TMS Scripter Manual

August, 2015

Table of Contents

Chapter I

Introduction

1 Introduction

1.1 Overview

TMS Scripter is a set of Delphi/C++Builder components that add scripting capabilities to your applications. With TMS Scripter 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
- **TScrMemo**: Lightweight syntax highlight memo, that can be used to edit scripts at run-time.

TatScripter, TatPascalScripter, TatBasicScripter and TIDEScripter (in this document, all of these componentes 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

In addition to the scripting engine, a full Integrated Development Environment (IDE) is provided in Pro edition, to edit scripts, design forms, debugging, and many other operations with Delphi/Visual Studio look and feel for both creating and running script projects. The following features are present in the IDE:

- Full IDE environment dialog
- Visual form designer
- Componet palette and palette buttons with Delphi 2007 style
- Integrated syntax memo with built-in code completion and breakpoint features

1.2 Copyright Notice

TMS Scripter components trial version 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 TMS Scripter. 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 autorization of the author.

Online registration for TMS Scripter is available at http://www.tmssoftware.com/site/orders.asp. Source code & license is sent immediately upon receipt of check or registration by email.

TMS Scripter is Copyright © 2002-2015 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 6.5.4 (Sep-2015)
- New: RAD Studio 10 Seattle support

version 6.5.3 (Aug-2015)
- Fixed: TFormDesigner.OnCreateComponentClass event type signature changed to work around issue in C++

version 6.5.2 (Jul-2015)
- New: Support for set properties when registering classes using new RTTI (DefineClassByRTTI)
- New: LastExceptionLine and LastExceptionCol functions available from script
- New: TAdvFormDesigner.OnCreateComponentClass event provides an opportunity to manually create an instance of a component needed by the designer
- New: TatCustomScripter.RemoveProperty method
- Improved: Import tool updated with XE8 support
- Fixed: published properties of type Int64 were not being automatically registered when registering a class
- Fixed: "True" and "False" constants had wrong types (integer instead of boolean) in Variant types (regression)

version 6.5.1 (Apr-2015)

- New: Delphi/C++Builder XE8 support
- New: LastExceptionLine and LastExceptionCol properties provide info about the source code position where last exception was raised.
- New: TCustomFormDesigner.GridStepPixel property sets the number of pixels to move/ resize controls when using Ctrl+Arrow keys
- Fixed: Application hanging when registering some specific classes/components using RTTI
- Fixed: Scripter IDE wrongly considering source was changed even though it was not modified
- Fixed: Sporadic "Index out of bounds" when executing compiled code previously saved

version 6.5 (Mar-2015)

- New: Packages structure changed. Now it allows using runtime packages with 64-bit applications. It's a breaking change.
- Improved: A breaking change was added for Delphi XE and lower, requiring you to add Vcl.ScriptInit unit to your project.
- Improved: TScrMemo replaces TAdvMemo as the syntax memo class. It's a breaking change.
- Improved: DefineClassByRTTI sets default indexed property automatically
- Improved: TIDEEngine.PrepareXXXXDialog methods (Save/Load/SaveProject/ Loadproject) made virtual protected
- Improved: TIDEPaletteToolbar.UpdatePalette now virtual
- Fixed: Format function giving wrong results when formatting multiple string values
- Fixed: Setting indexed elements of array parameters passed by reference were not changing the array in Basic language
- Fixed: Import tool generating importing code that might fail in 64-bit applications
- Fixed: Assigned function could fail in 64-bit applications
- Fixed: Pascal syntax now accepts spaces before and after dot for accessing object members (<Variable>.<Member>)
- Fixed: "Cannot Focus Disable Windows" error message when right-clicking some third-party components in form designer
- Fixed: Method calls in with clauses had lower precedence than global properties.
- Fixed: Margins were incorrect with anchored controls in forms using bsSingle border style
- Fixed: Memory leak when copying components in form designer
- Fixed: TatMethod and TatProperty assign now properly assign event properties

version 6.4 (Sep-2014)

- New: Rad Studio XE7 support
- New: TIDEEngine.ProjectExt property allows specifying a project extension name different than '.ssproj'
- Fixed: Syntax memo incorrectly displaying commented code
- Fixed: AV when accessing default indexed properties with more than one index
- Fixed: Setting default indexed properties for untyped variable objects
- Fixed: Ord function not working for non-string values.
- Fixed: import tool incorrectly creating event adapter for events that returned a value (functions)
- Fixed: OnSingleDebugHook event not being fired in TIDEScripter component

version 6.3.1 (May-2014)

- New: Rad Studio XE6 support
- New: LibInstance property in TatClass, TatMethod, TatProperty classes indicates the library which registered them

- Improved: deprecated methods: TatCustomScripter.AddClass, TatClasses.Add, TatClasses.AddDelphiClass. All must be replaced by DefineClass method.
- Fixed: Icon for non-visual components in form designer were not being affected by OnGetComponentImage event
- Fixed: Insufficient RTTI Information when trying to import indexed TBytes property. Now the property is ignored (Delphi limitation)
- Fixed: AV violation while destroying TIDEEngine in some rare situations.
- Fixed: some unicode characters being displayed incorrectly in object inspector
- Fixed: "Canvas does not allow drawing" error when using form designer in an styled VCL application
- Fixed: Rare Access Violation when closing all forms in scripter IDE
- Fixed: Controls not being displayed in some situations, after loading a project, when controls had their Visible property set to false.
- Fixed: setting object properties failing in some situations in 64-bit applications
- Fixed: error when forcing varLongWord variants to integer values
- Fixed: "SSImport_Icon not found" error when compiling ImportTool project in some Delphi versions

