Calling this function informs the program of this. In this case the program will continue with creation of other history nodes and will try to create the node again later.

Definition

```
Sub TryLater()
```

WriteData

Writes data to the data set.

You can write data either channel by channel or simultaneously for all channels. For reasons of performance, it is recommended that you write data to all channels simultaneously.

If you have used `CreateWithChannelMap`, you only need to write data to those channels that you actually wish to create from scratch. For example, if you have specified 0 twice as the number of the source channel, you should use `WriteData` in exactly the same way as if you wanted to create a data set containing only two channels.

You can use the `Position` and `Points` parameters to write the data section by section. This is necessary if the data set is too large for the available memory, for example. Although it is possible to write the sections out of sequence, this approach is not recommended for reasons of performance.

It is possible that each data point that is to be written is made up of multiple values. This is the case if you are writing multiple channels simultaneously, but also if the data set contains complex data, for instance. In this event, the values within a data point are always multiplexed.

Definition

```
Sub WriteData(ChannelNumber As Long, Position As Long, 
    Points As Long, Data() As Single)
```

Parameters

`ChannelNumber` Number of the channel to be written. The value 0 means that the block covers all channels. In this case the data must be available in multiplexed form.

`Position` Number of the first data point to be written (1 – ...)

`Points` Number of data points to be written

`Data` Data to be written

2.23.3 Properties

Averaged	Set this flag if the data set contains averaged data. Among other things, this specification is used to lock or unlock transforms in the *Transformations* tab.
Definition	`Averaged As Boolean`
Description	Reads or sets the description of the operation and its input parameters.
Definition	`Description As String`
Description2	Reads or sets the description of the operation results.
Definition	`Description2 As String`
SegmentationType	Reads or sets the segmentation type of the data set. The segmentation types are described in Section 4.3 on page 99. Among other things, the segmentation type is used to lock or unlock transforms in the *Transformations* tab.
Definition	`SegmentationType As VisionSegType`
Unit	Reads or sets the unit for the data in the data set, e.g. μV, μV^2 (see also Section 4.2 on page 98). This unit applies to all channels whose unit has not been set explicitly with `SetChannelUnit()`.
Definition	`Unit As VisionDataUnit`

2.24 ProgressBar

2.24.1 Description

The `ProgressBar` object represents a progress bar. Use a progress bar to keep the user informed of the progress of long-running calculations and to allow the calculation to be canceled.

If multiple progress bars are created at the same time, they are arranged vertically in the same window.

2.24.2 Example

There are two ways of creating an object of the `ProgressBar` class. If you have included the type library for Analyzer Automation, you can write

```
Dim pb as New ProgressBar
```

The type library is automatically included in the integrated BASIC, and you can use this statement. If you have not included the type library, you must instead write the following:

```
Set pb = CreateObject("VisionAnalyzer.ProgressBar")
```

The example below shows how to use two nested progress bars. The program is momentarily paused in the inner loop in order to prevent the progress bars from moving too fast. A real application would perform a calculation in place of this pause.

A check is performed in both the inner and outer loops to see whether the user has clicked *Cancel*. It is sufficient for the user to click once. This causes both progress bars to be placed in the state "Cancel" and their `UserCanceled` property to be set to `True`.

```
Sub Main

    Dim pb1 As New ProgressBar

    Dim pb2 As New ProgressBar

    ' Initialize objects

    pb1.Init "This title will not be shown", "First Bar"

    pb2.Init "ProgressBar Demo", "Second Bar"

    ' Set value range for progress bar

    pb1.SetRange 0, 5

    pb2.SetRange 0, 100

    pb1.SetStep 1

    pb2.SetStep 1

    For i = 1 To 5

        For j = 1 To 100

            ' Terminate loop if the user clicks "Cancel"

            If pb2.UserCanceled Then

                Exit For

            End If

            ' Move progress bar 2
```

```
              pb2.StepIt
              ' Wait in place of a calculation
              Wait .001
         Next j
         ' Terminate loop if the user clicks "Cancel"
         If pb1.UserCanceled Then
              Exit For
         End If
         ' Move progress bar 1
         pb1.StepIt
         ' Set progress bar 2 to start
         pb2.SetPosition(0)
    Next i
    pb1.SetText "Done 1"
    pb2.SetText "Done 2"
    ' Wait 2 seconds before the bars are removed
    Wait 2
End Sub
```

2.24.3 Methods

Hide

Hides the progress bar window.

Definition

`Sub Hide()`

Init

Initializes the progress bar and displays it.

Definition

`Sub Init(Title As String, Text As String)`

Parameters

`Title` Title of the progress bar. In the case of nested progress bars, only the title of the first bar is displayed.

`Text` The text associated with the progress bar. The text can be changed subsequently with `SetText`.

OffsetPosition

Sets the position of the progress bar relative to the current position. The position specifies the state that has been reached between the upper and lower range limits.

Definition

`Sub OffsetPosition(Position As Long)`

Parameters

`Position` New position of the progress bar relative to the old position

SetPosition

Sets the position of the progress bar. The position specifies the state that has been reached between the upper and lower range limits.

Definition

`Sub SetPosition(Position As Long)`

Parameters

`Position` New position of the progress bar

SetRange

Sets the upper and lower range limits. These values are set to 0 and 100 by default.

Definition

`Sub SetRange(Lower As Long, Upper As Long)`

Parameters

`Lower` The lower limit of the range shown

`Upper` Upper limit of the range shown

SetStep

Sets the step size that is used with `StepIt`. The default value is 10.

