



Scripter Studio Manual

September, 2011
Copyright (c) 2010 by tmssoftware.com bvba
Web: <http://www.tmssoftware.com>
E-mail: info@tmssoftware.com

Table of Contents

| | |
|--|-----------|
| Chapter I Introduction | 5 |
| 1 Overview | 5 |
| 2 Copyright Notice | 5 |
| 3 What's New | 6 |
| 4 Manual Installation | 13 |
| 5 Getting Support | 15 |
| 6 Version 4.5: News about RTTI | 15 |
| Registering a class in scripter | 16 |
| Registering a record in scripter | 16 |
| What is not supported | 17 |
| | |
| Chapter II Language Features | 19 |
| 1 Pascal syntax | 19 |
| Overview | 19 |
| Script structure | 19 |
| Identifiers | 20 |
| Assign statements | 20 |
| Character strings | 20 |
| Comments | 20 |
| Variables | 21 |
| Indexes | 21 |
| Arrays | 21 |
| If statements | 22 |
| while statements | 22 |
| repeat statements | 22 |
| for statements | 23 |
| case statements | 23 |
| function and procedure declaration | 23 |
| 2 Basic syntax | 24 |
| Overview | 24 |
| Script structure | 24 |
| Identifiers | 25 |
| Assign statements | 25 |
| New statement | 25 |
| Character strings | 25 |
| Comments | 26 |
| Variables | 26 |
| Indexes | 27 |
| Arrays | 27 |
| If statements | 27 |
| while statements | 27 |
| loop statements | 28 |
| for statements | 28 |
| select case statements | 29 |
| function and sub declaration | 29 |

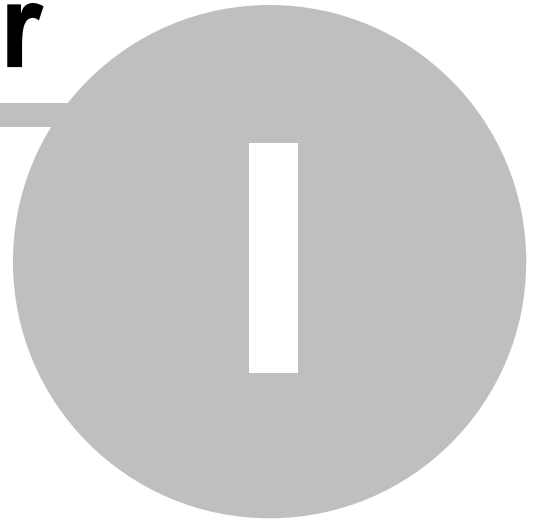
| | |
|--------------------------------------|-----------|
| 3 Calling dll functions | 30 |
| Overview | 30 |
| Pascal syntax | 30 |
| Basic syntax | 31 |
| Supported types | 31 |

Chapter III Working with scripiter 33

| | |
|---|-----------|
| 1 Getting started | 33 |
| 2 Cross-language feature: TatScripiter and TIDEScripiter | 33 |
| 3 Common tasks | 34 |
| Calling a subroutine in script | 34 |
| Returning a value from script | 35 |
| Passing parameters to script | 35 |
| 4 Accessing Delphi objects | 36 |
| Registering Delphi components | 36 |
| Access to published properties | 37 |
| Class registering structure | 37 |
| Calling methods | 37 |
| More method calling examples | 38 |
| Accessing non-published properties | 39 |
| Registering indexed properties | 39 |
| Retrieving name of called method or property | 40 |
| Registering methods with default parameters | 41 |
| 5 Accessing Delphi functions, variables and constants | 41 |
| Overview | 41 |
| Registering global constants | 41 |
| Accessing global variables | 42 |
| Calling regular functions and procedures | 43 |
| 6 Script-based libraries | 44 |
| 7 Declaring forms in script | 45 |
| 8 Declaring classes in script | 46 |
| 9 Using the Refactor | 47 |
| 10 Using libraries | 48 |
| Overview | 48 |
| Delphi-based libraries | 48 |
| The TatSystemLibrary library | 50 |
| Removing functions from the System library | 52 |
| The TatVBScriptLibrary library | 52 |
| 11 Debugging scripts | 54 |
| Overview | 54 |
| Using methods and properties for debugging | 54 |
| Using debug components | 55 |
| 12 Form-aware scripiter - TatPascalFormScripiter and TatBasicFormScripiter | 55 |
| 13 C++ Builder issues | 55 |
| Overview | 55 |
| Registering a class method for an object | 56 |

| | |
|---|-----------|
| Chapter IV The syntax highlighting memo | 58 |
| 1 Using the memo | 58 |
| | |
| Chapter V TSourceExplorer component | 60 |
| 1 Overview | 60 |
| 2 Using the component | 60 |
| | |
| Chapter VI C++Builder Examples | 63 |
| 1 Working with scripter | 63 |
| Getting started | 63 |
| Cross-language feature: TatScripter and TIDEScripter | 63 |
| Common tasks | 63 |
| Calling a subroutine in script..... | 63 |
| Returning a value from script..... | 63 |
| Passing parameters to script..... | 63 |
| Accessing Delphi objects | 64 |
| Registering Delphi components..... | 64 |
| Calling methods..... | 64 |
| More method calling examples..... | 64 |
| Accessing non-published properties..... | 64 |
| Registering indexed properties..... | 64 |
| Retrieving name of called method or property..... | 65 |
| Registering methods with default parameters..... | 65 |
| Accessing Delphi functions, variables and constants | 66 |
| Registering global constants..... | 66 |
| Accessing global variables..... | 66 |
| Calling regular functions and procedures..... | 66 |
| Using libraries | 67 |
| Delphi-based libraries..... | 67 |
| Removing functions from the System library..... | 67 |

Chapter



Introduction

1 Introduction

1.1 Overview

Scripter Studio and Scripter Studio Pro are a set of Delphi/C++Builder components that add scripting capabilities to your applications. With Scripter Studio your end-user can write his own scripts using visual tools and then execute the scripts with scripter component. Main components available are:

- **TatScripter**: Non-visual component with cross-language support. Executes scripts in both Pascal and Basic syntax.
- **TatPascalScripter**: Non-visual component that executes scripts written in Pascal syntax
- **TatBasicScripter**: Non-visual component that executes scripts written in Basic syntax
- **TatPascalFormScripter**: form-aware pascal scripter descendant from TatPascalScripter
- **TatBasicFormScripter**: form-aware basic scripter descendant from TatBasicScripter
- **TatScriptDebugger**: Dialog component for script debugging
- **TAdvMemo**: Lightweight syntax highlight memo, that can be used to edit scripts at run-time.
- **TSourceExplorer**: Code explorer treeview like the one in Delphi IDE.

TatScripter, TatPascalScripter, TatBasicScripter and (in Scripter Studio Pro version) TIDEScripter (in this document, all of these components are just called Scripter) descend from **TatCustomScripter** component, which has common properties and methods for scripting execution. The scripter has the following main features:

- Run-time Pascal and Basic language interpreter
- Access any Delphi object in script, including properties and methods
- Supports try..except and try..finally blocks in script
- Allows reading/writing of Delphi variables and reading constants in script
- Allows access (reading/writing) script variables from Delphi code
- You can build (from Delphi code) your own classes, with properties and methods, to be used in script
- Most of Delphi system procedures (conversion, date, formatting, string-manipulation) are already included (IntToStr, FormatDateTime, Copy, Delete, etc.)
- You can save/load compiled code, so you don't need to recompile source code every time you want to execute it
- Debugging capabilities (breakpoint, step into, run to cursor, pause, halt, and so on)
- Thread-safe
- COM (Microsoft Common Object Model) Support
- DLL functions calls

1.2 Copyright Notice

Scripter Studio and Scripter Studio Pro components trial versions are free for use in non-commercial applications, that is any software that is not being sold in one or another way or that does not generate income in any way by the use of the application.

For use in commercial applications, you must purchase a single license or a site license of Scripter Studio. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates for a full version cycle and priority email support. A single developer license allows ONE developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A single developer license is NOT transferable to another developer within the

company or to a developer from another company. Both licenses allow royalty free use of the components when used in binary compiled applications.

The component cannot be distributed in any other way except through free accessible Internet Web pages or ftp servers. The component can only be distributed on CD-ROM or other media with written authorization of the author.

Online registration for Scriptor Studio is available at <http://www.tmssoftware.com/orders.htm>. Source code & license is sent immediately upon receipt of check or registration by email. Payment grants users the right for a full version cycle source code updates (from version x.y to x+1.y-1)

Scripter Studio is Copyright © 2002-2011 TMS Software. ALL RIGHTS RESERVED.

No part of this help may be reproduced, stored in any retrieval system, copied or modified, transmitted in any form or by any means electronic or mechanical, including photocopying and recording for purposes others than the purchaser's personal use.

1.3 What's New

version 5.1 (Sep-2011)

- New: Delphi/C++Builder XE2 Support
- New: Delphi XE2 support in import tool
- Improved: Class registration using new RTTI - now also import classes not registered with RegisterClass
- Fixed: Issue with combined indexed default properties
- Fixed: Minor bug when saving compiled code
- Fixed: Import tool now importing published methods

version 5.0 (Apr-2011)

- New: Support for creating [script-based classes](#)
- New: New code insight class supporting parameter hints and improved code completion (to be used in custom IDE's)
- New: Updated import tool to also import parameter hints of methods
- New: Updated imported VCL units for all Delphi versions, now including parameter hints
- New: Additional parameter in DefineMethod allowing to specify the parameter hint for that method
- Improved: Several other improvements added from TAdvMemo 2.3 version (see AdvMemo.pas source code for more info)
- Fixed: Relative paths for script files not working with \$(APPDIR) and \$(CURDIR)

version 4.7.1 (Dec-2010)

- Fixed: Registered version installer not working properly with TMS VCL Subscription Manager.

version 4.7 (Dec-2010)

- New: Updated imported VCL units for all Delphi versions, now including indexed properties, default parameters and other minor tweaks
- Fixed: Issue with getter of boolean properties using DefineClassByRTTI
- Fixed: Issue with TStringList.Create in Delphi XE imported Classes library
- Fixed: Functions with "out" parameters not working in ap_DateUtils
- Fixed: Install conflict between Scriptor and other TMS packages
- Fixed: Instructions to return values for "out" parameters not generated by ImportTool
- Fixed: Issue with enumerated types in ImportTool

version 4.6.0.1 (Oct-2010)

- Improved: Information about CurrentClass in Context parameter for OnUnknownElementEvent event

- Fixed: Issue with InStr function in VB Script Library
- Fixed: Issues installing Scripter Studio on RAD Studio XE

version 4.6 (Sep-2010)

- New: RAD Studio XE Support
- New: Support for default indexed properties in script syntax (e.g. Lines[i] instead of Lines.Strings[i])
- Improved: C++ Builder source code examples included in Scripter manual
- Improved: Import Tool parser is now recognizing most of new Delphi syntax features and provides RAD Studio XE support
- Improved: Options in DefineClassByRTTI method to redefine an already defined class in scripter
- Fixed: Issue with getter of boolean properties
- Fixed: Issue with script executed step by step while watching a variable
- Fixed: Issues with DefineClassByRTTI method (registering of constructor overloads, return of var/out method parameters)
- Fixed: Issue with record declarations in units imported by ImportTool using enhanced RTTI
- Fixed: Issues with code completion (up to Delphi 2005)
- Fixed: Find and Replace in memo didn't work with Match Whole Word Only

version 4.5 (Jul-2010)

- New: [Automatic classes, methods and properties registration using new enhanced RTTI](#) (Delphi 2010 and later).
- New: extensive help component reference.
- New: fully documented source code.
- Fixed: error compiling some imported units in Delphi 2010.
- Fixed: issue with SaveCodeToFile when using form components of a non-registered class.
- Fixed: memory leak when using some rare syntax constructions in script.

version 4.4.6 (Jan-2010)

- New: TatCustomScripter.LoadFormEvents property allows setting event handlers when loading form dfm files saved in Delphi.
- Improved: char constants now accept hexadecimals (#\$0D as an alternative to #13).
- Fixed: VB function MsgBox was displaying incorrect window caption.
- Fixed: VB function Timer was performing wrong calculation with miliseconds.
- Fixed: issue with OnRuntimeError not providing correct source code row and col of error.

version 4.4.5 (Sep-2009)

- New: Delphi/C++ Builder 2010 support.
- New: Array properties supported in COM objects
- Improved: pascal syntax allows "end." (end dot) in main script block
- Improved: AdvMemo files updated to latest versions
- Fixed: issue with try..except and try..finally blocks

version 4.4 (May-2009)

- New: "Private" and "Public" keywords allow defining private global variables, private subs and private functions (not visible from other scripts) in Basic scripts
- New: Variable initialization in Basic scripts (e.g., Dim A as String = "Hello")
- New: Return statement in Basic scripts
- New: If..Then.. statements without "End If" for single line statements (in Basic scripts)
- New: Try..Catch..End Try syntax in addition to Try..Except..End (in Basic scripts)
- New: TCustomScripter.ScriptFormClass allows providing a different class (derived from TScriptForm) for forms created from script
- Improved: when scripter don't find a library, a compile error is raised (instead of an exception)

version 4.3 (Feb-2009)

- New: "new" clause in Basic script. e.g "MyVar = new TLabel(Self)"
- New: const declaration in Basic script
- New: VBScript functions Redim, RedimPreserve, Split, Join, StrReverse and Randomize
- New: TatCustomScripter methods BeginRefactor and EndRefactor to allow changing in source code without clearing events
- Improved: better load/save compiled code engine
- Improved: exposed TAdvMemo.VisiblePosCount as public property
- Improved: scrolling in memo when ActiveLine property is set
- Improved: VBScript functions LBound, UBound, MsgBox now have default parameters
- Fixed: memory leak in memo using word wrap
- Fixed: small issue with cursor position handling for wordwrapped memo
- Fixed: issue with backspace & selection in memo
- Fixed: issue with input of unicode characters in memo
- Fixed: issue with paste after delete in specific circumstances in memo
- Fixed: issue with horiz. scrollbar updating in memo
- Fixed: AV in some scripts accessing indexed properties
- Fixed: AV when setting breakpoint in begin clause

version 4.2 (Oct-2008)

- New: Delphi 2009/C++Builder 2009 support
- Fixed: issue with AssignFile procedure
- Fixed: issue when removing attached events
- Fixed: issue while using debug watches for global variables

version 4.1 (Jul-2008)

- New: method TAdvMemo.SaveToRTFStream
- New: property TatCustomScripter.Watches (TatScripterWatches class) with the concept of watches for the whole scripter, not only the current script being executed
- Improved: memo syntax highlighting with pascal syntax
- Improved: autocompletion list updating while typing
- Improved: local variables are now initialized to NULL
- Fixed: runtime error message was not displaying correct line and number of error
- Fixed: issue with parameters passed by value to subroutines behaving like by reference
- Fixed : issue with paste on non expanded line in TAdvMemo
- Fixed : issue with repainting after RemoveAllCodeFolding in TAdvMemo
- Fixed : issue with pasting into an empty memo in TAdvMemo
- Fixed : issue with TrimTrailingSpaces = false in TAdvMemo
- Fixed : issue in Delphi 5 with inserting lines in TAdvMemo
- Fixed : issue with scrollbar animation on Windows Vista in TAdvMemo
- Fixed : gutter painting update when setting Modified = false programmatically in TAdvMemo

version 4.0 (Apr-2008)