version 6.3 (Feb-2014)
- New: Automatic registration via RTTI now supports indexed properties (Delphi XE2 and up)
- Improved: Code completion for local variables
- Improved: New TPaletteButton.ToolClass property allows checking what is the component class associated with a palette button
- Fixed: Error when calling methods such as Outer.InnerClass.HelloWorld and both Outer and InnerClass return script-based classes
- Fixed: Issue with component palette icons when compiling to 64-bit
- Fixed: Access Violation when reading record properties declared using RTTI
- Fixed: Setting ScriptFormClass property had no effect when creating forms from script
- Fixed: Access Violation when using code completion in some situations
- Fixed: Losing event handlers when renaming a component which name is contained by another component's name
- Fixed: copy/paste operations were not copying event handlers properly

version 6.2 (Oct-2013)
- New: TIDEDialog.Show method now allows IDE to be displayed as non-modal
- New: Rad Studio XE5 support
- Improved: TatScript.FileName property made public
- Fixed: Pascal single-line comments (starting with "//") not working on Mac.
- Fixed: Error when trying to pass EmptyParam to OLE servers
- Fixed: AV when setting TIDEEngine.PaletteButtons at design-time
- Fixed: issue with DebugStepOverLine and DebugUntilReturn methods when scripter is running in threads
- Fixed: breakpoints were not being displayed after file was saved in IDE
- Fixed: small glitch when selecting controls in form designer with fast mouse movement
- Fixed: custom glyph grab handles not being displayed on graphical controls
- Fixed: MDI child script forms being displayed twice
- Fixed: Anchoring not properly working when creating forms with Position = poScreenCenter

version 6.1.1 (May-2013)
- New: Rad Studio XE4 support
- Fixed: AV when trying to insert a component from component palette using keyboard

and no form visible
- Fixed: Flickering when filtering components in component palette
- Fixed: IsDate function in VBLibrary now checks strings for valid dates
- Fixed: Memory leak when using Int64 values in Format function

version 6.1 (Mar-2013)
- New: Latest AdvMemo 3.1.1 improvements
- Improved: Components now available at design-time for 64-bit applications
- Fixed: loading forms now opens file in shareable mode to avoid problems with multiple projects accessing same form file.
- Fixed: Events in very specific components like TvrTimer were not being set when form loads
- Fixed: issues when compiling scripter with assertions off
- Fixed: wrong line/column debug information when running pre-compiled code
- Fixed: msg parameter not being passed as reference in DoCompileError method
- Fixed: Access violation when registering components in an specific order
- Fixed: Setting TatCustomScripter.ScriptFormClass property raised an incorrect exception
- Fixed: Wrong atScript.hpp header file in C++ Builder 2007
- Fixed: designer handles not being updated when selected control was resized automatically after a form resize

version 6.0 (Sep-2012)
- New: Support for executing scripts in Firemonkey applications
- New: Delphi XE3 and C++ Builder XE3 support
- Improved: DefineRecordByRTTI method now returns the generated class
- Improved: better performance in import tool by using .spu files if .pas file was not changed
- Fixed: Duplicated entries in code completion window
- Fixed: Import tool parameter hints with default string values were being exported with single quotes causing syntax error in imported files
- Fixed: Issue when clearing some scripts between first and second project execution (implicit class references)
- Fixed: Missing component names when loading forms at low level using TFDReader (without using designer component)
- Fixed: Access Violation when closing main form custom IDE's
- Fixed: Access Violation when placing a component over a grab handle in form designer
- Fixed: Support for int64 values in Format function
- New: Dropped support for Delphi 5, 6, 2005, 2006 and C++Builder 6, 2006

TMS Scripter 6 is a merge of former Scripter Studio and Scripter Studio Pro. You can check version history of such products here:

Scripter Studio version history
Scripter Studio Pro version history

1.4 Former Scripter Studio History

version 5.2 (Apr-2012)
- New: 64-bit support in Rad Studio XE2
- New: All new features of TAdvMemo 3.0 included
- New: Support to declare 64-bit integer (Int64) literal values in script. Better handling of Int64 arithmetic operations (Delphi 6 and up only)

- Improved: Import tool: Better handling of subtypes. It was ignoring properties/methods declared after subtype declaration
- Improved: All imported files for VCL updated for 64-bit support and some missing methods like TList.Count in Delphi XE and up.
- Fixed: Several Issue with default properties (using With clause, expression in indexes, global objects)
- Fixed: TatScriptDebugger issue when settings breakpoints in a second execution
- Fixed: Calling class functions using object references (eg. Button1.ClassName) failing in some situations
- Fixed: Issue with code completion in TatScriptDebugger and TatMemoInterface components
- Fixed: Import Tool issue with WideChar parameters
- Fixed: Issue with WideString parameters when defining classes using new RTTI
- Fixed: Import tool now splits string constants when they are longer than 255 chars

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 Scripter 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 scripter 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 scripter
- Fixed: Scripter meta info (ScriptInfo): TatVariableInfo.TypeDecl value now has the correct value (it was empty)
- Fixed: Some variable values were not being displayed when using TatWebScripter
- 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: Scripter 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 scripter to handle component events. It's possible to declare a script event handler from script code (e.g. MyObject.Event = 'MyScriptEventHandler'), even if the scripter 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 Scripter Studio manual with the new features and in a new format (chm)

version 3.0.1 (Jul-2006)
- New : TatCustomScripter.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, CDbl, 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 ilustrates 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 + Scripter 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 Scripter 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 webscripter
- Fixed : missing "begin..end" block iin webscripter 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 ElseIf constructor in Basic scripter
- New : Support to Uses and Imports declaration in Basic scripter (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 scripter 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 rearrengement
- 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, Txxxx.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

