

# VbScriptXtra

## Version 2.6

[www.xtramania.com](http://www.xtramania.com)

All trademarked names mentioned in this document and product are used for editorial purposes only, with no intention of infringing upon the trademarks.

---

<b>VbScriptXtra .....</b>	<b>1</b>
<b>About VbScriptXtra .....</b>	<b>1</b>
<b>About ActiveCompanionSet .....</b>	<b>1</b>
<b>VbScriptXtra features .....</b>	<b>1</b>
<b>'ActiveCompanionSet' Xtras License Agreement .....</b>	<b>2</b>
<b>VbScriptXtra Programmer's Guide.....</b>	<b>6</b>
<b>How to Use VbScriptXtra .....</b>	<b>6</b>
Typecasting.....	6
ProgId .....	7
Creating Object.....	7
Object Description.....	8
<b>Debugging and Errors Handling.....</b>	<b>8</b>
Lingo Errors.....	8
Programming Errors .....	8
Simple Debugging Mode.....	9
Advanced Debugging Mode.....	10
Using Put Command.....	10
Using Debugger and Object Inspector .....	11
<b>Samples .....</b>	<b>12</b>
<b>ADO Databasing.....</b>	<b>12</b>
Creating Recordset Object.....	12
Choosing which Database to Open.....	12
Opening Recordset Object.....	13
Getting Data from Database via Recordset .....	13
Modifying Data via Recordset.....	14
Closing Recordset.....	14
Connection Object .....	14
Object's Dynamic Properties .....	15
Using Transactions .....	15
<b>Save and Compact Microsoft Access Database .....</b>	<b>16</b>
<b>Automating Microsoft PowerPoint .....</b>	<b>17</b>
<b>Automating Microsoft Word .....</b>	<b>19</b>
<b>Automating Microsoft Excel.....</b>	<b>21</b>
<b>WMI Scripting .....</b>	<b>23</b>
<b>VbScriptXtra Programmer's Reference .....</b>	<b>24</b>
<b>Wrapping Objects.....</b>	<b>24</b>
<b>Common Features of Wrapping Objects .....</b>	<b>26</b>
Error Handling Support .....	26
<i>Succeeded</i> .....	27
<i>Failed</i> .....	27

---

---

<i>LastErrorCode</i> .....	28
<i>LastError</i> .....	28
Debugging Support.....	29
<i>DebugMode</i> .....	30
Type Casting Routines.....	31
<i>COM Automation to Lingo</i> .....	31
<i>Lingo to COM Automation</i> .....	32
Unicode Conversion Support.....	33
<i>CodePage</i> .....	34
<i>UnicodeAsBinaryMode</i> .....	36
<b>Automation Object Wrapper.....</b>	<b>37</b>
Methods.....	37
<i>Interface()</i> .....	37
<i>GetEnum( symName )</i> .....	38
<i>Calling Other Methods</i> .....	39
Properties.....	39
<i>InterfaceName</i> .....	39
<i>TypeLib</i> .....	40
<i>EventsHandler</i> .....	40
<i>__NewEnum</i> .....	42
<i>Getting Other Properties</i> .....	43
<i>Setting Other Properties</i> .....	44
Technical Details.....	45
<i>Underscore Handling</i> .....	45
<i>Named Method's Arguments</i> .....	45
<i>Passing Parameters by Reference</i> .....	46
<i>Optional and Missing Method's Arguments</i> .....	46
<i>Using Wrapper Instance as First Argument of a Method</i> .....	46
<i>Cascading Methods and Properties in Director 7</i> .....	47
<i>Using Square Brackets</i> .....	47
<b>Type Library Wrapper.....</b>	<b>48</b>
Methods.....	48
<i>Interface()</i> .....	48
<i>Load( strPath )</i> .....	49
Properties.....	49
<i>Name</i> .....	49
<i>GUID</i> .....	49
<i>Getting Other Properties</i> .....	50
<b>Typed Instance Wrapper.....</b>	<b>52</b>
Methods.....	52
<i>Interface()</i> .....	52
<i>New()</i> .....	52
<i>Calling Other Methods</i> .....	53
Properties.....	54
<i>InterfaceName</i> .....	54
<i>Ptr</i> .....	54
<i>Getting Other Properties</i> .....	55
<i>Setting Other Properties</i> .....	55

<b>Binary Data Wrapper .....</b>	<b>57</b>
Methods .....	58
<i>Interface()</i> .....	58
<i>Clear()</i> .....	58
<i>Allocate( nSize )</i> .....	58
<i>Resize( nSize )</i> .....	59
<i>GetAt( nIndex )</i> .....	59
<i>SetAt( nIndex, nValue )</i> .....	59
<i>ReadFromFile( strPath, nOffset, nBytesToRead )</i> .....	60
<i>WriteToFile( strPath )</i> .....	61
<i>AppendToFile( strPath, nOffset, bSetEndOfFile )</i> .....	61
<i>MakeImageFromRGB( nWidth, nHeight, bBitsPerPixel )</i> .....	62
<i>GetRGBFromImage( objImage )</i> .....	63
Properties .....	63
<i>Count</i> .....	63
<i>Size</i> .....	64
<i>ContentsType</i> .....	64
<i>UnsignedByte[ nIndex ]</i> .....	65
<i>SignedByte[ nIndex ]</i> .....	65
<i>Byte[ nStartIndex .. nEndIndex ]</i> .....	66
<i>Char[ nIndex ]</i> .....	66
<i>Char[ nStartIndex .. nEndIndex ]</i> .....	67
<i>WideChar[ nIndex ]</i> .....	67
<i>WideChar[ nStartIndex .. nEndIndex ]</i> .....	68
<i>Media</i> .....	68
<i>Picture</i> .....	69
<i>String</i> .....	69
<i>HexString</i> .....	70
<i>Value</i> .....	71
<b>Image Wrapper .....</b>	<b>72</b>
Methods .....	72
<i>Interface()</i> .....	72
<i>InitFromHBitmap( BitmapHandle, bRelease )</i> .....	72
Properties .....	73
<i>Image</i> .....	73
<i>Width</i> .....	73
<i>Height</i> .....	74
<i>Depth</i> .....	74
<i>AlphaDepth</i> .....	74
<i>StdOlePicture</i> .....	74
<b>Date/Time Data Wrapper .....</b>	<b>76</b>
Methods .....	76
<i>Interface()</i> .....	76
<i>FormatDate( strFormat )</i> .....	76
<i>FormatTime( strFormat )</i> .....	77
<i>MonthName( nMonth, bAbbreviated )</i> .....	78
<i>WeekdayName( nDay, bAbbreviated, nFirstDayOfWeek )</i> .....	79
Properties .....	79
<i>Value</i> .....	79
<i>Year</i> .....	80

<i>Month</i> .....	80
<i>MonthName</i> .....	81
<i>Weekday</i> .....	81
<i>WeekdayName</i> .....	81
<i>Day</i> .....	81
<i>Minute</i> .....	81
<i>Second</i> .....	82
<i>Millisecond</i> .....	82
<i>Local</i> .....	82
<i>Universal</i> .....	83
<b>Registry Key Wrapper</b> .....	<b>84</b>
Methods .....	84
<i>Interface()</i> .....	84
<i>Open( strParent, strName, symAccessType, bCreate )</i> .....	84
<i>OpenSubKey( strName, symAccessType, bCreate )</i> .....	86
<i>CreateSubKey( strSubKeyName )</i> .....	87
<i>DeleteSubKey( strSubKeyName )</i> .....	88
<i>DeleteValue( strValueName )</i> .....	88
<i>GetAt( Index )</i> .....	89
<i>SetAt( Index, Value, symType )</i> .....	90
Properties .....	91
<i>Count</i> .....	91
<i>Value</i> .....	91
<i>Value[ index ]</i> .....	91
<i>ValueType[ index ]</i> .....	92
<i>ValueNames</i> .....	93
<i>SubKeyNames</i> .....	93
<b>System Helper Object</b> .....	<b>94</b>
Methods .....	94
<i>Interface()</i> .....	94
<i>GetRunningObjectTable()</i> .....	94
<i>LoadKeyboardLayout( strLayoutName )</i> .....	96
<i>UnloadKeyboardLayout( hkl )</i> .....	96
<i>ActivateKeyboardLayout( hkl )</i> .....	97
Properties .....	97
<i>ProcessId</i> .....	97
<i>StageHWND</i> .....	98
<i>AppHWND</i> .....	98
<i>KeyboardLayout</i> .....	98
<i>KeyboardLayoutList</i> .....	99
<i>DefaultLocaleInfo</i> .....	99
<b>Xtra-level methods</b> .....	<b>100</b>
<i>Init( nDebug )</i> .....	100
<i>CreateObject( strProgId )</i> .....	101
<i>GetObject( strProgId )</i> .....	102
<i>GetObject2( strPath, strProgId )</i> .....	103
<i>Version()</i> .....	104
<i>SetBusyHandler( objHandler )</i> .....	104
<i>CreateWrapper( symWrapperType )</i> .....	106

# VbScriptXtra

## About VbScriptXtra

VbScriptXtra extends the Macromedia Director's Lingo functionality with capability to handle VB-scriptable objects. These are objects or external applications that support COM Automation technology.

Software, which supports Automation (and therefore is supported by VbScriptXtra) includes: Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Microsoft Access, Microsoft Internet Explorer, Microsoft Visual Source Safe, some system components including common open/save dialogs, system shell, Windows Scripting Host, data access components: ADO, ADOX, ADOMD, DAO; Adobe Photoshop, Adobe ImageReady, Microsoft NetMeeting, Collaboration Data Objects (CDO), Windows Management Instrumentation (WMI) etc.

VbScriptXtra is available for Macromedia Director (v7 and later) under Windows 98/ME/NT/2000/XP/Vista.

VbScriptXtra is not available for Shockwave.

**Note:** All trademarked names mentioned in this document and product are used for editorial purposes only, with no intention of infringing upon the trademarks.

## About ActiveCompanionSet

VbScriptXtra is shipped within ActiveCompanionSet. It is a bundle of xtras that provide COM, DCOM, OLE and ActiveX support for Macromedia Director. The set currently includes VbScriptXtra, OLE xtra, ActiveX xtra and ObjectBrowserXtra.

## VbScriptXtra features

VbScriptXtra integrates several types of wrapper objects to provide more flexible handling of a data of different nature that can be used by Automation objects.

VbScriptXtra can script almost any COM/DCOM Automation compliant object including IUnknown based objects.

VbScriptXtra's [Binary](#) data wrapper is used to handle [Unicode](#) text, BLOB or other binary data. It can also be used as a simple array of bytes or Unicode wide chars with possibility to read/write data from files. Binary wrapper of VbScriptXtra is free. You can freely use it in your projects. See [License Agreement](#) for more details.

VbScriptXtra's [Image wrapper](#) is used as a helper to exchange image data between Director and COM applications. It handles Lingo image translation to and from Bitmap handles (HBITMAP) and `StdOle.Picture` objects. Image wrapper of VbScriptXtra is free. You can freely use it in your projects. See [License Agreement](#) for more details.

VbScriptXtra's [System](#) wrapper provides further support for multilingual applications providing full control over current [keyboard layout](#). It allows switching layouts from one language to another and [load](#) specific keyboard layouts as necessary.

VbScriptXtra's [Date/time](#) wrapper is used to handle date/time data. It offers quite powerful formatting capabilities for date/time values. Date/time wrapper of VbScriptXtra is free. You can freely use it in your projects. See [License Agreement](#) for more details.

VbScriptXtra's [Registry key](#) wrapper is used to handle operations with system Registry. Registry key wrapper of VbScriptXtra is free. You can freely use it in your projects. See [License Agreement](#) for more details.

`GetObject` syntax is fully supported for accessing currently running objects or applications. See [GetObject](#) and [GetObject2](#) methods.

The system-level [Running Object Table](#) is supported providing scripting control of all running Automation objects registered in ROT.

VbScriptXtra's [type casting](#) routines provide native data types conversion from Automation types to Lingo and vice versa. Lingo symbols passed to VbScriptXtra type casting routines are translated as named constants from the loaded type libraries. Use [wrapper.GetEnum\(\)](#) method to get the value of the specified named constant if you need it in expression.

VbScriptXtra is shipped with `ObjectBrowserXtra`. You can invoke it with [wrapper.Interface\(\)](#) method to view the description of methods and properties provided by the wrapped object. `ObjectBrowserXtra` is free. See [License Agreement](#) for more details.

VbScriptXtra wrapper objects fully support Macromedia Director's [debugger](#) and object inspector. You can now expand object instance to view values of its properties.

## 'ActiveCompanionSet' Xtras License Agreement

This user license agreement (the AGREEMENT) is an agreement between you (individual or single entity) and MediaMacros, Inc. and Eugene Shoustrov for the included 'ActiveCompanionSet' XTRAS (the SOFTWARE) that are accompanying this AGREEMENT.

The SOFTWARE is the property of Eugene Shoustrov and is protected by copyright laws and international copyright treaties. The SOFTWARE is not sold, it is licensed.

If you accept the terms and conditions of this AGREEMENT, then you are granted the FREE LICENSE.

### I. FREE LICENSE

The FREE LICENSE allows using any functionality of the SOFTWARE except Automation and Typed Instance wrappers of VbScriptXtra.

The FREE LICENSE allows using Automation and Typed Instance wrappers of VbScriptXtra with evaluation purposes only.

By accepting the FREE LICENSE you have certain rights and obligations as follow:

YOU MAY:

Install and use the SOFTWARE (as LICENSE permits) on any computer within your company or home.

Make a copy of the SOFTWARE for archival purposes.

Distribute an unlimited number of copies of the SOFTWARE with your final runtimes provided that the original package contents stay unchanged including this EULA.

**YOU MAY NOT:**

Sublicense, rent or lease your license

Decompile, disassemble, reverse engineer or modify the SOFTWARE or any portion of it, or make any attempt to bypass, unlock, or disable any protective or initialization system on the SOFTWARE.

Copy the documentation accompanying the SOFTWARE for use in other software.

## **II. LIMITED LICENSE**

The LIMITED LICENSED VERSION means a Registered Version (using your personal registration number). The LIMITED LICENSE defines a certain set of ProgIds that are allowed to be used with the SOFTWARE.

The LIMITED LICENSE allows using any functionality of the SOFTWARE except for Automation and Typed Instance wrappers of VbScriptXtra for objects with ProgIds not covered by the LIMITED LICENSE.

The LIMITED LICENSE allows using Automation and Typed Instance wrappers of VbScriptXtra for objects with ProgIds covered by the LIMITED LICENSE.

The LIMITED LICENSE allows using Automation and Typed Instance wrappers of VbScriptXtra for objects with ProgIds not covered by the LIMITED LICENSE with evaluation purposes only.

If you accept the terms and conditions of this AGREEMENT, you have certain rights and obligations as follow:

**YOU MAY:**

Install and use the Registered SOFTWARE on any single computer.

Make a copy of the Registered SOFTWARE for archival purposes only.

Distribute an unlimited number of copies of the Xtra with your final runtimes provided that the source code is protected and your serial number is not accessible to any 3rd party.

**YOU MAY NOT:**

Copy and distribute the SOFTWARE with an accessible serial number.

Sublicense, rent or lease your license

Decompile, disassemble, reverse engineer or modify the SOFTWARE or any portion of it, or make any attempt to bypass, unlock, or disable any protective or initialization system on the SOFTWARE.

Copy the documentation accompanying the SOFTWARE for use in other software.

## **III. UNLIMITED/UNLIMITED PLUS LICENSE**

The UNLIMITED LICENSED VERSION means a Registered Version (using your personal special registration number).

The UNLIMITED LICENSE does not imply any restrictions on ProgIds being used with the SOFTWARE. This license allows using any functionality of the SOFTWARE except for Typed Instance wrappers of VbScriptXtra.

The UNLIMITED PLUS LICENSE does not imply any restrictions on ProgIds being used with the SOFTWARE. This license allows using any functionality of the SOFTWARE including Typed Instance wrappers of VbScriptXtra.

If you accept the terms and conditions of this AGREEMENT, you have certain rights and obligations as follow:

**YOU MAY:**

Install and use the Registered SOFTWARE on any single computer.

Make a copy of the Registered SOFTWARE for archival purposes only.

Distribute an unlimited number of copies of the Xtra with your final runtimes provided that the source code is protected and your serial number is not accessible to any 3rd party.

**YOU MAY NOT:**

Copy and distribute the SOFTWARE with an accessible serial number.

Sublicense, rent or lease your license

Decompile, disassemble, reverse engineer or modify the SOFTWARE or any portion of it, or make any attempt to bypass, unlock, or disable any protective or initialization system on the SOFTWARE.

Copy the documentation accompanying the SOFTWARE for use in other software.

**WARRANTY DISCLAIMER**

The SOFTWARE is supplied "AS IS". MediaMacros, Inc. and Eugene Shouistrov disclaim all warranties, expressed or implied, including, without limitation, the warranties of merchantability and of fitness for any purpose. The user must assume the entire risk of using this SOFTWARE.

**DISCLAIMER OF DAMAGES**

MediaMacros, Inc. and Eugene Shouistrov assume no liability for damages, direct or consequential, which may result from the use of this SOFTWARE, even if MediaMacros, Inc. and/or Eugene Shouistrov have been advised of the possibility of such damages.

**TERM**

This license is effective from the date of obtaining or purchasing the SOFTWARE and shall remain in force until terminated. You may terminate the license and this agreement at any time by destroying the SOFTWARE and its documentation, together with all copies in any form that reside on your computer or media.

**COPYRIGHT NOTICE:**

The Company and/or our Licensors hold valid copyright in the Software. Nothing in this Agreement constitutes a waiver of any rights under U.S. Copyright law or any other federal or state law.

**ACKNOWLEDGMENT:**

BY USING THIS SOFTWARE YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERCEDES ALL PROPOSALS OR PRIOR

ENDORSEMENTS, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

All trademarked names mentioned in this document and product are used for editorial purposes only, with no intention of infringing upon the trademarks.

# VbScriptXtra Programmer's Guide

## How to Use VbScriptXtra

VbScriptXtra allows using COM Automation objects right from Lingo. COM Automation technology is used by many applications to expose their functionality to external applications or to macro programming. Visual Basic, VbScript, JavaScript, C/C++ and others are languages that can be used for programming Automation objects. VbScriptXtra extends Lingo with possibility of programming Automation objects.

Any Automation object has a certain set of methods and properties. These methods and properties might accept or return either plain data types like numbers, strings, etc. or other Automation objects. Simple applications might expose their functionality via one single Automation object. More complicated applications might contain the whole internal data model with a large hierarchy of objects.

VbScriptXtra can handle Automation objects by wrapping them with special wrapper objects provided by the xtra. In Lingo these wrapper objects are referred via usual Lingo variables of type 'instance'. Methods and properties exposed by Automation object are automatically available at Lingo level with the wrapper object. So, any method or property called with wrapper object is passed to the wrapped Automation object. Returned value is passed back to Lingo level. The method's arguments as well as returned value all can be another wrapper objects.

## Typecasting

VbScriptXtra performs necessary typecasting operations required to pass Lingo values to Automation object and vice versa. Plain data types are mapped to appropriate Lingo types. If Automation objects use another Automation object as a parameter, VbScriptXtra can accept its own wrappers to extract the wrapped data and use it as an actual parameter. If Automation object returns another Automation object, VbScriptXtra automatically creates another wrapper object for the returning value. In this way cascading properties access is working.

Some Automation types cannot be mapped to Lingo directly. So typecasting operation is not always possible. For some types VbScriptXtra provides special wrappers. Binary data (BLOB) is wrapped with special [Binary](#) wrapper. It is a kind of array of bytes that could be handled by Lingo. Binary wrapper allows data to be written to file or used as media of Director cast members etc. Date/time data is wrapped by special [date/time](#) wrapper. This wrapper provides standard for VB functionality for formatting date/time values and other features.

COM Automation normally assumes all text to be in Unicode while Director handles text as MBCS according to the current Windows default locale for non-Unicode applications. VbScriptXtra automatically performs Unicode-MBCS text conversion by using either default system code page or specific code page that is set for the wrapper object. Starting from version 2.3 VbScriptXtra supports Unicode text as binary data. In this mode the xtra returns Unicode text as [Binary](#) objects allowing direct manipulation with Unicode text. See [Unicode conversion support](#) for more details.

## ProgId

Creatable Automation objects are identified by their ProgIds. ProgId is normally a string that consists of several dot separated items. The first item normally means the application name. The second item normally identifies an object type within Application. And the optional third item identifies the required version of object. If the third item is skipped then the currently installed version of application is used. For example the ProgId of Microsoft Word 2003 is "Word.Application.11". The ProgId of a document of Microsoft Word 2003 is "Word.Document.11". If the third item is missed "Word.Application" then this ProgId identifies the currently installed version of Microsoft Word.