- New: TatScripter component supporting cross-language scripts (both Pascal and Basic), allowing to replace TatPascalScripter and TatBasicScripter by a single component
- New: Forms support. You can now declare forms and instantiate them from scripts. You can create form methods and load forms from dfm files.
- New: TatScript.Refactor property retrieves a TatScriptRefactor object with methods for refactoring source code, like "DeclareRoutine" and "AddUsedUnit"
- New: Debugger now allows tracing into script-based function calls
- New: TatScript.UnitName property allows a script library to be registered using "uses MyLibrary" syntax without needing MyLibrary to be in a file
- New: Script-level breakpoints allow better control of breakpoints for debugging, instead of

- VirtualMachine-level breakpointsNew: Basic syntax allows declaring the variable type
- New: OnBreakpointStop event in scripiter component is called whenever the script execution stops at a breakpoint
- New: OnSingleDebugHook event allows better performance for debugging than OnDebugHook
- New: Demo project which shows how to use forms with scripiter
- Fixed: Scripiter meta info (ScriptInfo): TatVariableInfo.TypeDecl value now has the correct value (it was empty)
- Fixed: Some variable values were not being displayed when using TatWebScripiter
- Fixed: Minor bugs

version 3.3 (Oct-2007)

- New: TSourceExplorer component. Shows the script structure in a Delphi-like source explorer tree.
- New: C++ to Pascal converter demo shows the capabilities of TatSyntaxParser component.
- Improved: Scripiter Studio Manual includes a "getting started" section for TatSyntaxParser and TSourceExplorer components
- Improved: more accurated value in TatVariableInfo.DeclarationSourcePos property
- Improved: small optimizations in parser
- Improved: many warnings removed
- Fixed: Wrong event name in object inspector in Greatis integration demo

version 3.2 (Jul-2007)

- New: Delphi 2007 support
- New: improved Code Completion - now it retrieves methods and properties at multiple levels for declared global/local script variables (e.g. "var Form: TMyForm"), and retrieves local script functions and procedures.
- New: improved compilation speed.
- New: improved event handling. Now it allows multiple scripts in a single scripiter to handle component events. It's possible to declare a script event handler from script code (e.g. MyObject.Event = 'MyScriptEventHandler'), even if the scripiter component has multiple script objects.
- New: improved import tool for better importing: size of sets and record parameters by reference
- New: new OnUnknownElement event allows defining methods and properties on the fly during compilation when a unknown method or property is found by the compiler
- New: fixed problem with AV in watch viewer
- New: updated VCL import files
- Import tool: Support for Delphi 2007 in import tool

version 3.1 (Sep-2006)

- New: Support for [calling DLL functions](#) from script, allowing even more flexibility for scripts. This feature is enabled by AllowDLLCalls property.
- New: Support for [registering methods with default parameters](#).
- New: OnRuntimeError event
- New: "call dll functions" demo. Includes pascal and basic syntax, and also source code for CustomLib.dll (used by the demos)
- New: "methods with default parameters" demo for pascal and basic
- New: "simple demo" which creates the components at runtime.
- New: Turbo Delphi compatible
- Updated Scripiter Studio manual with the new features and in a new format (chm)

version 3.0.1 (Jul-2006)

- New : TatCustomScripiter.AddDataModule method
- New : AName parameter in TatScript.SelfUnregisterAsLibrary method
- Fixed : form events where not being saved by TSSEventSaver components

- Fixed : memory leak in some specific cases when an event handler was removed from dispatcher

version 3.0 (Mar-2006)

- New : Syntax highlighting memo with codefolding support added
- New : Delphi 2006 & C++Builder 2006 support added
- New : Registered versions comes with VCL ImportTool and full source code for ImportTool

version 2.9 (May-2005)

- New : TatVBScriptLibrary library which adds several function compatible with the available ones in VBScript. Functions added: Asc, Atn, CBool, CByte, CCur, CDate, CDbI, CInt, CLng, CreateObject, CSng, CStr, DatePart, DateSerial, DateValue, Day, Hex, Hour, InStr, Int, Fix, FormatCurrency, FormatDateTime, FormatNumber, InputBox, IsArray, IsDate, IsEmpty, IsNull, IsNumeric, LBound, LCase, Left, Len, Log, LTrim, RTrim, Mid, Minute, Month, MonthName, MsgBox, Replace, Right, Rnd, Second, Sgn, Space, StrComp, String, Timer, TimeSerial, TimeValue, UBound, UCase, Weekday, WeekdayName, Year
- New : OnExecHook event for callback while executing script. CallExecHookEvent property must be set to true to activate the event
- Updated : manual with list of available functions in system library and vbscript library
- Fixed : a couple of bugs in Basic - REM, DO statements, and others
- Fixed : Greatis demo - component properties were not listing components in the form
- Fixed : Wrong example in manual for Basic Syntax in Exit
- Fixed : D6 errors in imports

version 2.8 (Feb-2005)

- New : Script file libraries system: now it's possible to use other script files by declaring the files in the uses clause. This feature is enabled by LibOptions.UseScriptFiles property
- New : Script file libraries works with source files and p-compiled files
- New : LibOptions property allow settings of script file libraries system. Search path can be defined, as well the default extensions for the source files and compiled files.
- New : Added a samples subdirectory in "ide" demo with "newversion.psc" which shows illustrates script file libraries usage.
- New : Form scripters are now aware of components of the form (not only the controls)
- Fixed : Script IDE demo - showing duplicated messages
- Fixed : problems with Greatis integration and Greatis + Scriptor Studio demo
- Fixed : Minor bug fixes & improvements

version 2.7.1 (Oct-2004)

- New : Delphi 2005 support added

version 2.7.0 (Oct-2004)

- New: TSSInspector and TSSEventSaver components for smooth integration with Greatis Runtime Fusion components
- New : downto support in for loops (Pascal syntax)
- New : Added Widestring support in AddVariable method and GetInputArgAsWideString function
- New : new TAdvMemo v1.6 integration
- Fixed: OnCompileError was retrieving wrong line/row error when compiling script-based library.
- Fixed: Bug when destroying Scriptor Studio at design-time

version 2.6.4 (Aug-2004)

- New : script-based libraries can be used from different scripter components and even different languages (see updated "script-based libraries" demo)
- Fixed: parameter with names starting with "Var" was considered as by reference
- Fixed: MessageDlg call was not working in Delphi 5
- Fixed: It's now possible to Halt all running scripts

- Fixed: Errors with Create method expecting 0 parameters (important! current users see AScript.INC file)

version 2.6.3 (Jun-2004)

- Improved : debugger speed
- Fixed : Syntax Error with WriteLn in webscripiter
- Fixed : missing "begin..end" block iin webscripiter demo
- Fixed : TypeCast was not working in calls. Example: TStringList(S).Add('Hello');
- Fixed : SaveCodeToFile and LoadCodeFromFile were failing in some situations

version 2.6.2 (May-2004)

- New : ShortBooleanEval property to control optional short-circuit boolean evaluation

version 2.6.1 (Apr-2004)

- Improved : More overloaded AddVariable methods
- Improved : RangeChecks off directive in ascript.inc
- Fixed: bug with script libraries
- Improved : TAdvMemo syntax highlighting memo

version 2.6.0 (Apr-2004)

- New : Script-based libraries. It's now possible to call routines/set global variables from other scripts. See new "script-based libraries" demo to see how it works
- New : File-manipulation routines added: AssignFile, Reset, Rewrite, Append, CloseFile, Write, WriteLn, ReadLn, EOF, FilePos, FileSize (thanks to Keen Ronald)
- New : More system functions added: Abs, ArcTan, ChDir, Chr, Exp, Frac, Int, Ln, Odd, Ord, Sqr, Sqrt
- New : Support to Elself constructor in Basic scripiter
- New : Support to Uses and Imports declaration in Basic scripiter (thanks to Dean Franks)
- New : Code editor with Drag & drop support
- New : AdvCodeList component
- New : Code editor with wordwrap support (no wordwrap, wordwrap on memo width, wordwrap on right margin)
- New : Code editor with Code block start/end highlighting while typing
- New : Code editor with properties ActiveLineColor, ActiveLineTextColor properties added
- New : Code editor with BreakpointColor, BreakpointTextColor
- New : Code editor with Actions for most common editor actions.
- Improved : Could not use events or call subroutines on precompiled scripts (loaded from stream/file)
- Improved : CASE and SELECT CASE statements not working properly
- Improved : FOR statements with negative step not working properly
- Improved : Changing CanClose parameter in OnClose event has no effect
- Improved : Basic double double-quotes in strings not working properly
- Improved : Unknown variable error in FOR statements when OptionExplicit = true

version 2.5.3 (Mar-2004)

- Fixed: Small fixes and improvements

version 2.5.2

- New : Debugging can start from any script subroutine, not only main block
- New : Properties in TatScriptDebugger component: RoutineName, UpdateSourceCode and MemoReadOnly
- Improved : TatScripterDebugger.Execute method now works even if script is already running
- Improved : Values of global variables keep their values between scripiter executions
- Fixed : bug with variant arrays

- Fixed : bug with try..except blocks while debugging

version 2.5.1

- Fixed: Several bug fixes and stability improvements

version 2.5

- New : WITH clause language construct
- New : Type casting
- New : IS/AS operators (only between object and class)
- New : Typed variable declarations. E.g, var Table: TTable; It will only take effect for object variables
- New : global variables
- New : watches
- New : forward directives
- New : integrated autocompletion in IDE and debugger
- New : integrated hint for evaluation of variables during debug
- New : syntax memo with bookmark support
- New : IDE demo app
- Improved : WebScripter & PageProducer component for creating Pascal based ASP-like web applications
- Improved : multi-thread support

version 2.4.6

- Improved : WebScripter component
- New : PageProducer component to be used with WebScripter

version 2.4.5

- New : WebScripter component (written by and provided by Don Wibier) and Page producer component that parses Pascal or Basic ASP-like files and produces HTML files
- New : Basic Scripter: "Set" word supported. Example: Set A = 10
- New : Basic Scripter: "&" operator supported. Example: MyFullName = MyFirstName & " " & MyLastName
- New : Pascal Scripter: function declaration accepts result type (which is ignored): function MyFunction: string;
- New : Pascal Scripter: const section supported: const MyStr = 'This is a string';
- New : AdvMemo insert & overwrite mode
- Improved : AdvMemo numeric highlighting

version 2.4

- New : AdvMemo with parameter hinting
- New : AdvMemo with code completion
- New : AdvMemo with error marking
- Improved : various smaller scripter engine improvements
- New : DynaForms demo added

version 2.3.5

- New : Support for hexadecimal integers (\$10 in Pascal, 0x10 in Basic)
- New : Allow spaces between function names and parameters, eg.: ShowMessage ('Hello world!');
- New : uses clause (to use import libraries), eg.: uses Classes; {Load Classes library if TatClassesLibrary was previous registered}
- New : From Delphi function, it is possible to know name of method or property called, using CurrentPropertyName and CurrentMethodName functions from TatVirtualMachine object
- New : No need to assign OnDebugHook event to debug script
- New : Use of params by reference when calling script procedures from Delphi

- New : Changed class name of internal library, from TatSytemLibrary to TatInternalLibrary
- New : Minor bug fixes (array property)

version 2.3

- New : support for Pascal & Basic script engines for Kylix 2,3

version 2.2

- Improved: syntax highlighting memo, with improved speed, SaveToHTML function, Print
- Improved: design time script property editor
- Improved: debugger control

version 2.1

- New : Seamless and powerful Delphi component event handling allows event handling chaining between Delphi and Scripter in any sequence allows setting component event handling from Delphi or from Scripter or from both
- New : 4 sample applications for Pascal and Basic scripter that shows the new powerful event handling

version 2.0

- first release as Scripter Studio, suite of scripter tools for applications
- New : Run-time Pascal and Basic language interpreter
- New : Design-time and run-time debugger
- New : Pascal and Basic syntax highlighting memo with integrated debugging facilities
- New : FormScript, form-aware descendant scripter components for Basic and Pascal
- New : Scripter Studio developers guide
- New : Run-time script debugger dialog
- New : Arguments passed by reference on local procedures/function and on object methods capability added
- New : Safe multiprocessing/multi-threading features with new method signature and source code rearrangement
- New : Automatic variable declaration, now is controlled by OptionExplicit property
- New : Array properties, variant array constructor and string as array support was introduced
- New : Class methods and properties support and class references (allow to implement, for example, Txxx.Create(...))
- New : Additional system library usefull routines: Inc, Dec, Format, VarArrayHighBound, High, VarArrayLowBound, Low, TObject.Create, TObject.Free, VarToStr
- New : Extendable architecture open to add support for other languages in future updates
- Improved : Object Pascal syntax compatibility
(not,xor,shl,shr,~,div,mod,break,continue,exit,null,true,false,var,case,function)

version 1.5

- TatPascalScripter release

1.4 Manual Installation

If you selected manual installation during setup, follow this instructions to manually install Scripter Studio in Delphi or C++Builder:

1) From Delphi, C++Builder

Add the directory where the files are installed {\$SS} to the Delphi library path under Tools, Environment options, Directories

a) All Delphi/C++Builder Versions:

Add the directory `{SS}\source` to the library path

Add the directory `{SS}\source\AdvMemo` to the library path

Add the directory `{SS}\source\Designer` to the library path (if installing Scripster Studio PRO)

b) All C++Builder Versions:

Add the directory `{SS}\source` to the include path

Add the directory `{SS}\source\AdvMemo` to the include path

Add the directory `{SS}\source\Designer` to the include path (if installing Scripster Studio PRO)

c) For specific Delphi/C++Builder version

Delphi 5 : Add the directory `{SS}\source\Imports\Delphi5` to the library path

Delphi 6 : Add the directory `{SS}\source\Imports\Delphi6` to the library path

Delphi 7 : Add the directory `{SS}\source\Imports\Delphi7` to the library path

BCB 6 : Add the directory `{SS}\source\Imports\Delphi6` to the library path

Delphi 2005 : Add the directory `{SS}\source\Imports\Delphi2005` to the library path

Delphi/BCB 2006 : Add the directory `{SS}\source\Imports\Delphi2006` to the library path

Delphi/BCB 2007 : Add the directory `{SS}\source\Imports\Delphi2007` to the library path

Delphi/BCB 2009 : Add the directory `{SS}\source\Imports\Delphi2009` to the library path

Delphi/BCB 2010 : Add the directory `{SS}\source\Imports\Delphi2010` to the library path

RAD Studio XE : Add the directory `{SS}\source\Imports\Delphi2011` to the library path

2) From Delphi, C++Builder

Choose menu File, Open and browse for the correct package file for Delphi or C++Builder:

- aScript5.dpk: Delphi 5
- aScript6.dpk: Delphi 6
- aScript7.dpk: Delphi 7
- aScript2005.dpk: Delphi 2005
- aScript2006.dpk: Delphi 2006/C++Builder 2006
- aScript2007.dpk: Delphi 2007/C++Builder 2007
- aScript2009.dpk: Delphi 2009/C++Builder 2009
- aScript2010.dpk: Delphi 2010/C++Builder 2010
- aScript2011.dpk: Delphi XE/C++Builder XE
- aScriptc6.bpk: C++Builder 6

Installing the integration with Greatis Designer and Inspector

1) Make sure that Greatis Form Designer, Object Inspector and Runtime Fusion are installed. To install the Runtime Fusion support, it might be required to create a package for the Runtime Fusion components.
See also: www.greatis.com

2) Add the folder `{SS}\source\Greatis` to the environment library path

3) Open and install the Greatis integration support package

ssgreatis5.dpk : Delphi 5

ssgreatis6.dpk : Delphi 6

ssgreatis7.dpk : Delphi 7

ssgreatis2005.dpk : Delphi 2005

ssgreatis2006.dpk : Delphi 2006

ssgreatis2007.dpk : Delphi 2007

ssgreatis2009.dpk : Delphi 2009

ssgreatis2010.dpk : Delphi 2010

Upon install , two additional components are added to the Scripter Studio tab:
SSInspector and SSEventSaver

- 4) Check the demo in the Greatis folder that shows how to build a Delphi like IDE with Greatis and Scripter Studio.