1.5 Former Scripter Studio Pro History

version 2.2 (Apr-2012)
- New: 64-bit support in Rad Studio XE2
- New: All new features of TAdvMemo 3.0 included
- New: Support to declare 64-bit integer (Int64) literal values in script. Better handling of Int64 arithmetic operations (Delphi 6 and up only)
- New: TIDEDialog.PaletteStyle property allows use old-style Delphi 7 palette in newer Delphis
- New: If palette glyph is not available for a registered component, uses glyph from ancestor instead of using TComponent glyph
- Improved: Import tool: Better handling of subtypes. It was ignoring properties/methods declared after subtype declaration
- Improved: All imported files for VCL updated for 64-bit support and some missing methods like TList.Count in Delphi XE and up.
- Fixed: Several Issue with default properties (using With clause, expression in indexes, global objects)
- Fixed: TatScriptDebugger issue when settings breakpoints in a second execution
- Fixed: Calling class functions using object references (eg. Button1.ClassName) failing in some situations
- Fixed: Issue with TatDebugWatch
- Fixed: Files in Scripter IDE were being marked as modified even when no modifications were being done to project
- Fixed: Issue with code completion in TatScriptDebugger and TatMemoInterface components
- Fixed: Import Tool issue with WideChar parameters
- Fixed: Multi selection in form designer was being lost when controls were moved/resized
- Fixed: Issue with WideString parameters when defining classes using new RTTI
- Fixed: Small issue with form header being renamed twice when form unit is project main unit
- Fixed: Import tool now splits string constants when they are longer than 255 chars

version 2.1 (Sep-2011)
- New: Delphi/C++Builder XE2 Support
- New: Undo/Redo operations in IDE form designer
- New: TIDEEngine.UndoLevel property to control the level form designer undo operations
- New: ButtonHints property in Palette Toolbar component allows custom hints

- 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 2.0 (Apr-2011)
- New: Full support for parameter hints in syntax memo editor
- New: Smart code completion automatically suggests last choices made by user
- New: Support for creating script-based classes
- 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: Overall improved code completion experience with several issues fixed and better keyboard support for completion
- 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)
- Fixed: Issues with on dataset fields editor

version 1.7.1 (Dec-2010)
- Fixed: Registered version installer not working properly with TMS VCL Subscription Manager.

version 1.7 (Dec-2010)
- New: Updated imported VCL units for all Delphi versions, now including indexed properties, default parameters and other minor tweaks
- New: Visual editor for TWideStringList properties in object inspector
- New: Event on TIDEEngine for component selection in form designer
- 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 Scripter and other TMS packages
- Fixed: Instructions to return values for "out" parameters not generated by ImportTool
- Fixed: Issue with enumerated types in ImportTool

version 1.6.0.1 (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 1.6 (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])
- New: Fields Editor for TDataset components in the IDE
- New: Combobox editor for FieldName and TableName properties in Object Inspector
- 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
- Improved: Added property Modified (read only) in TIDEProjectFile
- 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
- Fixed: cursor position was not restoring in source code when toggling form/unit
- Fixed: unit ap_Mask missing at DB palette registering

version 1.5 (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.
- New: TIDEEngine.PreventDefaultEventCreation property.
- Fixed: Access Violation on Items property of TMainMenu and TPopupMenu.
- Fixed: error compiling some imported units in Delphi 2010.
- Fixed: issue with SaveCodeToFile when using form components of a non-registered class.
- Fixed: paste to editor was not pasting in correct position.
- Fixed: issue with scrollbars in form editor.
- Fixed: issue with tab set in Themed IDE.
- Fixed: memory leak when using some rare syntax constructions in script.

version 1.4.6 (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: component icons in toolbar were missing when compiling application with packages.
- 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.
- Fixed: issue with F9 key not being trapped by script forms.
- Fixed: editor not becoming invisible when closing a file in the ide (with THEMED_IDE directive defined).

version 1.4.5 (Sep-2009)
- New: Delphi/C++ Builder 2010 support.
- New: Design-time image list editor
- 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
- Fixed: issue with component placing in form designer
- Fixed: incompatibility when Greatis components are installed together with scripter pro

version 1.4 (May-2009)

- New: Themed IDE. By defining directive THEMED_IDE in AScript.INC file you can compile scripter package with TMS Component Pack and have your IDE in Office style (Luna, Olive, etc.).
- 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: TIDEDialog.AppStyler property allows setting the theme style of the whole IDE (requires TMS Component Pack)
- New: TIDEEngine.UnregisterComponent method
- New: TIDEEngine.OnGetComponentImage event allows providing an image for component icon (in toolbar and form designer) without needing to include resources
- New: TIDEEngine.OnComponentPlaced event is fired whenever a new component in placed in form designer
- New: TIDEPaletteButtons.CategoryColor and CategoryColorTo properties allow settings a background color for all categories in the control
- New: TCustomScripter.ScriptFormClass allows providing a different class (derived from TScriptForm) for the IDE forms
- Improved: included packages for specific compilation in C++Builder-only environments
- Improved: when scripter don't find a library, a compile error is raised (instead of an exception)
- Improved: in IDE, current file name is displayed in save dialogs
- Improved: IDE now uses default component icon for new components registered in IDE that don't have specific icon
- Fixed: issue with menu option "Compile" in scripter IDE
- Fixed: issue when double clicking the form's caption in form designer
- Fixed: issue when using arrow keys to move between controls in form designer
- Fixed: in IDE form designer, form was disappearing in Windows Vista when BorderStyle was set to bsNone
- Fixed: undesired behaviour when using Close Project menu option
- Fixed: issue with clipboard error in scripter IDE
- Fixed: issue with popup menu in object inspector when mouse buttons are swapped
- Fixed: "Select Unit" and "Watch Properties" windows are now dialogs (not sizeable, not minimizable)
- Fixed: AV in form designer when cutting controls to clipboard

version 1.3 (Feb-2009)
- New: "new" clause in Basic script. e.g "MyVar = new TLabel(Self)"
- New: const declaration in Basic script
- New: redo menu option in IDE
- New: compile menu option in IDE
- New: VBScript functions Redim, RedimPreserve, Split, Join, StrReverse and Randomize
- New: public method/property TIDEEngine.VisibleFileCount and TIDEEngine.VisibleFiles
- New: property TIDEEngine.AutoStyler allows avoiding the engine to set an automatic syntax styler for the memo
- New: TatCustomScripter methods BeginRefactor and EndRefactor to allow changing in source code without notifying the IDE
- Improved: better load/save compiled code engine
- Improved: cursor position in memo is preserved when switching units and/or running the script
- Improved: clipboard operations now working in designer, memo and inspector