Definition

`Sub SetStep(Step As Long)`

Parameters

`Step` Length of the increment for the progress bar

SetText

This function replaces the existing text.

Definition

`Sub SetText(Text As String)`

Parameters

`Text` The text associated with the progress bar

Show

Shows the progress bar window.

Definition

`Sub Show()`

StepIt

Moves the position forward by the step length set in `SetStep`. The position specifies the state that has been reached between the upper and lower range limits.

Definition

`Sub StepIt()`

2.24.4 Properties

UserCanceled

This flag is set if the user has clicked *Cancel*.
Query this value in the outer loop of a long calculation, for instance, to determine whether the calculation should be canceled.

Definition `UserCanceled As Boolean`

2.25 Segment

2.25.1 Description

The `Segment` object describes a single data segment within a history node.

2.25.2 Properties

Dataset

Write-protected
This `Dataset` object describes the data of a segment. The data set is a subset of the data set that contains the segment. All position specifications in this data set relate to the beginning of the segment.
The `Markers` collection of this object no longer contains any "New Segment" markers.

Definition `Dataset As Dataset`

DateTime

Write-protected
Date and time of the beginning of the segment.

Definition `DateTime As Date`

TimeZeroOffset

Write-protected
Position of the zero time point relative to the start of the segment in data points.

Definition `TimeZeroOffset As Long`

2.26 Segments

2.26.1 Description

The `Segments` object is a collection of `Segment` objects.

2.26.2 Properties

Count

Write-protected
Number of segments in the collection.

Definition

`Count As Long`

Item

Default element, write-protected
Returns a `Segment` object from the collection.

Definition

`Item(Index As Long) As Segment`

Parameters

`Index` Specifies the position of the segment in the collection (1 – ...)

2.27 Transformation

2.27.1 Description

You can call some primary transforms of the Analyzer with parameters using the `Transformation` object. A list of supported transforms and their parameters is given in Chapter 3 as of page 91.

The advantage of this approach compared with calling a transform via the *Transformations* tab is that it enables dynamic parameterization. In other words, the parameters for the transform can be determined at runtime. This means, for example, that results of previously completed operations can be taken into account.

For further information on how to use the `Transformation` object, refer to Section 1.5 as of page 30.

2.27.2 Methods

Do

Performs a transform.

The behavior of the transform is controlled by its parameters. 📖 Suitable parameters for each transform that can be called dynamically are defined in Chapter 3 as of page 91.

Definition

```
Sub Do(Transformation As String, Parameters As String,
    ParentNode As HistoryNode, [NodeName As String])
```

Parameters

`Transformation` Name of the transform

`Parameters` Transform parameters

`ParentNode` The node to which the transform is applied

`NodeName` (optional) Name of the node created by the transform. If no name is input, it is assigned by the transform.

TryLater

This function is only needed if a macro is used in a history template. If information from a history node which is neither the current node nor the parent of the current node is needed to calculate the new data set in this case, then this node may not have been calculated as yet. Calling this function informs the program of this. In this case the program will continue with creation of other history nodes and will try to create the node again later.

Definition

```
Sub TryLater()
```

2.28 Window

2.28.1 Description

The `Window` object describes a tab in the main window. A tab of this sort would typically contain an EEG view, for instance. You can use a `Window` object to control the behavior of an EEG view.

2.28.2 Methods

ActivateTransientTransformation

Allows a transient transform to be called. This method can only be used if an EEG view with a selected range is shown in the tab.

Definition

```
Sub ActivateTransientTransformation(Name As String)
```

Parameters

`Name` Name of the transient transform

Example

The example selects an interval in the currently open node and creates a transient FFT view.

```
ActiveNode.Windows(1).SetMarkedInterval(1001,512)

ActiveNode.Windows(1).ActivateTransientTransformation "FFT"
```

Close

Closes the tab.

Definition

```
Sub Close()
```

Copy

Copies the contents of the tab to the clipboard as an image. This method can only be used if an EEG view is shown in the tab.

Definition

```
Sub Copy()
```

Maximize

Maximizes the window. This method can only be used if data that is open in the main window is represented in windows and not in the form of tabs.

Definition

```
Sub Maximize()
```

Minimize

Minimizes the window. This method can only be used if data that is open in the main window is represented in windows and not in the form of tabs.

Definition

```
Sub Minimize()
```

Print

Prints the content of the tab. This method can only be used if an EEG view is shown in the tab.

Definition

```
Sub Print()
```

Restore

Restores the window to its original size. This method can only be used if data that is open in the main window is represented in windows and not in the form of tabs.

Definition

```
Sub Restore()
```

SetDisplayedInterval

Sets the data range displayed in the EEG view. This method can only be used if an EEG view is shown in the tab.

Definition

```
Sub SetDisplayedInterval(Position as Long, DataPoints as Long)
```

Parameters

`Position` The first data point displayed (1 – ...)
`DataPoints` Number of data points displayed

SetMarkedInterval

Sets the selected range in the EEG view. This method can only be used if an EEG view is shown in the tab. The range must lie within the displayed interval.