## Creating Object

To create an instance of the Automation object use one of xtra-level methods [CreateObject\( strProgId \)](#), [GetObject\( strProgId \)](#) or [GetObject2\( strPath, strProgId \)](#).

CreateObject method creates an Automation object of the requested ProgId. If successful, it returns VbScriptXtra wrapper for newly created Automation object. Otherwise it returns a string with error description.

```
vb = xtra( "VbScriptXtra" )
w = vb.CreateObject( "Word.Application" )
w.visible = true
```

GetObject method looks for currently running object of the specified ProgId. If it finds one then it wraps it with the VbScriptXtra wrapper. Otherwise it returns string with error description.

```
vb = xtra( "VbScriptXtra" )

-- Creating a new instance of Microsoft Word
w = vb.GetObject( "Word.Application" )

if not objectP(w) then
  w = vb.CreateObject( "Word.Application" )
end if
```

GetObject2 method is used for creating Automation object either from file or from user-friendly specially formatted string. In the first case VbScriptXtra creates an instance of the requested Automation object and then makes it to load the specified file.

```
vb = xtra( "VbScriptXtra" )

-- Getting a Word document from file
doc = vb.GetObject( "D:\file.doc", "Word.Document" )

doc.Application.visible = true
```

The second case is used to get Automation objects by specially formatted string describing which object is wanted. It is often used in WMI scripting for example.

```
vb = xtra( "VbScriptXtra" )

objDisk = vb.GetObject2("WinMgmts:win32_LogicalDisk.DeviceId='C:', """)
props = ObjSet.Properties_.__NewEnum

repeat with i = 1 to props.Count
  put props[i].Name & ":" && props[i].Value
end repeat
```

You can also access the Running Object Table (ROT) that is used by the system to keep track of running objects. See [GetRunningObjectTable](#) for more details and sample.

## Object Description

Automation objects usually provide a type library that defines methods and properties provided by the object. This information is useful for discovering what you can do with a particular object. VbScriptXtra can invoke ObjectBrowser window that shows the description of the wrapped Automation object. Use [Interface\(\)](#) method of any Automation wrapper object. Make sure ObjectBrowser.x32 is placed in Xtras folder of your Director installation.

```
vb = xtra( "VbScriptXtra" )
w = vb.CreateObject( "Word.Application" )
w.visible = true

-- Shows description of Documents collection of Word application
w.Documents.interface()
```

Another useful source of information about how to do anything with something is the documentation of the application. All Microsoft Office applications have complete documentation about programming them. This documentation describes everything in VB syntax, but it can be simply translated to Lingo syntax.

Recording a macro in Office applications allows you to see VB representation of certain operations. The same code could be written with Lingo and VbScriptXtra.

## Debugging and Errors Handling

There are two main levels of errors related to VbScriptXtra. They have completely different nature and therefore have to be handled differently.

### Lingo Errors

Lingo errors are similar to incorrect Lingo syntax run-time errors. They cause Director to show error alert saying something like "Method or property not found in object" or "One parameter expected". In Projector they might halt script execution etc. These errors usually mean that something is wrong with the programming. Wrong method call syntax is used or something similar to it. VbScriptXtra might return error codes to Director that make Director to show Lingo error alert box. It happens when wrapper object discovers the programming error at the Lingo level (wrong syntax, wrong parameters and other compile time evident programming errors).

### Programming Errors

This level includes errors that are actually exception conditions. They happen or do not happen depending on particular execution context. They are normal in programming practice and have to be handled programmatically. For example if file operation fails it does not have to worry end-user with Lingo error alert box. Instead developer should check whether operation completed successfully and perform what is appropriate.

VbScriptXtra provides programming errors handling support based on storing status of the last call within every wrapper object. In other words, every VbScriptXtra's wrapper object keeps the error code and description returned by the most recently called method or property. Before returning from the call to any wrapper object the last error information (if

any) is being set by the wrapper object. Right before calling the next method or property of the wrapper object the last error information is cleared.

To check the status of the most recent call to the object use [obj.Failed](#) or [obj.Succeeded](#) properties. The error message and error code are available via [obj.LastError](#) and [obj.LastErrorCode](#) properties.

If Lingo statement includes cascading property access, several wrapper objects might be involved. Most of these wrappers (except the first one) are temporarily and therefore they are not accessible after the Lingo statement. So the error information could be lost. Sometimes it is worth to store intermediate wrappers in a Lingo variable just to have an opportunity to check whether a call was successful.

This sample shows how to check error status when multiple wrappers are involved in cascading property access operation.

```
on OpenWordDocument strPath
    vb = xtra("VbScriptXtra")
    w = vb.CreateObject( "Word.Application" )

    docs = w.Documents
    doc = docs.Open( strPath )

    if docs.Succeeded then
        return doc
    end if

    alert doc.LastError
    return VOID
end
```

Compare the above sample to the following one.

```
on OpenWordDocument strPath
    vb = xtra("VbScriptXtra")
    w = vb.CreateObject( "Word.Application" )

    doc = w.Documents.Open( strPath )

    -- Incorrect check since "w.Documents" always works
    -- while Open( strPath ) might fail
    if w.Succeeded then
        return doc
    end if

    alert doc.LastError
    return VOID
end
```

## Simple Debugging Mode

Since errors are happening VbScriptXtra provides debugging modes to simplify debugging process.

In simple debugging mode any wrapper object puts error information into Messages window whenever error occurred. Usually simple debugging mode is useful to detect whether script is executed well or there is a problem somewhere. Error messages usually come from wrapped objects but there is no information about the context where error occurred.

To set the simple debugging mode for the xtra use:

```

on prepareMovie
  if the playerMode = "author" then
    xtra("VbScriptXtra").Init( 1 )
  end if
end

```

## Advanced Debugging Mode

Advanced debugging mode allows you to catch error right in Debugger whenever error occurred. In this mode VbScriptXtra tries to call movie-level handler `VbScriptXtra_DebugEvent( strMes, nCode )`. If there is no such handler, the xtra behaves as in simple debugging mode. This handler may contain any Lingo statements. Furthermore, you can place a break point inside this handler and use Director's debugging capabilities to view the calling context, variables etc.

Sample movie-level handler for advanced debugging.

```

on prepareMovie
  if the playerMode = "author" then
    xtra("VbScriptXtra").Init( 2 )
  end if
end

on VbScriptXtra_DebugEvent strMes, nCode
  put strMes -- Place the break point here
end

```

Debugging mode is kept separately for every VbScriptXtra wrapper object. Use [DebugMode](#) property to change the debugging mode of the particular object directly. Otherwise use xtra-level [Init\( nDebug \)](#) method to set the default debugging mode for newly created wrappers. This method does not affect objects that already exist at the time of calling this method.

## Using Put Command

Every wrapper object provides descriptive information about itself via `put` Lingo method. To see what the wrapper object contains simply `put` it in Messages window.

```

Vb = xtra("VbScriptXtra")

objDate = vb.CreateWrapper( #Date )

put objDate
--"< VbScriptXtra, Date/Time, 09/03/2004 20:22:44 >"

objWord = vb.CreateObject("Word.Application")

put objWord
--"< VbScriptXtra, Word._Application, 0x001FB29C, (1) >"

objBinary = vb.CreateWrapper( #Binary )

objBinary.String = "Test"
put objBinary
--"< VbScriptXtra, Binary, Size: 4 byte(s) >"

objBinary.ContentsType = #UnicodeString
objBinary.String = "Test"
put objBinary
--"< VbScriptXtra, UnicodeString, Size: 8 byte(s) >"

```

## Using Debugger and Object Inspector

VbScriptXtra wrappers support viewing their contents via Director Debugger and Object Inspector.

Automation wrapper allows expanding its entry in Debugger to view properties of the wrapped Automation object. It is quite convenient although it has side effect that conflicts with debugging modes. When wrapper's entry in debugger is expanded Director internally calls all properties available to view in debugger. Wrapper object cannot distinguish whether it is called by debugger or by Lingo script. Therefore last error information kept by the wrapper object is erased with the status of the last method or property that was called by Director but not Lingo script. In advanced debugging mode the `VbScriptXtra_DebugEvent` movie level handler could be called while Director asks object for its property values. So take care with that.

## Samples

### ADO Databasing

ActiveX Data Objects (ADO) provides a universal programming way of handling databases. VbScriptXtra allows using ADO within Lingo.

ADO documentation is probably already available at your Windows\Help folder. See ADO210.CHM file. Otherwise there is [MSDN](#).

### Creating Recordset Object

Use xtra-level method `CreateObject( strProgId )` to create wrapper for ADODB.Recordset object:

```
Vb = xtra("VbScriptXtra")
rst = vb.CreateObject( "ADODB.Recordset" )
```

Check resulting value to ensure that ADO is available. If function succeeded `rst` will be the Lingo object reference, otherwise it will be a string, describing error:

```
if objectP(rst) then
  put "Recordset created"
else
  put "Error:" && rst
end if
```

### Choosing which Database to Open

ADO usually uses a connection string to specify to which database to connect or which database to open. Connection string is usually the string in a form "PropertyName=PropertyValue;OtherPropertyName=OtherValue". Here are several samples, how the connection string may look like:

#### *MS Access databases*

```
strCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=D:\Temp\DB.mdb;
Mode=ReadWrite"
```

#### *MS Access databases (password protected)*

```
strCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=D:\Temp\DB.mdb;
Mode=ReadWrite; Jet OLEDB:Database Password=PasswordHere"
```

#### *MS Access databases via ODBC driver (DSN-less connection):*

```
strCnn = "DRIVER={Microsoft Access Driver (*.mdb)}; DBQ=D:\Temp\DB.mdb"
```

#### *MS SQL Server:*

```
strCnn = "Provider=SQLOLEDB.1; Integrated Security=SSPI;Persist
Security Info=False;Initial Catalog=DemoDB;Data Source=SqlServerName"
```

#### *Oracle databases:*

```
strCnn = "Provider=MSDAORA.1; Password=psw; User ID=admin; Data
Source=srv; Persist Security Info=True"
```

The most important property in connection string is "Provider". Its value usually determines the type of database to work with. Other properties specify additional

information that may be specific to the provider. Note that if you omit the provider property, the default will be used. Default provider for ADO is OLE DB Provider for ODBC.

Note that connection string may specify the type of access to data. In the first example "Mode=ReadWrite" specifies that connection to database is for reading and writing. All or almost all information specified in connection string may be adjusted directly by setting properties of the connection object. But in simple scenario you do not use Connection object directly, although ADO will create it implicitly during processing of the recordset's Open method. So, in simple scenario connection string is the only source of information about which database to open.

## Opening Recordset Object

To get actual database data with ADO you have to open a recordset with specified command text over specified connection. The command text may be a SQL query or command, a table name, a stored procedure name, or other provider specific command.

To open recordset you may call the recordset's Open method:

```
strCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=D:\Temp\DB.mdb;
Mode=ReadWrite;"

strSQL = "SELECT SomeFieldName, SomeOtherFieldName FROM SomeTable ORDER
BY SomeFieldName"

rst.Open( strSQL, strCnn )

if rst.succeeded then
    put "Recordset state:"&&rst.State
else
    put "Error:"&&rst.lastError
end if
```

Be sure to always check whether call was successful if you do not use VbScriptXtra's debugging mode, since ADO often (but not always) returns useful error descriptions, if you do something incorrectly. After Open call succeeded, check the state property of the recordset. Usually if source text specifies row-returning query (like SELECT), the `rst.state` property will be set to `adStateOpen (=1)`. If source text specifies command query (like INSERT), the state of recordset object will be set to `adStateClosed (=0)`.

## Getting Data from Database via Recordset

The recordset object with `rst.state = adStateOpen` is ready to provide access to the data. Recordset provides access to the data in record by record manner. So at any given moment you can only access the current record. Move the current record of a recordset with `rst.MoveNext()`, `rst.MovePrevious()`, `rst.MoveFirst`, `rst.MoveLast()` functions. Use `rst.EOF` and `rst.BOF` properties to determine whether recordset has reached the end or the beginning. Use `rst.Fields` collection to actually work with data:

```
repeat while not rst.eof
    put rst.fields["SomeFieldName" or SomeFieldIndex].Value
    rst.MoveNext()
end repeat
```

## Modifying Data via Recordset

By default, recordset's Open method will open read only forward only recordset. It means such recordset will not be able to modify data and will not be able to move the current record backward. This behavior is determined by other parameters of `rst.Open` method. See the description of `cursorType` and `lockType` parameters of `rst.Open` method. In general, `lockType` parameter determines the type of locking to be applied to the data. The default value is `adLockReadOnly`, which allows only read access to the data. The `cursorType` defines the capabilities of the recordset in relation to data changes made by others. The default value is `adOpenForwardOpen`, which defines a static copy of a set of records with forward only movement capability. Usually, in case you are going to modify data in database you may set the `lockType` parameter to `adLockPessimistic` and the `cursorType` parameter to `adOpenKeyset`:

```
rst.Open( strSQL, strCnn, #adOpenKeyset, #adLockPessimistic )
if rst.succeeded then
  put "Recordset state:" && rst.State
else
  put "Error:" && rst.lastError
end if
```

Now you are able to make modifications to data:

```
rst.Fields["SomeFieldName"].Value = SomeNewValue
rst.Fields["SomeOtherFieldName"].Value = SomeOtherNewValue
rst.Update()
```

The actual data modification is occurred on `Update` method. Always check whether call was succeeded, since data provider may deny attempt to modify data if data violates database integrity or other database rules.

## Closing Recordset

After you finish using particular recordset you may reopen it with other parameters. Use `rst.Close` method to release system resources associated with open recordset. Then you may reopen it with other parameters. If you do not need it any more, make sure to void out any Lingo variable that may store a reference to the VbScriptXtra wrapper object, thus completely releasing it from memory.

## Connection Object

In certain cases you may need to use alternative approach to perform required task. For example, you have to create connection object before opening recordset to open recordset inside a transaction. The other example is retrieving database schema information.

Use xtra-level method `CreateObject( strProgId )` to create wrapper for `ADODB.Connection` object:

```
Vb = xtra("VbScriptXtra")
cnn = vb.CreateObject( "ADODB.Connection" )
```

Check resulting value to ensure that ADO is available. If function succeeded `cnn` will be the Lingo object reference, otherwise it will be a string, describing error. Use `cnn.Version` property to determine ADO version:

```
if objectP( cnn ) then
  put "ADO version:" && cnn.Version
```

```
else
  put "Error:" && cnn
end if
```

Then you have to adjust connection parameters using Connection object's properties. See `cnn.ConnectionString`, `cnn.Provider` and other properties of the Connection object. Otherwise you may specify connection information as parameters of `cnn.Open` method.

## Object's Dynamic Properties

Connection object contains the collection of dynamic properties `cnn.Properties`. This collection contains multiple properties specific to the provider. You may access this collection after you specify which provider to use. If you do not specify any, the OLE DB provider for ODBC will be used. Once you set the provider of the connection object you cannot change it for this particular instance. After you specify provider you may look at dynamic properties it supports:

```
cnn.Provider = "Microsoft.Jet.OLEDB.4.0"
repeat with i = 0 to cnn.Properties.Count - 1
  put cnn.Properties[i].Name && "=" && cnn.Properties[i].Value
end repeat
```

You may adjust some dynamic properties:

```
cnn.Properties["SomePropertyName"] = SomeNewPropertyValue
```

The recordset object contains its own provider specific collection of the dynamic properties. They may be accessed in the same way.

## Using Transactions

You may use opened connection to start transaction. Use `cnn.BeginTrans` to start transaction. Use `cnn.CommitTrans` method to save changes or `cnn.RollbackTrans` method to cancel the changes being made inside the current transaction.

## Save and Compact Microsoft Access Database

Sometimes Jet (Microsoft Access) databases need to be compacted to decrease database size. There is a library called Microsoft Jet and Replication Objects that provides JetEngine object. One of its methods allows compacting/converting existing database into another file. It also allows setting or changing Jet database password.

At first create an instance of a JetEngine object:

```
vb = xtra( "VbScriptXtra" )
-- Creating a new instance of JetEngine
jet = vb.CreateObject( "JRO.JetEngine" )
```

Then you have to know the connection string for your existing database. **Note:** the database should not be opened by anyone else during save and compact procedure. Normally it is something like:

```
strSourceCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=D:\Temp\DB.mdb"
```

If you set the Jet password for your database, it looks like:

```
strSourceCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=D:\Temp\DB.mdb; Jet OLEDB:Database Password=PasswordHere"
```

Then you have to build a connection string for the new file that will be created by this operation.

```
strDestCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=D:\Temp\DB2.mdb"
```

You may specify new Jet database password or convert it to another engine type.

```
strDestCnn = "Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=D:\Temp\DB2.mdb; Jet OLEDB:New Database
Password=NewPasswordHere; Jet OLEDB:Engine Type=5"
```

To get more details about Jet properties and settings visit [MSDN](#).

Then you may call CompactDatabase method of the JetEngine object.

```
jet.CompactDatabase( strSourceCnn, strDestCnn )
if jet.Failed then alert jet.LastError
```

If the operation completes successfully Jet creates a new file of the specified type with the specified password if any. You may use any file management xtras to move newly created file into the original location. You can also move the file with VbScriptXtra and FileSystemObject.

```
-- Creating a new instance of FileSystemObject
fso = vb.CreateObject( "Scripting.FileSystemObject" )

-- Deleting old source file
fso.DeleteFile( "D:\Temp\DB.mdb" )

-- Moving new file in place of the old one
fso.MoveFile( "D:\Temp\DB2.mdb", "D:\Temp\DB.mdb" )
```

## Automating Microsoft PowerPoint

This sample makes a slide in a new PowerPoint presentation from the current Director frame.

```

vb = xtra("VbScriptXtra")

-- Create a new instance of PowerPoint
ppt = vb.CreateObject("PowerPoint.Application")

-- Create new presentation
p = ppt.Presentations.Add()

-- Add new slide to the newly created presentation
s = p.Slides.Add( 1, #ppLayoutBlank )

-- Scanning Director's current frame for texts and linked pictures
repeat with i = 1 to the lastChannel

  if sprite( i ).type <> 0 then
    mem = sprite(i).member
    r = sprite(i).rect

    case mem.type of
      #text:
        -- Add text box for a text member
        sh = s.shapes.AddTextbox( 1, r[1], r[2], r.width, r.height )
        sh.RTF = mem.rtf

      #bitmap:
        -- if bitmap member is linked
        if mem.fileName <> "" then
          -- Add picture shape for a bitmap member
          sh = s.shapes.AddPicture( mem.fileName, #true, #false, \
            r[1], r[2], r.width, r.height )
        end if
      end case
    end if
  end repeat

-- Now make PowerPoint visible
ppt.visible = #true

```

To find out how to save resulting presentation use ObjectBrowser xtra. Right after you get the instance of the Presentaion object, call [Interface\(\)](#) method.

```

-- Create new presentation
p = ppt.Presentations.Add()

p.Inteface()

```

If ObjectBrowser xtra is placed in Director's xtras folder, it will show up with the description of methods and properties available for Presentation object.

"Method: SaveAs

Arguments:

IN	String FileName
IN	PpSaveAsFileType FileFormat, Optional
IN	MsoTriState EmbedTrueTypeFonts, Optional

Returns: VOID

Call syntax:

```
SaveAs(fileName, FileFormat, EmbedTrueTypeFonts) "
```

So, since `FileFormat` and `EmbedTrueTypeFonts` are optional you may simply specify the new file name to save the presentation with default format.

```
p.SaveAs( the moviePath & "p.ppt" )
```

```
-- or
```

```
p.SaveAs( the moviePath & "p.ppt", #ppSaveAsPresentation, #false )
```

To quit PowerPoint set to `VOID` all references to PowerPoint objects. Call `Quit()` method of PowerPoint application object and set it to `VOID` too.

```
s = VOID
```

```
p = VOID
```

```
ppt.Quit()
```

```
ppt = VOID
```

Note: In Macromedia Director (version 7, 8, 8.5) 'quit' was reserved and did not passed to the xtra at all. To workaround this issue VbScriptXtra uses '[underscore handling](#)'. Add one underscore '\_' at the beginning of the `Quit`. VbScriptXtra will automatically remove it and will call real `Quit` method of the wrapped object.

```
-- Underscore will be removed by VbScriptXtra internally
```

```
ppt._Quit()
```

## Automating Microsoft Word

This sample creates a simple Microsoft Word document.

```
vb = xtra("VbScriptXtra")

-- Creating a new instance of Microsoft Word
wordApp = vb.CreateObject("Word.Application")

-- Checking whether object is created
if not objectP( wordApp ) then
  alert "Failed to create Word.Application:" & RETURN & wordApp
  exit
end if

-- Setting application's window params
wordApp.visible = #true
wordApp.left = 100
wordApp.top = 100
wordApp.width = 400
wordApp.height = 300

-- Creating new document
doc = wordApp.Documents.Add()

-- Arranging document window
wordApp.Windows.Arrange()

-- Typing simple message at the beginning of the document
doc.range().InsertAfter( "Hello Word!" )

-- Setting the font style of all text in document
doc.content.bold = #true
```

To find out how to open existing Word document use ObjectBrowser xtra. Right after you get the instance of the Word object, call [Interface\(\)](#) method of Documents collection.

```
-- Creating a new instance of Microsoft Word
wordApp = vb.CreateObject("Word.Application")

wordApp.Documents.Inteface()
```

If ObjectBrowser xtra is placed in Director's xtras folder, it will show up with the description of methods and properties available for Word's Application.Documents object.