Notes for users of the TMS Component Pack

If the TAdvMemo component (latest version v2.0) is already installed as part of the TMS Component Pack, the Scripter Studio can be installed in following way:

- 1) Make sure it is the latest version of the TMS Component pack that is installed, including TAdvMemo v1.5 or higher.
- 2) Remove AdvMemo*, AdvCode*, uMemoEdit* file references from the Scripter Studio package file.
- 3) Install the Scripter Studio package file. Delphi or C++Builder will automatically add a reference to the TMS Component Pack in the Scripter Studio package and all components will install as expected.

1.5 Getting Support

General notes

Before contacting support:

- Make sure to read the tips, faq and readme.txt or install.txt files in component distributions.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component you have a problem.
 - Specify which Delphi or C++Builder version you're using and preferably also on which OS.
 - In case of IntraWeb or ASP.NET components, specify with which browser the issue occurs.
 - For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.
- Send email from an email account that
 - 1) allows to receive replies sent from our server
 - 2) allows to receive ZIP file attachments
 - 3) has a properly specified & working reply address

Getting support

For general information: info@tmssoftware.com

Fax: +32-56-359696

For all questions, comments, problems and feature request for VCL components :

help@tmssoftware.com.

To improve efficiency and speed of help, refer to the version of Delphi, C++Builder, Visual Studio .NET you are using as well as the version of the component. In case of problems, always try to use the latest version available first

1.6 Version 4.5: News about RTTI

Taking advantage of new features related to RTTI and available from Delphi 2010, TMS Scripter Studio 4.5 implements methods to make easier the registration of classes, letting them available for use in scripts. So far we need to manually define each method/property of a class (except published properties) – at least there's a nice utility program named "ImportTool" – but from now we can register

almost all members of a class automatically and with minimum effort, as seen below.

1.6.1 Registering a class in scripiter

To register a class in Scripiter, usually we use `TatCustomScripiter.DefineClass` method to define the class, and helper methods to define each class member, and also we need to implement wrapper methods to make the calls for class methods, as well as getters and setters for properties. Example:

```
with Scripiter.DefineClass(TMyClass) do
begin
  DefineMethod('Create', 0, tkClass, TMyClass, __TMyClassCreate, true);
  DefineMethod('MyMethod', tkNone, nil, __TMyClassMyMethod);
  (...)
  DefineProp('MyProp', tkInteger, __GetTMyClassMyProp, __SetTMyClassMyProp);
  (...)
end;
```

With new features, just call `TatCustomScripiter.DefineClassByRTTI` method to register the class in scripiter, and automatically all their methods and properties:

```
Scripiter.DefineClassByRTTI(TMyClass);
```

This method has additional parameters that allow you to specify exactly what will be published in scripiter:

```
procedure TatCustomScripiter.DefineClassByRTTI(
  AClass: TClass;
  AClassName: string='';
  AVisibilityFilter: TMemberVisibilitySet = [mvPublic, mvPublished];
  ARecursive: boolean = False);
```

- `AClass`: class to be registered in scripiter;
- `AClassName`: custom name for registered class, the original class name is used if empty;
- `AVisibilityFilter`: register only members whose visibility is in this set, by default only public and published members are registered, but you can register also private and protected members;
- `ARecursive`: if true, scripiter will also register other types (classes, records, enumerated types) which are used by methods and properties of class being defined. These types are recursively defined using same option specified in visibility filter.

1.6.2 Registering a record in scripiter

Since scripiter does not provide support for records yet, our recommended solution is to use wrapper classes (inherited from `TatRecordWrapper`) to emulate a record structure by implementing each record field as a class property. Example:

```
TRectWrapper = class(TatRecordWrapper)
  (...)
published
  property Left: Longint read FLeft write FLeft;
  property Top: Longint read FTop write FTop;
  property Right: Longint read FRight write FRight;
  property Bottom: Longint read FBottom write FBottom;
end;
```

While scripiter still remains using classes to emulated records, is no longer necessary to implement an exclusive wrapper class for each record, because now scripiter implements a generic wrapper. Thus a record (and automatically all its fields) can be registered into scripiter by `TatCustomScripiter.DefineRecordByRTTI` method, as in example below:

```
Scripiter.DefineRecordByRTTI(TypeInfo(TRect));
```

The method only receives a pointer parameter to record type definition:

```
procedure TatCustomScripter.DefineRecordByRTTI(ATypeInfo: Pointer);
```

Records registered in scripiter will work as class and therefore need to be instantiated before use in your scripts (except when methods or properties return records, in this case scripiter instantiates automatically). Example:

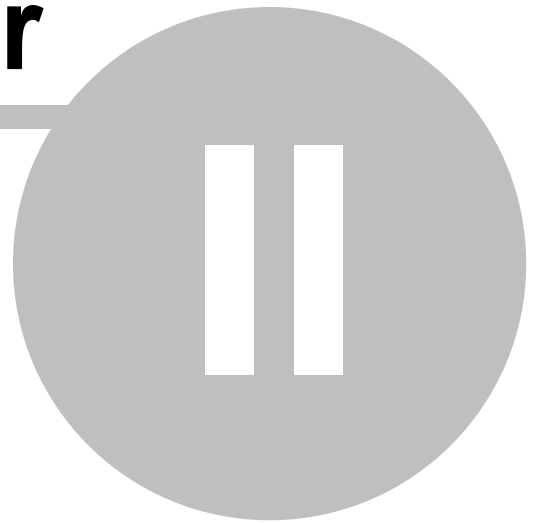
```
var
  R: TRect;
begin
  R := TRect.Create;
  try
    R.Left := 100;
    // do something with R
  finally
    R.Free;
  end;
end;
```

1.6.3 What is not supported

Due to Delphi RTTI and/or scripiter limitations, some features are not supported yet and you may need some workaround for certain operations.

- Indexed properties (arrays) are not automatically registered in scripiter. To define them use DefineProp method, passing the dimension of array property in AIndexCount argument.
- Scripiter automatically registers only methods declared in public and published clauses of a class, since methods declared as private or protected are not accessible via RTTI. When defining a class with private and protected in visibility filter, scripiter will only define fields and properties declared in these clauses.
- If a class method has overloads, scripiter will register only the first method overload declared in that class.
- Methods having parameters with default values, when automatically defined in scripiter, are registered with all parameters required. To define method with default parameters, use DefineMethod method, passing number of default arguments in ADefArgCount parameter, and implement the method handler (TMachineProc) to check the number of arguments passed to method by using TatVirtualMachine.InputArgCount function.
- Event handlers are not automatically defined by scripiter. You must implement a TatEventDispatcher descendant class and use DefineEventAdapter method.
- Some methods having parameters of "uncommon" types (such as arrays and others) are not defined in scripiter, since Delphi does not provide enough information about these methods.

Chapter



Language Features

2 Language Features

This chapter covers all the languages you can use to write scripts, and which language features you can use, language syntax, constructors, etc.

2.1 Pascal syntax

2.1.1 Overview

TatPascalScripter component executes scripts written in Pascal syntax. Current Pascal syntax supports:

- **begin .. end** constructor
- **procedure** and **function** declarations
- **if .. then .. else** constructor
- **for .. to .. do .. step** constructor
- **while .. do** constructor
- **repeat .. until** constructor
- **try .. except** and **try .. finally** blocks
- **case** statements
- **array** constructors (x:=[1, 2, 3];
- **^ , * , / , and , + , - , or , <> , >= , <= , = , > , < , div , mod , xor , shl , shr** operators
- access to object properties and methods (**ObjectName.SubObject.Property**)

2.1.2 Script structure

Script structure is made of two major blocks: a) procedure and function declarations and b) main block. Both are optional, but at least one should be present in script. There is no need for main block to be inside begin..end. It could be a single statement. Some examples:

```
SCRIPT 1:
procedure DoSomething;
begin
    CallSomething;
end;
```

```
begin
    CallSomethingElse;
end;
```

```
SCRIPT 2:
begin
    CallSomethingElse;
end;
```

```
SCRIPT 3:
function MyFunction;
begin
    result:='Ok!';
end;
```

```
SCRIPT 4:
```

```
CallSomethingElse;
```

Like in pascal, statements should be terminated by ";" character. Begin..end blocks are allowed to group statements.

2.1.3 Identifiers

Identifier names in script (variable names, function and procedure names, etc.) follow the most common rules in pascal : should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric chars or '_' char. Cannot contain any other character os spaces.

Valid identifiers:

```
VarName
_Some
V1A2
_____Some_____
```

Invalid identifiers:

```
2Var
My Name
Some-more
This,is,not,valid
```

2.1.4 Assign statements

Just like in Pascal, assign statements (assign a value or expression result to a variable or object property) are built using ":=". Examples:

```
MyVar := 2;
Button.Caption := 'This ' + 'is ok.';
```

2.1.5 Character strings

Strings (sequence of characters) are declared in pascal using single quote (') character. Double quotes (") are not used. You can also use #nn to declare a character inside a string. There is no need to use '+' operator to add a character to a string. Some examples:

```
A := 'This is a text';
Str := 'Text '+'concat';
B := 'String with CR and LF char at the end'#13#10;
C := 'String with '#33#34' characters in the middle';
```

2.1.6 Comments

Comments can be inserted inside script. You can use // chars or (* *) or { } blocks. Using // char the comment will finish at the end of line.

```
//This is a comment before ShowMessage
ShowMessage('Ok');

(* This is another comment *)
ShowMessage('More ok!');

{ And this is a comment
  with two lines }
ShowMessage('End of okays');
```

2.1.7 Variables

There is no need to declare variable types in script. Thus, you declare variable just using var directive and its name. There is no need to declare variables if scripter property OptionExplicit is set to false. In this case, variables are implicit declared. If you want to have more control over the script, set OptionExplicit property to true. This will raise a compile error if variable is used but not declared in script. Examples:

SCRIPT 1:

```
procedure Msg;
var S;
begin
  S:='Hello world!';
  ShowMessage(S);
end;
```

SCRIPT 2:

```
var A;
begin
  A:=0;
  A:=A+1;
end;
```

SCRIPT 3:

```
var S: string;
begin
  S:='Hello World!';
  ShowMessage(S);
end;
```

Note that if script property OptionExplicit is set to false, then var declarations are not necessary in any of scripts above.

2.1.8 Indexes

Strings, arrays and array properties can be indexed using "[" and "]" chars. For example, if Str is a string variable, the expression Str[3] returns the third character in the string denoted by Str, while Str[l + 1] returns the character immediately after the one indexed by l. More examples:

```
MyChar:=MyStr[2];
MyStr[1]:='A';
MyArray[1,2]:=1530;
Lines.Strings[2]:='Some text';
```

2.1.9 Arrays

Script support array constructors and support to variant arrays. To construct an array, use "[" and "]" chars. You can construct multi-index array nesting array constructors. You can then access arrays using indexes. If array is multi-index, separate indexes using ",".

If variable is a variant array, script automatically support indexing in that variable. A variable is a variant array is it was assigned using an array constructor, if it is a direct reference to a Delphi variable which is a variant array (see Delphi integration later) or if it was created using VarArrayCreate procedure.

Arrays in script are 0-based index. Some examples:

```
NewArray := [ 2,4,6,8 ];
```

```
Num:=NewArray[1]; //Num receives "4"  
MultiArray := [ ['green','red','blue'] , ['apple','orange','lemon'] ];  
Str:=MultiArray[0,2]; //Str receives 'blue'  
MultiArray[1,1]:='new orange';
```

2.1.10 If statements

There are two forms of if statement: if...then and the if...then...else. Like normal pascal, if the if expression is true, the statement (or block) is executed. If there is else part and expression is false, statement (or block) after else is execute. Examples:

```
if J <> 0 then Result := I/J;  
if J = 0 then Exit else Result := I/J;  
if J <> 0 then  
begin  
    Result := I/J;  
    Count := Count + 1;  
end  
else  
    Done := True;
```

2.1.11 while statements

A while statement is used to repeat a statement or a block, while a control condition (expression) is evaluated as true. The control condition is evaluated before the statement. Hence, if the control condition is false at first iteration, the statement sequence is never executed. The while statement executes its constituent statement (or block) repeatedly, testing expression before each iteration. As long as expression returns True, execution continues. Examples:

```
while Data[I] <> X do I := I + 1;  
while I > 0 do  
begin  
    if Odd(I) then Z := Z * X;  
    I := I div 2;  
    X := Sqr(X);  
end;  
  
while not Eof(InputFile) do  
begin  
    Readln(InputFile, Line);  
    Process(Line);  
end;
```

2.1.12 repeat statements

The syntax of a repeat statement is *repeat statement1; ...; statementn; until expression* where expression returns a Boolean value. The repeat statement executes its sequence of constituent statements continually, testing expression after each iteration. When expression returns True, the repeat statement terminates. The sequence is always executed at least once because expression is not evaluated until after the first iteration. Examples:

```
repeat  
    K := I mod J;  
    I := J;  
    J := K;  
until J = 0;
```

```
repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

2.1.13 for statements

Scripter support for statements with the following syntax: *for counter := initialValue to finalValue do statement*

For statement set counter to initialValue, repeats execution of statement (or block) and increment value of counter until counter reaches finalValue. Examples:

```
SCRIPT 1:
for c:=1 to 10 do
  a:=a+c;
```

```
SCRIPT 2:
for i:=a to b do
begin
  j:=i^2;
  sum:=sum+j;
end;
```

2.1.14 case statements

Scripter support case statements with following syntax:

```
case selectorExpression of
  caseexpr1: statement1;
  ...
  caseexprn: statementn;
else
  elstatement;
end
```

if selectorExpression matches the result of one of caseexprn expressions, the respective statement (or block) will be execute. Otherwise, elstatement will be execute. Else part of case statement is optional. Different from Delphi, case statement in script doesn't need to use only ordinal values. You can use expressions of any type in both selector expression and case expression. Example:

```
case uppercase(Fruit) of
  'lime': ShowMessage('green');
  'orange': ShowMessage('orange');
  'apple': ShowMessage('red');
else
  ShowMessage('black');
end;
```

2.1.15 function and procedure declaration

Declaration of functions and procedures are similar to Object Pascal in Delphi, with the difference you don't specify variable types. Just like OP, to return function values, use implicated declared result variable. Parameters by reference can also be used, with the restriction mentioned: no need to specify variable types. Some examples:

```
procedure HelloWorld;
```

```
begin
  ShowMessage('Hello world!');
end;

procedure UpcaseMessage(Msg);
begin
  ShowMessage(Uppercase(Msg));
end;

function TodayAsString;
begin
  result:=DateToStr(Date);
end;

function Max(A,B);
begin
  if A>B then
    result:=A
  else
    result:=B;
end;

procedure SwapValues(var A, B);
Var Temp;
begin
  Temp:=A;
  A:=B;
  B:=Temp;
end;
```

2.2 Basic syntax

2.2.1 Overview

TatBasicScripter component executes scripts written in Basic syntax. Current Basic syntax supports:

- **sub .. end** and **function .. end** declarations
- **byref** and **dim** directives
- **if .. then .. else .. end** constructor
- **for .. to .. step .. next** constructor
- **do .. while .. loop** and **do .. loop .. while** constructors
- **do .. until .. loop** and **do .. loop .. until** constructors
- **^**, *****, **/**, **and**, **+**, **-**, **or**, **<>**, **>=**, **<=**, **=**, **>**, **<**, **div**, **mod**, **xor**, **shl**, **shr** operators
- **try .. except** and **try .. finally** blocks
- **try .. catch .. end try** and **try .. finally .. end try** blocks
- **select case .. end select** constructor
- **array** constructors (x:=[1, 2, 3];)
- **exit** statement
- access to object properties and methods (**ObjectName.SubObject.Property**)

2.2.2 Script structure

Script structure is made of two major blocks: a) function and sub declarations and b) main block. Both are optional, but at least one should be present in script. Some examples:

```
SCRIPT 1:
SUB DoSomething
  CallSomething
END SUB
```

```
CallSomethingElse
```

```
SCRIPT 2:
CallSomethingElse
```

```
SCRIPT 3:
FUNCTION MyFunction
    MyFunction = "Ok!"
END FUNCTION
```

Like in normal basic, statements in a single line can be separated by ":" character.

2.2.3 Identifiers

Identifier names in script (variable names, function and procedure names, etc.) follow the most common rules in basic : should begin with a character (a..z or A..Z), or '_' , and can be followed by alphanumeric chars or '_' char. Cannot contain any other character os spaces.

Valid identifiers:

```
VarName
_Some
V1A2
_____Some_____
```

Invalid identifiers:

```
2Var
My Name
Some-more
This,is,not,valid
```

2.2.4 Assign statements

Assign statements (assign a value or expression result to a variable or object property) are built using "=". Examples:

```
MyVar = 2
Button.Caption = "This " + "is ok."
```

2.2.5 New statement

Scripter Studio provides the "new" statement for Basic syntax. Since you don't provide the method name in this statement, scripter studio looks for a method named "Create" in the specified class. If the method doesn't exist, the statement fails. Example:

```
MyLabel = new TLabel(Form1)
MyFont = new TFont
```

In the above examples, a method named "Create" for TLabel and TFont class will be called. The method must be registered. If the method receives parameters, you can pass the parameters in parenthesis, like the TLabel example above.

2.2.6 Character strings

strings (sequence of characters) are declared in basic using double quote (") character. Some examples:

```
A = "This is a text"  
Str = "Text "+"concat"
```

2.2.7 Comments

Comments can be inserted inside script. You can use ' chars or REM. Comment will finish at the end of line. Examples:

```
' This is a comment before ShowMessage  
ShowMessage("Ok")
```

```
REM This is another comment  
ShowMessage("More ok!")
```

```
' And this is a comment  
' with two lines  
ShowMessage("End of okays")
```

2.2.8 Variables

There is no need to declare variable types in script. Thus, you declare variable just using DIM directive and its name. There is no need to declare variables if scripter property OptionExplicit is set to false. In this case, variables are implicit declared. If you want to have more control over the script, set OptionExplicit property to true. This will raise a compile error if variable is used but not declared in script. Examples:

```
SCRIPT 1:  
SUB Msg  
    DIM S  
    S = "Hello world!"  
    ShowMessage(S)  
END SUB
```

```
SCRIPT 2:  
DIM A  
A = 0  
A = A+1  
ShowMessage(A)
```

Note that if script property OptionExplicit is set to false, then variable declarations are not necessary in any of scripts above.

You can also declare global variables as private or public using the following syntax:

```
SCRIPT 2:  
PRIVATE A  
PUBLIC B  
B = 0  
A = B + 1  
ShowMessage(A)
```

Variable declared with DIM statement are public by default. Private variables are not accessible from other scripts.

Variables can be default initialized with the following syntax

```
DIM A = "Hello world"  
DIM B As Integer = 5
```

2.2.9 Indexes

Strings, arrays and array properties can be indexed using "[" and "]" chars. For example, if Str is a string variable, the expression Str[3] returns the third character in the string denoted by Str, while Str[l + 1] returns the character immediately after the one indexed by l. More examples:

```
MyChar = MyStr[2]
MyStr[1] = "A"
MyArray[1,2] = 1530
Lines.Strings[2] = "Some text"
```

2.2.10 Arrays

Script support array constructors and support to variant arrays. To construct an array, use "[" and "]" chars. You can construct multi-index array nesting array constructors. You can then access arrays using indexes. If array is multi-index, separate indexes using ",".

If variable is a variant array, script automatically support indexing in that variable. A variable is a variant array is it was assigned using an array constructor, if it is a direct reference to a Delphi variable which is a variant array (see Delphi integration later) or if it was created using VarArrayCreate procedure.

Arrays in script are 0-based index. Some examples:

```
NewArray = [ 2,4,6,8 ]
Num = NewArray[1] //Num receives "4"
MultiArray = [ ["green","red","blue"] , ["apple","orange","lemon"] ]
Str = MultiArray[0,2] //Str receives 'blue'
MultiArray[1,1] = "new orange"
```

2.2.11 If statements

There are two forms of if statement: if...then..end if and the if...then...else..end if. Like normal basic, if the if expression is true, the statements are executed. If there is else part and expression is false, statements after else are executed. Examples:

```
IF J <> 0 THEN Result = I/J END IF
IF J = 0 THEN Exit ELSE Result = I/J END IF
IF J <> 0 THEN
  Result = I/J
  Count = Count + 1
ELSE
  Done = True
END IF
```

If the IF statement is in a single line, you don't need to finish it with END IF:

```
IF J <> 0 THEN Result = I/J
IF J = 0 THEN Exit ELSE Result = I/J
```

2.2.12 while statements

A while statement is used to repeat statements, while a control condition (expression) is evaluated as true. The control condition is evaluated before the statements. Hence, if the control condition is false at first iteration, the statement sequence is never executed. The while statement executes its constituent statement repeatedly, testing expression before each iteration. As long as expression returns True, execution continues. Examples:

```
WHILE (Data[I] <> X) I = I + 1 END WHILE
WHILE (I > 0)
  IF Odd(I) THEN Z = Z * X END IF
  X = Sqr(X)
```

```

END WHILE

WHILE (not Eof(InputFile))
  Readln(InputFile, Line)
  Process(Line)
END WHILE

```

2.2.13 loop statements

Scripter support loop statements. The possible syntax are:

```

DO WHILE expr statements LOOP
DO UNTIL expr statements LOOP
DO statements LOOP WHILE expr
DO statement LOOP UNTIL expr

```

statements will be execute WHILE expr is true, or UNTIL expr is true. if expr is before statements, then the control condition will be tested before iteration. Otherwise, control condition will be tested after iteration. Examples:

```

DO
  K = I mod J
  I = J
  J = K
LOOP UNTIL J = 0

DO UNTIL I >= 0
  Write("Enter a value (0..9): ")
  Readln(I)
LOOP

DO
  K = I mod J
  I = J
  J = K
LOOP WHILE J <> 0

DO WHILE I < 0
  Write("Enter a value (0..9): ")
  Readln(I)
LOOP

```

2.2.14 for statements

Scripter support for statements with the following syntax: FOR counter = initialValue TO finalValue STEP stepValue statements NEXT. For statement set counter to initialValue, repeats execution of statement until "next" and increment value of counter by stepValue, until counter reaches finalValue. Step part is optional, and if omitted stepValue is considered 1. Examples:

```

SCRIPT 1:
FOR c = 1 TO 10 STEP 2
  a = a + c
NEXT

SCRIPT 2:
FOR I = a TO b
  j = i ^ 2
  sum = sum + j
NEXT

```

2.2.15 select case statements

Scripter support select case statements with following syntax:

```

SELECT CASE selectorExpression
  CASE caseexpr1
    statement1
  ...
  CASE caseexprn
    statementn
CASE ELSE
  elsestatement
END SELECT

```

if selectorExpression matches the result of one of caseexprn expressions, the respective statements will be execute. Otherwise, elsestatement will be executed. Else part of case statement is optional.

Example:

```

SELECT CASE uppercase(Fruit)
  CASE "lime" ShowMessage("green")
  CASE "orange"
    ShowMessage("orange")
  CASE "apple" ShowMessage("red")
CASE ELSE
  ShowMessage("black")
END SELECT

```

2.2.16 function and sub declaration

Declaration of functions and subs are similar to basic. In functions to return function values, use implicated declared variable which has the same name of the function, or use Return statement. Parameters by reference can also be used, using BYREF directive. Some examples:

```

SUB HelloWorld
  ShowMessage("Hello world!")
END SUB

SUB UpcaseMessage(Msg)
  ShowMessage(Uppercase(Msg))
END SUB

FUNCTION TodayAsString
  TodayAsString = DateToStr(Date)
END FUNCTION

FUNCTION Max(A,B)
  IF A>B THEN
    MAX = A
  ELSE
    MAX = B
  END IF
END FUNCTION

SUB SwapValues(BYREF A, B)
  DIM TEMP
  TEMP = A
  A = B
  B = TEMP
END SUB

```

You can also declare subs and functions as private or public using the following syntax:

```
PRIVATE SUB Hello
END SUB
```

```
PUBLIC FUNCTION Hello
END FUNCTION
```

Subs and functions are public by default. Private subs and functions are not accessible from other scripts.

You can use Return statement to exit subs and functions. For functions, you can also return a valid value. Examples:

```
SUB UppcaseMessage(Msg)
  ShowMessage(Uppercase(Msg))
  Return
  'This line will be never reached
  ShowMessage("never displayed")
END SUB
```

```
FUNCTION TodayAsString
  Return DateToStr(Date)
END FUNCTION
```

2.3 Calling dll functions

2.3.1 Overview

Scripter allows importing and calling external DLL functions, by inserting special directives on declaration of script routines, indicating library name and, optionally, the calling convention, beyond the function signature.

External libraries are loaded by Scripter on demand, before function calls, if not loaded yet (dynamically or statically). To load and unload libraries explicitly, functions LoadLibrary and FreeLibrary from unit Windows can be used.

Note: to enable DLL function calls, you must set AllowDLLCalls property to true.

2.3.2 Pascal syntax

```
function functionName(arguments): resultType; [callingConvention]; external
'libName.dll' [name ExternalFunctionName];
```

For example, the following declaration:

```
function MyFunction(arg: integer): integer; external 'CustomLib.dll';
```

imports a function called MyFunction from CustomLib.dll. Default calling convention, if not specified, is register. Scripter also allows to declare a different calling convention (stdcall, register, pascal, cdecl or safecall) and to use a different name for DLL function, like the following declaration:

```
function MessageBox(hwnd: pointer; text, caption: string; msgtype: integer):
integer; stdcall; external 'User32.dll' name 'MessageBoxA';
```

that imports 'MessageBoxA' function from User32.dll (Windows API library), named 'MessageBox' to be used in script.

Declaration above can be used to functions and procedures (routines without result value).

2.3.3 Basic syntax

```
function lib 'libName.dll' [alias ExternalFunctionName] [callingConvention]
functionName(arguments) as resultType;
```

For example, the following declaration:

```
function lib "CustomLib.dll" MyFunction(arg as integer) as integer
```

imports a function called MyFunction from CustomLib.dll. Default calling convention, if not specified, is stdcall. Scripter also allows to declare a different calling convention (stdcall, register, pascal, cdecl or safecall) and to use a different name for DLL function, like the following declaration:

```
function MessageBox lib "User32.dll" alias "MessageBoxA" stdcall (hwnd as pointer,
text as string, caption as string, msgtype as integer) as integer
```

that imports 'MessageBoxA' function from User32.dll (Windows API library), named 'MessageBox' to be used in script.

Declaration above can be used to functions and subs (routines without result value).

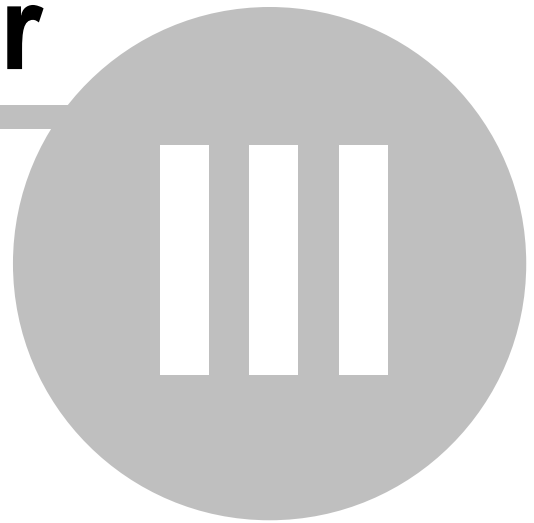
2.3.4 Supported types

Scripter support following basic data types on arguments and result of external functions:

- Integer*
- Boolean*
- Char*
- Extended*
- String*
- Pointer*
- PChar*
- Object*
- Class*
- WideChar*
- PWideChar*
- AnsiString*
- Currency*
- Variant*
- Interface*
- WideString*
- Longint*
- Cardinal*
- Longword*
- Single*
- Byte*
- Shortint*
- Word*
- Smallint*
- Double*
- Real*
- DateTime*
- TObject descendants (class must be registered in scripter with DefineClass)*

Others types (records, arrays, etc.) are not supported yet. Arguments of above types can be passed by reference, by adding var (Pascal) or byref (Basic) in param declaration of function.

Chapter



Working with scripter

3 Working with scripter

This chapter provides information about how to use the scripter component in your application. How to run scripts, how to integrate Delphi objects with the script, and other tasks are covered here.

3.1 Getting started

To start using scripter, you just need to know one property (SourceCode) and one method (Execute). Thus, to start using scripter to execute a simple script, drop it on a form and use the following code (in a button click event, for example):

```
Scripter.SourceCode.Text := 'ShowMessage(''Hello world!'');';  
Scripter.Execute;
```

And you will get a "Hello world!" message after calling Execute method. That's it. From now, you can start executing scripts. To make it more interesting and easy, drop a TAdvMemo component in form and change code to:

```
Scripter.SourceCode := AdvMemo1.Lines;  
Scripter.Execute;
```

[C++Builder example](#)

Now you can just type scripts at runtime and execute them.

From this point, any reference to scripter object (methods, properties, events) refers to TatCustomScripter object and can be applied to TatPascalScripter and TatBasicScripter - except when explicit indicated. The script examples will be given in Pascal syntax.

3.2 Cross-language feature: TatScripter and TIDEScripter

Scripter Studio provides a single scripter component that allows cross-language scripting: TatScripter. In addition, Scripter Studio Pro provides also TIDEScripter, which descends from TatScripter and provide the same features.

Replacing old TatPascalScripter and TatBasicScripter by the new TatScripter (or TIDEScripter) is simple and straightforward. It's full compatible with the previous one, and the cross-language works smoothly. There only two things that are **not backward compatible by default**, but you can change it using properties. The differences are:

1. OptionExplicit property now is "true" by default

The new TIDEScripter component requires that all variables are declared in script, different from TatPascalScripter or TatBasicScripter. So, if you want to keep the old default functionality, you must set OptionExplicit property to false.

2. ShortBooleanEval property now is "true" by default

The new TIDEScripter component automatically uses short boolean evaluation when evaluation boolean expressions. If you want to keep the old default functionality, set ShortBooleanEval to false.

In addition to the changes above, the new TatScripter and TIDEScripter includes the following properties and methods:

New DefaultLanguage property