- 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
- Improved: active line indicator now is hidden after script finished execution
- Improved: better performance in designer when using big scripts
- Improved: position of non-visual componentes being saved now
- Improved: default popup menu (copy, paste, etc.) in object inspector
- Fixed: issue with KeyPreview property in inspector
- Fixed: issue with wide string properties in Delphi 2009
- Fixed: issue with PasswordChar and other properties of type Char
- Fixed: issue with inspector becoming blank when using scroll bars
- Fixed: watches not being updated properly in some situations
- Fixed: losing form events in some situations
- Fixed: designer handles not appearing after paste operation
- Fixed: engine not recognizing basic syntax file extension in some situations
- Fixed: visual issues in form designer in Windows Vista
- Fixed: non-visual components appearing behind visual components in designer
- Fixed: duplicated "save as" dialog when using menu option "Save Project As"
- 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 1.2 (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 1.1 (Jul-2008)
- New: non-modal menu editor in the IDE allows better integration with the IDE while editing a menu
- New: Undo menu option in IDE Dialog
- New: "Find" and "Find and Replace" menu options in IDE Dialog
- New: TIDEProjectFile.SaveFormToString method
- New: TIDEProjectFile.FormResource property
- 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
- New: AddNotifier and RemoveNotifier in TIDEEngine allows to receive notifications about changed in the IDE
- New: TIDEEngine.ActiveFileModified allows notify the IDE that the current file was updated
- New: read/write TIDEEngine.SelectedComponent property identifies which is the current component selected in the ide form designer
- Improved: Scroll bars now appear in the form designer when the form is bigger than client editor area
- Improved: Clicking on caption bar now selects the form being designed
- Improved: renaming internal classes for compatibility with other 3rd party tools (Greatis,

ReportBuilder)
- Improved: memo syntax highlighting with pascal syntax
- Improved: autocompletion list updating while typing
- Improved: local variables are now initialized to NULL
- Fixed: center in window option in alignment tool was not working properly
- Fixed: issue with editing TForm.WindowMenu property
- Fixed: issue with editing TForm.ActiveControl property
- Fixed: menu items now can be selected in the object inspector and component combobox.
- Fixed: runtime error message was not displaying correct line and number of error
- Fixed: issue with watches not being updated or disappearing while debugging
- 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 1.0 (Apr-2008)
- First release, based on Scripter Studio 4.0.

1.6 Rebuilding Packages

If for any reason you want to rebuild source code, you should do it using the "Packages Rebuild Tool" utility that is installed. There is an icon for it in the Start Menu.
Just run the utility, select the Delphi versions you want the packages to be rebuilt for, and click "Install".
If you are using Delphi XE and up, you can also rebuild the packages manually by opening the dpk/dproj file in Delphi/Rad Studio IDE.
Do NOT manually recompile packages if you use Delphi 2010 or lower. In this case always use the rebuild tool.

1.7 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 attachements
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

1.8 Breaking Changes

List of changes in each version that breaks backward compatibility.

Version 6.5

- There was a big package restructuration in version 6.0. More info in the dedicated topic.
- Now you are required to add Vcl.ScripterInit unit to your project if you are using Delphi XE or lower, otherwise an error message will appear when you try to use the scripter components:

```
uses
  Vcl.ScripterInit;
```

Version 6.0

- Changes in package structure to support Firemonkey. More info here.

1.8.1 Version 6.5 - Package Restructuration

TMS Scripter packages have been restructured. The packages are now separated into runtime and design-time packages, and into several smaller ones allowing a better usage of them in an application using runtime packages (allows it to work with 64-bit applications using runtime packages, for example). Also, Libsuffix option is now being used so the dcp files are generated with the same name for all Delphi versions. Here is an overview of what's changed:

Before version 6.5, packages were last restructured in version 6.0. You can check the topic about Version 6.0 Breaking Changes to see how it was.

From version 6.5 and on, there are twelve packages:

TMSScripter.dpk (Core Package)

TMSScripter_Memo.dpk (Syntax Highlight Memo)
TMSScripter_Imports_RTL.dpk (Imports for RTL Units)
TMSScripter_Imports_VCL.dpk (Imports for VCL Components)
TMSScripter_Imports_DB.dpk (Imports for DB Components)
TMSScripter_Imports_ADODB.dpk (Imports for ADODB Components)
TMSScripter_FMX.dpk (Units to Support Scripter in Firemonkey Applications)
TMSScripter_VCL.dpk (Units to Support Scripter in VCL Applications)
TMSScripter_IDE.dpk (TMS Scripter IDE Components)
TMSScripter_Legacy.dpk (Legacy TMS Scripter Components)
dclTMSScripter.dpk (Design-Time Core Package)
dclTMSScripter_Memo.dpk (Design-Time Memo Package)

Dcp files are generated with same name, and only bpl files are generated with the suffix indicating the Delphi version. The suffix, however, is the same used by the IDE packages (numeric one indicating IDE version: 160, 170, etc.). The new package structure is as following (note that when 6.5 was released, latest Delphi version was XE7. Packages for newer versions will follow the same structure):