"Method: Open

Arguments:

IN	Variant*	FileName
IN	Variant*	ConfirmConversions, Optional
IN	Variant*	ReadOnly, Optional
IN	Variant*	AddToRecentFiles, Optional
IN	Variant*	PasswordDocument, Optional
IN	Variant*	PasswordTemplate, Optional
IN	Variant*	Revert, Optional
IN	Variant*	WritePasswordDocument, Optional
IN	Variant*	WritePasswordTemplate, Optional
IN	Variant*	Format, Optional
IN	Variant*	Encoding, Optional
IN	Variant*	Visible, Optional
IN	Variant*	OpenAndRepair, Optional
IN	Variant*	DocumentDirection, Optional

IN Variant\* NoEncodingDialog, Optional  
IN Variant\* XMLTransform, Optional

**Returns: Document\***

**Call syntax:**

```
Open(FileName, ConfirmConversions, ReadOnly, AddToRecentFiles,  
PasswordDocument, PasswordTemplate, Revert, WritePasswordDocument,  
WritePasswordTemplate, Format, Encoding, Visible, OpenAndRepair,  
DocumentDirection, NoEncodingDialog, XMLTransform) "
```

As you see, most of parameters are optional. So you may simply specify the file name.

```
-- Creating a new instance of Microsoft Word  
wordApp = vb.CreateObject("Word.Application")  
  
-- Opening the document  
docs = wordApp.Documents  
docs.Open( the moviePath & "sample.doc" )  
  
-- Check whether document is opened successfully  
if docs.Failed then  
    alert "Failed to open document." & RETURN & docs.LastError  
    exit  
end if  
  
-- Making Word visible  
wordApp.Visible = #true
```

There is an alternative way to open Word document.

```
-- Opening the document  
doc = vb.GetObject2( the moviePath & "sample.doc", "Word.Document" )  
  
-- Check whether document is opened successfully  
if not objectP( doc ) then  
    alert "Failed to open document." & RETURN & doc  
    exit  
end if  
  
-- Making Word visible  
doc.Application.Visible = #true
```

## Automating Microsoft Excel

This sample creates a new workbook and outputs a simple message to the first cell of the first worksheet.

```

vb = xtra("VbScriptXtra")

-- Creating a new instance of Microsoft Excel
excel = vb.CreateObject( "Excel.Application" )

-- Checking whether object is created
if not objectP(excel) then
  alert "Failed to create Excel.Application:" & RETURN & excel
  exit
end if

-- Setting application's window params
excel.visible = #true
excel.DisplayFullScreen = #false
excel.left=100
excel.top=100
excel.width=400
excel.height=300

-- Creating new workbook
workbook= excel.workbooks.add()

-- Getting access to the first worksheet of the newly created workbook
sheet = workbook.Worksheets(1)

-- Arranging work book window
excel.Windows.Arrange()

-- Setting the value of the left-top cell
-- Extra parentheses required for Director 7
(sheet.Cells(1,1)).Value = "Hello Excel!"

-- Setting the font style of the left-top cell
sheet.cells(1,1).Style.Font.Bold = #true

-- Setting the width and height of the cell using range property
sheet.range("A1:A1").rowHeight = 64
sheet.range("A1:A1").columnWidth = 16

```

Good source of information about how to do something with Microsoft Office application is to record a macro within that application. Then see the Visual Basic code of the newly recorded macro. It will show you which methods and properties you should call to complete the required task. In most cases macro code could be directly translated to Lingo and VbScriptXtra.

Visual Basic macros usually use so-called named arguments where every method parameter is identified with its name, like in the following line, created by macro recorder in Excel.

```
Workbooks.Open Filename:="D:\Temp\Book2.xls"
```

To translate this statement to Lingo named argument should be translated to usual ordinal argument. In most cases you can simply skip the argument name. So in Lingo it should be:

```
Workbooks.Open( "D:\Temp\Book2.xls" )
```

Sometimes you will have to check the method definition to know correct order of arguments expected by the method. Use either ObjectBrowser xtra or documentation for the application being automated.

Inside Visual Basic macro reference to the application object is assumed by default. So that statement is actually a `Workbooks` property of an Excel's `Application` object. So, in Lingo you should use a VbScriptXtra's object holding Excel's `Application` object.

```
vb = xtra("VbScriptXtra")  
  
-- Creating a new instance of Microsoft Excel  
excel = vb.CreateObject( "Excel.Application" )  
  
excel.Workbooks.Open( "D:\Temp\Book2.xls" )
```

## WMI Scripting

Windows Management Instrumentation is a system level component that provides management information and control in an enterprise environment. Here is the sample that enumerates different properties of the logical drive C:.

```
vb = xtra("VbScriptXtra")
-- Obtain WMI object for drive C:
devC = vb.GetObject2("WinMgmts:win32_LogicalDisk.DeviceId='C:', """)
-- Getting the enumeration of available properties
props = ObjSet.Properties_.__NewEnum
-- Output available properties from a collection
repeat with i = 1 to props.count
  if props[i].value <> #Null then
    put props[i].name & ":" && props[i].value
  end if
end repeat
```

This sample outputs following information with my drive C:.

```
-- "Caption: C:"
-- "Compressed: 0"
-- "CreationClassName: Win32_LogicalDisk"
-- "Description: Local Fixed Disk"
-- "DeviceID: C:"
-- "DriveType: 3"
-- "FileSystem: FAT32"
-- "FreeSpace: 158916608"
-- "MaximumComponentLength: 255"
-- "MediaType: 12"
-- "Name: C:"
-- "Size: 4194902016"
-- "SupportsFileBasedCompression: 0"
-- "SystemCreationClassName: Win32_ComputerSystem"
-- "SystemName: EUGENE"
-- "VolumeName: SYS"
-- "VolumeSerialNumber: 77963952"
```

The sample below looks for memory usage values of the calling process. These are the same value as Windows Task Manager shows as 'VM Size' and 'Mem usage' for Director (or Projector) application. It uses [ProcessId](#) property of the new [System](#) wrapper object to get the identifier of the calling process and find appropriate record with WMI.

```
vb = xtra("VbScriptXtra")
sys = vb.CreateWrapper(#System)
ObjSet = xtra("VbScriptXtra").GetObject2("WinMgmts:root\cimv2","")
strQuery = "Select * from Win32_Process where " & \
  "ProcessId = " & sys.ProcessId & ""
colItems = ObjSet.ExecQuery( strQuery )
appInfo = colItems.__NewEnum[1]
put appInfo.PageFileUsage
-- 31289344
put integer(appInfo.WorkingSetSize)
-- 20926464
```

More details about using WMI are available at [MSDN](#).

---

# VbScriptXtra Programmer's Reference

## Wrapping Objects

VbScriptXtra provides its basic functionality via so-called wrapper objects. Wrapper objects allow using the wrapped contents from Lingo. Wrapper object provides methods and properties accessing wrapped contents or provided by wrapped contents.

[Automation object](#) wrapper is a key component of VbScriptXtra. Wrapper object keeps the pointer to the real Automation object. When Lingo calls any method or property from wrapper object it passes it to the wrapped Automation object providing necessary type casting and error checking support. Automation object wrapper is created automatically by typecasting routines when IDispatch value is detected.

To explicitly create this wrapper use xtra-level [CreateObject](#) or [GetObject](#) method:

```
objAuto = xtra("VbScriptXtra").CreateObject( strProgId )
```

[Binary data](#) wrapper is provided by VbScriptXtra for handling binary data and Unicode text. It is a kind of array of bytes or wide chars that could be handled by Lingo. Binary wrapper is created automatically by typecasting routines when BLOB value is detected. If the object is set to handle [Unicode as Binary](#) then Binary wrapper is created to represent the Unicode text string in Lingo.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
binaryWrapper = xtra("VbScriptXtra").CreateWrapper( #Binary )
unicodeWrapper = xtra("VbScriptXtra").CreateWrapper( #UnicodeString )
mbcsWrapper = xtra("VbScriptXtra").CreateWrapper( #MBCSString )
```

[Date/Time](#) wrapper is provided by VbScriptXtra for handling date/time data. It is created automatically by typecasting routines when VB date/time value is detected. This wrapper provides standard for VB functionality for formatting date/time values and other features.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
dateWrapper = xtra("VbScriptXtra").CreateWrapper( #Date )
```

[Registry key](#) wrapper is provided by VbScriptXtra for handling operations with system Registry. It allows browsing Registry keys checking for values and subordinate keys.

To create this wrapper use xtra-level [CreateWrapper](#) method:

```
registryKeyWrapper = xtra("VbScriptXtra").CreateWrapper( #RegistryKey )
```

[System helper](#) object is provided by VbScriptXtra starting with version 2.2. It support several system-level routines.

To create this object use xtra-level [CreateWrapper](#) method:

```
systemObject = xtra("VbScriptXtra").CreateWrapper( #System )
```

[Image wrapper](#) is provided by VbScriptXtra starting with version 2.5 as a helper to exchange image data between Director and COM applications. It handles Lingo image translation to and from Bitmap handles (HBITMAP) and StdOle.Picture objects.

To create this wrapper use xtra-level [CreateWrapper](#) method:

```
imageWrapper = xtra("VbScriptXtra").CreateWrapper( #Image )
```

Further versions of VbScriptXtra might include other wrapper types as well.

## Common Features of Wrapping Objects

Every wrapper object is built on the same prototype that provides some basic functionality common for all wrapper objects implemented by VbScriptXtra. Basic wrapper functionality includes: [error handling](#) and [debugging](#) support, [type casting](#) routines, [Unicode conversion](#) support.

### Error Handling Support

There are two main levels of errors related to VbScriptXtra. They have completely different nature and therefore have to be handled differently.

#### Lingo Errors

Lingo errors are similar to incorrect Lingo syntax run-time errors. They cause Director to show error alert saying something like "Method or property not found in object" or "One parameter expected". In Projector they might halt script execution etc. These errors usually mean that something is wrong with the programming. Wrong method call syntax is used or something similar to it. VbScriptXtra might return error codes to Director that make Director to show Lingo error alert box. It happens when wrapper object discovers the programming error at the Lingo level (wrong syntax, wrong parameters and other evident programming errors).

#### Programming Errors

This level includes errors that are actually exception conditions. They happen or do not happen depending on particular execution context. They are normal in programming practice and have to be handled programmatically. For example if file operation fails it does not have to worry end-user with Lingo error alert box. Instead developer should check whether operation completed successfully and perform what is appropriate.

VbScriptXtra provides programming errors handling support based on storing status of the last call within every wrapper object. In other words, every VbScriptXtra's wrapper object keeps the error code and description returned by the most recently called method or property. Before returning from the call to any wrapper object the last error information (if any) is being set by the wrapper object. Right before calling the next method or property of the wrapper object the last error information is cleared.

To check the status of the most recent call to the object use [obj.Failed](#) or [obj.Succeeded](#) properties. The error message and error code are available via [obj.LastError](#) and [obj.LastErrorCode](#) properties.

If Lingo statement includes cascading property access, several wrapper objects might be involved. Most of these wrappers (except the first one) are temporarily and therefore they are not accessible after the Lingo statement. So the error information could be lost. Sometimes it is worth to store intermediate wrappers in a Lingo variable just to have an opportunity to check whether a call was successful.

This sample shows how to check error status when multiple wrappers are involved in cascading property access operation.

```
on OpenWordDocument strPath
    vb = xtra("VbScriptXtra")
    w = vb.CreateObject( "Word.Application" )

    docs = w.Documents
    doc = docs.Open( strPath )
```

```
    if docs.Succeeded then
        return doc
    end if

    alert doc.LastError
    return VOID
end
```

Compare the above sample to the following one.

```
on OpenWordDocument strPath
    vb = xtra("VbScriptXtra")
    w = vb.CreateObject( "Word.Application" )

    doc = w.Documents.Open( strPath )

    -- Incorrect check since "w.Documents" always works
    -- while Open( strPath ) might fail
    if w.Succeeded then
        return doc
    end if

    alert doc.LastError
    return VOID
end
```

---

## Succeeded

Returns whether the most recent call to the wrapped contents was successful.

### Syntax

```
bResult = obj.Succeeded
```

### Return values

True

If the previous call to the wrapper's contents was successful

False

If the previous call to the wrapper's contents was not successful. The error code and description are available via [LastErrorCode](#) and [LastError](#) properties.

### Remarks

This property as well as other properties described in this section does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

---

## Failed

Returns whether the most recent call to the wrapped contents has failed.

### Syntax

```
bResult = obj.Failed
```

---

**Return values**

True

If the previous call to the wrapper's contents was not successful. The error code and description are available via `LastErrorCode` and `LastError` properties.

False

If the previous call to the wrapper's contents was successful

**Remarks**

This property as well as other properties described in this section does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

---

**LastErrorCode**

Returns the code of the last error (if any) happened while calling the contents of a wrapper object.

**Syntax**

```
nCode = obj.LastErrorCode
```

**Return values**

Integer

Integer value that indicates the error code of the most recent call to the wrapped contents. If the most recent call completed successfully, the error code is 0.

**Remarks**

This property as well as other properties described in this section does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

Most of error codes are coming from the wrapped Automation objects. They define their own error codes usually described in component's documentation.

Other error codes are defined by COM. Here come errors produced by passing incorrect parameters or skipping required parameter etc.

Several error codes are defined by VbScriptXtra. They could occur if VbScriptXtra failed to typecast Lingo value into COM variant or vice versa.

---

**LastError**

Returns the description of the last error (if any) happened while calling the contents of a wrapper object.

**Syntax**

```
strErrorMessage = obj.LastError
```

**Return values**

String

String value that contains the error description of the most recent call to the wrapped contents. If the most recent call completed successfully, the error description is empty.

**Remarks**

This property as well as other properties described in this section does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

**Debugging Support**

Every wrapper object created by VbScriptXtra can detect errors returned by wrapped objects. Internal VbScriptXtra errors (type casting problems etc) could happen too. Normally these errors could be trapped programmatically by checking object's last error status after any meaningful call to the object. See [error handling](#) support properties for more details. To simplify debugging process VbScriptXtra provides debugging mode.

**Simple Debugging Mode**

In simple debugging mode any wrapper object puts error information into Messages window whenever error occurred. Usually simple debugging mode is useful to detect whether script is executed well or there is a problem somewhere. Error messages usually come from wrapped objects but there is no information about the context where error occurred.

**Advanced Debugging Mode**

Advanced debugging mode allows you to catch error right in Debugger whenever error occurred. In this mode VbScriptXtra tries to call movie-level handler `VbScriptXtra_DebugEvent( strMes, nCode )`. If there is no such handler, the xtra behaves as in simple debugging mode. This handler may contain any Lingo statements. Furthermore, you can place a break point inside this handler and use Director's debugging capabilities to view the calling context, variables etc.

Sample movie-level handler for advanced debugging.

```
on prepareMovie
  if the playerMode = "author" then
    xtra("VbScriptXtra").Init( 2 )
  end if
end

on VbScriptXtra_DebugEvent strMes, nCode
  put strMes -- Place the break point here
end
```

Debugging mode is kept separately for every VbScriptXtra wrapper object. Use [DebugMode](#) property to change the debugging mode of the particular object directly. Otherwise use xtra-level [Init\( nDebug \)](#) method to set the default debugging mode for newly created wrappers. This method does not affect objects that already exist at the time of calling this method.

## Using Put Command

Every wrapper object provides descriptive information about itself via put Lingo method. To see what the wrapper object contains simply put it in Messages window.

```
Vb = xtra("VbScriptXtra")

objDate = vb.CreateWrapper( #Date )

put objDate
--"< VbScriptXtra, Date/Time, 09/03/2004 20:22:44 >"

objWord = vb.CreateObject("Word.Application")

put objWord
--"< VbScriptXtra, Word._Application, 0x001FB29C, (1) >"

objBinary = vb.CreateWrapper( #Binary )

objBinary.String = "Test"
put objBinary
--"< VbScriptXtra, Binary, Size: 4 byte(s) >"
```

## Using Debugger and Object Inspector

VbScriptXtra wrappers support viewing their contents via Director Debugger and Object Inspector.

Automation wrapper allows expanding its entry in Debugger to view properties of the wrapped Automation object. It is quite convenient although it has side effect that conflicts with debugging modes. When wrapper's entry in debugger is expanded Director internally calls all properties available to view in debugger. Wrapper object cannot distinguish whether it is called by debugger or by Lingo script. Therefore last error information kept by the wrapper object is erased with the status of the last method or property that was called by Director but not Lingo script. In advanced debugging mode the VbScriptXtra\_DebugEvent movie level handler could be called while Director asks object for its property values. So take care with that.

---

## DebugMode

Sets or gets the debugging mode for the specific wrapper object.

### Syntax

```
nDebugMode = obj.DebugMode
obj.DebugMode = nDebugMode
```

### Parameters

nDebugMode - Integer

Debugging mode for newly created objects. This parameter can be one of the following values.

Value	Meaning
0	No debugging support. Release behavior.
1	Simple debugging. Any error is automatically printed in Messages window.
2	Advanced debugging. When any error is occurred, the xtra calls movie level handler VbScriptXtra_DebugEvent( strMes, nCode ).

## Return values

Integer

Integer value that indicates the current debugging mode applied to the wrapper.

## Remarks

This property as well as other properties described in this section does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

Debugging mode is inherited by wrapper objects that are produced by the current object during calls to the wrapped contents.

Temporary wrapper objects produced by cascading properties access Lingo statement get the debugging mode from their parent object.

New wrappers created by [CreateObject](#) or [CreateWrapper](#) xtra-level methods inherits default xtra-level debug mode that is set by [Init](#) method.

## Type Casting Routines

COM Automation defines certain set of possible types that values could be. Lingo defines another set of types that Lingo values could be. VbScriptXtra performs necessary type casting operations to map one set of types into another.

In some cases VbScriptXtra cannot know how to convert the value from one type to another. So it can fail with error message saying "Cannot type cast Lingo or COM value". This type of errors could be programmatically detected via standard VbScriptXtra's wrappers error checking properties.

---

## COM Automation to Lingo

This conversion happens when VbScriptXtra wrapper returns any value returned by the wrapped object or when updating arguments passed by reference. This includes getting property values of the wrapped automation object.

The table below describes COM Automation types which are recognized by VbScriptXtra wrapper and into which Lingo types they are converted.

Automation type	Lingo type or value
EMPTY	VOID
NULL	Symbol #Null
Integer (signed/unsigned), 1,2,4 bytes	Integer signed 4 bytes native to Director value
Error	Integer
Float 4,8 bytes	Float
Numeric	Float
Date	VbScriptXtra Date/time wrapper
Unicode String	MBCS String or Binary
Boolean	Integer (1 or 0)
Currency	Float
GUID	String

SafeArray of Variants	Linear List (recursive)
IUnknown	Tries to get IDispatch. If it is not available, creates a <a href="#">Typed Instance</a> wrapper of Unknown type.
Automation object (IDispatch)	VbScriptXtra <a href="#">Automation</a> object wrapper
Empty pointer to IDispatch or IUnknown	Symbol #Nothing
SafeArray of Bytes (BLOB, OLE, Image)	VbScriptXtra Binary data wrapper

By default Unicode String is converted into MBCS string that is native to Lingo. If [UnicodeAsBinaryMode](#) is specified, then typecasting routines produce Binary wrapper for the Unicode string. The [ContentsType](#) of the wrapper is set to #UnicodeString in this case. See more details [below](#).

---

### Lingo to COM Automation

This conversion happens when VbScriptXtra wrapper passes any arguments to the wrapped automation object. This includes assigning property values of the wrapped automation object.