```
TScriptLanguage = (slPascal, slBasic);  
property DefaultLanguage: TScriptLanguage;
```

TatScripter and descendants add the new property DefaultLanguage which is the default language of the scripts created in the scripter component using the old way (Scripter.Scripts.Add). Whenever a script object is created, the language of this new script will be specified by DefaultLanguage. The default value is slPascal. So, to emulate a TatBasicScripter component with TatScripter, just set DefaultLanguage to slBasic. If you want to use pascal language, it's already set for that.

New AddScript method

```
function AddScript(ALanguage: TScriptLanguage): TatScript;
```

If you create a script using old Scripts.Add method, the language of the script being created will be specified by DefaultLanguage. But as an alternative you can just call AddScript method, which will create a new TatScript object in the Scripts collection, but the language of the script will be specified by ALanguage parameter. So, for example, to create a Pascal and a Basic script in the TatScripter component:

```
MyPascalScript := atScripter1.AddScript(slPascal);  
MyBasicScript := atScripter1.AddScript(slBasic);
```

[C++Builder example](#)

Using cross-language feature

There is not much you need to do to be able to use both Basic and Pascal scripts. It's just transparent, from a Basic script you can call a Pascal procedure and vice-versa.

3.3 Common tasks

3.3.1 Calling a subroutine in script

If the script has one or more functions or procedures declared, than you can directly call them using ExecuteSubRoutine method:

```
Pascal script:  
procedure DisplayHelloWorld;  
begin  
    ShowMessage('Hello world!');  
end;
```

```
procedure DisplayByeWorld;  
begin  
    ShowMessage('Bye world!');  
end;
```

```
Basic script:  
sub DisplayHelloWorld  
    ShowMessage("Hello world!");  
end sub  
  
sub DisplayByeWorld  
    ShowMessage("Bye world!");  
end sub
```

CODE:

```
Scripter.ExecuteSubRoutine('DisplayHelloWorld');  
Scripter.ExecuteSubRoutine('DisplayByeWorld');
```

[C++Builder example](#)

This will display "Hello word!" and "Bye world!" message dialogs.

3.3.2 Returning a value from script

Execute method is a function, which result type is Variant. Thus, if script returns a value, then it can be read from Delphi code. For example, calling a script function "Calculate":

Pascal script:

```
function Calculate;  
begin  
    result:=(10+6)/4;  
end;
```

Basic script:

```
function Calculate  
    Calculate = (10+6)/4  
end function
```

CODE:

```
FunctionValue:=Scripter.ExecuteSubRoutine('Calculate');
```

FunctionValue will receive a value of 4. Note that you don't need to declare a function in order to return a value to script. Your script and code could be just:

Pascal script:

```
result:=(10+6)/4;
```

CODE:

```
FunctionValue:=Scripter.Execute;
```

[C++Builder example](#)

(*) In Basic syntax, to return a function value you must use "FunctionName = Value" syntax. You can also return values in basic without declaring a function. In this case, use the reserved word "MAIN" : "MAIN = (10+6)/4".

3.3.3 Passing parameters to script

Another common task is to pass values of variables to script as parameters, in order to script to use them. To do this, just use same Execute and ExecuteSubRoutine methods, with a different usage (they are overloaded methods). Note that parameters are Variant types:

Pascal script:

```
function Double(Num);  
begin  
    result:=Num*2;  
end;
```

Basic script:

```
function Double(Num)
```

```
    Double = Num*2
End function
```

CODE:

```
FunctionValue:=Scripter.ExecuteSubRoutine('Double', 5);
```

FunctionValue will receive 10. If you want to pass more than one parameter, use a Variant array or an array of const:

Pascal script:

```
function MaxValue(A,B);
begin
    if A>B then
        result:=A
    else
        result:=B;
end;

procedure Increase(var C; AInc);
begin
    C := C + AInc;
end;
```

CODE:

```
var
    MyVar: Variant;
begin
    FunctionValue:=Scripter.ExecuteSubRoutine('MaxValue',VarArrayOf([5,8]));
    Scripter.ExecuteSubRoutine('Increase',[MyVar, 3]);
end;
```

[C++Builder example](#)

NOTE: To use parameter by reference when calling script subroutines, the variables must be declared as variants. In the example above, the Delphi variable MyVar must be of Variant type, otherwise the script will not update the value of MyVar.

NOTE: Script doesn't need parameter types, you just need to declare their names.

3.4 Accessing Delphi objects

3.4.1 Registering Delphi components

One powerful feature of scripter is to access Delphi objects. This way you can make reference to objects in script, change its properties, call its methods, and so on. However, every object must be registered in scripter so you can access it. For example, suppose you want to change caption of form (named Form1). If you try to execute this script:

SCRIPT:

```
Form1.Caption:='New caption';
```

you will get "Unknown identifier or variable not declared: Form1". To make scripter work, use AddComponent method:

CODE:

```
Scripter.AddComponent(Form1);
```

[C++Builder example](#)

Now scripter will work and form's caption will be changed.

3.4.2 Access to published properties

After a component is added, you have access to its published properties. That's why the caption property of the form could be changed. Otherwise you would need to register property as well. Actually, published properties are registered, but scripter does it for you.

3.4.3 Class registering structure

Scripter can call methods and properties of objects. But this methods and properties must be registered in scripter. The key property for this is `TatCustomScripter.Classes` property. This property holds a collection of registered classes (`TatClass` object), which in turn holds its collection of registered properties and methods (`TatClass.Methods` and `TatClass.Properties`). Each registered method and property holds a name and the wrapper method (the Delphi written code that will handle method and property).

When you registered `Form1` component in the previous example, scripter automatically registered `Tform` class in `Classes` property, and registered all published properties inside it. To access methods and public properties, you must registered them, as showed in the following topics.

3.4.4 Calling methods

To call an object method, you need to register it. For instance, if you want to call `ShowModal` method of a newly created form named `form2`. So we must add the form it to scripter using `AddComponent` method, and then register `ShowModal` method:

CODE:

```
procedure TForm1.ShowModalProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(TCustomForm(CurrentObject).ShowModal);
end;

procedure TForm1.PrepareScript;
begin
    Scripter.AddComponent(Form2);
    With Scripter.DefineClass(TCustomForm) do
        begin
            DefineMethod('ShowModal', 0, tkInteger, nil, ShowModalProc);
        end;
end;
```

[C++Builder example](#)

SCRIPT:

```
ShowResult:=Form2.ShowModal;
```

This example has a lot of new concepts. First, component is added with `AddComponent` method. Then, `DefineClass` method was called to register `TCustomForm` class. `DefineClass` method automatically check if `TCustomForm` class is already registered or not, so you don't need to do test it.

After that, `ShowModal` is registered, using `DefineMethod` method. Declaration of `DefineMethod` is:

```
function DefineMethod(AName:string; AArgCount:integer; AResultDataType: TatTypeKind;
AResultClass:TClass; AProc:TMachineProc; AIsClassMethod:boolean=false): TatMethod;
```

AName receives 'ShowModal' – it's the name of method to be used in script.

AArgCount receives 0 – number of input arguments for the method (none, in the case of ShowModal)

AResultDataType receives tkInteger – it's the data type of method result. ShowModal returns an integer. If method is not a function but a procedure, AResultDataType should receive tkNone.

AResultClass receives nil – if method returns an object (not this case), then AResultClass must contain the object class. For example, TField.

AProc receives ShowModalProc – the method written by the user that works as ShowModal wrapper.

And, finally, there is ShowModalProc method. It is a method that works as the wrapper: it implements a call to ShowModal. In this case, it uses some useful methods and properties of TatVirtualMachine class:

property CurrentObject – contains the instance of object where the method belongs to. So, it contains the instance of a specified TCustomForm.

method ReturnOutputArg – it returns a function result to scripter. In this case, returns the value returned by TCustomForm.ShowModal method.

You can also register the parameter hint for the method using UpdateParameterHints method

3.4.5 More method calling examples

In addition to previous example, this one illustrates how to register and call methods that receive parameters and return classes. In this example, FieldByName:

SCRIPT:

```
AField:=Table1.FieldByName('CustNo');
ShowMessage(AField.DisplayLabel);
```

CODE:

```
procedure TForm1.FieldByNameProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ReturnOutputArg(integer(TDataset(CurrentObject).FieldByName(GetInputArgAsString(0))));
end;

procedure TForm1.PrepareScript;
begin
  Scripter.AddComponent(Table1);
  With Scripter.DefineClass(TDataset) do
    begin
      DefineMethod('FieldByName',1,tkClass,TField,FieldByNameProc);
    end;
end;
```

[C++Builder example](#)

Very similar to [Calling methods](#) example. Some comments:

- FieldByName method is registered in TDataset class. This allows use of FieldByName method by any TDataset descendant inside script. If FieldByName was registered in a TTable class, script would not recognize the method if component was a TQuery
- DefineMethod call defined that FieldByName receives one parameters, its result type is tkClass, and class result is TField.
- Inside FieldByNameProc, GetInputArgAsString method is called in order to get input parameters. The 0 index indicates that we want the first parameter. For methods that receive 2 or more parameters, use GetInputArg(1), GetInputArg(2), and so on.
- To use ReturnOutputArg in this case, we need to cast resulting TField as integer. This must be done to return any object. This is because ReturnOutputArg receives a Variant type, and objects must

then be cast to integer

3.4.6 Accessing non-published properties

Just like methods, properties that are not published must be registered. The mechanism is very similar to method registering, with the difference we must indicate one wrapper to get property value and another one to set property value. In the following example, the "Value" property of TField class is registered:

SCRIPT:

```
AField:=Table1.FieldByName('Company');
ShowMessage(AField.Value);
```

CODE:

```
procedure TForm1.GetFieldValueProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    ReturnOutputArg(TField(CurrentObject).Value);
end;

procedure TForm1.SetFieldValueProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
    TField(CurrentObject).Value:=GetInputArg(0);
end;

procedure TForm1.PrepareScript;
begin
  With Scripter.DefineClass(TField) do
    begin
      DefineProp('Value',tkVariant,GetFieldValueProc,SetFieldValueProc);
    end;
end;
```

[C++Builder example](#)

DefineProp is called passing a tkVariant indicating that Value property is Variant type, and then passing two methods GetFieldValueProc and SetFieldValueProc, which, in turn, read and write value property of a field object. Note that in SetFieldValueProc method was used GetInputArg (instead of GetInputArgAsString). This is because GetInputArg returns a variant.

3.4.7 Registering indexed properties

A property can be indexed, specially when it is a TCollection descendant. This applies to dataset fields, grid columns, string items, and so on. So, the code below illustrates how to register indexed properties. In this example, Strings property of TStrings object is added in other to change memo content:

SCRIPT:

```
ShowMessage(Mem1.Lines.Strings[3]);
Mem1.Lines.Strings[3]:=Mem1.Lines.Strings[3]+' with more text added';

//This is a comment
```

CODE:

```
procedure TForm1.GetStringsProc(AMachine: TatVirtualMachine);
begin
  with AMachine do
```

```

    returnOutputArg(TStrings(CurrentObject).Strings[GetArrayIndex(0)]);
end;

procedure TForm1.SetStringsProc(AMachine: TatVirtualMachine);
begin
    with AMachine do
        TStrings(CurrentObject).Strings[GetArrayIndex(0)] := GetInputArgAsString(0);
end;

procedure TForm1.PrepareScript;
begin
    Scripter.AddComponent(Memo1);
    with Scripter.DefineClass(TStrings) do
        begin
            DefineProp('Strings', tkString, GetStringsProc, SetStringsProc, nil, false, 1);
        end;
end;

```

[C++Builder example](#)

Some comments:

- DefineProp receives three more parameters than DefineMethod: **nil** (class type of property. It's nil because property is string type), **false** (indicating the property is not a class property) and **1** (indicating that property is indexed by 1 parameter. This is the key param. For example, to register Cells property of the grid, this parameter should be 2, since Cells depends on Row and Col).
- In GetStringsProc and SetStringsProc, GetArrayIndex method is used to get the index value passed by script. The 0 param indicates that it is the first index (in the case of Strings property, the only one).
- To define an indexed property as the default property of a class, set the property `TatClass.DefaultProperty` after defining the property in Scripter. In above script example (`Memo1.Lines.Strings[i]`), if the 'Strings' is set as the default property of TStrings class, the string lines of the memo can be accessed by "Memo1.Lines[i]". Code example (defining TStrings class with Strings default property):

```

procedure TForm1.PrepareScript;
begin
    Scripter.AddComponent(Memo1);
    with Scripter.DefineClass(TStrings) do
        begin
            DefaultProperty :=
                DefineProp('Strings', tkString, GetStringsProc, SetStringsProc, nil, false, 1);
        end;
end;

```

3.4.8 Retrieving name of called method or property

You can register the same wrapper for more than one method or property. In this case, you might need to know which property or method was called. In this case, you can use `CurrentPropertyName` or `CurrentMethodName`. The following example illustrates this usage

```

procedure TForm1.GenericMessageProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        If CurrentMethodName = 'MessageHello' then
            ShowMessage('Hello')
        Else if CurrentMethodName = 'MessageWorld' then
            ShowMessage('World');
end;

procedure TForm1.PrepareScript;
begin
    With Scripter do
        begin

```

```

    DefineMethod('MessageHello',1,tkNone,nil,GenericMessageProc);
    DefineMethod('MessageWorld',1,tkNone,nil,GenericMessageProc);
end;
end;

```

[C++Builder example](#)

3.4.9 Registering methods with default parameters

You can also register methods which have default parameters in scripter. To do that, you must pass the number of default parameters in the DefineMethod property. Then, when implementing the method wrapper, you need to check the number of parameters passed from the script, and then call the Delphi method with the correct number of parameters. For example, let's say you have the following procedure declared in Delphi:

```

function SumNumbers(A, B: double; C: double = 0; D: double = 0; E: double = 0):
double;

```

To register that procedure in scripter, you use DefineMethod below. Note that the number of parameters is 5 (five), and the number of default parameters is 3 (three):

```

Scripter.DefineMethod('SumNumbers', 5 {number of total parameters}, tkFloat, nil,
SumNumbersProc, false, 3 {number of default parameters});

```

Then, in the implementation of SumNumbersProc, just check the number of input parameters and call the function properly:

```

procedure TForm1.SumNumbersProc(AMachine: TatVirtualMachine);
begin
  With AMachine do
  begin
    Case InputArgCount of
      2: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1)));
      3: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
        GetInputArgAsFloat(2)));
      4: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
        GetInputArgAsFloat(2), GetInputArgAsFloat(3)));
      5: ReturnOutputArg(SumNumbers(GetInputArgAsFloat(0), GetInputArgAsFloat(1),
        GetInputArgAsFloat(2), GetInputArgAsFloat(3), GetInputArgAsFloat(4)));
    end;
  end;
end;

```

[C++Builder example](#)

3.5 Accessing Delphi functions, variables and constants

3.5.1 Overview

In addition to access Delphi objects, scripter allows integration with regular procedures and functions, global variables and global constants. The mechanism is very similar to accessing Delphi objects. In fact, scripter internally consider regular procedures and functions as methods, and global variables and constants are props.

3.5.2 Registering global constants

Registering a constant is a simple task in scripter: use AddConstant method to add the constant and the name it will be known in scripter:

CODE:

```

Scripter.AddConstant('MaxInt',MaxInt);

```

```
Scripter.AddConstant('Pi',pi);
Scripter.AddConstant('MyBirthday',EncodeDate(1992,5,30));
```

[C++Builder example](#)

SCRIPT:

```
ShowMessage('Max integer is '+IntToStr(MaxInt));
ShowMessage('Value of pi is '+FloatToStr(pi));
ShowMessage('I was born on '+DateToStr(MyBirthday));
```

Access the constants in script just like you do in Delphi code.

3.5.3 Accessing global variables

To register a variable in scripter, you must use AddVariable method. Variables can be added in a similar way to constants: passing the variable name and the variable itself. In addition, you can also add variable in the way you do with properties: use a wrapper method to get variable value and set variable value:

CODE:

```
var
    MyVar: Variant;
    ZipCode: string[15];

procedure TForm1.GetZipCodeProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(ZipCode);
end;

procedure TForm1.SetZipCodeProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ZipCode:=GetInputArgAsString(0);
end;

procedure TForm1.PrepareScript;
begin
    Scripter.AddVariable('ShortDateFormat',ShortDateFormat);
    Scripter.AddVariable('MyVar',MyVar);
    Scripter.DefineProp('ZipCode',tkString,GetZipCodeProc,SetZipCodeProc);
    Scripter.AddObject('Application',Application);
end;

procedure TForm1.Run1Click(Sender: TObject);
begin
    PrepareScript;
    MyVar:='Old value';
    ZipCode:='987654321';
    Application.Tag:=10;
    Scripter.SourceCode:=Memo1.Lines;
    Scripter.Execute;
    ShowMessage('Value of MyVar variable in Delphi is '+VarToStr(MyVar));
    ShowMessage('Value of ZipCode variable in Delphi is '+VarToStr(ZipCode));
end;
```

[C++Builder example](#)

SCRIPT:

```
ShowMessage('Today is '+DateToStr(Date)+' in old short date format');
ShortDateFormat:='dd-mmmm-yyyy';
```

```
ShowMessage('Now today is '+DateToStr(Date)+' in new short date format');

ShowMessage('My var value was "'+MyVar+'");
MyVar:='My new var value';

ShowMessage('Old Zip code is '+ZipCode);
ZipCode:='109020';

ShowMessage('Application tag is '+IntToStr(Application.Tag));
```

3.5.4 Calling regular functions and procedures

In scripter, regular functions and procedures are added like methods. The difference is that you don't add the procedure in any class, but in scripter itself, using DefineMethod method. The example below illustrates how to add QuotedStr and StringOfChar methods:

SCRIPT:

```
ShowMessage(QuotedStr(StringOfChar('+',3)));
```

CODE:

```
{ TSomeLibrary }
procedure TSomeLibrary.Init;
begin
    Scripter.DefineMethod('QuotedStr',1,tkString,nil,QuotedStrProc);
    Scripter.DefineMethod('StringOfChar',2,tkString,nil,StringOfCharProc);
end;

procedure TSomeLibrary.QuotedStrProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(QuotedStr(GetInputArgAsString(0)));
end;

procedure TSomeLibrary.StringOfCharProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(StringOfChar(GetInputArgAsString(0)[1],GetInputArgAsInteger(1)
));
end;

procedure TForm1.Run1Click(Sender: TObject);
begin
    Scripter.AddLibrary(TSomeLibrary);
    Scripter.SourceCode:=Memol.Lines;
    Scripter.Execute;
end;
```

[C++Builder example](#)

Since there is no big difference from defining methods, the example above introduces an extra concept: libraries. Note that the way methods are defined didn't change (a call to DefineMethod) and neither the way wrapper are implemented (QuotedStrProc and StringOfCharProc). The only difference is the way they are located: instead of TForm1 class, they belong to a different class named TSomeLibrary. The following topic covers the use of libraries

3.6 Script-based libraries

Script-based library is the concept where a script can "use" other script (to call procedures, set global variables, etc..

Take, for example, the following scripts:

```
//Script 1
uses Script2;

begin
  Script2GlobalVar := 'Hello world!';
  ShowScript2Var;
end;

//Script2
var
  Script2GlobalVar: string;

procedure ShowScript2Var;
begin
  ShowMessage(Script2GlobalVar);
end;
```

When you execute the first script, it "uses" Script2, and then it is able to read/write global variables and call procedures from Script2.

The only issue here is that script 1 must "know" where to find Script2.

When the compiler reaches a identifier in the uses clause, for example:

```
uses Classes, Forms, Script2;
```

Then it tries to "load" the library in several ways. This is the what the compiler tries to do, in that order:

1. Tries to find a registered Delphi-based library with that name.

In other words, any library that was registered with RegisterScripterLibrary. This is the case for the imported VCL that is provided with Scripter Studio, and also for classes imported by the import tool. This is the case for Classes, Forms, and other units.

2. Tries to find a script in Scripts collection where UnitName matches the library name

Each TScript object in the Scripter.Scripts collection has a UnitName property. You can manually set that property so that the script object is treated as a library in this situations. In the example above, you could add a script object, set its SourceCode property to the script 2 code, and then set UnitName to 'Script2'. This way, the script1 could find the script2 as a library and use its variables and functions.

3. Tries to find a file which name matches the library name (if LibOptions.UseScriptFiles is set to true)

If LibOptions.UseScriptFiles is set to true, then the scripter tries to find the library in files. For example, if the script has "uses Script2;", it looks for files named "Script2.psc". There are several sub-options for this search, and LibOptions property controls this options:

LibOptions.SearchPath:

It is a TStrings object which contains file paths where the scripter must search for the file. It accepts two constants: "\$(CURDIR)" (which contains the current directory) and "\$(APPDIR)" (which contains the application path).

LibOptions.SourceFileExt:

Default file extension for source files. So, for example, if sourcefileext is ".psc", the scripter will look for

a file named Script2.psc. The scripter looks first for compiled files, then source files.

LibOptions.CompileFileExt:

Default file extension for compiled files. So, for example, if compilefileext is ".pcu", the scripter will look for a file name Script2.pcu. The scripter looks first for compiled files, then source files.

LibOptions.UseScriptFiles:

Turns on/off the support for script files. If UseScriptFiles is false, then the scripter will not look for files.

3.7 Declaring forms in script

A powerful feature in scripter is the ability to declare forms and use dfm files to load form resources. With this feature you can declare a form to use it in a similar way than Delphi: you create an instance of the form and use it.

Take the following scripts as an example:

```
//Main script
uses
  Classes, Forms, MyFormUnit;

var
  MyForm: TMyForm;
begin
  {Create instances of the forms}
  MyForm := TMyForm.Create(Application);

  {Initialize all forms calling its Init method}
  MyForm.Init;

  {Set a form variable. Each instance has its own variables}

  MyForm.PascalFormGlobalVar := 'my instance';

  {Call a form "method". You declare the methods in the form script like procedures}
  MyForm.ChangeButtonCaption('Another click');

  {Accessing form properties and components}
  MyForm.Edit1.Text := 'Default text';

  MyForm.Show;
end;

//My form script
{$FORM TMyForm, myform.dfm}

var
  MyFormGlobalVar: string;

procedure Button1Click(Sender: TObject);
begin
  ShowMessage('The text typed in Edit1 is ' + Edit1.Text +
    #13#10 + 'And the value of global var is ' + MyFormGlobalVar);
end;

procedure Init;
begin
  MyFormGlobalVar := 'null';
  Button1.OnClick := 'Button1Click';
end;

procedure ChangeButtonCaption(ANewCaption: string);
begin
  Button1.Caption := ANewCaption;
end;
```

The sample scripts above show how to declare forms, create instances, and use their "methods" and variables. The second script is treated as a regular [script-based library](#), so it follows the same concept of registering and using. See the related topic for more info.

The \$FORM directive is the main piece of code in the form script. This directive tells the compiler that the current script should be treated as a form class that can be instantiated, and all its variables and procedures should be treated as form methods and properties. The directive should be in the format {\$FORM FormClass, FormFileName}, where FormClass is the name of the form class (used to create instances, take the main script example above) and FormFileName is the name of a dfm form which should be loaded when the form is instantiated. The dfm form file is searched the same way that other script-based libraries, in other words, it uses LibOptions.SearchPath to search for the file.

As an option to load dfm files, you can set the form resource through TatScript.DesignFormResource string property. So, in the TatScript object which holds the form script source code, you can set DesignFormResource to a string which contains the dfm-file content in [binary](#) format. If this property is not empty, then the compiler will ignore the dfm file declared in \$FORM directive, and will use the DesignFormResource string to load the form.

The dfm file is a regular Delphi-dfm file format, in text format. You cannot have event handlers define in the dfm file, otherwise a error will raise when loading the dfm.

Another thing you must be aware of is that all existing components in the dfm form must be previously registered. So, for example, if the dfm file contains a TEdit and a TButton, you must add this piece of code in your application (only once) before loading the form:

```
RegisterClasses([TEdit, TButton]);
```

Otherwise, a "class not registered" error will raise when the form is instantiated.

3.8 Declaring classes in script

It's now possible to declare classes in a script. With this feature you can declare a class to use it in a similar way than Delphi: you create an instance of the class and reuse it.

Declaring the class

Each class must be declared in a separated script, in other words, you need to have a script for each class you want to declare.

You turn the script into a "class script" by adding the \$CLASS directive in the beginning of the script, followed by the class name:

```
//Turn this script into a class script for TSomeClass  
{ $CLASS TSomeClass }
```

Methods and properties

Each global variable declared in a class script actually becomes a property of the class. Each procedure/function in script becomes a class method.

The main routine of the script is always executed when a new instance of the class is created, so it can be used as a class initializer and you can set some properties to default value and do some proper class initialization.

```
//My class script  
{ $CLASS TMyClass }  
uses Dialogs;  
  
var  
  MyProperty: string;  
  
procedure SomeMethod;
```

```
begin
  ShowMessage('Hello, world!');
end;

// class initializer
begin
  MyProperty := 'Default Value';
end;
```

Using the classes

You can use the class from other scripts just by creating a new instance of the named class:

```
uses MyClassScript;
var
  MyClass: TMyClass;
begin
  MyClass := TMyClass.Create;
  MyClass.MyProperty := 'test';
  MyClass.SomeMethod;
end;
```

Implementation details

The classes declared in script are "pseudo" classes. This means that no new Delphi classes are created, so for example although in the sample above you call `TMyClass.Create`, the "TMyClass" name is just meaning to the scripting system, there is no Delphi class named `TMyClass`. All objects created as script-based classes are actually instances of the class `TScriptBaseObject`. You can change this behavior to make instances of another class, but this new class must inherit from `TScriptBaseObject` class. You define the base class for all "pseudo"-classes objects in scripiter property `ScriptBaseObjectClass`.

Memory management

Although you can call `Free` method in scripts to release memory associated with instances of script-based classes, you don't need to do that. All objects created in script that are based on script classes are eventually destroyed by the scripiter component.

Limitations

Since scripiter doesn't create new real Delphi classes, there are some limitations about what you can do with it. The main one is that inheritance is not supported. Since all classes in script are actually the same Delphi class, you can't create classes that inherit from any other Delphi class except the one declared in `TScriptBaseObject` class.

3.9 Using the Refactor

Every `TatScript` object in `Scripter.Scripts` collection has its own refactor object, accessible through `Refactor` property. The Refactor object is just a collection of methods to make it easy and safe to change source code. As long as new versions of Scripter Studio is released, some new refactoring methods might be added. For now, these are the current available methods:

{Create (or update) the FORM directive in the script giving the AFormClass (form class name) and AFileName (form file name). For example, the code below:

```
UpdateFormHeader('TMyForm', 'myform.dfm');
```

will create (or update) the form directive in the script as following (in this case, the example is in Basic syntax):

```
#FORM TMyForm, myform.dfm}
procedure UpdateFormHeader(AFormClass, AFileName: string); virtual;
```

{Declare a routine named ProcName in source code, and return the line number of the declared routine. The line number returned is not the line where the routine is declared, but the line with the first statement. For example, in pascal, it returns the line after the "begin" of the procedure}

```
function DeclareRoutine(ProcName: string): integer; overload;
```

{Declare a routine in source code, and return the line number of the declared routine. The line number returned is not the line where the routine is declared, but the line with the first statement. For example, in pascal, it returns the line after the "begin" of the procedure.

This method uses the AInfo property to retrieve information about the procedure to be declared. Basically it uses AInfo.Name as the name of routine to be declared, and also uses AInfo.Variables to declare the parameters. This is a small example:

```
AInfo.Name := 'MyRoutine';
AInfo.IsFunction := true;
AInfo.ResultTypeDecl := 'string';
With AInfo.Variables.Add do
begin
  VarName := 'MyParameter';
  Modifier := moVar;
  TypeDecl := 'integer';
end;
With AInfo.Variables.Add do
begin
  VarName := 'SecondPar';
  Modifier := moNone;
  TypeDecl := 'TObject';
end;
ALine := Script.DeclareRoutine(AInfo);
```

The script above will declare the following routine (in pascal):

```
function MyRoutine(var MyParameter: integer; SecondPar: TObject): string; }
function DeclareRoutine(AInfo: TatRoutineInfo): integer; overload; virtual;
```

{Add the unit named AUnitName to the list of used units in the uses clause. If the unit is already used, nothing is done. If the uses clause is not present in the script, it is included. Example:

```
AddUsedUnit('Classes');}
procedure AddUsedUnit(AUnitName: string); virtual;
```

3.10 Using libraries

3.10.1 Overview

Libraries are just a concept of extending scripiter by adding more components, methods, properties, classes to be available from script. You can do that by manually registering a single component, class or method. A library is just a way of doing that in a more organized way.

3.10.2 Delphi-based libraries

In script, you can use libraries for registered methods and properties. Look at the two codes below, the first one uses libraries and the second use the mechanism used in this doc until now:

CODE 1:

```
type
  TExampleLibrary = class(TatScripiterLibrary)
  protected
    procedure CurrToStrProc(AMachine: TatVirtualMachine);
    procedure Init; override;
    class function LibraryName: string; override;
  end;
```

```

class function TExampleLibrary.LibraryName: string;
begin
    result := 'Example';
end;

procedure TExampleLibrary.Init;
begin
    Scripter.DefineMethod('CurrToStr', 1, tkInteger, nil, CurrToStrProc);
end;

procedure TExampleLibrary.CurrToStrProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(CurrToStr(GetInputArgAsFloat(0)));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    Scripter.AddLibrary(TExampleLibrary);
    Scripter.SourceCode:=Memol.Lines;
    Scripter.Execute;
end;

```

CODE 2:

```

procedure TForm1.PrepareScript;
begin
    Scripter.DefineMethod('CurrToStr', 1, tkInteger, nil, CurrToStrProc);
end;

procedure TForm1.CurrToStrProc(AMachine: TatVirtualMachine);
begin
    With AMachine do
        ReturnOutputArg(CurrToStr(GetInputArgAsFloat(0)));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    PrepareScript;
    Scripter.SourceCode:=Memol.Lines;
    Scripter.Execute;
end;

```

[C++Builder example](#)

Both codes do the same: add CurrToStr procedure to script. Note that scripeter initialization method (Init in Code 1 and PrepareScript in Code 2) is the same in both codes. And so is CurrToStrProc method – no difference. The two differences between the code are:

- The class where the methods belong to. In Code1, methods belong to a special class named TExampleLibrary, which descends from TatScripterLibrary. In Code 2, the belong to the current form (TForm1).
- In Code 1, scripeter preparation is done adding TExampleLibrary class to scripeter, using AddLibrary method. In Code 2, PrepareScript method is called directly.

So when to use one way or another? There is no rule – use the way you feel more comfortable. Here are pros and cons of each:

Declaring wrapper and preparing methods in an existing class and object:

- Pros: More convenient. Just create a method inside form, or datamodule, or any object.
- Cons: When running script, you must be sure that object is instantiated. It's more difficult to reuse code (wrapper and preparation methods)

Using libraries, declaring wrapper and preparing methods in a TatScripiterLibrary class descendant:

- Pros: No need to check if class is instantiated – scripiter does it automatically. It is easy to port code – all methods are inside a class library, so you can add it in any scripiter you want, put it in a separate unit, etc..
- Cons: Just the extra work of declaring the new class

In addition to using AddLibrary method, you can use RegisterScripiterLibrary procedure. For example:

```
RegisterScripiterLibrary(TExampleLibrary);  
RegisterScripiterLibrary(TAnotherLibrary, True);
```

RegisterScripiterLibrary is a global procedure that registers the library in a global list, so all scripiter components are aware of that library. The second parameter of RegisterScripiterLibrary indicates if the library is load automatically or not. In the example above, TAnotherLibrary is called with Explicit Load (True), while TExampleLibrary is called with Explicit Load false (default is false).

When explicit load is false (case of TExampleLibrary), every scripiter that is instantiated in application will automatically load the library.

When explicit load is true (case of TAnotherLibrary), user can load the library dinamically by using uses directive:

SCRIPT:

```
Uses Another;
```

```
//Do something with objects and procedures register by TatAnotherLibrary
```

Note that "Another" name is informed by TatAnotherLibrary.LibraryName class method.

3.10.3 The TatSystemLibrary library

There is a library that is added by default to all scripiter components, it is the TatSystemLibrary. This library is declared in the uSystemLibrary unit. It adds commonly used routines and functions to scripiter, such like ShowMessage and IntToStr.

Functions added by TatSystemLibrary

The following functions are added by the TatSystemLibrary (refer to Delphi documentation for an explanation of each function):

```
Abs  
AnsiCompareStr  
AnsiCompareText  
AnsiLowerCase  
AnsiUpperCase  
Append  
ArcTan  
Assigned  
AssignFile  
Beep  
Chdir  
Chr  
CloseFile  
CompareStr  
CompareText  
Copy  
Cos  
CreateOleObject
```

Date
DateTimeToStr
DateToStr
DayOfWeek
Dec
DecodeDate
DecodeTime
Delete
EncodeDate
EncodeTime
EOF
Exp
FilePos
FileSize
FloatToStr
Format
FormatDateTime
FormatFloat
Frac
GetActiveOleObject
High
Inc
IncMonth
InputQuery
Insert
Int
Interpret (*)
IntToHex
IntToStr
IsLeapYear
IsValidIdent
Length
Ln
Low
LowerCase
Machine (*)
Now
Odd
Ord
Pos
Raise
Random
ReadLn
Reset
Rewrite
Round
Scripter (*)
SetOf (*)
ShowMessage
Sin
Sqr
Sqrt
StrToDate
StrToDateTime
StrToFloat
StrToInt
StrToIntDef
StrToTime

Time
TimeToStr
Trim
TrimLeft
TrimRight
Trunc
UpperCase
VarArrayCreate
VarArrayHighBound
VarArrayLowBound
VarIsNull
VarToStr
Write
WriteLn

All functions/procedures added are similar to the Delphi ones, with the exception of those marked with a "*", explained below:

```
procedure Interpret(Ascript: string);  
Executes the script source code specified by Ascript parameter
```

```
function Machine: TatVirtualMachine;  
Returns the current virtual machine executing the script.
```

```
function Scripter: TatCustomScripter;  
Returns the current scripter component.
```

```
function SetOf(array): integer;  
Returns a set from the array passed. For example:
```

```
MyFontStyle := SetOf([fsBold, fsItalic]);
```

3.10.4 Removing functions from the System library

To remove a function from the system library, avoiding the end-user to use the function from the script, you just need to destroy the associated method object in the SystemLibrary class:

```
MyScripter.SystemLibrary.MethodByName('ShowMessage').Free;
```

[C++Builder example](#)

3.10.5 The TatVBScriptLibrary library

The TatVBScriptLibrary adds many VBScript-compatible functions. It's useful to give to your end-user access to the most common functions used in VBScript, making it easy to write Basic scripts for those who are already used to VBScript.

How to use TatVBScriptLibrary

Ulinke to TatSystemLibrary, the TatVBScriptLibrary is not automatically added to scripter components. To add the library to scripter and thus make use of the functions, you just follow the regular steps described in the section [Delphi-based libraries](#), which are described here again:

a) First, you must use the uVBScriptLibrary unit in your Delphi code:

```
Uses uVBScriptLibrary;
```

b) Then you just add the library to the scripter component, from Delphi code:

```
atBasicScripter1.AddLibrary(TatVBScriptLibrary);
```

or, enable the VBScript libraries from the script code itself, by adding VBScript in the uses clause:

```
'My Basic Script  
uses VBScript
```

Functions added by TatVBScriptLibrary

The following functions are added by the TatVBScriptLibrary (refer to MSDN documentation for the explanation of each function):

- Asc
- Atn
- CBool
- CByte
- CCur
- CDate
- CDbl
- Cint
- CLng
- CreateObject
- CSng
- CStr
- DatePart
- DateSerial
- DateValue
- Day
- Fix
- FormatCurrency
- FormatDateTime
- FormatNumber
- Hex
- Hour
- InputBox
- InStr
- Int
- IsArray
- IsDate
- IsEmpty
- IsNull
- IsNumeric
- LBound
- LCase
- Left
- Len
- Log
- LTrim
- Mid
- Minute
- Month
- MonthName
- MsgBox
- Replace
- Right
- Rnd
- RTrim
- Second
- Sgn
- Space
- StrComp

String
Timer
TimeSerial
TimeValue
UBound
UCase
Weekday
WeekdayName
Year

3.11 Debugging scripts

3.11.1 Overview

Scripter Studio contains components and methods to allow run-time script debugging. There are two major ways to debug scripts: using scripter component methods and properties, or using debug components. Use of methods and properties gives more flexibility to programmer, and you can use them to create your own debug environment. Use of components is a more high-level debugging, where in most of case all you need to do is drop a component and call a method to start debugging.

3.11.2 Using methods and properties for debugging

Scripter component has several properties and methods that allows script debugging. You can use them inside Delphi code as you want. They are listed here:

```
property Running: boolean;
```

read/write property. While script is being executed, running is true. Note that the script might be paused but still running. Set running to true is equivalent to call Execute method.

```
property Paused: boolean read GetPaused write SetPaused;
```

read/write property. Use it to pause script execution, or get script back to execution.

```
procedure DebugTraceIntoLine;
```

Executes only current line. If the line contains a call to a subroutine, execution point goes to first line of subroutine. Similar to Trace Into option in Delphi.

```
procedure DebugStepOverLine;
```

Executes only current line and execution point goes to next line in code. If the current line contains a call to a subroutine, it executes the whole subroutine. Similar to Step Over option in Delphi.

```
procedure DebugRunUntilReturn;
```

Executes code until the current subroutine (procedure, function or script main block) is finished. Execution point stops one line after the line which called the subroutine.

```
procedure DebugRunToLine(ALine: integer);
```

Executes script until line specified by ALine. Similar to Run to Cursor option in Delphi.

```
function DebugToggleBreakLine(ALine: integer): pSimplifiedCode;
```

Enable/disable a breakpoint at the line specified by ALine. Execution stops at lines which have breakpoints set to true.

```
function DebugExecutionLine: integer;
```

Return the line number which will be executed.

```
procedure Halt;
```

Stops script execution, regardless the execution point.

```
property Halted: boolean read GetHalted;
```

This property is true in the short time period after a call to Halt method and before script is effectively terminated.

```
property BreakPoints: TatScriptBreakPoints read GetBreakPoints;
```

Contains a list of breakpoints set in script. You can access breakpoints using `Items[Index]` property, or using method `BreakPointByLine(ALine: integer)`. Once you access the breakpoint, you can set properties `Enabled` (which indicates if breakpoint is active or not) and `PassCount` (which indicates how many times the execution flow will pass through breakpoint until execution is stopped).

```
property OnDebugHook: TNotifyEvent read GetOnDebugHook write SetOnDebugHook;
```

During debugging (step over, step into, etc.) `OnDebugHook` event is called for every step executed.

```
property OnPauseChanged: TNotifyEvent read GetOnPauseChanged write  
SetOnPauseChanged;
```

```
property OnRunningChanged: TNotifyEvent read GetOnRunningChanged write  
SetOnRunningChanged;
```

These events are called whenever paused or running properties change.

3.11.3 Using debug components

Scripter Studio has specific component for debugging. It is `TatScriptDebugDlg`. Its usage is very simple: drop it on a form and assign its `Scripter` property to an existing script component. Call `Execute` method and a debug dialog will appear, displaying script source code and with a toolbar at the top of window. You can then use tool buttons or shortcut keys to perform debug actions (run, pause, step over, and so on). Shortcut keys are the same used in Delphi:

- F4 – Run to cursor
- F5 – Toggle breakpoint
- F7 – Step into
- F8 – Step Over
- F9 – Run
- Shift+F9 - Pause
- Ctrl+F2 – Reset
- Shift+F11 – Run until return

3.12 Form-aware scripters - `TatPascalFormScripter` and `TatBasicFormScripter`

`TatPascalFormScripter` and `TatBasicFormScripter` are scripters that descend from `TatPascalScripter` and `TatBasicScripter` respectively. They have the same functionality of their ancestor, but in addition they already have registered the components that are owned by the form where scripter component belongs to.

So, if you want to use scripter to access components in the form, like buttons, edits, etc., you can use form-aware scripter without needing to register form components.

3.13 C++ Builder issues

3.13.1 Overview

Since Scripter Studio works with objects and classes types and typecasting, it might be some tricky issues to do some tasks in C++ Builder. This section provides useful information on how to write C++ code to perform some common tasks with Scripter Studio.

3.13.2 Registering a class method for an object

Let's say you have created a class named testclass, inherited from TObject:

```
[in .h file]
class testclass : public TObject
{
public:
AnsiString name;
int number;
virtual __fastcall testclass();
};

[in .cpp file]
__fastcall testclass::testclass()
: TObject()
{
this->name = "test";
this->number = 10;
ShowMessage("In constructor");
}
```

If you want to add a class method "Create" which will construct a testclass from script and also call the testclass() method, you must register the class in script registration system:

```
scr->DefineMethod("create", 0, Typinfo::tkClass, __classid(testclass), constProc, true);
```

Now you must implement constProc method which will implement the constructor method itself:

```
void __fastcall TForm1::constProc(TatVirtualMachine* avm)
{
testclass *l_tc;

l_tc = (testclass *) avm->CurrentObject;
l_tc = new testclass;
avm->ReturnOutputArg((long)(l_tc));
}
```

Chapter



IV

**The syntax
highlighting memo**

4 The syntax highlighting memo

4.1 Using the memo

TAdvMemo provides syntax highlighting for your Pascal or Basic scripts. To start using the memo component, drop the memo component on the form together with either an AdvPascalMemoStyler or an AdvBasicMemoStyler component. Assign the AdvPascalMemoStyler or the AdvBasicMemoStyler to the TAdvMemo.SyntaxStyles property. Upon assigning, the text in the memo will be rendered with the syntax highlighting chosen. You can also programmatically switch the syntax highlighting by assigning at runtime a memo styler components:

```
AdvMemo1.SyntaxStyles := AdvPascalMemoStyler;
```

To change the colors of the syntax highlighting, the various properties of the language elements are kept in the TAdvPascalMemoStyler or TAdvBasicMemoStyler. Text and background colors and font can be set for comments, numbers in the MemoStyler properties or for keywords, symbols or values between brackets in the AllStyles properties. TAdvPascalMemoStyler or TAdvBasicMemoStyler have predefined keywords for the Pascal language or Basic language. If colors need to be changed for custom introduced keywords in the scripter, this can be easily done by adding a TElementStyle in the AllStyles property. Set the styletype to stKeyword and add the keywords to the Keywords stringlist.

TAdvMemo has a gutter that displays the line numbers of the source code. In addition it can also display executable code, breakpoints and active source code line. This is done automatically in the TAtScriptDebugDlg component that uses the TAdvMemo for displaying the source code. It can be done separately through following public properties:

```
TAdvMemo.ActiveLine: Integer;
```

Sets the active source code line. This line will be displayed in active line colors

```
TAdvMemo.BreakPoints[RowIndex]: Boolean;
```

When true, the line in source code has a breakpoint. Only executable lines can have breakpoints. It is through the scripter engine debug interfaces that you can retrieve whether a line is executable or not. A breakpoint is displayed as a red line with white font.

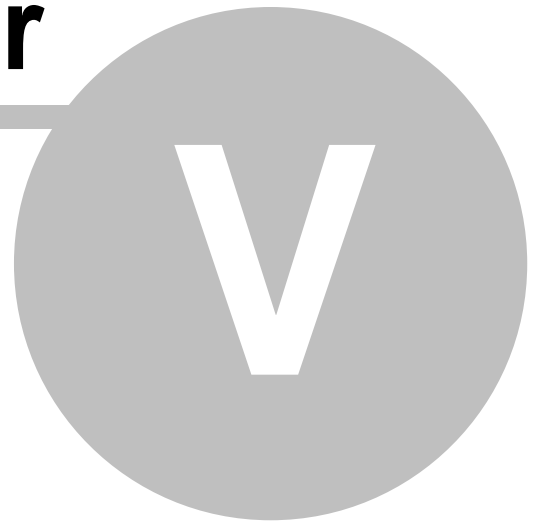
```
TAdvMemo.Executable[RowIndex]: Boolean;
```

When true, a marker is displayed in the gutter that the line is executable.

Using the memo with scripter is as easy as assigning the AdvMemo lines to the scripter SourceCode property and execute the code:

```
AtPascalScripter.SourceCode.Assign(AdvMemo.Lines);  
AtPascalScripter.Execute;
```

Chapter



**TSourceExplorer
component**

5 TSourceExplorer component

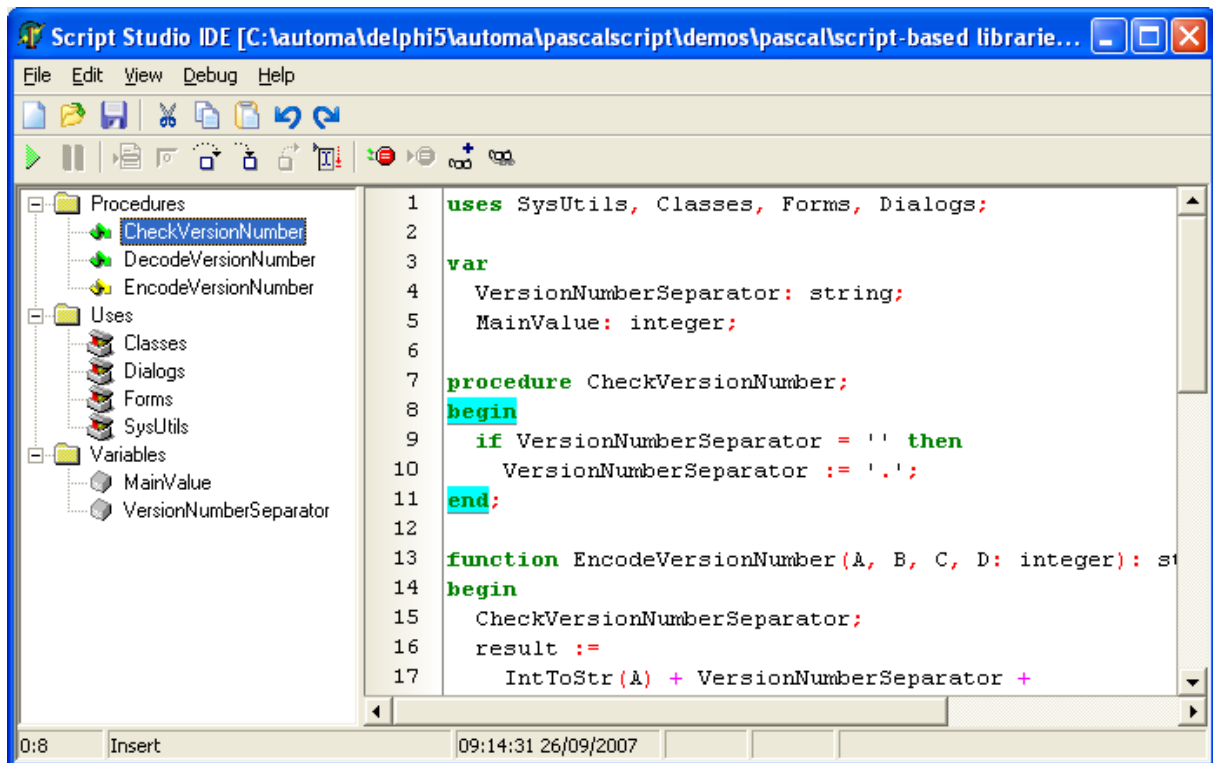
5.1 Overview

The TSourceExplorer component is a treeview-like component which display the structure of the script, just like the Source Explorer (or Code Structure) in Delphi IDE. The component displays:

- The procedures and functions declared in script;
- Units used in script (in the uses clause);
- Global variables declared in script.

The tree items are updated automatically as you change the source code. It has a timer and thread system so that it updates itself in background, like in Delphi IDE. When you double-click an item the source explorer, it automatically puts the cursor in the correct position in the source code memo, at the place where the item is declared.

The picture below displays the TSourceExplorer component, at the left of the window.



5.2 Using the component

Using TSourceExplorer component is easy:

1. Drop a TSourceExplorer component in the form
2. Assign the Scripter property to a scripter component

3. Assign the AdvMemo property to a TAdvMemo component (optional)

The component handles everything automatically. It wraps the OnChange event of the memo, and whenever the source code is changed in the memo, the tree uses the Scripter component to check the script information and updates the tree itself.

If you don't perform the optional item 3 above (assigning the memo), then you have to ask for the source explorer to update itself, by calling the method UpdateTree, and passing the source code of the script:

```
SourceExplorer1.UpdateTree(MySourceCodeStrings);
```

TSourceExplorer component descends from TCustomTreeView, so almost all of properties of a TTreeView component can be found in the TSourceExplorer component (like Align, HideSelection, ChangeDelay, OnCollapsed, etc.).

Here is a summary of the key properties of TSourceExplorer component:

Properties

```
property Scripter: TCustomScripter;
```

You must set scripter property to a scripter component so that the source explorer and compile the script and retrieve information about it.

```
property AdvMemo: TAdvMemo;
```

You can assign a TAdvMemo component to AdvMemo property. This makes the source explorer to work more automatically, retrieving the source code from the memo and allowing integration with double click. This property is optional, if you don't assign a memo to this property, you will have to call UpdateTree method to update the structure tree of source explorer.

Methods

```
procedure UpdateTree; overload;  
procedure UpdateTree(ASource: TStrings); overload;
```

Call UpdateTree to update the structure tree of source explorer. You can pass the source code in ASource parameter. If AdvMemo property is assigned, then you can call UpdateTree without any parameter, the TSourceExplorer component will retrieve the source code from the TAdvMemo.

Chapter



VI

**C++Builder
Examples**

6 C++Builder Examples

This section contains C++Builder examples equivalent to every Delphi example in this manual. Each example provides a link to the related topic, and vice versa.

6.1 Working with scripter

6.1.1 Getting started

```
Scripter->SourceCode->Text = "ShowMessage('Hello world!');";  
Scripter->Execute();
```

```
Scripter->SourceCode->Text = AdvMemol->Lines->Text;  
Scripter->Execute();
```

[Original topic](#)

6.1.2 Cross-language feature: TatScripter and TIDEScripter

```
TatScript *MyPascalScript, *MyBasicScript;  
  
MyPascalScript = atScripter1->AddScript(slPascal);  
MyBasicScript = atScripter1->AddScript(slBasic);
```

[Original topic](#)

6.1.3 Common tasks

6.1.3.1 Calling a subroutine in script

```
Scripter->ExecuteSubroutine("DisplayHelloWorld");  
Scripter->ExecuteSubroutine("DisplayByeWorld");
```

[Original topic](#)

6.1.3.2 Returning a value from script

```
Variant FunctionValue;  
  
FunctionValue = Scripter->ExecuteSubroutine("Calculate");  
  
FunctionValue = Scripter->Execute();
```

[Original topic](#)

6.1.3.3 Passing parameters to script

```
Variant FunctionValue;  
  
FunctionValue = Scripter->ExecuteSubroutine("Double", 5);  
  
Variant MyVar;  
  
FunctionValue = Scripter->ExecuteSubroutine("MaxValue",  
  VarArrayOf(OPENARRAY(Variant, (5, 8))));  
Scripter->ExecuteSubroutine("Increase", VarArrayOf(  
  OPENARRAY(Variant, (MyVar, 3))));
```

[Original topic](#)

6.1.4 Accessing Delphi objects

6.1.4.1 Registering Delphi components

```
Scripter->AddComponent(Form1);
```

[Original topic](#)

6.1.4.2 Calling methods

```
void __fastcall TForm1::ShowModalProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg((TCustomForm*)
    AMachine->CurrentObject)->ShowModal();
}

void __fastcall TForm1::PrepareScript()
{
    Scripter->AddComponent(Form2);
    TatClass *customFormClass = Scripter->DefineClass(__classid(TCustomForm));
    customFormClass->DefineMethod("ShowModal", 0, Atscript::tkInteger, NULL,
    ShowModalProc);
}
```

[Original topic](#)

6.1.4.3 More method calling examples

```
void __fastcall TForm1::FieldByNameProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg((long) ((TDataSet*)
    AMachine->CurrentObject)->FieldByName(AMachine->GetInputArgAsString(0)));
}

void __fastcall TForm1::PrepareScript()
{
    Scripter->AddComponent(Table1);
    TatClass *datasetClass = Scripter->DefineClass(__classid(TDataSet));
    datasetClass->DefineMethod("FieldByName", 1, Atscript::tkClass,
    __classid(TField), FieldByNameProc);
}
```

[Original topic](#)

6.1.4.4 Accessing non-published properties

```
void __fastcall TForm1::GetFieldValueProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg((TField*) AMachine->CurrentObject)->Value;
}

void __fastcall TForm1::SetFieldValueProc(TatVirtualMachine *AMachine)
{
    ((TField*) AMachine->CurrentObject)->Value = AMachine->GetInputArg(0);
}

void __fastcall TForm1::PrepareScript()
{
    TatClass *fieldClass = Scripter->DefineClass(__classid(TField));
    fieldClass->DefineProp("Value", Atscript::tkVariant, GetFieldValueProc,
    SetFieldValueProc);
}
```

[Original topic](#)

6.1.4.5 Registering indexed properties

```
void __fastcall TForm1::GetStringsProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg((TStrings*)
    AMachine->CurrentObject)->Strings[AMachine->GetArrayIndex(0)];
}
```

```

}

void __fastcall TForm1::SetStringsProc(TatVirtualMachine *AMachine)
{
  ((TStrings*) AMachine->CurrentObject)->Strings[AMachine->GetArrayIndex(0)] =
  AMachine->GetInputArgAsString(0);
}

void __fastcall TForm1::PrepareScript()
{
  Scripter->AddComponent(Mem01);
  TatClass *stringsClass = Scripter->DefineClass(__classid(TStrings));
  stringsClass->DefineProp("Strings", Atscript::tkString, GetStringsProc,
  SetStringsProc, NULL, false, 1);
}

```

[Original topic](#)

6.1.4.6 Retrieving name of called method or property

```

void __fastcall TForm1::GenericMessageProc(TatVirtualMachine *AMachine)
{
  if(AMachine->CurrentMethodName() == "MessageHello")
    ShowMessage("Hello");
  else if(AMachine->CurrentMethodName() == "MessageWorld")
    ShowMessage("World");
}

void __fastcall TForm1::PrepareScript()
{
  Scripter->DefineMethod("MessageHello", 1, tkNone, NULL, GenericMessageProc);
  Scripter->DefineMethod("MessageWorld", 1, tkNone, NULL, GenericMessageProc);
}

```

[Original topic](#)

6.1.4.7 Registering methods with default parameters

```

float SumNumbers(float a, float b, float c = 0, float d = 0, float e = 0);

Scripter->DefineMethod("SumNumbers",
  5 /*number of total parameters*/,
  Atscript::tkFloat, NULL, SumNumbersProc, false,
  3 /*number of default parameters*/);

void __fastcall TForm1::SumNumbersProc(TatVirtualMachine *AMachine)
{
  switch(AMachine->InputArgCount())
  {
    case 2:
      AMachine->ReturnOutputArg(SumNumbers(AMachine->GetInputArgAsFloat(0),
      AMachine->GetInputArgAsFloat(1)));
      break;
    case 3:
      AMachine->ReturnOutputArg(SumNumbers(AMachine->GetInputArgAsFloat(0),
      AMachine->GetInputArgAsFloat(1), AMachine->GetInputArgAsFloat(2)));
      break;
    case 4:
      AMachine->ReturnOutputArg(SumNumbers(AMachine->GetInputArgAsFloat(0),
      AMachine->GetInputArgAsFloat(1), AMachine->GetInputArgAsFloat(2),
      AMachine->GetInputArgAsFloat(3)));
      break;
    case 5:
      AMachine->ReturnOutputArg(SumNumbers(AMachine->GetInputArgAsFloat(0),
      AMachine->GetInputArgAsFloat(1), AMachine->GetInputArgAsFloat(2),
      AMachine->GetInputArgAsFloat(3), AMachine->GetInputArgAsFloat(4)));
      break;
  }
}

```

[Original topic](#)

6.1.5 Accessing Delphi functions, variables and constants

6.1.5.1 Registering global constants

```
Scripter->AddConstant("MaxInt", MaxInt);
Scripter->AddConstant("Pi", M_PI);
Scripter->AddConstant("MyBirthday", EncodeDate(1992, 5, 30));
```

[Original topic](#)

6.1.5.2 Accessing global variables

```
Variant MyVar;
AnsiString ZipCode;

void __fastcall TForm1::GetZipCodeProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg(ZipCode);
}

void __fastcall TForm1::SetZipCodeProc(TatVirtualMachine *AMachine)
{
    ZipCode = AMachine->GetInputArgAsString(0);
}

void __fastcall TForm1::PrepareScript()
{
    Scripter->AddVariable("ShortDateFormat", ShortDateFormat);
    Scripter->AddVariable("MyVar", MyVar);
    Scripter->DefineProp("ZipCode", Atscript::tkString, GetZipCodeProc,
        SetZipCodeProc);
    Scripter->AddObject("Application", Application);
}

void __fastcall TForm1::Run1Click(TObject *Sender)
{
    PrepareScript();
    MyVar = "Old value";
    ZipCode = "987654321";
    Application->Tag = 10;
    Scripter->SourceCode = Mem1->Lines;
    Scripter->Execute();
    ShowMessage("Value of MyVar variable in C++ Builder is " + VarToStr(MyVar));
    ShowMessage("Value of ZipCode variable in C++ Builder is " +
        VarToStr(ZipCode));
}
```

[Original topic](#)

6.1.5.3 Calling regular functions and procedures

```
void __fastcall TSomeLibrary::Init()
{
    Scripter->DefineMethod("QuotedStr", 1, Atscript::tkString, NULL,
        QuotedStrProc);
    Scripter->DefineMethod("StringOfChar", 2, Atscript::tkString, NULL,
        StringOfCharProc);
}

void __fastcall TSomeLibrary::QuotedStrProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg(QuotedStr(AMachine->GetInputArgAsString(0)));
}

void __fastcall TSomeLibrary::StringOfCharProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg(StringOfChar(AMachine->GetInputArgAsString(0)[1],
        AMachine->GetInputArgAsInteger(1)));
}

void __fastcall TForm1::Run1Click(TObject *Sender)
{
}
```

```

    Scripter->AddLibrary(__classid(TSomeLibrary));
    Scripter->SourceCode = Memol->Lines;
    Scripter->Execute();
}

```

[Original topic](#)

6.1.6 Using libraries

6.1.6.1 Delphi-based libraries

CODE 1:

```

class TExampleLibrary: public TatScripterLibrary
{
protected:
    void __fastcall CurrToStrProc(TatVirtualMachine *AMachine);
    virtual void __fastcall Init();
    virtual AnsiString __fastcall LibraryName();
};

AnsiString __fastcall TExampleLibrary::LibraryName()
{
    return "Example";
}

void __fastcall TExampleLibrary::Init()
{
    Scripter->DefineMethod("CurrToStr", 1, Atscript::tkInteger, NULL,
        CurrToStrProc);
}

void __fastcall TExampleLibrary::CurrToStrProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg(CurrToStr(AMachine->GetInputArgAsFloat(0)));
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Scripter->AddLibrary(__classid(TExampleLibrary));
    Scripter->SourceCode = Memol->Lines;
    Scripter->Execute();
}

```

CODE 2:

```

void __fastcall TForm1::PrepareScript()
{
    Scripter->DefineMethod("CurrToStr", 1, Atscript::tkInteger, NULL,
        CurrToStrProc);
}

void __fastcall TForm1::CurrToStrProc(TatVirtualMachine *AMachine)
{
    AMachine->ReturnOutputArg(CurrToStr(AMachine->GetInputArgAsFloat(0)));
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    PrepareScript();
    Scripter->SourceCode = Memol->Lines;
    Scripter->Execute();
}

```

[Original topic](#)

6.1.6.2 Removing functions from the System library

```

delete MyScripter->SystemLibrary()->MethodByName("ShowMessage");

```

[Original topic](#)