Version	Package File Name	Bpl File Name	Dcp File Name
Delphi 7	TMSScripter.dpk	TMSScripter70.bpl	TMSScripter.dcp
	TMSScripter_Memo.dpk	TMSScripter_Memo70.bpl	TMSScripter_Memo.dcp
	TMSScripter_Imports_RTL.dpk	TMSScripter_Imports_RTL70.bpl	TMSScripter_Imports_RTL.dcp
	TMSScripter_Imports_VCL.dpk	TMSScripter_Imports_VCL70.bpl	TMSScripter_Imports_VCL.dcp
	TMSScripter_Imports_DB.dpk	TMSScripter_Imports_DB70.bpl	TMSScripter_Imports_DB.dcp
	TMSScripter_Imports_ADODB.dpk	TMSScripter_Imports_ADODB70.bpl	TMSScripter_Imports_ADODB.dcp
	TMSScripter_FMX.dpk	TMSScripter_FMX70.bpl	TMSScripter_FMX.dcp
	TMSScripter_VCL.dpk	TMSScripter_VCL70.bpl	TMSScripter_VCL.dcp
	TMSScripter_IDE.dpk	TMSScripter_IDE70.bpl	TMSScripter_IDE.dcp
	TMSScripter_Legacy.dpk	TMSScripter_Legacy70.bpl	TMSScripter_Legacy.dcp
	dclTMSScripter.dpk	dclTMSScripter70.bpl	dclTMSScripter.dcp
	dclTMSScripter_Memo.dpk	dclTMSScripter_Memo70.bpl	dclTMSScripter_Memo.dcp
Delphi 2007	TMSScripter.dpk	TMSScripter100.bpl	TMSScripter.dcp
	TMSScripter_Memo.dpk	TMSScripter_Memo100.bpl	TMSScripter_Memo.dcp
	TMSScripter_Imports_RTL.dpk	TMSScripter_Imports_RTL100.bpl	TMSScripter_Imports_RTL.dcp
	TMSScripter_Imports_VCL.dpk	TMSScripter_Imports_VCL100.bpl	TMSScripter_Imports_VCL.dcp
	TMSScripter_Imports_DB.dpk	TMSScripter_Imports_DB100.bpl	TMSScripter_Imports_DB.dcp
	TMSScripter_Imports_ADODB.dpk	TMSScripter_Imports_ADODB100.bpl	TMSScripter_Imports_ADODB.dcp
	TMSScripter_FMX.dpk	TMSScripter_FMX100.bpl	TMSScripter_FMX.dcp
	TMSScripter_VCL.dpk	TMSScripter_VCL100.bpl	TMSScripter_VCL.dcp
	TMSScripter_IDE.dpk	TMSScripter_IDE100.bpl	TMSScripter_IDE.dcp
	TMSScripter_Legacy.dpk	TMSScripter_Legacy100.	TMSScripter_Legacy.dcp
			dclTMSScripter.dcp
			dclTMSScripter_Memo.dcp

	dclTMSScripter.dpk dclTMSScripter_Memo.dpk	bpl dclTMSScripter100.bpl dclTMSScripter_Memo100.bpl	
Delphi 2009	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter120.bpl TMSScripter_Memo120.bpl TMSScripter_Imports_RTL120.bpl TMSScripter_Imports_VCL120.bpl TMSScripter_Imports_DB120.bpl TMSScripter_Imports_ADODB120.bpl TMSScripter_FMX120.bpl TMSScripter_VCL120.bpl TMSScripter_IDE120.bpl TMSScripter_Legacy120.bpl dclTMSScripter120.bpl dclTMSScripter_Memo120.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi 2010	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter140.bpl TMSScripter_Memo140.bpl TMSScripter_Imports_RTL140.bpl TMSScripter_Imports_VCL140.bpl TMSScripter_Imports_DB140.bpl TMSScripter_Imports_ADODB140.bpl TMSScripter_FMX140.bpl TMSScripter_VCL140.bpl TMSScripter_IDE140.bpl TMSScripter_Legacy140.bpl dclTMSScripter140.bpl dclTMSScripter_Memo140.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi XE	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk	TMSScripter150.bpl TMSScripter_Memo150.bpl TMSScripter_Imports_RTL150.bpl TMSScripter_Imports_VCL150.bpl TMSScripter_Imports_DB1	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.

	TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	50.bpl TMSScripter_Imports_ADODB150.bpl TMSScripter_FMX150.bpl TMSScripter_VCL150.bpl TMSScripter_IDE150.bpl TMSScripter_Legacy150.bpl dclTMSScripter150.bpl dclTMSScripter_Memo150.bpl	dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi XE2	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter160.bpl TMSScripter_Memo160.bpl TMSScripter_Imports_RTL160.bpl TMSScripter_Imports_VCL160.bpl TMSScripter_Imports_DB160.bpl TMSScripter_Imports_ADODB160.bpl TMSScripter_FMX160.bpl TMSScripter_VCL160.bpl TMSScripter_IDE160.bpl TMSScripter_Legacy160.bpl dclTMSScripter160.bpl dclTMSScripter_Memo160.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi XE3	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter170.bpl TMSScripter_Memo170.bpl TMSScripter_Imports_RTL170.bpl TMSScripter_Imports_VCL170.bpl TMSScripter_Imports_DB170.bpl TMSScripter_Imports_ADODB170.bpl TMSScripter_FMX170.bpl TMSScripter_VCL170.bpl TMSScripter_IDE170.bpl TMSScripter_Legacy170.bpl dclTMSScripter170.bpl dclTMSScripter_Memo170.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi	TMSScripter.dpk	TMSScripter180.bpl	TMSScripter.dcp