Lingo type or value	Automation type or value
Symbol #Null	NULL
Symbol #Nothing	IDispatch (empty pointer)
Symbol #True	Boolean (true)
Symbol #False	Boolean (false)
Symbol other symbols	Enumeration value from type libraries currently known to VbScriptXtra
Integer	signed integer 4 bytes
Float	Float 8 bytes
MBCS String	Unicode String
Date	Float
Linear List	SafeArray of Variants (recursive)
Property List	Handled as <a href="#">named arguments</a>
VOID	Missing value
Parent Script Instance	Uses 'Value' property of the instance to allow passing <a href="#">arguments by reference</a>
VbScriptXtra Automation wrapper	IDispatch
VbScriptXtra Typed Instance wrapper	IUnknown
VbScriptXtra Binary data wrapper	SafeArray (Vector) of Bytes
VbScriptXtra Unicode string wrapper	Unicode String
VbScriptXtra Date/Time wrapper	Date

## Unicode Conversion Support

Macromedia Director (up to the current version MX 2004) uses MBCS text encoding. MBCS stands for Multi Byte Code String. In MBCS each character is encoded by one or more bytes. Mapping of the particular character and its numerical code is based on the current Code Page (default for user's system). Some languages (English, French, German other European languages) do not use more than one byte for encoding one character. Other languages (Japanese, Arabic and other) do really use multi byte feature of MBCS.

Unicode defines numerical values for all known characters of almost all used languages. Unicode (or wide) character is normally encoded by two bytes, although hieroglyphs might be encoded with a couple of wide characters.

COM technology internally assumes that all text data is in Unicode. So, here comes the problem of conversion text inside VbScriptXtra at the moment of passing text data from Director to COM and vice versa.

Unicode to MBCS conversion always assumes some specific code page for MBCS encoding. Normally system default code page is used. In most cases the default code page provides correct conversion between Unicode and MBCS. However default code page might be incorrect choice for some multilingual applications build with Macromedia Director.

Consider an application that stores some text data in database. Application is expected to be distributed all over the World. Suppose text data in database is in French and it is encoded in Unicode. While application is being developed in France everything is fine since in France most systems probably has ANSI Latin 1 code page as a system default code page. When French text data in Unicode comes to Director through VbScriptXtra, it is being converted to MBCS with ANSI Latin 1 code page and then it is displayed on stage with fonts that know how to display ANSI Latin 1 characters.

What happens if we run this application under system with another default code page? For example with Cyrillic code page. Text data from database comes in Unicode. If the xtra tries to convert it by using default (Cyrillic) code page it most likely will be converted as some Latin characters and a lot of '?' questions, because certain characters that are in Latin 1 code page do not present in Cyrillic code page. So after this conversion we get some MBCS data, but French fonts will not be able to display it properly, since some characters have been replaced by questions during conversion. So end-user will not be able to see French text under Cyrillic system.

That is why VbScriptXtra provides a special property that controls which code page is used for text conversion between Unicode and MBCS. In the above example, even under Cyrillic system VbScriptXtra could convert Unicode text into MBCS with Latin 1 (1252) code page.

Code pages for hieroglyphic languages normally define some MBCS characters with single byte (English letters), but others with several bytes. That is why it is actually called Multi Byte Code String. Using proper `CodePage` it is possible to get correctly encoded MBCS text from Unicode text stored in database. To display it with Director you will need the older format `TrueType` font that is encoded for this code page. `OpenType` fonts may behave strange either in Projector or in Director authoring environment.

[System](#) helper wrapper provides control over keyboard layouts that may be useful for multilingual applications.

## CodePage

Controls code page number used by the wrapper object's Unicode - MBCS text conversion routines.

### Syntax

```
nCodePage = obj.CodePage
```

```
obj.CodePage = nCodePage
```

### Parameters

nCodePage - Integer

Integer value that indicates which code page to use while conversion text from/to Unicode/MBCS. See MSDN for possible [code page identifiers](#).

### Return values

Integer

Integer value that indicates the current code page number applied to the wrapper object.

### Remarks

This property does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

The `CodePage` property allows the xtra to be used in multilingual environment.

Director uses MBCS (Multi Byte Code String). Every character is encoded by one or more bytes. The encoding is based on the particular code page number. COM assumes that all text data is Unicode encoded.

This property defines particular code page number to be used in text conversion routines of the xtra.

By default, the `CodePage` property is 0. It defines the default behavior for the system. It is the code page of the default locale for non-Unicode applications.

The `CodePage` property affects all text conversion operations initiated by this instance of the wrapper. All wrappers created by this wrapper inherit the value of `CodePage` property. In other words, all VbScriptXtra wrappers created by xtra-level methods [CreateObject](#) or [CreateWrapper](#) get the default value of the code page, which is zero. Wrapper instances derived from other wrapper instance inherits the code page setting of the parent wrapper object.

Take care when changing the default value of this property, since inappropriate code page number may result in empty string as a result of text conversion. Below is the list of possible code page numbers:

Code page number	Meaning
0	System default code page (ANSI by default)
2	Macintosh code page
1	OEM code page
42	Symbol code page (Win2k)
3	The current thread's ANSI code page (Win2k)
65000	Translate using UTF-7 (Win2k, NT 4.0)

65001	Translate using UTF-8 (Win2k, NT 4.0)
037	EBCDIC
437	MS-DOS United States
500	EBCDIC "500V1"
708	Arabic (ASMO 708)
709	Arabic (ASMO 449+, BCON V4)
710	Arabic (Transparent Arabic)
720	Arabic (Transparent ASMO)
737	Greek (formerly 437G)
775	Baltic
850	MS-DOS Multilingual (Latin I)
852	MS-DOS Slavic (Latin II)
855	IBM Cyrillic (primarily Russian)
857	IBM Turkish
860	MS-DOS Portuguese
861	MS-DOS Icelandic
862	Hebrew
863	MS-DOS Canadian-French
864	Arabic
865	MS-DOS Nordic
866	MS-DOS Russian
869	IBM Modern Greek
874	Thai
875	EBCDIC
932	Japanese
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1026	EBCDIC
1200	Unicode (BMP of ISO 10646)
1250	Windows 3.1 Eastern European
1251	Windows 3.1 Cyrillic
1252	Windows 3.1 US (ANSI)
1253	Windows 3.1 Greek
1254	Windows 3.1 Turkish
1255	Hebrew
1256	Arabic
1257	Baltic
1361	Korean (Johab)
10000	Macintosh Roman
10001	Macintosh Japanese
10006	Macintosh Greek I
10007	Macintosh Cyrillic

---

10029	Macintosh Latin 2
10079	Macintosh Icelandic
10081	Macintosh Turkish

---

### UnicodeAsBinaryMode

Controls whether Unicode text returned by Automation objects will be translated into [Binary](#) wrapper containing Unicode string. By default, all text is translated into Lingo strings according [CodePage](#) settings.

#### Syntax

```
bUnicodeAsBinaryMode = obj.UnicodeAsBinaryMode  
obj.UnicodeAsBinaryMode = bUnicodeAsBinaryMode
```

#### Parameters

bUnicodeAsBinaryMode

Boolean value that indicates whether typecasting routines should translate Unicode strings into Binary wrapper objects.

#### Return values

Boolean

value that indicates whether typecasting routines should translate Unicode strings into Binary wrapper objects. True means that any string returned by the Automation object will be represented as Binary object whose contents type is set to #UnicodeString.

False means default behavior when text is automatically converted into Lingo strings according to the current code page of the object.

#### Remarks

This mode is inherited by wrapper objects that are produced by the current object during calls to the wrapped contents.

New wrappers created by [CreateObject](#) or [CreateWrapper](#) xtra-level methods inherits default xtra-level mode that is set by [Init](#) method.

**Note:** although this property is listed among common properties, it only affects those text data that is coming from COM Automation objects. So, it is directly applies only to the [Automation](#) object wrappers.

## Automation Object Wrapper

Automation object wrapper is a key component of VbScriptXtra. Wrapper object keeps the pointer to the real Automation object. When Lingo calls any method or property from wrapper object it passes it to the wrapped Automation object providing necessary type casting and error checking support. Automation object wrapper is created automatically by typecasting routines when IDispatch value is detected.

To explicitly create this wrapper use xtra-level [CreateObject](#) or [GetObject](#) method:

```
objAuto = xtra("VbScriptXtra").CreateObject( strProgId )
```

Most of [methods](#) and [properties](#) called from the wrapper are simply passed to the wrapped Automation object.

Use [Interface\(\)](#) method to get the type library information of the wrapped Automation object via [ObjectBrowser](#) xtra.

Use [GenEnum\( name \)](#) method to get the value of named constant from wrapped object type library.

To handle events provided by the wrapped automation object use [EventsHandler](#) property.

Collection enumeration support is available via special [NewEnum](#) property.

## Methods

---

### Interface()

Invokes [ObjectBrowser](#) xtra to display methods and properties provided by the wrapped Automation object.

#### Syntax

```
strInfo = objAuto.Interface()
```

#### Return values

String

Returns a string message with supported methods and properties together with calling [ObjectBrowser](#) xtra status.

#### Remarks

This method does not clear the last error flag. It means this method does not affect the last error information for the particular wrapper object.

#### Sample

The sample creates an instance of Microsoft Word and invokes [ObjectBrowser](#) to display methods and properties provided by [Documents](#) collection.

```
vb = xtra( "VbScriptXtra" )
w = vb.CreateObject( "Word.Application" )
w.visible = true
put w.documents.Interface()
```

---

**GetEnum( symName )**

Gets the named enumeration value from currently loaded by VbScriptXtra type libraries.

Any COM Automation object is usually described by a type library provided by the object. Type libraries often contain a set of named constants. They are used as parameters to object methods or properties or in any other means depending on the particular object.

Once VbScriptXtra detects new Automation object it scans its type library for these enumerations that contain named constants. After that these values are accessible via this method.

**Syntax**

```
Value = objAuto.GetEnum( Symbol symName )
```

**Parameters**

symName

String or Symbol with the name of enumeration value

**Return values**

Returns the appropriate enumeration value. If no matching enumeration is found then wrapper raises Lingo error "Invalid parameter".

**Remarks**

This method does not clear the last error flag. It means this method does not affect the last error information for the particular wrapper object.

**Note:** VbScriptXtra type casting routine translates Lingo symbols the same way as this method does. So in most cases you may simply place enumeration value name as symbol, but this will only work if you use it as a parameter to any method that will be passed to the wrapped Automation object. Also you cannot use symbols in arithmetic expressions. Also you cannot compare the property value to a Lingo symbol. Be aware that Lingo symbol is not the value of appropriate enumeration. It turns to it only when VbScriptXtra type casting routine is involved.

**Note:** Some type libraries refer to other type libraries that could define their own enumerations. For example, it is the way how Microsoft Office applications type libraries are built. Once VbScriptXtra detects an object from another type library it scans enumerations from their, but before that moment VbScriptXtra might know nothing about enumerations from external library.

**Sample**

This sample demonstrates enumeration values usage.

```
Vb = xtra("VbScriptXtra")
ppt = vb.CreateObject("PowerPoint.Application")

p = ppt.presentations.Add()
s = p.Slides.Add( 1, #ppLayoutBlank )

-- or
s = p.Slides.Add( 1, p.GetEnum(#ppLayoutBlank) )
```

In the following sample it is important to use GetEnum method but not plain symbols, since bitwise operations and comparisons do have sense only with enumeration values but not with symbols.

```

if bitand( rst.state, rst.GetEnum(#adStateFetching) ) = \
    rst.GetEnum(adStateFetching) then
    -- if fetching is in progress exiting handler
    exit
end if

```

---

## Calling Other Methods

### Syntax

```
result = objAuto.MethodName( parameters )
```

Wrapper passes any method name to the wrapped Automation object. If Automation object does not accept the method name, wrapper raises Lingo error "Handler not found in object". Also see [Underscore handling](#) for more details.

### Parameters

Any parameters are converted to appropriate Automation types, if possible. See [type casting](#) for details about supported Lingo types. See [Technical details](#) for more info.

### Return values

Returns whatever is returned by wrapped Automation object translated to the appropriate Lingo type, if possible. See [type casting](#) about supported COM Automation types. Returned value could be either plain Lingo type or another VbScriptXtra wrapper. So cascaded method and property calls are possible.

### Remarks

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

Error code	Meaning
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

## Properties

---

### InterfaceName

Returns the name of the wrapped object.

### Syntax

```
strName = objAuto.InterfaceName
```

---

**Return values**

String

Returns the name of the wrapped object's interface. The name is built from the name of the type library and the name of the interface itself separated by dot.

**Sample**

Sample demonstrates using `InterfaceName` property with `Word.Application` object.

```
vb = xtra( "VbScriptXtra" )
w = vb.CreateObject( "Word.Application" )
w.visible = true
put w.interfaceName
-- "Word._Application"
```

---

**TypeLib**

Returns the object representing the Type Library that the current instance belongs to.

**Syntax**

```
objTypeLib = objAuto.TypeLib
```

**Return values**

Instance

Returns the object representing the type library that contains a definition of the wrapped object's interface.

**Remarks**

This property is available in version 2.5 of VbScriptXtra. [Type library wrapper object](#) is used for scripting IUnknown based COM Automation compliant interfaces. To use them, the [typed wrapper instance](#) has to be created first. Typed wrapper instance 'knows' the type of the wrapped IUnknown object as it is described in the type library. The type library wrapper object can create typed wrapper instances of the requested type.

---

**EventsHandler**

Gets or sets the events handler for the wrapped Automation object. The handler could be either parent script instance or sprite reference. Using sprite reference is useful for handling events fired by visual ActiveX controls or OLE objects activated in-place. Refer to ActiveX xtra and OLE xtra documentation for more details.

**Syntax**

```
objParentScript = objAuto.EventsHandler
objAuto.EventsHandler = objParentScript
```

**Parameters**

`objParentScript`

The instance of the parent script or sprite reference that handles events or VOID if there should not be any.

**Return values**

Object

Current parent script instance or sprite reference that handles events or VOID if there is no one.

**Remarks**

Some Automation objects can provide feedback via so-called events. Events usually used to inform clients about anything or to ask whether server should or should not do something. For example Microsoft Word notifies via events that document is about to be closed allowing event's handler to prevent closing of the document if time for it has not come yet.

Use `ObjectBrowser` to see whether particular object provides events.

Setting this property to a parent script instance makes the wrapper object to connect to the even source and start listening for events. Once some event occurs, wrapper object tries to call the parent script instance with that event.

Parameters to event handlers are passed by using property lists. This is done to allow event handlers to operate with parameters passed by reference.

For every event attached script instance is called twice. At first wrapper object tries to call the handler with the event name.

```
on eventName me, lstArgs
    put "eventName:" && lstArgs
end
```

Then `IncomingEvent` handler is called. **Note:** `lstArgs` coming to this handler might be modified by the previous handler (if any).

```
on IncomingEvent me, symEvent, lstArgs
    put symEvent, lstArgs
end
```

It is up to developer to choose which handling method to use. Both handlers are always called. If there is no handler to handle the event it is ignored.

**Sample**

Sample demonstrates using events with `ADODB.Connection` object.

```
-- *****
-- Here is the code for EventHandler parent script
on new me
    return me
end

on IncomingEvent me, event, args
    put event, args
end

on ConnectComplete me, args
    pError = args[1]
    adStatus = args[2]
    pConnection = args[3]

    put "ConnectComplete"

    if ( adStatus <> 1 ) then alert pError.Description
end
```

```

on Disconnect me, args
  adStatus = args[1]
  pConnection = args[2]

  put "Disconnect"
end

on WillConnect me, args
  ConnectionString = args[1]
  UserID = args[2]
  Password = args[3]
  Options = args[4]
  adStatus = args[5]
  pConnection = args[6]

  put "WillConnect"

  -- Creating new connection string
  -- Microsoft Jet provider for MS Access databases
  cnnStr = "Provider=Microsoft.Jet.OLEDB.4.0;"
  cnnStr = cnnStr & "Data Source=D:\Temp\TestDB.mdb;"
  cnnStr = cnnStr & "Mode=Read|Write;"

  -- return it to the connection object via referenced parameter
  args[1] = cnnStr
end

-- End of the code for EventHandler parent script
-- *****

```

Name this script as "ConnectionEvents". Then try to execute following lines right in Director's messages window.

```

Vb = xtra("VbScriptXtra")

-- Setting debug mode to true
vb.Init(true)

-- Creating an instance of the ADODB.Connection object
cnn = vb.CreateObject("ADODB.Connection")

-- Creating an instance of the events handler parent script
evnts = new( script("ConnectionEvents") )

-- Attaching handler to a wrapper
cnn.EventsHandler = evnts

-- Opening connection without explicitly specifying connection params
-- Connection string should be set by the events handler
cnn.Open()

cnn.Close()

```

---

## **NewEnum**

Returns the special collection enumerator object that allows access to collection elements. This property is normally hidden to VB users. It is called automatically by 'for each' statement. Consider following VB sample:

```

Set w = GetObject("Word.Application")

For Each Doc in w.Documents
  MsgBox Doc.Name
next

```

Lingo does have equivalent language construction 'repeat with each', but it is unavailable to be used by xtra (or at least it is undocumented). So VbScriptXtra provides the special collection enumerator object that uses the same mechanism of collection enumerating as VB does.

Most of real collections provide its own Count property and Item( Index ) method allowing enumerating the collection, but sometimes this internal enumeration support might be necessary.

**Note:** the exact name of this property in VB is '\_NewEnum' (one underscore '\_'). Due to VbScriptXtra [Underscore Handling](#) it is necessary to add one extra underscore when getting this property.

### Syntax

```
objCollection = objAuto.__NewEnum -- Note two underscore
count = objCollection.Count
element = objCollection[ nIndex ]
element = objCollection.GetAt( nIndex )
```

### Parameters

nIndex

One-based index of the required element. Integer value in the range from 1 to collection count is expected. If it is less than 1 method will raise Lingo error "Value out of range". If requested index is not accessible (for example it exceeds the number of elements in collection) the method sets the last error information of the collection enumerator object. It is available via common [error handling](#) properties.

### Return values

Object

Wrapper object for the collection enumerator.

### Remarks

Collection enumerator object only supports accessing element methods and properties shown in syntax above.

### Sample

The sample demonstrates enumerating documents collection by using \_\_NewEnum property.

```
w = vb.GetObject("Word.Application")
col = w.Documents.__NewEnum
repeat with i = 1 to col.Count
  doc = col[i]
  put doc.Name
end repeat
```

---

## Getting Other Properties

### Syntax

```
result = objAuto.PropertyName
```

```
result = objAuto.PropertyName[ valIndex ]
```

Wrapper passes any property name to the wrapped Automation object. If Automation object does not accept the property name, wrapper raises Lingo error "Property not found". Also see [Underscore handling](#) for more details.

### Parameters

Parameter (if any) is converted to appropriate Automation type, if possible. See [type casting](#) for details about supported Lingo types.

### Return values

Returns whatever is returned by wrapped Automation object translated to the appropriate Lingo type, if possible. See [type casting](#) about supported COM Automation types. Returned value could be either plain Lingo type or another VbScriptXtra wrapper. So cascaded method and property calls are possible.

### Remarks

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

Error code	Meaning
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

---

## Setting Other Properties

### Syntax

```
objAuto.PropertyName = SomeValue
objAuto.PropertyName[ valIndex ] = SomeValue
```

Wrapper passes any property name to the wrapped Automation object. If Automation object does not accept the property name, wrapper raises Lingo error "Property not found". Also see [Underscore handling](#) for more details.

### Parameters

Parameters are converted to appropriate Automation types, if possible. See [type casting](#) for details about supported Lingo types.

### Remarks

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

Error code	Meaning
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

## Technical Details

### Underscore Handling

Macromedia Director (D7, D8, and D8.5) sometimes behaves strangely with certain names. Director does not allow some names to be used as properties or method names. In Director MX and MX 2004 this problem seems to be fixed.

To workaround this issue Automation wrapper at first tries to see whether wrapped object knows passed name. If so, it is processed. If it does not know it and the name starts with underscore '\_' wrapper removes the first underscore and tries again.

Currently noticed names are: 'Delete' (D7, D8, and D8.5) and 'Append' (D8.5).

Attempt to invoke Delete method of the wrapped Automation object generates a Lingo error before Delete is even passed to the xtra (D7, D8, D8.5). The same way behaves D8.5 with Append method. To eliminate this problem use:

```
Object._Delete()
object._Append()
```

VbScriptXtra will remove the first underscore before passing method name to the wrapped Automation object. So methods are called correctly.

### Named Method's Arguments

Lingo does not support named arguments as Visual Basic does, but starting with VbScriptXtra 2.3.3 you may use syntax very similar to VB named arguments. Simply place arguments into a property list. In this case VbScriptXtra will treat them as named arguments.

For example in VB:

```
obj.SomeMethod NameOfSomeArg:"SomeValue"
```

In Lingo just put named arguments into a property list:

```
obj.SomeMethod( [ #NameOfSomeArg:"SomeValue" ] )
```

Named and ordinal arguments can be mixed similar to VB, but ordinal arguments have to come first:

```
obj.SomeMethod( 1, 2, [ #NameOfSomeArg:"SomeValue" ] )
```

Note that property name will be checked against method argument names according to object's type definition. Lingo error 'Out of range' will be raised by VbScriptXtra in case invalid property name is used in named arguments property list.