Definition

```
Sub SetMarkedInterval(Position as Long, DataPoints as Long)
```

Parameters

`Position` The first selected data point (1 – ...)
`DataPoints` Number of data points selected

MoveMarkedInterval	Moves the selected range in the EEG view. This method can only be used if an EEG view is shown in the tab.
Definition	`Sub MoveMarkedInterval(Points as Long)`
Parameters	`Points` Number of data points by which the range is to be moved

SetScalingRange	Sets the scaling range of the EEG view. This method can only be used if an EEG view is shown in the tab.
Definition	`Sub SetScalingRange(MinValue as Single, MaxValue as Single)`
Parameters	`MinValue` Lower limit of the scaling range `MaxValue` Upper limit of the scaling range

ResetScalingRange	Resets the scaling range of the EEG view to the default. This method can only be used if an EEG view is shown in the tab.
Definition	`Sub ResetScalingRange()`

2.28.3 Properties

DisplayBaselineCorrection	Switches baseline correction of the EEG view on or off. Only the baseline of the display is changed, not the data itself. This property can only be used if an EEG view is shown in the tab.
Definition	`DisplayBaselineCorrection As Boolean`

DisplayDataPoints	Write-protected Number of data points shown in the EEG view. This property can only be used if an EEG view is shown in the tab.
Definition	`DisplayDataPoints as Long`

DisplayStartPosition	Write-protected The first data point shown in the EEG view (1 – ...). This property can only be used if an EEG view is shown in the tab.
Definition	`DisplayStartPosition as Long`

HistoryNode	Write-protected Returns the history node whose data is being displayed. This property can only be used if an EEG view is shown in the tab.
Definition	`HistoryNode As HistoryNode`

MarkedIntervalDataPoints

Write-protected
Number of data points selected in the EEG view. This property can only be used if an EEG view is shown in the tab.

Definition

```
MarkedIntervalDataPoints as Long
```

MarkedIntervalStartPosition

Write-protected
The first data point selected in the EEG view (1 – ...). This property can only be used if an EEG view is shown in the tab.

Definition

```
MarkedIntervalStartPosition as Long
```

Title

Title of the tab.

Definition

```
Title As String
```

Type

Write-protected
Type of the tab. This value is "EEGData" if an EEG view is shown in the tab.

Definition

```
Type As String
```

Return value

`EEGData` EEG data window
`Macro` Macro editing window (for editing the BASIC source code)
`Template` Template editor (for editing history templates)

Example

The example selects a data range in the currently active tab, provided that this is actually a view showing EEG data.

```
If ActiveNode.Windows(1).Type = "EEGData" Then

    ActiveNode.Windows(1).SetMarkedInterval(1000,512)

End If
```

2.29 Windows

2.29.1 Description

The `Windows` object is a collection of `Window` objects corresponding to the tabs in the main window.

2.29.2 Properties

Count

Write-protected
Number of tabs.

Definition
`Count As Long`

Item

Default element, write-protected
Returns a `Window` object from the collection.

Definition
`Item(TitleOrIndex As Variant) As Window`

Parameters
`TitleOrIndex` Specifies the position of the tab in the collection (1 – ...) or its title

2.30 Workspace

2.30.1 Description

The `Workspace` object describes an Analyzer workspace.

2.30.2 Properties

ExportFileFolder

Write-protected
Default folder for exported files.

Definition
`ExportFileFolder As String`

FullName

Write-protected
Name of the workspace file including fully-qualified path.

Definition
`FullName As String`

HistoryFileFolder

Write-protected
Folder for history files.

Definition
`HistoryFileFolder`

Name

Write-protected
Base name of the workspace file without folder and file name extension.

Definition
`Name As String`

RawFileFolder

Write-protected
Folder for raw data.

Definition
`RawFileFolder As String`

2.31 Workspaces

2.31.1 *Description*

The `Workspaces` object is a collection of `Workspace` objects. It is used in the `Application` object to list all the workspaces in the Workfiles folder.

2.31.2 *Methods*

Refresh

Rereads the workspace files that are present in the Workfiles folder.

Definition
`Sub Refresh()`

2.31.3 *Properties*

Count

Write-protected
Number of workspaces available.

Definition
`Count As Long`

Item

Default element, write-protected
Returns a `Workspace` object from the collection.

Definition
`Item(NameOrIndex As Variant) As Workspace`

Parameters
`NameOrIndex` Specifies the position of the workspace in the collection (1 – ...) or its name

Chapter 3 Callable transforms

The currently available transforms and their parameters are listed in this chapter. They can be called as follows:

```
Transformation.Do(Transformation As String, Parameters As String, _
   ParentNode As HistoryNode, [NodeName As String])
```

The sections below describe the `Transformation` and `Parameters` arguments for the different transforms.

The following general syntax is used irrespective of the transform that is being called in order to pass the transform parameters to the `Do` method.

The notation for the parameters always takes the form of variable/value pairs. Variable names are not case-sensitive. If multiple variables are specified, they are separated by semicolons (;). The following example uses the variables `Highcutoff`, `Lowcutoff` and `Notch`:

```
Transformation.Do "Filters", _
   "Highcutoff=70;Lowcutoff=2;Notch=50", ActiveNode,"Test"
```

If a variable has multiple values, these are separated by commas (,):

```
Transformation.Do "Filters", _
   "Highcutoff=70.48;Lowcutoff=2.24;Notch=50", ActiveNode,"Test"
```

If variables are defined as vectors, the elements are indexed with parentheses `()`. The first index is 1. A value without parentheses is interpreted the same as a value with index (1), i.e. `Highcutoff` is equal to `Highcutoff(1)`:

```
Transformation.Do "Filters", "Highcutoff = 12,48;" & _
    "Highcutoff(3)=70,48; Lowcutoff(3) = 2; Notch(3)=50", _
   ActiveNode, "Test"
```

The variables can occur in any sequence in the parameters.