XE4	TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter_Memo180.bpl TMSScripter_Imports_RTL180.bpl TMSScripter_Imports_VCL180.bpl TMSScripter_Imports_DB180.bpl TMSScripter_Imports_ADODB180.bpl TMSScripter_FMX180.bpl TMSScripter_VCL180.bpl TMSScripter_IDE180.bpl TMSScripter_Legacy180.bpl dclTMSScripter180.bpl dclTMSScripter_Memo180.bpl	TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi XE5	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter190.bpl TMSScripter_Memo190.bpl TMSScripter_Imports_RTL190.bpl TMSScripter_Imports_VCL190.bpl TMSScripter_Imports_DB190.bpl TMSScripter_Imports_ADODB190.bpl TMSScripter_FMX190.bpl TMSScripter_VCL190.bpl TMSScripter_IDE190.bpl TMSScripter_Legacy190.bpl dclTMSScripter190.bpl dclTMSScripter_Memo190.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp
Delphi XE6	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dp	TMSScripter200.bpl TMSScripter_Memo200.bpl TMSScripter_Imports_RTL200.bpl TMSScripter_Imports_VCL200.bpl TMSScripter_Imports_DB200.bpl TMSScripter_Imports_ADODB200.bpl TMSScripter_FMX200.bpl TMSScripter_VCL200.bpl TMSScripter_IDE200.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp

	k dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter_Legacy200.bpl dclTMSScripter200.bpl dclTMSScripter_Memo200.bpl	dclTMSScripter_Memo.dcp
Delphi XE7	TMSScripter.dpk TMSScripter_Memo.dpk TMSScripter_Imports_RTL.dpk TMSScripter_Imports_VCL.dpk TMSScripter_Imports_DB.dpk TMSScripter_Imports_ADODB.dpk TMSScripter_FMX.dpk TMSScripter_VCL.dpk TMSScripter_IDE.dpk TMSScripter_Legacy.dpk dclTMSScripter.dpk dclTMSScripter_Memo.dpk	TMSScripter210.bpl TMSScripter_Memo210.bpl TMSScripter_Imports_RTL210.bpl TMSScripter_Imports_VCL210.bpl TMSScripter_Imports_DB210.bpl TMSScripter_Imports_ADODB210.bpl TMSScripter_FMX210.bpl TMSScripter_VCL210.bpl TMSScripter_IDE210.bpl TMSScripter_Legacy210.bpl dclTMSScripter210.bpl dclTMSScripter_Memo210.bpl	TMSScripter.dcp TMSScripter_Memo.dcp TMSScripter_Imports_RTL.dcp TMSScripter_Imports_VCL.dcp TMSScripter_Imports_DB.dcp TMSScripter_Imports_ADODB.dcp TMSScripter_FMX.dcp TMSScripter_VCL.dcp TMSScripter_IDE.dcp TMSScripter_Legacy.dcp dclTMSScripter.dcp dclTMSScripter_Memo.dcp

1.8.2 Version 6.5 - TScrMemo replaces TAdvMemo

As of TMS Scripter 6.5, TAdvMemo component is no longer available. This doesn't mean there is no syntax highlight memo component anymore - it was just renamed to TScrMemo.
This was done to get rid of dependency and conflict with TMS Component Pack. Both products have a TAdvMemo component and although registered versions of both products could be installed together, it was not an ideal setup.

For most TMS Scripter users, this will be a transparent change. If you use TIDEMemo, it's still there. If you use TIDEDialog component to show the TMS Scripter IDE, it will still work. If you have TMS Component Pack installed, you will also have no problems.

The only issue that might appear is if you have TAdvMemo components in your application forms and you don't have TMS Component Pack installed. In this case, Delphi IDE will complain that a TAdvMemo component does not exist, and you will have to manually edit your dfm/pas file and replace any TAdvMemo reference by a TScrMemo reference.

1.8.3 Version 6.0 - Breaking Changes

1. Packages changed

For Delphi XE2 and up, packages were restructured. Package ascriptproxe2.dpk doesn't exist anymore, and was split into the following packages:

tmsscripter_xe2: Runtime package with core/non-visual classes and scripting engine
tmsscriptervcl_xe2: Runtime package with VCL components and imported VCL
tmsscripterreg_xe2: Design-time package
tmsscripterfmx_xe2: Runtime package with Firemonkey components

2. Existing applications need a small change

For Delphi XE2 and up, since the scripter engine can work with either VCL or Firemonkey, you must specify which framework you are using in your application, by adding a proper unit to the uses clause of any unit in your project.

To use scripter with VCL, add unit Vcl.ScripterInit:

```
uses
  Vcl.ScripterInit;
```

To use scripter with Firemonkey add unit FMX.ScripterInit:

```
uses
  FMX.ScripterInit;
```

3. Firemonkey compatibility

TMS Scripter engine is now is compatible with Firemonkey. It means you can execute scripts in Firemonkey applications, even with forms. But note that several VCL components don't have Firemonkey equivalents yet, especially the visual ones, so the scripter IDE (form designer, syntax memo, object inspector, etc.) are not available for Firemonkey applications.

1.9 Use in Firemonkey applications

TMS Scripter engine can now be used in Firemonkey applications. You can execute scripts in FM applications even with forms.
But note that several VCL components don't have Firemonkey equivalents yet, especially the visual ones, so the scripter IDE (form designer, syntax memo, object inspector, etc.) are not available for Firemonkey applications.
All you need to do in your Firemonkey application is add unit FMX.ScripterInit to your project or the uses clause of any unit. Then you can use the scripter component normally just as you would do with in VCL (see chapter Working with Scripter).

There are several demos in TMS Scripter distributing showing how to use it with Firemonkey application, including manual debugging.