---

## Passing Parameters by Reference

VbScriptXtra Automation wrappers support arguments passed by reference, although it requires some special conventions. Lingo passes simple type values by value, but automation objects sometimes rely on arguments passed by reference. VbScriptXtra wrapper accepts parent script instances as method arguments. Once wrapper encounters such argument it will use its 'Value' property as an actual argument of the automation object's method. Then after method is executed, wrapper will put the updated argument value to 'value' property of the parent script instance.

So, if you expecting modified argument value you will have to create a simple parent script instance, set its value property with actual argument value, use that instance as an argument to wrapper object's method and then get the updated value from that instance. Most automation objects' method do not use arguments passed by reference, but sometimes, there is no other way.

Starting with VbScriptXtra 2.3.3 there is an alternative approach. It is based on passing named arguments within property list. In this case VbScriptXtra will refresh arguments that are expected to be references. After calling such method arguments property list will contain updated values.

```
args = [ #NameOfSomeArg:"SomeValue" ]
obj.SomeMethod( args )

put args
-- [ #NameOfSomeArg:"SomeOtherValue" ]
```

---

## Optional and Missing Method's Arguments

Optional and missing arguments are supported by VbScriptXtra but Lingo requires you to use VOID value to indicate missing argument in the middle of the parameters list. Missing arguments in the end of the arguments list may be safely skipped. Default values will be used by automation object.

Also you may use named arguments syntax to avoid using a lot of VOIDs if your method needs only 5th argument for example and does not need others.

---

## Using Wrapper Instance as First Argument of a Method

Take care while passing VbScriptXtra wrapper instance as the first argument to a movie handler. This may cause problems with some automation objects.

The problem arises from Lingo supporting both original and dot syntax. When Lingo interpreter encounters a method call, it checks whether its first argument is an object instance. So it tries to invoke a method of that object with the same name. If this call fails Lingo searches for a movie handler with this name and calls it if successful.

VbScriptXtra wrapper instance accepts any method name and tries to pass it to the wrapped automation object. Most of automation objects support a fixed set of methods, so the wrapper is capable to find out whether wrapped object supports the particular method or not. Such objects do not cause problems and are correctly passed to the movie handler if they do not support the same method.

There is at least one automation object, which behaves differently. It is ADODB.Connection object. Its instances accept any method names (not only supported

directly), since Connection object may try to execute the stored database procedure, which may exist in database. If stored procedure exists it is executed, otherwise it generates corresponding error. This behavior does not allow VbScriptXtra wrapper to know whether such automation object supports particular name or not.

So, avoid passing wrapper instances as the first argument of movie handlers, since you may unintentionally call a method of this object instead of your movie handler.

---

### **Cascading Methods and Properties in Director 7**

Director 7 has a bug in Lingo interpreter, which requires placing extra brackets to access properties of an object returned by some method. For example, the following statement will generate Lingo error.

```
put objAuto.someMethod().someProp -- Lingo error here
```

To avoid it you have to bracket method call:

```
put (objAuto.someMethod()).someProp -- Ok
```

Director 8 and later do not have this problem.

---

### **Using Square Brackets**

When calling method with single argument or accessing indexed property with single index, it is possible to use either normal or square brackets. For example following Lingo syntax is possible with VbScriptXtra wrappers:

```
put rst.fields["FileName"].Value -- works in D7 too
```

```
put rst.fields("FileName").Value -- does not work in D7, see above
```

## Type Library Wrapper

Type Library wrapper serves as a factory of typed instance wrappers. This is a key component of supporting non IDispatch COM Automation compliant objects (IUnknown based objects). Typed instance wrappers 'know' the type of the object they can contain. Thus when typed instance wrapper gets the pointer to some object, it first checks whether this object provides the type that wrapper 'knows'. If it does, then the wrapper can redirect Lingo calls to the wrapped object according to that type of the object.

This approach is very similar to VBA variables that are declared to be of the certain type. Type Library wrapper is used to create such variables in Lingo.

The instance of the Type Library wrapper can be created automatically by accessing [TypeLib](#) property of the [Automation Object](#) wrapper.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
typeLibraryWrapper = xtra("VbScriptXtra").CreateWrapper( #TypeLib )
```

Use [Load](#) method to initialize the newly created wrapper with a type library loaded from a file. Type libraries often exist as .tlb files. They are also may be loaded from .dll and .ocx files.

Use [Interface\(\)](#) method to get the type library information of the wrapped type library via `ObjectBrowser` xtra.

Use [Name](#) and [GUID](#) properties to find out the name and GUID of the loaded type library.

Type Library wrapper is used to create [typed instance wrappers](#). These wrappers are based on the type defined in the type library. Two kinds of types can be used for that: classes and interfaces. To create a new wrapper use the name of the requested type as a property of TypeLibrary wrapper. See [getting other properties](#) for a sample.

## Methods

---

### Interface()

Invokes `ObjectBrowser` xtra to display methods and properties provided by the wrapped Automation object.

#### Syntax

```
strInfo = objAuto.Interface()
```

#### Return values

String

Returns a string message with supported methods and properties together with calling `ObjectBrowser` xtra status.

#### Remarks

This method does not clear the last error flag. It means this method does not affect the last error information for the particular wrapper object.

**Load( strPath )**

Clears the current contents and fills it with a type library loaded from the specified file.

**Syntax**

```
objTypeLib.Load( String strPath )
```

**Parameters**

strPath

The path to a file containing type library data ( \*.tlb, \*.dll, \*.ocx, etc).

**Remarks**

This method is used to initialize newly created type library wrapper. It loads the library from a file.

**Sample**

The sample below shows how to use new method.

```
vb = xtra("VbScriptXtra")
-- Create TypeLibrary wrapper
adodb20 = vb.CreateWrapper(#TypeLib)
-- Load type library from file
adodb20.Load("C:\Program Files\Common Files\System\ado\msado20.tlb")
-- Use it to create the wrapper for Recordset object and
-- create a new instance of the ADO.DB.Recordset object
rst = adodb20.Recordset.New()
```

**Properties**

---

**Name**

Returns the name of the wrapped type library.

**Syntax**

```
strName = objTypeLib.Name
```

**Return values**

String

Returns the name of the wrapped type library.

---

**GUID**

Returns the GUID of the wrapped type library.

**Syntax**

```
strGUID = objTypeLib.GUID
```

**Return values**

String

Returns the GUID of the wrapped type library.

---

**Getting Other Properties**

Wrapper uses the name of the property to find appropriate interface in the wrapped type library. If it finds one, it creates the new typed instance wrapper and returns it.

**Syntax**

```
objTypedInstance = objTypeLib.InterfaceName
```

```
objTypedInstance = objTypeLib.ClassName
```

**Return values**

Instance

Returns the new empty typed instance wrapper.

**Remarks**

The newly created wrapper is empty. It does not contain any 'wrapped' object unless you assign it to the wrapper. Use [objTypedInstance.New\(\)](#) to initialize the wrapper with the new instance of the class. Otherwise use [Ptr](#) property to fill the wrapper with the pointer to the interface to another existing object if it provides one.

**Sample**

The sample below shows how to create wrappers for ADO DB objects.

```
vb = xtra("VbScriptXtra")
-- Create TypeLibrary wrapper
adodb20 = vb.CreateWrapper(#TypeLib)
-- Load type library from file
adodb20.Load("C:\Program Files\Common Files\System\ado\msado20.tlb")
-- Use it to create the wrapper for Recordset object and
rst = adodb20.Recordset
-- Use it to create the wrapper for Connection object and
cnn = adodb20.Connection
-- Initialize it with the new instance of the ADO DB.Connection object
cnn.New()
```

The sample below shows how to use [Ptr](#) property to use IUnknown-based interfaces of **MontiVision SmartControl**.

```
vb = xtra("VbScriptXtra")
-- Get the Automation object from Montivision ActiveX control
objSmartControl = sprite(spriteNum).GetObject()
-- Get the type library object from it
MV = objSmartControl.TypeLib
-- Create the wrapper of type IMVSCVideoWindow2
objVideoWindow2 = MV.IMVSCVideoWindow2
```

```
-- Retrieve the IMVSCVideoWindow2 interface from
-- the SmartControl object and assign it to the wrapper
objVideoWindow2.ptr = objSmartControl

-- Now objVideoWindow2 may be used to call methods and properties
-- specified by IMVSCVideoWindow2 interface
put objVideoWindow2.VideoXResolution, objVideoWindow2.VideoYResolution
```

---

## Typed Instance Wrapper

Typed instance is a key component of supporting non IDispatch COM Automation compliant objects (IUnknown based objects). Typed instance wrappers 'know' the type of the object they can contain. Thus when typed instance wrapper gets the pointer to some object, it first checks whether this object provides the type that wrapper 'knows'. If it does, then the wrapper can redirect Lingo calls to the wrapped object according to the type of the object.

This approach is very similar to VBA variables that are declared to be of the certain type. [Type Library wrapper](#) is used to [create](#) such variables in Lingo.

The newly created typed instance wrapper is empty. It does not contain any 'wrapped' object unless you assign it to the wrapper. Use [objTypedInstance.New\(\)](#) to initialize the wrapper with the new instance of the class. Otherwise use [Ptr](#) property to fill the wrapper with the pointer to the interface to another existing object if it provides one.

Use [Interface\(\)](#) method to get the type library information of the wrapped object via `ObjectBrowser xtra`.

Other [methods](#) and [properties](#) defined by the type used to create the wrapper are simply passed to the wrapped COM object.

## Methods

---

### Interface()

Invokes `ObjectBrowser xtra` to display methods and properties provided by the wrapped Automation object.

### Syntax

```
strInfo = objCOM.Interface()
```

### Return values

String

Returns a string message with supported methods and properties together with calling `ObjectBrowser xtra` status.

### Remarks

This method does not clear the last error flag. It means this method does not affect the last error information for the particular wrapper object.

---

### New()

Initializes the typed instance wrapper with the new instance of the class that is used to create this wrapper.

### Syntax

```
objCOM.New()
```

**Remarks**

This method is valid only for typed instance wrappers based on classes but not on interfaces. Type library may define several classes. Normally creatable classes have ProgId assigned to them but this is not necessary. This method allows alternative way to create objects.

**Sample**

The sample below shows how to use new method.

```
vb = xtra("VbScriptXtra")
-- Create TypeLibrary wrapper
adodb20 = vb.CreateWrapper(#TypeLib)
-- Load type library from file
adodb20.Load("C:\Program Files\Common Files\System\ado\msado20.tlb")
-- Use it to create the wrapper for Recordset object and
-- create a new instance of the ADO.DB.Recordset object
rst = adodb20.Recordset.New()
```

**Calling Other Methods****Syntax**

```
result = objCOM.MethodName( parameters )
```

Wrapper searches the type definition that it is based on for a passes method name. If it finds the match then it calls it on the wrapped object. If type definition does not contain the method name, wrapper raises Lingo error "Handler not found in object". Also see [Underscore handling](#) for more details.

**Parameters**

Any parameters are converted to appropriate Automation types, if possible. See [type casting](#) for details about supported Lingo types. See [Technical details](#) for more info.

**Return values**

Returns whatever is returned by wrapped COM object translated to the appropriate Lingo type, if possible. See [type casting](#) about supported COM Automation types. Returned value could be either plain Lingo type or another VbScriptXtra wrapper. So cascaded method and property calls are possible.

**Remarks**

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

Error code	Meaning
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

## Properties

---

### InterfaceName

Returns the name of the wrapped object's type.

#### Syntax

```
strName = objCOM.InterfaceName
```

#### Return values

String

Returns the name of the wrapped object's interface. The name is built from the name of the type library and the name of the interface itself separated by dot.

---

### Ptr

Gets or sets the pointer to the real object contained within this wrapper.

#### Syntax

```
objCOM.ptr = objUnknown
```

#### Parameters

objUnknown

The instance that contains COM Automation compliant object either IDispatch based or IUnknown based. It can be either [Automation object wrapper](#) or Typed instance wrapper.

#### Remarks

Set operation on this property retrieves the pointer to a COM object contained within objUnknown. Then it tries to query it for the interface that objCOM wrapper is based on. COM object may either provide this interface or do not do it. If it does, then assignment operation completes successfully, otherwise the current contents of objCOM wrapper is cleared and operation fails with COM error saying why it failed to query for the requested interface.

#### Sample

The sample below shows how to use Ptr property to use IUnknown based interfaces of MontiVision SmartControl.

```
vb = xtra("VbScriptXtra")  
  
-- Get the Automation object from MontiVision ActiveX control  
objSmartControl = sprite(spriteNum).GetObject()  
  
-- Get the type library object from it  
MV = objSmartControl.TypeLib  
  
-- Create the wrapper of type IMVSCVideoWindow2  
objVideoWindow2 = MV.IMVSCVideoWindow2
```

```
-- Retrieve the IMVSCVideoWindow2 interface from
-- the SmartControl object and assign it to the wrapper
objVideoWindow2.ptr = objSmartControl

-- Now objVideoWindow2 may be used to call methods and properties
-- specified by IMVSCVideoWindow2 interface
```

---

## Getting Other Properties

### Syntax

```
result = objCOM.PropertyName
result = objCOM.PropertyName[ valIndex ]
```

Wrapper searches the type definition that it is based on for a passes property name. If it finds the match then it queries it on the wrapped object. If type definition does not contain the specified property name, wrapper raises Lingo error "Property not found". Also see [Underscore handling](#) for more details.

### Parameters

Parameter (if any) is converted to appropriate Automation type, if possible. See [type casting](#) for details about supported Lingo types.

### Return values

Returns whatever is returned by wrapped COM object translated to the appropriate Lingo type, if possible. See [type casting](#) about supported COM Automation types. Returned value could be either plain Lingo type or another VbScriptXtra wrapper. So cascaded method and property calls are possible.

### Remarks

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

Error code	Meaning
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

---

## Setting Other Properties

### Syntax

```
objAuto.PropertyName = SomeValue
objAuto.PropertyName[ valIndex ] = SomeValue
```

Wrapper searches the type definition that it is based on for a passes property name. If it finds the match then it queries it on the wrapped object. If type definition does not contain

the specified property name, wrapper raises Lingo error "Property not found". Also see [Underscore handling](#) for more details.

### Parameters

Parameters are converted to appropriate Automation types, if possible. See [type casting](#) for details about supported Lingo types.

### Remarks

If VbScriptXtra cannot type cast Lingo values passed as parameters or returned COM Automation value, method sets the last error information that is available via common [error handling](#) properties.

VbScriptXtra specific errors include:

<b>Error code</b>	<b>Meaning</b>
1021	Cannot convert COM value to Lingo value. VbScriptXtra does know how to convert used COM Automation type to Lingo.
1022	Cannot convert Lingo value to COM value. VbScriptXtra does know how to convert used Lingo type to COM Automation.

Other errors could be produced by Automation object itself or COM or type casting code in some cases.

## Binary Data Wrapper

Binary data wrapper is provided by VbScriptXtra for handling binary and Unicode data. It is a kind of array of bytes or wide chars that could be handled by Lingo. Binary wrapper is created automatically by typecasting routines when BLOB value is detected. If the object is set to handle [Unicode as Binary](#) then Binary wrapper is created to represent the Unicode text string in Lingo.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
binaryWrapper = xtra("VbScriptXtra").CreateWrapper( #Binary )
unicodeWrapper = xtra("VbScriptXtra").CreateWrapper( #UnicodeString )
mbcsWrapper = xtra("VbScriptXtra").CreateWrapper( #MBCSString )
```

Newly created binary wrapper is initialized as an empty array.

Use [Interface\(\)](#) method to get the short description of methods and properties provided by this object.

Use [ContentsType](#) property to define what is contained in the object: binary data or Unicode string.

Use [Allocate\( nSize \)](#) or [Resize\( nSize \)](#) methods to set the wrapped data to the requested amount of bytes. [Clear\(\)](#) method releases all allocated memory. Use [Count](#) or [Size](#) properties or methods to get the allocated size of the wrapped data.

Binary wrapper supports list-like element access via [GetAt\( nIndex \)](#) and [SetAt\( nIndex, nValue \)](#) methods. They are implicitly called by using square brackets `objBinary[ nIndex ]`.

To initialize the binary data from a file use [ReadFromFile\(...\)](#) method. It allows reading either whole file or the portion of it.

[WriteToFile\(...\)](#) and [AppendToFile\(...\)](#) methods allow creating or modifying existing files with the contents of the binary wrapper.

Use [UnsignedByte\[ nIndex \]](#) or [SignedByte\[ nIndex \]](#) operators to get the numeric value of the specified element of the wrapped data either as unsigned or as signed value.

Use [Byte\[ nStartIndex .. nEndIndex \]](#) to get the new binary wrapper object initialized with the specified portion of wrapped data.

Use [String](#) property to use wrapper as a String. If either `#UnicodeString` or `#MBCSString` is set as a [ContentsType](#), then necessary Unicode to MBCS conversion will occur according to the current object's [CodePage](#).

[HexString](#) allows working with wrapper contents as with hex encoded data.

Use [Value](#) property to save/load Lingo lists or other values in binary form.

Use [MakeImageFromRGB\(...\)](#) and [GetRGBFromImage\(...\)](#) to operate directly on raw image data and create Lingo Image objects from it.

---

## Methods

---

### Interface()

Returns a short description of what you can do with this wrapper

#### Syntax

```
put binaryWrapper.Interface()
```

#### Return values

String

String value with short description of methods and properties provided by this wrapper

#### Remarks

This method does not clear the last error flag. It means this property does not affect the last error information for the particular wrapper object.

---

### Clear()

Clears the contents of the wrapper object. Initializes the object to the default state.

#### Syntax

```
objBinary.Clear()
```

---

### Allocate( nSize )

Clears the wrapped array and allocates requested number of bytes initialized by zeros.

#### Syntax

```
objBinary.Allocate( Integer nSize )
```

#### Parameters

nSize

The number of bytes to be allocated. Zero or positive integer number is expected. Otherwise method will raise Lingo error "Value out of range".

#### Remarks

If there are no memory available to complete the allocation request, method sets the last error information that is available via common [error handling](#) properties. If the allocation request fails array stays in the empty initialized state.

#### Sample

Trying to allocate more memory than available:

```
Vb = xtra("VbScriptXtra")
objBinary = vb.CreateWrapper( #Binary )
objBinary.Allocate( 1000000000 )

put b.Failed
-- 1
```

```
put b.LastErrorCode
-- 14

put b.LastError
-- "Not enough storage is available to complete this operation."
```

---

**Resize( nSize )**

Reallocates wrapped array to be of requested number of bytes preserving existing data.

**Syntax**

```
objBinary.Resize( Integer nSize )
```

**Parameters**

nSize

The number of bytes to be allocated. Zero or positive integer number is expected. Otherwise method will raise Lingo error "Value out of range".

**Remarks**

If there are no memory available to complete the allocation request, method sets the last error information that is available via common [error handling](#) properties. If the allocation request fails array stays in the empty initialized state.

If the requested array size is smaller than the current size, the current data is truncated.

If the requested array size is larger than the current size, the extra bytes are initialized with zeros.

---

**GetAt( nIndex )**

Gets the requested element of the array as an unsigned byte.

**Syntax**

```
nValue = objBinary.GetAt( Integer nIndex )
nValue = objBinary[ Integer nIndex ]
```

**Parameters**

nIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

**Return values**

Integer

Value of the requested byte. Returned value is unsigned, in the range from 0 to 255.

---

**SetAt( nIndex, nValue )**

Sets the requested element of the array to a new value.

**Syntax**

```
objBinary.SetAt( Integer nIndex, Integer nValue )
objBinary.SetAt( Integer nIndex, String strValue )
objBinary[ Integer nIndex ] = nValue
objBinary[ Integer nIndex ] = strValue
```

**Parameters**

nIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

nValue

New value to be set at the specified element. The low byte of the integer value is used.

strValue

New value to be set at the specified element. The first character of the string is used. If empty string is passed, method will raise Lingo error "Integer expected".

---

**ReadFromFile( strPath, nOffset, nBytesToRead )**

Clears the current contents and fills it with a data from file starting at the specified offset.

**Syntax**

```
nBytesRead = objBinary.ReadFromFile(
    String strPath,
    Integer nOffset,
    Integer nBytesToRead )
```

**Parameters**

strPath

The path to a file to be red.

nOffset

Optional. The offset within a file in bytes where to start reading data. Positive offset is calculated from the beginning of a file. Negative offset is calculated from the end of a file. Zero means reading from the beginning of a file. Default is zero.

nBytesToRead

Optional. The maximum number of bytes to be red. Default is -1 that means reading file from the specified offset till the end of file.

**Return values**

Integer

Number of bytes being actually red from a file.

**Remarks**

This method tries to open a file first. Then it calculates the resulting size of an array based on the file size and specified parameters. It resizes the array to the calculated size. After that it actually reads the file contents.