For detailed information on the transforms below, refer to the Analyzer User Manual. Here, we shall only describe the parameter syntax for the transforms.

3.1 Band Rejection

Name of the transform: `BandRejection`

Table 3-1. Parameters for Band Rejection

Variable	Description
`Filter`	A band rejection filter is defined. The variable can be indexed since multiple filters can be defined. A filter is always described by three values: Frequency, bandwidth and order. The order can only be 2 or 4. Example: `Filter(1)=17,2,4;Filter(2)=50,2,2;`
`Channels`	This variable lists the channels to be filtered by number. Example: `Channels=1,2,15` The variable is not allowed to be defined at the same time as the `NamedChannels` variable. If neither `Channels` nor `NamedChannels` has been defined, all channels are filtered.
`NamedChannels`	Here, the channels to be filtered can be listed by name. Example: `NamedChannels=Fp1,F7,Oz` This variable is not allowed to be defined at the same time as the `Channels` variable.

Examples:

```
Transformation.Do "BandRejection", "Filter=20,2,4", ActiveNode
```

This defines a band rejection filter of 20 Hz with a bandwidth of 2 Hz and an order of 4. All channels are filtered.

```
Transformation.Do "BandRejection", _
    "Filter(1)=20,2,4; Filter(2)=30,3,4;Channels=2,4,16", _
  ActiveNode
```

This defines a band rejection filter of 20 Hz with a bandwidth of 2 Hz and an order of 4, plus a filter of 30 Hz, a bandwidth of 3 Hz and an order of 4. Channels 2, 4 and 16 are filtered.

```
Transformation.Do "BandRejection", _
  "Filter=20,2,2;NamedChannels=Fp1", ActiveNode
```

Here, the Fp1 channel is filtered with a band rejection filter of 20 Hz, a bandwidth of 2 Hz and an order of 2.

3.2 Complex Demodulation

Name of the transform: `ComplexDemodulation`

Table 3-2. Parameters for Complex Demodulation

Variable	**Description**
`output`	Use this variable to specify whether the power or phase is to be output. This variable accepts the values `power` and `phase`.
`begin`	Start of the frequency band in hertz.
`end`	End of the frequency band in hertz.

Example:

```
Transformation.Do "ComplexDemodulation", _
   "output=phase;begin=10;end=20", ActiveNode
```

The phase is output for the frequency band 10 Hz through 20 Hz.

3.3 Formula Evaluator

Name of the transform: `Formula`

Table 3-3. Parameters for Formula Evaluator

Variable	**Description**
`Formula`	This variable describes the formula for calculating a new channel as text. Since several formulas can be defined, the variable can be indexed. The formula is input in accordance with the syntax of the Formula Evaluator. For a description of the syntax, refer to the Analyzer User Manual. Example: `Formula(1) = Fp1Power = Fp1 * Fp1`

Table 3-3. Parameters for Formula Evaluator

Variable	**Description**

`Unit`	This variable describes the unit for a newly calculated channel. If the unit is not specified, microvolts are taken by default. The possible values are (`µ` can be replaced by `u`, `²` can be replaced by `2`, and no distinction is made between uppercase and lowercase):
	`None` (without a unit)
	`µV` or `uV`
	`µV/Hz` or `uV/Hz`
	`µV²` or `uV2`
	`µV²/Hz` or `uV2/Hz`
	`µV/m²` or `uV/m2`
	Example:
	`Unit(1) = uV2`

`KeepOldChannels`	This variable accepts the values `False` and `True`. It defines whether the data of the parent node is to be included in the new data set. Example: `KeepOldChannels = True`

Examples:

```
Transformation.Do "Formula", _
    "Formula(1) = RelationFp1Fp2 = Fp1 / Fp2; Unit(1) = none", _
  ActiveNode
```

The new data set contains a new channel named *RelationFp1Fp2*. The data does not have a unit. The data of the parent node is not kept.

```
Transformation.Do "Formula", "Formula(1) = Fp1' = " & _
    "(shift(Fp1, -1) + Fp1 + shift(Fp1, 1)) / 3;" & _
    "Formula(2) = Fp2' = (shift(Fp2, -1) + Fp2 + " & _
    "shift(Fp2, 1)) / 3; KeepOldChannels = True", _
  ActiveNode, "Test"
```

Two new channels, Fp1' and Fp2', are created. The unit for these channels is µV, as the unit was not explicitly defined. The data of the parent node is kept.

3.4 IIR Filters

Name of the transform: `Filters`

Table 3-4. Parameters for IIR Filters

Variable	Description
`LowCutoff`	A low-cutoff filter is defined. The variable can be indexed. In this case, the index signifies the number of the channel to be filtered. The filter is described by two values – frequency and slope in db/octave. The slope can have the value 12, 24 or 48. If it is not specified, 12 is taken by default. Example: `LowCutoff(1)=2,24;LowCutoff(2)=4;`
`HighCutoff`	This variable relates to a high-cutoff filter. Otherwise the description of the low-cutoff filter above also applies to this filter. Example: `HighCutoff(1)=70,24;HighCutoff(2)=70;`
`Notch`	A band rejection filter can be specified for power-line noise. A channel can be indexed here, too, in the same way as for low-cutoff and high-cutoff filters. The permissible values for this filter are 50 or 60. Example: `Notch=50`
`IndividualFilters`	This variable accepts the values `False` and `True`. This variable defines whether the channels are to be filtered individually or if they are all to be given the same filters. Normally the program will autonomously decide whether individual filtering is required. If an index greater than 1 is used somewhere in the filter parameters, it switches to individual filtering. Otherwise all channels are filtered in the same way. The variable therefore only has to be set to `True` if just the first channel needs to be filtered. Example: `IndividualFilters=True`