Chapter II

Integrated Development Environment

2 Integrated Development Environment

TMS Scripter includes a ready-to-use IDE for writing scripts and designing forms. This chapter covers how to use that IDE and how to use additional components to build your own IDE. The IDE is only available for VCL applications.

2.1 Specific IDE components

TMS Scripter is a full scripting package for editing, debugging and running scripts and forms in Delphi and C++ Builder environment.

2.1.1 Basic concepts

TMS Scripter provides a set of components for scripting and designing. In summary, we can separate the usage in **runtime** and **design time**.

Runtime
For runtime execution, the main component to use is TIDEScripter. This component descends from TatScripter which descends from TatCustomScripter, so it has all functionalities of other scripter components present in previous versions and editions of TMS Scripter.

TIDEScripter is the scripter engine which runs scripts. So, basically, it holds a collection of one or more scripts that can be executed. To see all tasks that can be done with TIDEScripter component, please refer to Working with scripter topic. To see a reference about the languages supported in script, and the syntax of each language, please refer to Language Features.

Design time
TMS Scripter provides several components that will allow your end-user to write and design scripts and script projects. Basically you can provide an Integrated Development Environment for your end-user to build script projects, create forms, and write scripts. Please refer to the Integrated Development Environment chapter.

2.1.2 Component overview

TIDEScripter
This component is the non-visual component for running/debugging scripts. Check the topic "The TIDEScripter component" for more information.

TIDEEngine
This is the core engine component for the IDE. Check the topic "The TIDEEngine component" for more information.

TIDEDialog
This is the wrapper for the IDE window. Use this component to show the IDE. Check the topic "Running the IDE: TIDEDialog component" for more information.

TIDEPaletteToolbar

TIDEInspector

TIDEMemo

TIDEFormDesignControl

TIDEComponentComboBox

TIDEPaletteButtons

TIDEWatchListView

The components above are used to build your own custom IDE. Check the section "Building your own IDE" for more information

2.1.3 The TIDEScripter component

The TIDEScripter component is a non-visual component used to execute scripts. It descends from TatCustomScripter and is fully compatible with other scripter components like TatPascalScripter and TatBasicScripter.

The chapters "Language Features" and "Working with scripter" describes how to use the scripter component to execute scripts, access Delphi objects, integrate the scripter with your application, and also know the valid syntax and languages available.

2.2 Running the IDE: TIDEDialog component

The TIDEDialog component provides quick access to the ready-to-use IDE. It is a wrapper for a IDE form which already contains the memo, object inspector, among others. To invoke the IDE:

1. Drop a **TIDEScripter** component in the form

2. Drop a **TIDEEngine** component in the form

3. Drop a **TIDEDialog** component in the form

4. Link the TIDEScripter component to the TIDEEngine component through the **TIDEEngine.Scripter** property

5. Link the TIDEEngine component to the TIDEDialog component through the **TIDEDialog.Engine** property

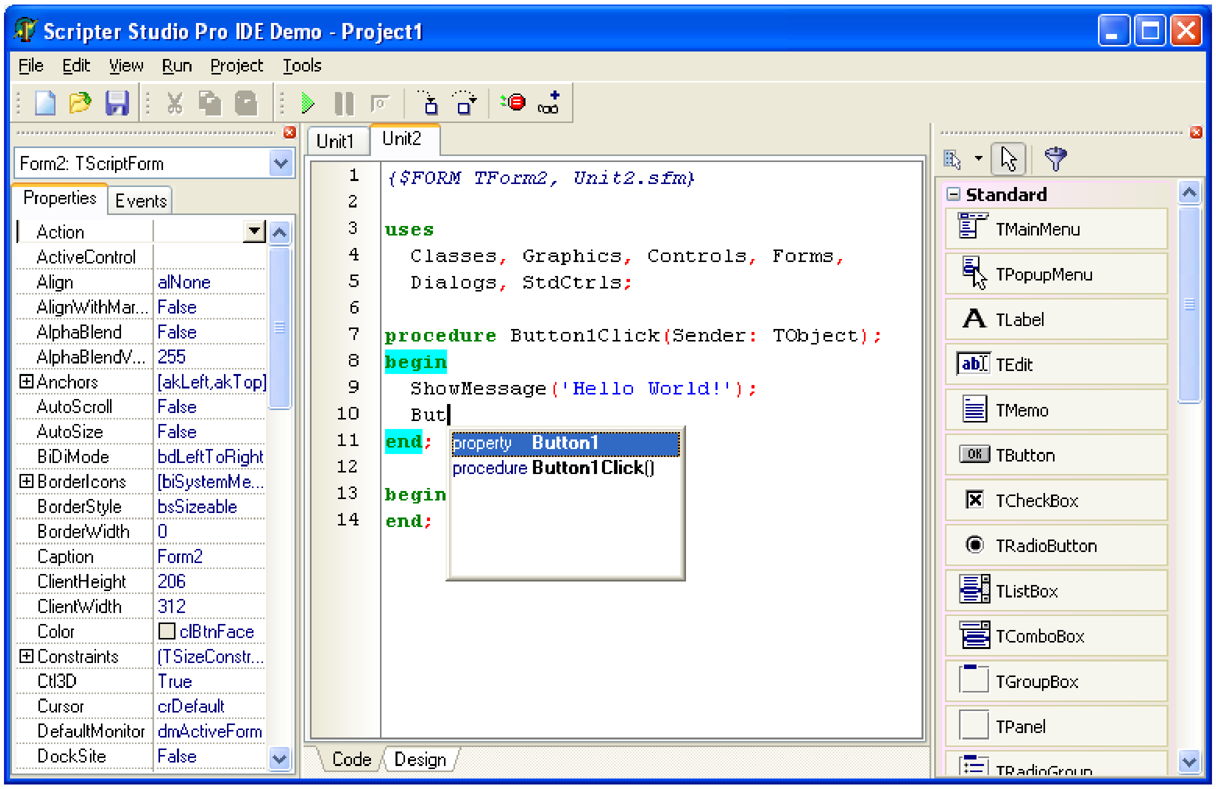
4. Call **TIDEDialog.Execute** method:

```
IDEDialog1.Execute;
```

This will open the IDE window.