If there are no memory available to complete the allocation request or there are problems reading the specified file method sets the last error information that is available via common [error handling](#) properties.

**WriteToFile( strPath )**

Saves the contents of array into the specified file.

**Syntax**

```
nBytesWritten = objBinary.WriteToFile( String strPath )
```

**Parameters**

strPath

The path to a file to be written.

**Return values**

Integer

Number of bytes being actually written to a file. It should match the size of array.

**Remarks**

This method creates or overwrites any existing file with the data from array.

If there are problems writing to the specified file method sets the last error information that is available via common [error handling](#) properties.

**AppendToFile( strPath, nOffset, bSetEndOfFile )**

Modifies the existing file with the data from array.

**Syntax**

```
nBytesWritten = objBinary.AppendToFile(
    String strPath,
    Integer nOffset,
    Boolean bSetEndOfFile )
```

**Parameters**

strPath

The path to an existing file to be written.

nOffset

Optional. The offset within a file in bytes where to start writing data. Positive offset is calculated from the beginning of a file. Negative offset is calculated from the end of a file. Zero means starting from the beginning of a file. If this parameter is missed data will be appended to the end of file.

bSetEndOfFile

Optional. Indicates whether file should be truncated with the last written byte from the array. It is only has sense if the array data is written in the middle of a file and the data size added to starting offset does not exceed the length of a file. If this parameter is `True` the length of file will be adjusted to the last written byte. If it is `False` the length of file will stay the same.

### Return values

Integer

Number of bytes being actually written to a file. It should match the size of array.

### Remarks

If there are problems writing to the specified file method sets the last error information that is available via common [error handling](#) properties.

## **MakeImageFromRGB( nWidth, nHeight, bBitsPerPixel )**

Creates a Lingo Image object of the specified size and depth and initializes it with the wrapped binary data.

### Syntax

```
objImage = objBinary.MakeImageFromRGB(
    Integer nWidth,
    Integer nHeight,
    Integer nBitsPerPixel )
```

### Parameters

nWidth

The width of the image to be created.

nHeight

The height of the image to be created.

nBitsPerPixel

The number of bits per pixel for the new image. Supported values are: 1, 2, 4, 8, 16, 24, 32. Other values will cause Lingo error "Value out of range".

### Return values

Image

Newly created Lingo Image object.

### Remarks

The binary wrapper should contain at least  $nWidth * nHeight * nBitsPerPixel / 8$  bytes. Otherwise this method will fail.

Note that Lingo Image objects do not really use 24 bits per pixels. It uses 32 bits per pixel instead. This difference is handled by the xtra. If `nBitsPerPixel` is set to 24, then the xtra uses every three bytes of binary data to set the value of every pixel of the Lingo Image. New Lingo Image object is created as having 32 bits per pixel without alpha channel.

If `nBitsPerPixel` is set to 32, then the xtra uses every four bytes of binary data to set the value of every pixel of the Lingo Image. New Lingo Image object is created as having 32 bits per pixel with or without alpha channel. It depends on whether binary data contain anything other than zeroes in bytes containing alpha data.

Raw binary data is expected to be in generic Windows bitmap format. Every pixel is represented by one or several bits/bytes depending on the `nBitsPerPixel` value. If 32 bits per pixel is expected, then every pixel is represented by four bytes where red green blue and alpha bytes go one after another. Every row of image data is aligned to four bytes boundary.

---

## GetRGBFromImage( objImage )

Initializes the contents of the wrapper with the raw image data.

### Syntax

```
objBinary.GetRGBFromImage( Image objImage )
```

### Parameters

`objImage`

The Lingo Image object whose image data is to be copied into the wrapper.

### Remarks

The size of the binary wrapper after calling this method should be `objImage.Width * objImage.Height * objImage.Depth` bytes or a bit more due to aligning of the every row of image data to the four bytes boundary.

To restore original image from the binary data use [MakeImageFromRGB\(\)](#) method. Note that you have to know width, height and depth of the image to be restored, since this information is not saved in raw RGB data.

Raw binary data is in generic Windows bitmap format. Every pixel is represented by one or several bits/bytes depending on the `nBitsPerPixel` value. If 32 bits per pixel is expected, then every pixel is represented by four bytes where red green blue and alpha bytes go one after another. Every row of image data is aligned to four bytes boundary.

## Properties

---

### Count

Returns the size of the wrapped data in bytes.

### Syntax

```
nSize = objBinary.Count  
nSize = objBinary.Count()
```

### Return values

Integer

Number of bytes that wrapped binary data occupies.

---

**Size**

Returns the size of the wrapped data in bytes. Same as `Count`.

**Syntax**

```
nSize = objBinary.Size
```

```
nSize = objBinary.Size()
```

**Return values**

Integer

Number of bytes that wrapped binary data occupies.

---

**ContentsType**

Returns the type of the wrapped binary data.

**Syntax**

```
symType = objBinary.ContentsType
```

```
objBinary.ContentsType = symType
```

**Parameters**

Symbol

One of symbols below that describes the data type of the value.

Lingo Value	Meaning
#Binary	Default contents type. Just binary data.
#UnicodeString	Object contains raw Unicode (UTF16) string.
#MBCSString	Object contains MBCS string.

**Remarks**

Modifying this property does not change the wrapped data. It only affects how this data is returned or set by [String](#) property of the object. For #Binary data object returns it as is, while for #UnicodeString or #MBCSString data it will perform necessary translation between Unicode UTF8/UTF16 and MBCS in order to get correct representation in Director. Translation that involves MBCS relies on the current [code page](#) setting of the object.

When Binary object is used as an argument to COM Automation method or property then typecasting routine behavior also depends on the type of object's contents. #UnicodeString data is used as normal COM String as is. #Binary data is represented as safe array of bytes.

Binary objects with `ContentsType` set to #UnicodeString may be automatically produced by xtra's typecasting routines for Automation object wrappers using [UnicodeAsBinaryMode](#).

---

**UnsignedByte[ nIndex ]**

Gets or sets the requested element of the array as an unsigned byte.

**Syntax**

```
nValue = objBinary.UnsignedByte[ Integer nIndex ]  
nValue = objBinary.Byte[ Integer nIndex ]  
put objBinary.Byte.Count  
objBinary.Byte[ Integer nIndex ] = nValue  
objBinary.UnsignedByte[ Integer nIndex ] = strValue
```

**Parameters**

nIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

nValue

New value to be set at the specified element. The low byte of the integer value is used.

strValue

New value to be set at the specified element. The first character of the string is used. If empty string is passed, method will raise Lingo error "Integer expected".

**Return values**

Integer

Value of the requested byte. Returned value is unsigned, in the range from 0 to 255.

---

**SignedByte[ nIndex ]**

Gets or sets the requested element of the array as a signed byte.

**Syntax**

```
nValue = objBinary.SignedByte[ Integer nIndex ]  
put objBinary.SignedByte.Count  
objBinary.SignedByte[ Integer nIndex ] = nValue  
objBinary.SignedByte[ Integer nIndex ] = strValue
```

**Parameters**

nIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

nValue

New value to be set at the specified element. The low byte of the integer value is used.

strValue

New value to be set at the specified element. The first character of the string is used. If empty string is passed, method will raise Lingo error "Integer expected".

### Return values

Integer

Value of the requested byte. Returned value is signed, in the range from -128 to 127.

## Byte[ nStartIndex .. nEndIndex ]

Gets the requested elements of the array as a new binary wrapper object.

### Syntax

```
objBinary2 = objBinary.Byte[ nStartIndex .. nEndIndex ]
put objBinary.Byte.Count
```

### Parameters

nStartIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

nEndIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

### Return values

Binary wrapper

New binary wrapper initialized with the specified range of data from original wrapper object.

## Char[ nIndex ]

Gets or sets the requested element of the array as a one character string.

### Syntax

```
strChar = objBinary.Char[ Integer nIndex ]
put objBinary.Char.Count
objBinary.Char[ Integer nIndex ] = strValue
```

### Parameters

nIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

strValue

New value to be set at the specified element. The first character of the string is used. If empty string is passed, method will raise Lingo error "Integer expected".

---

**Return values**

String

One character string value of the requested element.

---

**Char[ nStartIndex .. nEndIndex ]**

Gets the requested elements of the array as a string.

**Syntax**

```
strSubstring = objBinary.Char[ nStartIndex .. nEndIndex ]  
put objBinary.Char.Count
```

**Parameters**

nStartIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

nEndIndex

One-based index of the required element. Integer value in the range from 1 to array size is expected. Otherwise method will raise Lingo error "Value out of range".

**Return values**

String

String value from the specified range of array.

---

**WideChar[ nIndex ]**

Gets or sets the requested element of the array as an Integer.

**Syntax**

```
nUnicodeChar = objBinary.WideChar[ Integer nIndex ]  
objBinary.WideChar[ Integer nIndex ] = nUnicodeChar  
put objBinary.WideChar.Count
```

**Parameters**

nIndex

One-based index of the required element. Integer value in the range from 1 to the half of array size is expected. Otherwise method will raise Lingo error "Value out of range".

nUnicodeChar

New value to be set at the specified element.

**Return values**

Integer

Integer number of the specified Unicode char.

**Remarks**

This property is intended for use with objects whose [ContentsType](#) is set to #UnicodeString. It allows direct manipulations with Unicode (two-bytes) characters.

**WideChar[ nStartIndex .. nEndIndex ]**

Gets the requested elements of the array as a new binary wrapper object.

**Syntax**

```
objBinary2 = objBinary.Byte[ nStartIndex .. nEndIndex ]
put objBinary.WideChar.Count
```

**Parameters**

nStartIndex

One-based index of the required element. Integer value in the range from 1 to the half of the array size is expected. Otherwise method will raise Lingo error "Value out of range".

nEndIndex

One-based index of the required element. Integer value in the range from 1 to the half of the array size is expected. Otherwise method will raise Lingo error "Value out of range".

**Return values**

Binary wrapper

New binary wrapper initialized with the specified range of data from original wrapper object. The new wrapper inherits the [ContentsType](#) property of the original object.

**Remarks**

This property is intended for use with objects whose [ContentsType](#) is set to #UnicodeString. It allows extracting a Unicode substring from the original string.

**Media**

Allows represent the contents of a wrapper as a media of member.

**Syntax**

```
hMedia = objBinary.Media
member("SomeMember").Media= hMedia

hMedia = member("SomeMember").Media
objBinary.Media = hMedia
```

**Parameters**

hMedia

The Lingo value that keeps "media" data of any cast member.

---

**Return values**

Object

The Lingo value that keeps "media" data.

**Remarks**

**Note:** this property does not modify the wrapped data in any way. It only represent it is a "media" value by allowing Lingo to use it as real media values are used. Actually they can only be assigned to other members.

Binary wrapper allows storing media data in external file or in database for example.

---

**Picture**

Allows represent the contents of a wrapper as a picture value.

**Syntax**

```
hPicture = objBinary.Picture
member("SomeMember").Picture = hPicture

hPicture = member("SomeMember").Picture
objBinary.Picture = hPicture
```

**Parameters**

hPicture

The Lingo value that keeps "picture" data of appropriate cast member.

**Return values**

Object

The Lingo value that keeps "picture" data.

**Remarks**

**Note:** this property does not modify the wrapped data in any way. It only represent it is a "picture" value by allowing Lingo to use it as real picture values are used. Actually they can only be assigned to other members.

Binary wrapper allows storing picture data in external file or in database for example.

---

**String**

Gets or sets the array data as a string.

**Syntax**

```
strValue = objBinary.String
objBinary.String = strValue
```

**Parameters**

strValue

The new string value that should be stored in a binary wrapper. The internal array is resized to the length of the string.

**Return values**

String

String representation of the data in array.

**Remarks**

The way how data is passed through this property depends on the object's [ContentType](#) and its current [CodePage](#). If either #UnicodeString or #MBCSString is set as a ContentType, then necessary Unicode to MBCS conversion will occur according to the current object's CodePage.

**Sample**

The sample below shows Unicode text translation .

```
vb = xtra("VbScriptXtra")
-- Create wrapper
s = vb.CreateWrapper( #UnicodeString )
s.String = "Some text"

put s
--"< VbScriptXtra, UnicodeString, Size: 18 byte(s) >"

put s.HexString
-- "53006F006D00650020007400650078007400"

put s.String
-- "Some text"

s.String = "Some cyrillic text: АBB"

put s.WideChar[21..23].HexString
-- "100411041204"

s.CodePage = 65001 -- UTF8 encoding

put s.String
-- "Some cyrillic text: Ә Ә'Ә'"
```

**HexString**

Gets or sets the array data as a hex encoded string, where every byte of array is encoded by a couple of hexadecimal numbers.

**Syntax**

```
strHex = objBinary.HexString
objBinary.HexString = strHex
```

**Parameters**

strHex

The hex encoded string with data that should be placed in the wrapper. Characters other than hexadecimal digits are ignored.

---

**Return values**

String

Hex encoded string representation of the data.

---

**Value**

Accesses the binary contents as a Lingo value that is saved in binary format similar to MUS protocol format.

**Syntax**

```
val = objBinary.Value  
objBinary.Value = [ 1, [#a:2], [3, "SomeText] ]
```

**Parameters**

val

The Lingo value like number, string or list.

**Return values**

Object

The Lingo value recovered from binary data.

**Remarks**

This property is intended for storing Lingo values in binary form. This data can be saved into a file or database for later recovering. There is also a good idea to extract it as [HexString](#) and store in field cast member.

Assigning the Lingo value to the wrapper will [fail](#) if xtra does not know how to serialize the specific data type.

## Image Wrapper

Image wrapper is provided by VbScriptXtra as a helper to exchange image data between Director and COM applications.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
imageWrapper = xtra("VbScriptXtra").CreateWrapper( #Image )
```

Newly created image wrapper is not initialized.

Use [Interface\(\)](#) method to get the short description of methods and properties provided by this object.

Use [Image](#) property to access Lingo Image object wrapped by this object.

Use [Width](#), [Height](#), [Depth](#) and [AlphaDepth](#) to get specific attributes of the wrapped image.

Use [InitFromHBitmap\(...\)](#) method to create the Lingo Image object from Bitmap handle.

Use [StdOlePicture](#) property to get wrapped image in a form of StdOle.Picture object, that can be used to pass images to COM Automation components. If you need to pass Bitmap handle to COM Automation component use the Handle property of StdOle.Picture object. See the sample below.

Use [StdOlePicture](#) property to initialize wrapped image from StdOle.Picture object, that may be created by some COM Automation components.

## Methods

---

### Interface()

Returns a short description of what you can do with this wrapper

#### Syntax

```
put imageWrapper.Interface()
```

#### Return values

String

String value with short description of methods and properties provided by this wrapper

---

### InitFromHBitmap( BitmapHandle, bRelease )

Creates new Lingo Image object from the specified bitmap handle.

#### Syntax

```
objImage.InitFromHBitmap(
    HBITMAP nBitmapHandle,
    Optional Boolean bRelease )
```

---

**Parameters**

nBitmapHandle

The HBITMAP handle that specifies Windows bitmap object. Some ActiveX controls may return these objects. A HBITMAP handle is just an integer value, it is not an object itself.

bRelease

Optional Boolean value that specify whether Bitmap object should be released after extracting image data. The default method behavior assumes that bitmaps created as DibSection are always owned by creator. Bitmaps created by other means should be released by this method in any case. The default behavior is implied when bRelease parameter is omitted or set to VOID.

Although default behavior covers the most of cases it is a good idea to check COM object documentation whether HBITMAP is expected to be released or not. Using this parameter Incorrectly may cause either memory leaks or component failures.

If bRelease is set to true, then this method will release the bitmap object after successful completion. If method fails no release occurs.

**Remarks**

The method checks whether passed handle is actually a bitmap handle. If so, it checks it dimensions and bit depth. Then it creates a Lingo Image object with the same parameters. If bitmap has 32 bits per pixel, then xtra checks whether Image object should contain alpha channel or not. It depends on whether image data contain anything other then zeroes in bytes containing alpha data. In some cases you may need to switch off the alpha channel in the resulting Image object.

```
objImage.Image.UseAlpha = false
```

**Properties**

---

**Image**

Returns the wrapped Lingo Image object.

**Syntax**

```
img = objImage.Image
```

**Return values**

Image

Wrapped Lingo Image object if any, or VOID otherwise.

---

**Width**

Returns the width of the wrapped Lingo Image object.

**Syntax**

```
nWidth = objImage.Width
```

---

**Return values**

Integer

The width of the wrapped Lingo Image object.

---

**Height**

Returns the height of the wrapped Lingo Image object.

**Syntax**

```
nHeight = objImage.Height
```

**Return values**

Integer

The height of the wrapped Lingo Image object.

---

**Depth**

Returns the bit depth of the wrapped Lingo Image object.

**Syntax**

```
nDepth = objImage.Depth
```

**Return values**

Integer

The depth of the wrapped Lingo Image object.

---

**AlphaDepth**

Returns the depth of the alpha channel of the wrapped Lingo Image object.

**Syntax**

```
nAlphaDepth = objImage.AlphaDepth
```

**Return values**

Integer

The bit depth of the alpha channel of the wrapped Lingo Image object. Usually it is either 8 or zero.

---

**StdOlePicture**

Sets or gets Lingo Image in a form of StdOle.Picture object.

**Syntax**

```
StdOlePic = objImage.StdOlePicture
```

```
objImage.StdOlePicture = StdOlePic
```

## Parameters

StdOlePic

[Automation wrapper](#) containing an instance of StdOle.Picture object.

## Return values

Object

Newly created Automation wrapper containing a new instance of StdOle.Picture object.

## Remarks

StdOle.Picture object is a COM-provided way for handling images. It can carry Windows bitmaps (HBITMAP), icon, metafiles and some other image types.

When setting this property new Lingo Image object is created. When StdOle.Picture object carries bitmap, Lingo Image is created of the same dimensions and depth (similar to [InitFromHBitmap\(\)](#) method). For other types of images Lingo Image is created with 32 bits per pixel without alpha channel.

Reading this property causes the creation of a new bitmap object, then new instance of the StdOle.Picture object is created and initialized with newly created bitmap. After that new Automation wrapper object is created to wrap the StdOle.Picture object. It may be used as any other COM Automation objects. Its Handle property allows accessing to the bitmap handle of the wrapped image.

## Sample

The sample below shows how to set overlay image to be shown over video stream of VideoCapX ActiveX control from [Fath software](#).

```

vid = sprite( "VideoCapX" ).GetObject()
-- Connect to the default video devise (probably web CAM)
vid.connected = true
-- Make it to show anything
vid.preview = true
-- Create an image wrapper
img = xtra("VbScriptXtra").CreateWrapper(#Image)
-- Initialize it with the image of the stage
img.image = the stage.image
-- Get the image in a form of StdOle.Picture object
StdOlePic = img.StdOlePicture
-- Set overlay image for the current video stream
vid.SetBitmapOverlay( StdOlePic.Handle, 0, 0, 0, 100 )

```

**Note:** vid.SetBitmapOverlay() expects bitmap handle. To get the Lingo image in a form of Windows bitmap we create StdOle.Picture object first. It carries bitmap that we can access via Handle property. Be aware that this bitmap is owned by StdOle.Picture object. It is released when StdOlePic goes out of the current scope.

## Date/Time Data Wrapper

Date/Time wrapper is provided by VbScriptXtra for handling date/time data. It is created automatically by typecasting routines when VB date/time value is detected. This wrapper provides standard for VB functionality for formatting date/time values and other features.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
dateWrapper = xtra("VbScriptXtra").CreateWrapper( #Date )
```

Newly created date/time wrapper is initialized with the current system time.

Use [Interface\(\)](#) method to get the short description of methods and properties provided by this object.

Use [FormatDate](#) and [FormatTime](#) methods to get date and time portions of the wrapped value in text representation formatted according the specified format.

Properties provided by this object allows working with date/time value in [float representation](#) and getting user friendly date/time value parts as [year](#), [month](#), [day](#), etc.

Also the wrapper provides conversion date/time value from [local](#) user's time zone to [universal](#) time and vice versa.

## Methods

---

### Interface()

Returns a short description of what you can do with this wrapper

#### Syntax

```
put dateWrapper.Interface()
```

#### Return values

String

String value with short description of methods and properties provided by this wrapper

---

### FormatDate( strFormat )

Gets the formatted string representation of the date part of date/time value according to current user's locale.

#### Syntax

```
strDate = dateWrapper.FormatDate( String strFormat )
```

#### Parameters

strFormat

Optional. Format string for example "DD/MM/YY". See remarks for more details.

#### Return values

String

String representation of the date/time value formatted according to the specified format.

## Remarks

If the method is called without parameters, it uses default short date representation from user's locale.

Format string consists of following elements.