Examples:

```
Transformation.Do "Filters", "HighCutoff=70", ActiveNode
```

A high-cutoff filter of 70 Hz is defined. Since the slope is not specified, 12 db/octave is used. All channels are filtered.

```
Transformation.Do "Filters", _
   "LowCutoff(10)=0.535,48;HighCutoff(10)=70,48;Notch(10)=50", _
  ActiveNode
```

Only channel ten is filtered here. This is done with a low-cutoff filter of 0.535 Hz, 48 db/octave, a high-cutoff filter of 70 Hz, 48 db/octave and a notch filter of 50 Hz.

```
Transformation.Do "Filters", "IndividualFilters=True; " & _
   "LowCutoff(1)=2", ActiveNode, "Test"
```

Here, only the first channel is filtered. Filtering is performed with a low-cutoff filter of 2 Hz.

●

Chapter 4 Enumerator types

This chapter describes the various enumerator types.

Note that the integrated BASIC interpreter does not permit declaration of enumerator variables.

The example below will trigger an error message to this effect:

```
Dim vdt As VisionDataType

vdt = viDtTimeDomain
```

Therefore, wherever you wish to use enumerators as variables, you should declare the variable as the type `Long`:

```
Dim vdt As Long

vdt = viDtTimeDomain
```

4.1 VisionDataType

The enumerator type `VisionDataType` defines constants for the various data types that a history node can manage.

The individual constants and the values associated with them are listed in Table 4-1. The numeric values are specified as hexadecimal numbers in BASIC notation. The numeric values were not selected arbitrarily. The last hexadecimal digit of real data types is always 1 and the last hexadecimal digit of complex data types is always 2.

Table 4-1. Values of the enumerator type "VisionDataType"

Identifier	**Numeric value**	**Meaning**
`viDtTimeDomain`	`&H101`	Data in the time domain
`viDtTimeDomainComplex`	`&H102`	Complex data in the time domain
`viDtFrequencyDomain`	`&H201`	Data in the frequency domain
`viDtFrequencyDomainComplex`	`&H202`	Complex data in the frequency domain
`viDtTimeFrequencyDomain`	`&H301`	Data in the time-frequency domain (e.g. wavelet data)
`viDtTimeFrequencyDomainComplex`	`&H302`	Complex data in the time-frequency domain (e.g. wavelet data)
`viDtUserDefined`	`&H10001`	User-defined data type

Table 4-1. Values of the enumerator type "VisionDataType"

Identifier	**Numeric value**	**Meaning**
`viDtUserDefinedComplex`	&H10002	User-defined data type, complex
`viDtUserDefinedNoMatrix`	&H100FF	User-defined data type that does not fit in the standard matrix

4.2 VisionDataUnit

The enumerator type `VisionDataUnit` defines constants for the various units that can be used in EEG channels.

If a channel uses `viDuUnitless` as the value of its `Unit` property, the convention is that it can also use a user-defined unit. In this case, the user-defined unit is entered as a value of the `UnitString` property. This convention was chosen in order to allow known units to continue to be processed automatically (`viDuMicrovoltSquare` as power values) while at the same time completely excluding user-defined units from any such processing.

The individual constants and the values associated with them are listed in Table 4-2.

Table 4-2. Values of the enumerator type "VisionDataUnit"

Identifier	**Numeric value**	**Meaning**
`viDuMicrovolt`	0	µV
`viDuUnitless`	1	Without a unit
`viDuMicrovoltsPerHertz`	2	µV/Hz
`viDuMicrovoltSquare`	3	μV^2
`viDuMicrovoltSquarePerHertz`	4	μV^2/Hz
`viDuMicrovoltPerMeterSquare`	5	$\mu V/m^2$

4.3 VisionSegType

The enumerator type `VisionSegType` defines constants for the various segmentation types that can be used in EEG data sets.

Data sets whose segmentation type is "not segmented" or `viStNotSegmented` can nevertheless contain "New Segment" markers. In this event, the "New Segment" markers indicate interruptions during recording rather than segments in the traditional sense.

The convention is that the segments of a data set whose segmentation type is `viStMarker`, `viStMarkerAndABE` or `viStFixedTime` must all be the same length. This assumption is important in many scenarios, e.g. for the Average transform. You should avoid creating data sets, for example with `NewHistoryNode`, that do not observe this convention and which can therefore not be processed meaningfully.

The individual constants and the values associated with them are listed in Table 4-3.

Table 4-3. Values of the enumerator type "VisionSegType"

Identifier	Numeric value	Meaning
`viStNotSegmented`	0	Not segmented
`viStMarker`	1	Segmented relative to marker position
`viStMarkerAndABE`	2	Segmented relative to marker position with the aid of an ABE expression
`viStFixedTime`	3	Segmented in fixed time units
`vistManual`	4	Segmented manually; segments of different lengths are possible

4.4 VisionLayerIncFunction

The enumerator type `VisionLayerIncFunction` defines constants for the increment functions between the frequency levels (layers) of a data set. Frequency levels occur with continuous wavelets, for instance.