Element	Meaning
d	Day of month as digits with no leading zero for single-digit days.
dd	Day of month as digits with leading zero for single-digit days.
ddd	Day of week as a three-letter abbreviation.
dddd	Day of week as its full name.
M	Month as digits with no leading zero for single-digit months.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as a three-letter abbreviation.
MMMM	Month as its full name.
Y	Year as last two digits, but with no leading zero for years less than 10.
YY	Year as last two digits, but with leading zero for years less than 10.
YYYY	Year represented by full four digits.
gg	Period/era string. This element is ignored if the date to be formatted does not have an associated era or period string.

**Note:** format string elements are case-sensitive.

Characters that do not match any of format string elements will appear at the same location in the output string.

Characters in the format string that are enclosed in single quotation marks will appear in the same location and unchanged in the output string.

To include a single quote in the output string it should be entered twice and enclosed in a couple of single quotation marks. So it comes four times. For example to get "Aug'31" use format string "MMM' ' 'dd".

---

## FormatTime( strFormat )

Gets the formatted string representation of the time part of date/time value according to current user's locale.

### Syntax

```
strTime = dateWrapper.FormatDate( String strFormat )
```

### Parameters

strFormat

Optional. Format string for example "DD/MM/YY". See remarks for more details.

### Return values

String

String representation of the date/time value formatted according to the specified format.

**Remarks**

If the method is called without parameters, it uses default short date representation from user's locale.

Format string consists of following elements.

Element	Meaning
h	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	Hours with leading zero for single-digit hours; 12-hour clock.
H	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	Hours with leading zero for single-digit hours; 24-hour clock.
m	Minutes with no leading zero for single-digit minutes.
mm	Minutes with leading zero for single-digit minutes.
s	Seconds with no leading zero for single-digit seconds.
ss	Seconds with leading zero for single-digit seconds.
t	One character time-marker string, such as A or P.
tt	Multicharacter time-marker string, such as AM or PM.
h	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	Hours with leading zero for single-digit hours; 12-hour clock.

**Note:** format string elements are case-sensitive.

Characters that do not match any of format string elements will appear at the same location in the output string.

Characters in the format string that are enclosed in single quotation marks will appear in the same location and unchanged in the output string.

To include a single quote in the output string it should be entered twice and enclosed in a couple of single quotation marks. So it comes four times. For example to get "22:01'51" use format string "HH' : 'mm' ' ' 'ss".

**MonthName( nMonth, bAbbreviated )**

Gets the name of the month in abbreviated or complete form.

**Syntax**

```
strMonthName = dateWrapper.MonthName (
    Integer nMonth,
    Boolean bAbbreviated )
```

**Parameters**

nMonth

Integer month number in the range from 1 to 12.

bAbbreviated

Optional. If `true` the abbreviated form of the month name is returned. If `false` or skipped then unabbreviated form of the month name is returned.

---

**Return values**

String

String value with the name of the month.

---

**WeekdayName( nDay, bAbbreviated, nFirstDayOfWeek )**

Gets the name of the month in abbreviated or complete form.

**Syntax**

```
strDayName = dateWrapper.WeekdayName(  
    Integer nDay,  
    Boolean bAbbreviated,  
    Integer nFirstDayOfWeek )
```

**Parameters**

nDay

Integer month number in the range from 1 to 7.

bAbbreviated

Optional. If true the abbreviated form of the month name is returned. If false or skipped then unabbreviated form of the month name is returned.

nFirstDayOfWeek

Optional. Indicates the first day of week. 0 = system default, 1 = Sunday, 2 = Monday etc. It has to be in the range from 0 to 7.

**Return values**

String

String value with the name of the specified day of week.

**Properties**

---

**Value**

Sets or gets the wrapped date/time value hold by the object.

**Syntax**

```
fltDate = dateWrapper.Value  
dateWrapper.Value = fltDate  
dateWrapper.Value = strDate  
dateWrapper.Value = date( 2004, 8, 31)
```

**Parameters**

fltDate

Float representation of the date/time value, representing a date between January 1, 100 and December 31, 9999, inclusive. The value 2.0 represents January 1, 1900; 3.0 represents January 2, 1900, and so on. Adding 1 to the value increments the date by a

day. The fractional part of the value represents the time of day. Therefore, 2.5 represents noon on January 1, 1900; 3.25 represents 6:00 A.M. on January 2, 1900, and so on. Negative numbers represent the dates prior to December 30, 1899.

`strDate`

String date/time representation according to one of standard format for the current locale. If VbScriptXtra failed to recognize date/time from the specified string, it sets the last error flag for the date wrapper object.

Lingo date/time value

Wrapper can accept Lingo date/time values.

### **Return values**

Float

Float representation of date/time value.

### **Sample**

Float date/time representation allows simply date/time arithmetic operations. To calculate how much time is between two date/time values, simply calculate a difference between them. The code below calculates how many hours are between `objDate1` and `objDate2`:

```
fltDif = objDate2.Value - objDate1.Value  
fltOneHour = 1.0/24  
nHours = integer( fltDif / fltOneHour )
```

---

### **Year**

Gets the year part of the date/time value.

### **Syntax**

```
nYear = dateWrapper.Year
```

### **Return values**

Integer

Integer year part of the date/time value.

---

### **Month**

Gets the month part of the date/time value.

### **Syntax**

```
nMonth = dateWrapper.Month
```

### **Return values**

Integer

Integer month part of the date/time value. January = 1, February = 2, and so on.

**MonthName**

Gets the month part of the date/time value as a name of the month.

**Syntax**

```
strMonth = dateWrapper.MonthName
```

**Return values**

String

String month part of the date/time value as a name of the month.

---

**Weekday**

Gets the weekday part of the date/time value.

**Syntax**

```
nWeekday = dateWrapper.Weekday
```

**Return values**

Integer

Integer weekday part of the date/time value. Sunday = 0, Monday = 1, and so on.

---

**WeekdayName**

Gets the weekday part of the date/time value as a name of the weekday.

**Syntax**

```
strWeekday = dateWrapper.WeekdayName
```

**Return values**

String

String weekday part of the date/time value as a name of the weekday.

---

**Day**

Gets the day of month part of the date/time value.

**Syntax**

```
nDay = dateWrapper.Day
```

**Return values**

Integer

Integer day of month part of the date/time value.

---

**Minute**

Gets the minutes part of the date/time value.

---

---

**Syntax**

```
nMinute = dateWrapper.Minute
```

**Return values**

Integer

Integer minutes part of the date/time value.

---

**Second**

Gets the seconds part of the date/time value.

**Syntax**

```
nSecond = dateWrapper.Second
```

**Return values**

Integer

Integer seconds part of the date/time value.

---

**Millisecond**

Gets the milliseconds part of the date/time value.

**Syntax**

```
nMilliseconds = dateWrapper.Milliseconds
```

**Return values**

Integer

Integer milliseconds part of the date/time value.

---

**Local**

Gets the date/time value shifted to the local time zone.

**Syntax**

```
objLocalTime = dateWrapper.Local
```

**Return values**

Date/time wrapper

New date/time wrapper that holds the date/time value shifted from universal time to local time.

**Remarks**

This property allows getting user's local time from universal time. It treats currently hold time as universal. So it calculates time shift between universal time zone and user's time zone and adds this difference to the time value. New date/time value is returned as a new date/time wrapper object.

## Universal

Gets the date/time value shifted to the universal time zone.

### Syntax

```
objUniversalTime = dateWrapper.Universal
```

### Return values

Date/time wrapper

New date/time wrapper that holds the date/time value shifted from local time to universal time.

### Remarks

This property allows getting universal time from user's local time. It treats currently hold time as user's local time. So it calculates time shift between universal time zone and user's time zone and subtracts this difference from the time value. New date/time value is returned as a new date/time wrapper object.

## Registry Key Wrapper

Registry key wrapper is provided by VbScriptXtra for handling operations with system registry. It is useful for checking some installation details of ActiveX or OLE objects. Also it can be used for storing user's preferences in system registry.

To explicitly create this wrapper use xtra-level [CreateWrapper](#) method:

```
registryKeyWrapper = xtra("VbScriptXtra").CreateWrapper( #RegistryKey )
```

Newly created Registry key wrapper is not initialized. It has to be [opened](#) first.

Once registry key is opened, its named values are available either by name or by index via array-like syntax or [GetAt](#) and [SetAt](#) methods. Names of values that belong to the registry keys are available through [ValueNames](#) property.

To open subordinate key use [OpenSubKey](#) method. Names of subordinate keys are available through [SubKeyNames](#) property.

## Methods

---

### Interface()

Returns a short description of what you can do with this wrapper

#### Syntax

```
put registryKeyWrapper.Interface()
```

#### Return values

String

String value with short description of methods and properties provided by this wrapper

---

### Open( strParent, strName, symAccessType, bCreate )

Opens the specified registry key using requested access type.

#### Syntax

```
registryKeyWrapper.Open(
    String strParent,
    String strSubKeyName,
    Optional Symbol symAccessType,
    Optional Boolean bCreateIfMissing )
```

#### Parameters

strParent

String name of the basic root registry keys or another Registry key wrapper to serve as parent key for the required subordinate key. It could be one of the following:

Value	Meaning
"HKEY_CLASSES_ROOT"	Basic types (or classes) of documents and the properties associated with those types.
"HKEY_CURRENT_USER"	Preferences of the current user.

Value	Meaning
"HKEY_LOCAL_MACHINE"	Physical state of the computer and installed hardware and software.
"HKEY_USERS"	Default user configuration for new users on the local computer and the user configuration for the current user.
"HKEY_CURRENT_CONFIG"	Contains information about the current hardware profile of the local computer system.

strSubKeyName

String with the path and name of the specific key to open.

symAccessType

Optional. Symbol or Integer value indicating the type of access requested. This value could be one of the following:

Value	Meaning
#KEY_ALL_ACCESS	All types of access is requested.
#KEY_READ	Reading operations are requested.
#KEY_WRITE	Writing operations are requested
Integer value	Bitwise mask of requested operations.

If this parameter is missed or set to VOID, #KEY\_ALL\_ACCESS is used.

bCreateIfMissing

Optional. If true and the requested key is not found then this method will try to create the specified key. By default it is true unless symAccessType is set to #KEY\_READ.

### Return values

VOID

Does not return anything.

### Error codes

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. Some of them are:

Name	Value	Meaning
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_ACCESS_DENIED	5	Try using #KEY_READ as symAccessType.
ERROR_INVALID_HANDLE	6	Wrapper key used as parent is not a valid registry key handle.

### Sample

The sample looks for applications that start with every system boot right after user login.

```
vb = xtra("VbScriptXtra")
-- Create wrapper
key = vb.CreateWrapper( #RegistryKey )
-- Open it
key.Open( "HKEY_LOCAL_MACHINE", \
```

```

        "SOFTWARE\Microsoft\Windows\CurrentVersion\Run", \
        #KEY_READ )

if key.Failed then exit

-- Iterate through values of the key
repeat with i = 1 to key.count
    put key[i]
end repeat

```

---

### OpenSubKey( **strName**, **symAccessType**, **bCreate** )

Opens the specified subordinate registry key using requested access type and returns it within new Registry key wrapper.

#### Syntax

```

subKey = registryKeyWrapper.OpenSubKey(
    String strSubKeyName,
    Optional Symbol symAccessType,
    Optional Boolean bCreateIfMissing )

```

#### Parameters

`strSubKeyName`

String with the path and name of the specific key to open.

`symAccessType`

Optional. Symbol or Integer value indicating the type of access requested. This value could be one of the following:

Value	Meaning
#KEY_ALL_ACCESS	All types of access is requested.
#KEY_READ	Reading operations are requested.
#KEY_WRITE	Writing operations are requested
Integer value	Bitwise mask of requested operations.

If this parameter is missed or set to VOID, #KEY\_ALL\_ACCESS is used.

`bCreateIfMissing`

Optional. If `true` and the requested key is not found then this method will try to create the specified key. By default it is true unless `symAccessType` is set to #KEY\_READ.

#### Return values

Registry key wrapper

New Registry key wrapper that holds the requested subordinate registry key.

#### Error codes

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. Some of them are:

Name	Value	Meaning
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_ACCESS_DENIED	5	Try using #KEY_READ as <code>symAccessType</code> .

Name	Value	Meaning
ERROR_INVALID_HANDLE	6	Wrapper is not opened.

### Sample

The sample looks shows extensions and their master keys.

```

vb = xtra("VbScriptXtra")
-- Create wrapper
key = vb.CreateWrapper( #RegistryKey )
-- Open it
key.Open( "HKEY_CLASSES_ROOT", "", #KEY_READ )
if key.Failed then exit
-- Iterate through subkeys of the key
lstSubKeyNames = key.SubKeyNames
repeat with i = 1 to lstSubKeyNames.count
    -- Check for the first dot '.'
    if lstSubKeyNames[i].char[1] = "." then
        subKey = key.OpenSubKey(lstSubKeyNames[i], #KEY_READ )
        if key.succeeded then
            defaultValue = subKey.value
            if subKey.succeeded then
                -- Output the key name and its default value
                put lstSubKeyNames[i] & ": " & subKey.value
            else if subKey.LastErrorCode = 2 then
                -- There is no default value
                put lstSubKeyNames[i] & ": " & "<No value>"
            else
                -- Something is wrong
                put lstSubKeyNames[i] & ": " & subKey.LastError
            end if
        end if
    end if
end if
end repeat

```

---

### CreateSubKey( strSubKeyName )

Creates the specified subordinate registry key and returns it within new Registry key wrapper.

#### Syntax

```

subKey = registryKeyWrapper.CreateSubKey(
    String strSubKeyName )

```

#### Parameters

strSubKeyName

String with the path and name of the specific key to create.

#### Return values

Registry key wrapper

New Registry key wrapper that holds the requested subordinate registry key.

## Error codes

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. Some of them are:

Name	Value	Meaning
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_ACCESS_DENIED	5	You are not allowed to perform the operation.
ERROR_INVALID_HANDLE	6	Wrapper is not opened.

---

## DeleteSubKey( strSubKeyName )

Deletes the specified subordinate registry key.

### Syntax

```
registryKeyWrapper.DeleteSubKey(
    String strSubKeyName )
```

### Parameters

strSubKeyName

String with the path and name of the specific key to delete.

### Return values

VOID

Does not return anything.

## Error codes

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. Some of them are:

Name	Value	Meaning
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_ACCESS_DENIED	5	You are not allowed to perform the operation.
ERROR_INVALID_HANDLE	6	Wrapper is not opened.

---

## DeleteValue( strValueName )

Deletes the specified named value of the registry key.

### Syntax

```
registryKeyWrapper.DeleteValue(
    String strValueName )
```

### Parameters

strValueName

String with the name of the specific value to delete.

**Return values**

VOID

Does not return anything.

**Error codes**

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. Some of them are:

Name	Value	Meaning
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_ACCESS_DENIED	5	You are not allowed to perform the operation.
ERROR_INVALID_HANDLE	6	Wrapper is not opened.

**GetAt( Index )**

Gets the requested named value of the key.

**Syntax**

```
val = registryKeyWrapper.GetAt( Index )
```

```
val = registryKeyWrapper[ Index ]
```

**Parameters**

Integer Index

One-based index of the value to retrieve. Integer value in the range from 1 to the number of values is expected.

String Index

The name of the value to retrieve.

**Return values**

Returns the value of the key depending on the data type stored in registry. The table below shows mapping between registry types and Lingo types for the typecasting operation of the Registry key wrapper:

Registry type	Lingo type
REG_SZ, REG_EXPAND_SZ	String
REG_DWORD	Integer
REG_BINARY	<a href="#">Binary data wrapper</a>
Other registry data types	<a href="#">Binary data wrapper</a>

**Error codes**

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API and the wrapper itself. Some of them are:

Name	Value	Meaning
E_VB_CANNOT_HANDLE_DATA_TYPE	-2129330175	VbScriptXtra cannot handle this data type.

ERROR_NO_MORE_ITEMS	259	No more data is available (probably index is out of bounds).
ERROR_FILE_NOT_FOUND	2	The requested key is not found.
ERROR_INVALID_HANDLE	6	Wrapper is not opened.

---

### SetAt( Index, Value, symType )

Sets the requested value of the key to a new value.

#### Syntax

```
registryKeyWrapper.SetAt( Index, Value, Optional Symbol symType )
registryKeyWrapper[ Index ] = Value
```

#### Parameters

Integer Index

One-based index of the value to retrieve. Integer value in the range from 1 to the number of values is expected.

String Index

The name of the value to set.

Value

New value to be set at the specified key's value. The table below shows default mapping between registry types and Lingo types for the typecasting operation of the Registry key wrapper:

Lingo type	Registry type
String	REG_SZ
Integer	REG_DWORD
<a href="#">Binary data wrapper</a>	REG_BINARY

If the `symType` parameter is specified it overrides the default settings.

`symType`

The type of data to be set for a value. It can be one of the following values:

Lingo value	Meaning
#REG_SZ	Normal string
#REG_EXPAND_SZ	String that contains unexpanded references to environment variables (for example, "%PATH%").
#REG_BINARY	Binary data in any form
#REG_DWORD	Integer 32-bit value
Integer value	See MSDN for other possible values

#### Remarks

This method has some specific behavior while handling which registry data type to assign to the value. The default behavior is applied if `symType` argument is not specified.

**Default behavior:** Method checks whether the requested value is already exists. If so, it reads its registry data type and writes new value over existing data keeping the data type unchanged. If the value is the new one, method uses default registry data types according to the type of Lingo data passed to the method, as described in the `Value` parameter above.

Default behavior take place with any way of calling this method, unless `symType` is specified.

**Note:** This method is called internally by both Director and other wrapper properties. All statements below cause this method to be called with `symType` omitted:

```
registryKeyWrapper[ Index ] = newValue
registryKeyWrapper.Value[ Index ] = newValue
registryKeyWrapper.Value = newValue
```

If `symType` is specified, its value is used as registry data type in any case.

Default behavior allows making most operations without worrying about data types of registry values. If existing values have to be overwritten, their data types stay unchanged. New values are added with correct data type based on Lingo value passed to the method.

## Properties

---

### Count

Returns the number of named values for the wrapped registry key.

#### Syntax

```
nCount = registryKeyWrapper.Count
nCount = registryKeyWrapper.Value.Count
```

#### Return values

Integer

Number of named values for the wrapped registry key.

---

### Value

Sets or gets the default value for the wrapped registry key.

#### Syntax

```
defaultValue = registryKeyWrapper.Value
registryKeyWrapper.Value = defaultValue
```

#### Remarks

It is the same as using [registryKeyWrapper.GetAt\(""\)](#) for reading default value and [registryKeyWrapper.SetAt\("", val\)](#) for writing default value.

---

### Value[ index ]

Sets or gets the specified value for the wrapped registry key.

**Syntax**

```
value = registryKeyWrapper.Value[ index ]
value = registryKeyWrapper[ index ]
registryKeyWrapper.Value[ index ] = newValue
registryKeyWrapper[ index ] = newValue
```

**Parameters**

Integer Index

One-based index of the value to retrieve. Integer value in the range from 1 to the number of values is expected.

String Index

The name of the value to retrieve.

**Remarks**

It is the same as using [registryKeyWrapper.GetAt\(index\)](#) for reading specified value and [registryKeyWrapper.SetAt\(index, newValue\)](#) for writing specified value.

Using the property without index returns the value of the default key value (if any).

**ValueType[ index ]**

Returns the registry type of the specified value for the wrapped registry key.

**Syntax**

```
type = registryKeyWrapper.ValueType[ index ]
type = registryKeyWrapper.ValueType
```

**Parameters**

Integer Index

One-based index of the value to retrieve. Integer value in the range from 1 to the number of values is expected.

String Index

The name of the value to retrieve.

**Parameters**

Symbol

One of symbols below that describes the data type of the value.

Lingo Value	Meaning
#REG_SZ	Normal string
#REG_EXPAND_SZ	String that contains unexpanded references to environment variables (for example, "%PATH%").
#REG_BINARY	Binary data in any form

#REG_DWORD	Integer 32-bit value
------------	----------------------

Integer

Integer value for rarely used registry data types. See MSDN for details.

### Remarks

Using the property without index returns the type of the default key value (if any).

---

### ValueNames

Returns the list of value names for the wrapped registry key.

### Syntax

```
lstValueNames = registryKeyWrapper.ValueNames
```

### Return values

Linear list

A Lingo list with names of values for the wrapped registry key.

---

### SubKeyNames

Returns the list of subordinate key names for the wrapped registry key.

### Syntax

```
lstSubKeyNames = registryKeyWrapper.SubKeyNames
```

### Return values

Linear list

A Lingo list with names of subordinate keys for the wrapped registry key.

## System Helper Object

VbScriptXtra starting with version 2.2 provides the special helper object with some system-level routines.

To create this object use xtra-level [CreateWrapper](#) method:

```
systemObject = xtra("VbScriptXtra").CreateWrapper( #System )
```

Use [GetRunningObjectTable\(\)](#) method to get access to all running object known to the system that support Automation.