The increment function specifies what frequencies are assigned to the individual frequency levels. The frequencies of the top and bottom levels and the number of levels are predetermined. The frequencies of the intermediate levels are then determined in such a way that the range between the top and bottom frequencies is divided in the ways specified by the increment function. For example, if `viLifLinear` is used, the frequency levels are arranged at constant intervals.

The individual constants and the values associated with them are listed in Table 4-4.

Table 4-4. Values of the enumerator type "VisionLayerIncFunction"

Identifier	**Numeric value**	**Meaning**
`viLifLinear`	0	Linear (i.e. a constant interval between the frequency levels)
`viLifLogarithmic`	1	Logarithmic distance between the frequency levels

●

Chapter 5 Error codes

This chapter lists the error codes used by Analyzer Automation along with the associated messages. These error codes are set when an error occurs during execution of an Automation call.

The error numbers only indicate the lower 15 bits of the error code. To extract the error number, the upper bits of the error code have to be masked out.

If a macro has not defined any custom error handling, the error message associated with the error number is displayed in the status bar of the macro editing window. The programming example below shows how to use the error handling provided by the integrated BASIC interpreter to show custom error messages:

```
Sub Main
    ' Initialize error handling
    On Error GoTo CheckError

    Set hf = HistoryFiles(1)
    MsgBox "First channel name: " + hf(1).Dataset.Channels(1).Name
    Exit Sub

CheckError:
    ' Extract Automation error number
    nError = Err.Number And &h7fff
    Select Case nError
    Case 1501
         ' Code for "History file is closed."
         MsgBox "History file is closed."
    End Select
End Sub
```

Table 5-1 lists all the error codes used.

Table 5-1. Error codes

Code	**Message**
1500	Display name of History File: Invalid characters found in '%s'. '%s' can't be used for naming.

Table 5-1. Error codes

Code	Message
1501	History file is closed.
1502	Can't handle this data type.
1503	History node is invalid.
1504	History node not found.
1505	The data set is currently not available.
1506	History file is invalid.
1507	History node does not contain data.
1508	History file not found.
1509	Can't access workspace.
1510	Index is out of range.
1511	A history node with the same name already exists.
1512	Rename History Node: Invalid characters found in '%s'. '%s' can't be used for naming.
1513	Channel not found.
1514	No workspace is loaded.
1515	Window does no longer exist.
1516	Requested data is out of segment range.
1517	Window not found.
1518	History node collection is invalid.
1519	History template not found.
1520	History Template: Type mismatch.
1521	User canceled history template processing.
1522	The start node '%s' was not found in the history template '%s'.
1523	User defined message.
1524	A history file with this display name already exists in the current workspace.
1525	Can't change display name on open history file.
1526	'Create' has not been called.
1527	Invalid data size or format.
1528	Channel is out of range.
1529	NewHistoryNode.Create: Invalid data type.
1530	NewHistoryNode.Create: Parameters mismatch.
1531	NewHistoryNode.AddMarker: Marker out of range.

Table 5-1. Error codes

Code	Message
1532	NewHistoryNode.WriteData: Can't write data. Data set inherits data.
1533	Transformation.Do: Transformation '%s' not found.
1534	Transformation.Do: '%s', parameters '%s' are not correct.
1535	Transformation.Do: Type mismatch
1536	‹Error, warning or other message from Transformation›
1537	Dataset.GetData: The "ChannelList" parameter is incorrect.
1538	The requested data layer is out of range.
1539	The requested number of layers is invalid.
1540	Wrong function for this type of data. Use NewHistoryNode.CreateEx().
1541	The requested layer function is not supported.
1542	Workspace not found.
1543	Landmark not found.
1544	Marked interval can only be inside of a displayed interval.
1545	User canceled operation.
1546	Invalid characters in history node name. '\/:' can't be used for naming.
1547	Progress bar is not initialized.
1548	Menu item not found.
1549	FastArray: First element is out of range.
1550	FastArray: Division by Zero.
1551	FastArray: Source data array is not initialized.
1552	FastArray: Out of memory.
1553	Fast Array: Target data array is not initialized.
1554	FastArray: Only one dimensional floating point single precision arrays are supported.
1555	FastArray.Parameters lead into infinite loop.
1556	FastArray: Parameter exceeds the limit of 536870912.
1557	FastArray: The parameters would lead into more than 536870912 assignments. This means long lasting operations that can't be interrupted.

Chapter 6 Analyzer Automation .NET

As of Version 2.0, you can access the Analyzer not only via the OLE Automation facility but also by using the Microsoft .NET Framework. For this purpose, an interface library has been set up that you can use directly without the need to access the COM type library.

Currently, no separate Reference Manual is available for .NET Automation. However, the content of the interfaces for .NET Automation largely correspond to the object classes for COM Automation. This chapter is intended to provide a guideline for developers who wish to use .NET Automation that will allow them to use the existing documentation for COM Automation efficiently for their purposes.

.NET Automation is primarily intended to be accessed from within Analyzer program components (transforms, add-ins). We shall assume that this application scenario applies. We shall not discuss how to create a new project for developing Analyzer program components under .NET, but will instead assume that you wish to make use of Automation from an existing project. We shall also assume that you are using Visual Studio and the C# programming language.

6.1 Overview

.NET Automation follows the programming conventions of C and C# (rather than those of BASIC). The most important consequence of this is that lists and arrays are numbered starting with 0.

In the same way, -1 is used instead of 0 to indicate an invalid list item. If, for example, a marker is to apply to all channels, the value of its `Channel` property is -1.

You can include .NET Automation by adding a reference to the library *AnalyzerAutomation.dll* to your add-in or transform project. All .NET Automation classes are in the namespace `BrainVision.AnalyzerAutomation`. This namespace will not be explicitly shown below.

The interfaces used by Automation correspond to the object classes for OLE Automation, with the names being prefixed by the letter `I`. This means, for example, that `IChannel` corresponds to the object class `Channel`. The object class hierarchy is shown in Figure 1-3 on page 20.

The names of the interface members correspond to the names of their counterparts in the object classes for OLE Automation. You can use the Object Browser in Visual Studio to view the exact definition of properties and methods. Alternatively, you can use the *Go To Definition* function in Visual Studio to view individual interface definitions.

The `IApplication` interface plays a key role in the same way as the `Application` object class. The `AutomationSupport` class returns an instance that represents the application. The sample program below opens the first history file in a workspace:

```
AutomationSupport.Application.HistoryFiles[0].Open()
```

Unlike the approach taken by OLE Automation, instances of `INewHistoryNode` are created by calling one of the overloads of

```
IApplication.CreateNode()
```

The `ActiveNode` variable is used in the integrated BASIC interpreter to create nodes that are capable of being used as templates. The node that is active in the Analyzer main window can be determined with the `IApplication.ActiveNode` property in .NET Automation.

Some of the object classes for OLE Automation define a default object to allow you to access a child collection directly. In .NET Automation, this behavior is implemented in the form of appropriate indexers in the parent object. The two lines of code below are equivalent:

```
IChannel channel = historyNode.Dataset.Channels[0]
```

```
IChannel channel = historyNode.Dataset[0]
```

Some collections (such as `Channels`) permit indexing via the name or title of the objects they contain. This option is also available for the corresponding collection in .NET Automation (such as `IChannels`).

.NET Automation has no equivalent of the `FastArray` object class, because the program code in Analyzer components is executed extremely efficiently and there is therefore no need for fast array operations.

6.2 Subscribing to Automation events

Events in the `IApplication` object and the `IHistoryFiles` object are new features in .NET Automation. You can define event handlers that are called when certain changes occur in the program. Thus, for example, you can define a method that is executed if the user wishes to close a history file.

The events are implemented in accordance with the normal .NET Framework conventions. You should, however, note that the usual argument `sender` for the event handler has not been used, because the events are generated by objects that are unique throughout the entire program.

Some Automation events allow the event handler to prevent pending changes in the program. This allows you, for instance, to prevent the user from closing a history file whose data is still required for calculations that have not yet been completed. In .NET Automation, this functionality has been implemented using the argument of the event handler.

The example below shows an event handler that prevents the first history file of the workspace from being closed:

```
public void Execute()
{
    // Define event handler
    AutomationSupport.Application.HistoryFiles.TestFileIsLockedO-
    pen
          += new TestLockedOpenEventHandler(TestLockedOpen);
}

void TestLockedOpen(ITestLockedOpenArgument e)
{
    if (e.HistoryFile ==
          AutomationSupport.Application.HistoryFiles[0])
    {
          // File remains open.
          e.SetLockedOpen();
    }
}
```

6.3 Using "NewHistoryNode"

The normal procedure for creating new data sets (see Section 1.3 as of page 21) also generally applies to `INewHistoryNode`. However, three overloads of `IApplication.CreateNode()` are now available, so that it is easier to identify the appropriate call for the current application scenario:

- If you wish to create a new history file, use the overload

  `INewHistoryNode.CreateNode(string sFileName)`

- If you wish to create a new child node containing the same data as its parent node, use the overload

  `INewHistoryNode.CreateNode(string sName, IHistoryNode parent, bool bInheritData)`

- If you wish to create a new child node and define the data for this node yourself, you should also use the overload

  `INewHistoryNode.CreateNode(string sName, IHistoryNode parent, bool bInheritData)`

- If you want to create a child node that both inherits the data of some of the channels of its parent node and also modifies or rearranges channels, use the overload

  `INewHistoryNode.CreateNode(string sName, IHistoryNode parent, int[] channelMap)`

Immediately after you have called `CreateNode`, you should use the `INewHistoryNode.SetDimensions()` method to make further basic specifications for the data set. If you wish to create a data set with multiple frequency levels, you can also use `INewHistoryNode.SetLayerInformation()` to make specifications regarding the frequency levels.

If you wish to change the markers of the new data set, you can use the `INewHistoryNode.Markers` collection. You can change the markers contained in the collection directly by assigning new values to their properties. You can delete markers from the collection or add new markers. If you wish to move markers to a new position in the data set, you do not have to take account of the sequence of the markers. The markers in the collection are automatically sorted by their positions when you call `INewHistoryNode.Finish()`.

The `ActiveNode` variable is used in the integrated BASIC interpreter to create nodes that are capable of being used as templates. You can determine the node that is active in the Analyzer main window in .NET Automation using the `IApplication.ActiveNode` property.

When you create the node, you can specify the view that is to be used by default when it is opened. To do this, assign the unique identifier of the view to the `NewHistoryNode.RequestedView` property. You can take the identifier of the view from its XML definition (`id` attribute of the `<View>` tag, the value can, for example, be "EE10458E-8BA8-4276-B469-E15E785264C2" as in the file *StandardView.xml*).