[ProcessId](#) property together with WMI scripting allows getting detailed memory usage information for the current process. It returns the same values as Windows Task Manager does.

[StageHWND](#) and [AppHWND](#) returns window handles of the stage and the top-level application window.

Use [DefaultLocaleInfo](#) property to find out the current system language and code page. It may be important to know for multilingual applications.

Switching keyboard layout is available with [ActivateKeyboardLayout](#) method and [KeyboardLayoutList](#) property. Current keyboard layout is available via [KeyboardLayout](#) property.

It is possible to load or unload specific keyboard layouts with [LoadKeyboardLayout](#) and [UnloadKeyboardLayout](#) methods.

## Methods

---

### Interface()

Returns a short description of what you can do with this wrapper

#### Syntax

```
put systemObject.Interface()
```

#### Return values

String

String value with short description of methods and properties provided by this wrapper

---

### GetRunningObjectTable()

Returns the list with currently running automation objects known to the system.

#### Syntax

```
lstObjects = systemObject.GetRunningObjectTable()
```

```
lst = sys.GetRunningObjectTable( strDisplayNameFilter, strNameFilter )
```

## Parameters

`strDisplayNameFilter`

String that is used as filter of the resulting list. Only those entries that contain this string (case insensitive) in display name will be placed into resulted list. This parameter is optional, if `VOID` or empty string is used it does not affect the result.

`strNameFilter`

String that is used as filter of the resulting list. Only those entries that contain this string (case insensitive) in the interface name will be placed into resulted list. This parameter is optional, if `VOID` or empty string is used it does not affect the result.

## Return values

`List`

List of property lists for every entry of the running object table that supports Automation and matches the specified filters.

## Remarks

Every matching entry of the table is described with the property list as shown below.

Symbol	Meaning
<code>#DisplayName</code>	Contains the display name of the object. It may be path to the open document or GUID of the application or something else.
<code>#Name</code>	Contains the automation interface name preceded by the name of the object's type library (separated by dot).
<code>#Object</code>	Contains the <a href="#">Automation</a> wrapper with the object from the table.

Normally Office applications register itself in the running object table (ROT). If the application has been run more than once, the running object table will contain multiple entries for this application. Every opened document is normally also registered in the ROT.

This method allows connecting to all running applications that support automation while [GetObject](#) method can only return the last active one.

It is recommended to use filters since this method will try to create wrapper objects for all detected applications and documents going to the resulting list. It may takes noticeable delay (several seconds for the first call) since the xtra needs to scan type libraries of every newly detected object.

## Sample

The sample looks for currently loaded Word documents:

```

vb = xtra("VbScriptXtra")
-- Create System object
sys = xtra("VbScriptXtra").CreateWrapper(#System)
put sys.GetRunningObjectTable( ".doc" )
-- [[#DisplayName: "E:\...\VbScriptXtra.doc", #name: "Word._Document",
#object: < VbScriptXtra, Word._Document, 0x0023911C, (2) >]]

```

Call it unfiltered to see what other applications and documents are contained in the ROT:

```

put sys.GetRunningObjectTable()

```

---

**LoadKeyboardLayout( strLayoutName )**

Adds the specified layout to the list of available keyboard layouts.

**Syntax**

```
hkl = sys.LoadKeyboardLayout( String strLayoutName )  
hkl = sys.LoadKeyboardLayout( Integer nLanguageId,  
                               Optional Integer nLayoutId = 0 )
```

**Parameters**

strLayoutName

String with the keyboard layout name to be loaded. It can be "00000409" for U.S. English, "00010409" for U.S. English (Dvorak) and so on.

nLanguageId

Integer language identifier. See MSDN for possible [language identifiers](#).

nLayoutId

Integer identifier of the physical keyboard layout. By default it is zero.

**Return values**

hkl

Integer value that identifies loaded keyboard layout. Use it in further calls to [ActivateKeyboardLayout](#) or [UnloadKeyboardLayout](#).

**Error codes**

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. It may fail if requested layout is not found or it cannot be loaded due to security reasons.

**Remarks**

Layout name is actually a hex string representation of the Language Identifier and the ordinal number of specific physical keyboard layout. The second form of this method simply combines nLanguageId and nLayoutId into proper hexadecimal string.

**Note:** loaded keyboard layout appears in the list of available keyboard layouts and stays there until it is unloaded.

---

**UnloadKeyboardLayout( hkl )**

Removes the specified layout from the list of available keyboard layouts.

**Syntax**

```
sys.UnloadKeyboardLayout( Integer hkl )
```

**Parameters**

hkl

Integer value returned by previous call to [LoadKeyboardLayout](#) or [KeyboardLayoutList](#).

---

## Return values

hkl

If successful it is an integer value that identifies the previous keyboard layout. Otherwise it is zero.

---

## ActivateKeyboardLayout( hkl )

Activates the specified layout for the current Projector or Director thread.

### Syntax

```
sys.ActivateKeyboardLayout( Integer hkl )
```

```
sys.ActivateKeyboardLayout( Integer nLanguageId )
```

### Parameters

hkl

Integer value returned by previous call to `LoadKeyboardLayout` or `KeyboardLayoutList`. Use 0 to activate the previous layout in the list and 1 to activate the next one.

nLanguageId

Integer language identifier. See MSDN for possible [language identifiers](#).

### Error codes

The method may return several useful error codes through [wrapper.LastErrorCode](#) property. Error codes are coming from Win32 API. It may fail if requested layout is not found or it cannot be unloaded due to security reasons.

### Remarks

If the list of available keyboard layouts contains layout for the requested language, it will be activated. There could be several keyboard layouts for the specific language, in this case use hkl value to specify the one that is needed. Use `KeyboardLayoutList` property to get the list of available layouts and their hkl handles.

## Properties

---

### ProcessId

Returns the identifier of the running process.

### Syntax

```
nProcessId = systemObject.ProcessId
```

### Return values

Integer

Identifier of the Director or Projector process that calls this property.

---

**Sample**

The sample looks for memory usage values of the calling process. These are the same value as Windows Task Manager shows as 'VM Size' and 'Mem usage'

```
vb = xtra("VbScriptXtra")

-- Get WMI root
ObjSet = xtra("VbScriptXtra").GetObject2("WinMgmts:root\cimv2", "")

-- Create System object
sys = xtra("VbScriptXtra").CreateWrapper(#System)

strQuery = "Select * from Win32_Process where " & \
           "ProcessId = " & sys.ProcessId & ""

colItems = ObjSet.ExecQuery( strQuery )
appInfo = colItems.__NewEnum[1]

put appInfo.PageFileUsage
-- 31289344

put integer(appInfo.WorkingSetSize)
-- 20926464
```

---

**StageHWND**

Returns the handle of the stage window.

**Syntax**

```
hwndStage = systemObject.StageHWND
```

**Return values**

Integer

Windows handle (HWND) of the stage window.

---

**AppHWND**

Returns the handle of the top-level application window.

**Syntax**

```
hwndApp = systemObject.AppHWND
```

**Return values**

Integer

Windows handle (HWND) of the top-level application window.

---

**KeyboardLayout**

Returns the property list with active keyboard layout details.

**Syntax**

```
put sys.KeyboardLayout
-- [#LayoutName: "00000409", #LanguageId: 1033, #PhysicalLayout: 1033,
#HKL: 67699721]
```

---

**Return values**

Property list

Currently active keyboard layout details. See MSDN for possible [language identifiers](#). #HKL property contains the handle of the layout. It may be used to identify the layout in later call of [ActivateKeyboardLayout](#) method.

---

**KeyboardLayoutList**

Returns the linear list with currently available keyboard layouts. Every list entry contains a property list with layout details.

**Syntax**

```
put sys.KeyboardLayoutList[1]
-- [#LanguageIdHex: "0409", #LanguageId: 1033, #PhysicalLayout: 1033,
#HKL: 67699721]
```

**Return values**

Linear list

Currently available keyboard layouts details. See MSDN for possible [language identifiers](#). #HKL property contains the handle of the layout. It may be used to identify the layout in later call of [ActivateKeyboardLayout](#) method.

---

**DefaultLocaleInfo**

Returns the property list with some system information about the current locale.

**Syntax**

```
put sys.DefaultLocaleInfo
-- [#SystemDefaultLanguageID: 1033, #UserDefaultLanguageID: 1033,
#SystemDefaultLocaleID: 1033, #UserDefaultLocaleID: 1033,
#ANSICodePage: 1250, #OEMCodePage: 0]
```

**Return values**

Property list

Some details about current system settings. See MSDN for possible [language identifiers](#) and [code page identifiers](#).

## Xtra-level methods

### Init( nDebug )

Performs generic xtra initialization. Allows setting debugging mode as a default for newly created wrappers.

### Syntax

```
bSuccess = xtra("VbScriptXtra").Init( Integer nDebug )
```

```
bSuccess = xtra("VbScriptXtra").Init( Symbol symMode )
```

### Parameters

nDebug

Optional. Debugging mode for newly created objects. This parameter can be one of the following values.

Value	Meaning
0	No debugging support. Release behavior.
1	Simple debugging. Any error is automatically printed in Messages window.
2	Advanced debugging. When any error is occurred, the xtra calls movie level handler <code>VbScriptXtra_DebugEvent( strMes, nCode )</code> .

symMode

Optional. Default mode for newly created objects. This parameter can be one of the following values.

Value	Meaning
#JSMODE	Used for better integration with JS scripts. It affects typecasting routines making them returning JS Null instead of Lingo symbol #Null or Lingo VOID
#UnicodeAsBinaryMode	Used for fine control over Unicode strings returned by COM Automation objects. If this mode is used, typecasting routines won't try to convert the text into MBCS. Instead they will wrap it with Binary wrapper.

### Return values

If the method succeeds, it returns true. Otherwise it returns false.

### Remarks

See [error handling](#) and [debugging](#) support for more details about debugging modes.

**Note:** This method does not affect objects that already exist at the time of calling this method. You may use [DebugMode](#) property to change the debugging mode of the particular object directly. Use [UnicodeAsBinaryMode](#) property to change the Unicode conversion mode of the particular object directly.

---

**CreateObject( strProgId, strServer )**

Creates a new instance of the object specified by its ProgId.

**Syntax**

```
objAuto = xtra("VbScriptXtra").CreateObject(  
    String strProgId,  
    Optional String strServer )
```

**Parameters**

strProgId

ProgId string value specifying which object to create. For example, use "Word.Application" to create a new instance of Microsoft Word.

strServer

Name or IP address of remote computer where specified object has to be created via DCOM (Distributed COM).

**Return values**

Object

If object is created successfully the method returns the new instance of VbScriptXtra wrapper object, that holds newly created automation object of requested ProgId.

String

If the xtra failed to create requested object the method returns a string with error description.

**Remarks**

This method is an analogue to Visual Basic's CreateObject. It is used to create new automation objects. Therefore, it is a main entry point while using VbScriptXtra. Usual usage scenario starts from calling this method to create an instance of external application or another automation object and then operating on that object.

Specifying remote computer name with strServer parameter allows remote object creation. To make this to work server should be allowed for accepting DCOM connections and specified object has to be properly configured. Normally it requires calling user to have create or launch permissions for the specified object. See dcomcnfg.exe utility that serves for configuring DCOM features.

**Sample**

This sample creates an instance of Internet Explorer and then makes it to navigate to www.xtramania.com, then shows it in full screen mode without menu bar, toolbar, status bar. Then it waits while downloading is still in progress and when it is done IE appears to a user.

```
on navigate  
    vb = xtra("VbScriptXtra")  
  
    -- Creating a new instance of Internet Explorer  
    ie = vb.CreateObject("InternetExplorer.Application" )
```

```
-- Navigate to URL
ie.navigate("www.XtraMania.com")

-- ie is hidden now, making it fullscreen
ie.fullscreen=true

-- hiding extra interface elements
ie.menuBar=false
ie.toolbar=false
ie.statusbar=false

-- Waiting while page is complete
repeat while ie.busy
  put "Waiting for IE"
end repeat

-- Here we are, ready and fullscreen!
ie.visible=true

put "Ops!"
end
```

---

## GetObject( strProgId )

Gets a running current instance of the object specified by its ProgId.

There are three semantics of calling this method in VB. This xtra-level method represents the following VB statement:

```
set obj = GetObject( , strProgId )
```

Comma before strProgId means that the first parameter of the method is missed.

### Syntax

```
objAuto = xtra("VbScriptXtra").GetObject( String strProgId )
```

### Parameters

strProgId

ProgId string value specifying which object to get. For example, use "Word.Application" to get a currently running instance of Microsoft Word.

### Return values

Object

If object is got successfully the method returns the new instance of VbScriptXtra wrapper object that holds currently active automation object of requested ProgId.

String

If the xtra failed to get requested object the method returns a string with error description.

### Remarks

This method will not try to create a new instance of requested object if there is no active one. So it can be used to check whether there is a running application of requested ProgId or not.

GetObject method is based on the running object table (ROT) used by the system to keep track of the running object. The last registered object of the specified type is returned by

this method. When there are several similar objects running, `GetObject` will only return the last one made active. VbScriptXtra's [System](#) object provides an access to the ROT itself, so it is possible to get all running object and handle them as needed, See [GetRunningObjectTable](#) method for more details and samples.

## Sample

This sample checks whether there is an instance of Microsoft Word running. If so it attaches to it and returns, otherwise it creates a new Microsoft Word instance.

```
on GetWordApplication
  vb = xtra( "VbScriptXtra" )

  -- Creating a new instance of Microsoft Word
  w = vb.GetObject( "Word.Application" )

  if not objectP(w) then
    w = vb.CreateObject( "Word.Application" )
  end if

  return w
end
```

---

## GetObject2( strPath, strProgId )

Gets the object from a combination of file and a ProgId.

There are three semantics of calling this method in VB. This xtra-level method represents two following VB statement:

```
set obj = GetObject( strPath, strProgId )
set obj = GetObject( strPath )
```

The first statement is used to create an instance of the specified ProgId and then make it to load the specified file.

The second statement is used to create an object either from file or from user friendly name of the object and/or its items.

## Syntax

```
ObjAuto = xtra("VbScriptXtra").GetObject2(
  String strPath,
  String strProgId)
```

## Parameters

`strPath`

Path to a file that has to be loaded or other object specification.

`strProgId`

ProgId string value specifying which object to get. For example, use "Word.Document" to get a Microsoft Word document loaded from the specified file.

## Return values

Object

If object is got successfully the method returns the new instance of VbScriptXtra wrapper object that holds currently active automation object of requested ProgId.

String

If the xtra failed to get requested object the method returns a string with error description.

### Sample

This sample checks whether there is an instance of Microsoft Word running. If so it attaches to it and returns, otherwise it creates a new Microsoft Word instance.

```
on GetWordDocument
  vb = xtra( "VbScriptXtra" )

  -- Getting a Word document from file
  doc = vb.GetObject2( "D:\file.doc", "Word.Document" )

  doc.Application.visible = true

  return doc
end
```

See [WMI scripting](#) sample that demonstrates the usage of GetObject2 method.

---

## Version( )

### Syntax

```
strVersion = xtra("VbScriptXtra").Version()
```

### Return values

String

Version string in a form of 5 point delimited items: "VbScriptXtra.2.1.1.71".

The first item is the xtra's name "VbScriptXtra".

The second item is the major xtra's version.

The third item is the subversion number. It indicates noticeable changes.

The forth item is the minor version number. It indicates minor changes.

The last item is the absolute build number. It is auto incremented with every release build of the xtra.

---

## SetBusyHandler( objHandler )

Sets the xtra's level handler of "Busy" states.

Busy state can happen when external application cannot handle Automation requests from its clients. For example, Microsoft Excel cannot respond to Automation commands while it shows an Open/Save dialog to a user. Normally application responds to calling client "I am busy, retry later". Client behavior could decide what to do: either just wait for some time or inform user that external application is busy and probably it waits for user input. This notification is normally done via system level standard busy state dialog.

Another key point with busy states happens when user tired of waiting for something and starts clicking mouse and pressing keys. COM allows detecting these events from user and asks client application what to do either wait further, or show busy state notification or terminate the call to the server.

So this method allows customizing how to handle busy states. See remarks section for further details.

**Syntax**

```
bSucceeded = xtra("VbScriptXtra").SetBusyHandler(
    Object objHandler )
```

**Parameters**

objHandler - Parent script instance or VOID

An instance of a parent script that will handle the "Busy" states or Void to reset the handler to xtra's default state.

**Return values**

true

If succeeded.

false

Otherwise.

**Remarks**

This method sets the instance of a parent script as a "busy state" handle, which is called when xtra detects busy condition.

When COM server returns the busy state, xtra calls a method of a handler instance:

```
on ServerBusy me, milliseconds
```

Where milliseconds indicate the time elapsed since the initial call of the method. The handler can return:

Value	Meaning
-1	Cancels the call, it most probably produces a Lingo error.
0	Immediately retry the call.
1	Displays system "Server Busy" dialog.
> 1	Retry the call in the specified number of milliseconds.

While call to COM server is in progress, xtra can detect whether any messages are waiting to be processed by Director Application. If so, it calls another method of a handler instance:

```
on MessagePending me, milliseconds, category
```

Where milliseconds indicate a time elapsed since initial call to the server, category indicates the type of waiting message. Category can be one of the following values.

Value	Meaning
0	Menu or other Windows-level message.
1	Keyboard message.
2	Mouse message.

This handler should return either 0 (false) to do nothing or 1(true) to display a busy dialog.

To use xtra's default handler set it to void:

```
xtra("VbScriptXtra").SetBusyHandler( void )
```

The xtra's default behavior is to wait 10 seconds on `ServerBusy` by retrying call every 200 milliseconds. Then xtra shows the "Busy" dialog. If important messages (system-level, mouse, keyboard) are detected while COM call is in progress for more than 3 seconds, the xtra also shows "Busy" dialog.

### Sample

This sample shows how custom "Busy" state handler could look like. To make it the xtra's handler use:

```
objHandler = script("BusyHandler").new()
bSucceeded = xtra("VbScriptXtra").SetBusyHandler( objHandler )
```

This sample handler waits for 25 seconds while server does not respond and then informs user via system level busy state dialog.

If any incoming message is received while waiting for server to respond, handler checks the type of message. If it is windows-level message, it informs user, otherwise puts to the Messages what is happening.

```
-- Parent script
on new me
  return me
end

-- Called when server application returned a busy status
on ServerBusy me, milliseconds
  if milliseconds > 25000 then
    return 1
  end if

  return 1000
end

-- Called when messages come to Director
on MessagePending me, milliseconds, category
  case category of
    0:
      put "Menu or other Windows-level message"
      return 1
    1:
      put "KeyPressed"
      return 0
    2:
      put "Mouse:" && the mouseLoc
      return 0
  end case

  return 0
end
```

---

### CreateWrapper( symWrapperType )

Creates a new instance of VbScriptXtra wrapper for the requested custom data type.

### Syntax

```
obj = xtra("VbScriptXtra").CreateWrapper( Symbol symType )
```

## Parameters

symType

Indicates which wrapper to create. This parameter can be one of the following values.

Value	Meaning
#Date	The wrapper for VB <a href="#">Date/Time</a> values.
#Binary	The wrapper for <a href="#">binary data</a> including BLOB. Previous versions of VbScriptXtra used BinaryXtra that implemented this kind of wrappers. VbScriptXtra v2 includes binary data wrapper.
#UnicodeString	The <a href="#">Binary</a> wrapper with <a href="#">ContentsType</a> set to #UnicodeString
#MBCSString	The <a href="#">Binary</a> wrapper with <a href="#">ContentsType</a> set to #MBCSString
#Image	The <a href="#">Image</a> wrapper object.
#RegistryKey	The wrapper for <a href="#">Registry keys</a> .
#System	The <a href="#">system</a> helper object.
#TypeLib	The <a href="#">Type Library</a> wrapper.

## Return values

Object

A newly created instance of the requested data type or

String

String value with error description.

## Remarks

Normally these wrappers are created by VbScriptXtra internally when it detects the data of specific type. For example, when getting a value from database field of type BLOB, VbScriptXtra returns to Lingo a binary data wrapped to the wrapper instance of type #Binary.

To store BLOB data into database, you should create a Binary wrapper first, then fulfill it with a data and then assign the wrapper to the respective field of a recordset object (ADO).

Image wrapper is useful as an image data exchange helper between Lingo Image objects and COM Automation components.

Date/Time wrapper is useful for handling Date/Time values. It offers a lot of functionality of date/time values representation. It is created automatically when VbScriptXtra detects a data of type Date/Time.

Registry key wrapper is used for handling operations with system Registry.

System helper object contains several useful system-level methods and properties.

Type Library wrapper is used for supporting IUnknown-based Automation compliant COM objects.