In contrast to the behavior with OLE Automation in the integrated BASIC interpreter, execution of an add-in is not automatically terminated when you call the `INewHistoryNode.Finish()` or `ITransformation.Do()` methods.

The example below works in the same way as the example in Section 1.3.2 on page 23:

```
public void Execute()

{

    // Create the object and define basic properties.

    IApplication application = AutomationSupport.Application;

    INewHistoryNode newNode =
```

```
     application.CreateNode("Automation.NET",
     application.ActiveNode, false);
newNode.SetDimensions(250, 32, 1);
newNode.Datatype = VisionDataType.TimeDomain;
newNode.DataUnit = VisionDataUnit.Microvolt;

// Define channel properties. All other channel properties
// retain the properties inherited from the parent node as
// defaults.
newNode.SetChannelName(0, "Channel B");
newNode.SetChannelName(1, "Channel A");
newNode.SetChannelPosition(3, 1, 0, 90);

// Set an interval marker.
newNode.AddMarker(-1, 200, 20, "Bad Interval", "");

// Specify data: Read 250 points from 3rd channel of the
// parent node and write them to the 1st channel of the new
// node. All other data points retain the
// default value 0.0.
float[] data = application.ActiveNode.Dataset.GetData(0, 250,
     new int[] { 2 });
newNode.WriteChannelData(0, 0, data);

// Write a sample text for "Operation Infos"
newNode.Description = "Test for .NET automation.";

// Complete the note.
// Insert the GUID that identifies your add-in here.
```

```
    newNode.Finish(new Guid(AddInGuid));
}
```

6.4 Additional extensions

.NET Automation includes a number of minor extensions that provide new functions.

The `IApplication` interface contains an extended list of functions for displaying messages to the user: `AskYesNo`, `AskOKCancel`, `Message`, `Warning` and `Error`. If the add-in is run inside a history template, and messages are only output to a log, execution is not interrupted by the functions.

The `FindNode` and `FindNextNode` functions of the object class `HistoryFile` have been replaced by the `IHistoryFile.FindNodes()` function. This function returns all the nodes with matching names in an array.

With OLE Automation, access to the properties of markers, channels and data sets is achieved on the basis of their property names using the methods or properties `PropertyName`, `PropertyValue` and `PropertyCount`. With .NET Automation, this is achieved by the methods `EnumerateProperties` and `GetUserProperty`.

It is now possible to display a data set using a precisely specified EEG view. To do this, pass the unique identifier of the view to the `IHistoryNode.ShowView` function. You can get the identifier of the view from its XML definition (`id` attribute of the `<View>` tag). The following call opens a history node using the grid view:

```
historyNode.ShowView(new Guid(
   "D654817E-4429-4d9b-AF23-6F09F5A471B5"))
```

●

Glossary

A

Add-in: Add-ins are Analyzer program components that offer additional functions. Add-ins can also be created by users themselves and are freely programmable. While, for example, they can act as transforms or export components, they internally use a simplified program mechanism.

C

Child node: In the history tree, this refers to the EEG data sets that are subordinate to the current node and represent the following processing steps.

D

Dongle: Pluggable copy protection device.

E

EEG view: Method of representing the EEG, such as the grid view, the head view, and the mapping view. A view determines how the channels are arranged in the window, for example.

Export component: Analyzer program element which writes the content of the current data set to a file so that this can be used outside of the Analyzer.

H

History Explorer: An element in the Analyzer user interface which allows users to edit raw data nodes and created history nodes.

History file: The file on your computer in which the processing steps applied to an EEG file are stored. Also refers to the EEG file displayed in the History Explorer.

History node: Representation of a processing step applied to an EEG file in the Analyzer user interface.

History template: File in which processing steps from the history tree are stored. The processing steps can be executed again automatically elsewhere.

History tree: The processing steps applied to the EEG and displayed in the form of a tree.

L

License: Allows the user to work with one of the optional program components of the Analyzer.

N

Network dongle: Pluggable copy protection device for operating the Analyzer on multiple workstations in a network environment.

O

Operation Infos: The descriptive text that summarizes the settings used for the execution of a processing step. The Operation Infos are saved automatically and can be viewed again later.

P

Parent node: In the history tree, the uniquely defined EEG data set directly above the current node.

Primary history file: Primary history files are history files that are based on the EEG raw data, in contrast to secondary history files.

Primary transform: Primary transforms are processing steps which are applied to an existing data set in a history file. This leads to the creation of a new data set below the original data set.

Program component: Analyzer program element that is located outside of the actual program file and is dynamically loaded. By adding new components it is possible to expand the Analyzer's functionality.

R

Raw data node: The top-level EEG data set in a history file. This contains the unmodified EEG data read in from the raw file.

Raw file: The EEG file obtained directly during recording without any modifications.

S

Secondary history file: Secondary history files are history files that are based on data compiled as the results of processing steps from multiple history files.

T

Transform: Transforms are Analyzer program components that process input data and then output data either in the form of a new EEG data set or directly for display.

Transient transform: Transient transforms are processing steps which are only used for visualization purposes. The data output from a transient transform does not generate a new data set but is instead displayed directly.

W

Workfile: A file containing information on workspaces (*.wksp2), montages (*.mont2) and other user-defined settings.

Workspace: Configuration file which contains storage locations for raw files, history files and exported data. Extension: .wksp2.