2.3 Overview of the IDE

This is a screenshot of the TMS Scripter IDE:



Action	
ActiveControl	
Align	alNone
AlignWithMar...	False
AlphaBlend	False
AlphaBlendV...	255
⊞Anchors	[akLeft,akTop]
AutoScroll	False
AutoSize	False
BiDiMode	bdLeftToRight
⊞BorderIcons	[biSystemMe...
BorderStyle	bsSizeable
BorderWidth	0
Caption	Form2
ClientHeight	206
ClientWidth	312
Color	☐ clBtnFace
⊞Constraints	(TSizeConstr...
Ctl3D	True
Cursor	crDefault
DefaultMonitor	dmActiveForm
DockSite	False

```
{$FORM TForm2, Unit2.sfm}

uses
  Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

procedure Button1Click(Sender: TObject);
begin
  ShowMessage('Hello World!');
  But
end;

begin
end;
```

```
property  Button1
procedure Button1Click()
```

It's very similar to a Delphi or Visual Studio IDE. The object inspector is at the left, the syntax code editor memo is in center, menus and toolbars at the top, and the tool palette is at right. Please not that the tool palette is only available from Delphi 2005 and up. For previous versions of Delphi, a toolbar is available with Delphi 7 style (at the top of the IDE).

Shortcuts are available for most used actions, you can see the shortcuts available in the main menu of the IDE.

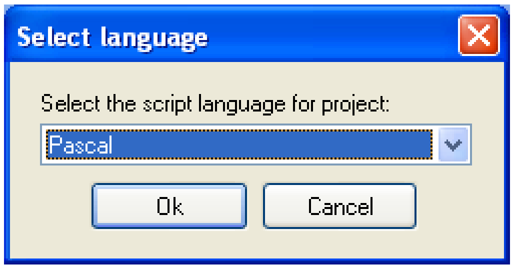
2.3.1 Managing projects and files

Project concept and structure
A project in TMS Scripter is a collection of scripts (files), and each file can be a unit (a single script file) or a form (a script file and a form file). A project file is just a list of the script files belonging to that project and the information of which script is the main script.

Mixing languages
You can mix scripts with different languages, i.e., in a project you can have a Basic script which creates and executes a Pascal form.

Main script
Each project has a "main" script. The main script is the script which will be executed when you press F9 or click the "Run" button (or menu option).

Creating a new project
To create a new project, choose "File | New Project" menu option. This dialog will be displayed:



Keep in mind that here you are choosing the language for the units that will be created automatically by the IDE. It's not the language of the "project" itself, since such concept doesn't exist. It's the language of the main units.
After you choose the language of the main units, the IDE will create a main unit and a form unit. This is the basic project and if you execute it right away you will have a running blank form in your screen.

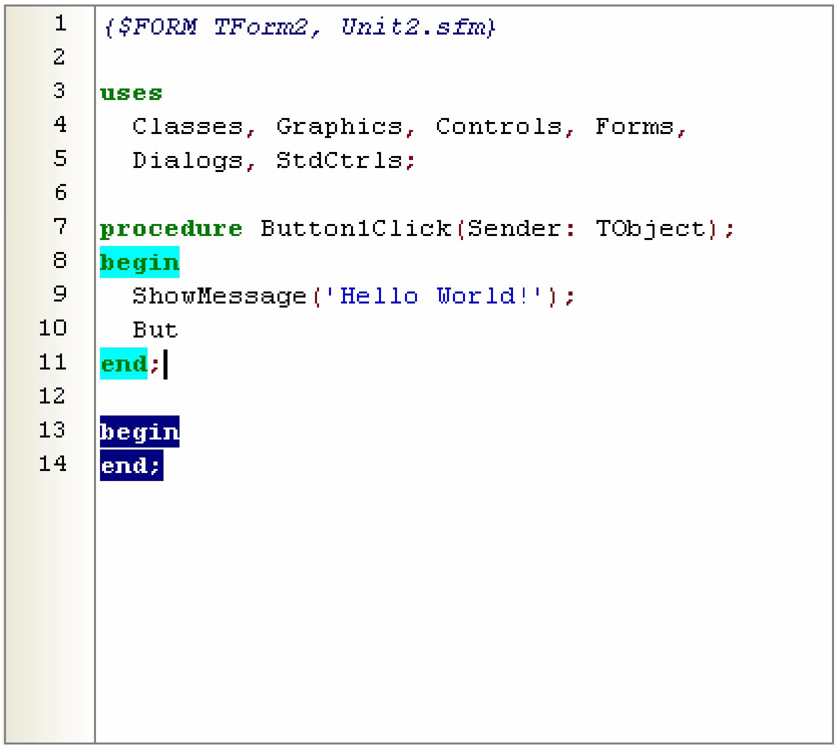
Note: before running this simple example, you must add the following units to your Delphi/C++Builder uses/include clause: ap_Classes, ap_Controls, ap_Forms, ap_Dialogs, ap_Graphics and ap_StdCtrls.

Creating/adding units/forms to the project
You can create or add existing units/forms to the project by choosing the "File | New unit", "File | New Form" and "File | Open (add to project)" menu options. If you are creating a new one, you will be prompted with the same dialog as above, to choose the language of the new unit. If you're adding an existing unit, then the IDE will detect the script language based on the file extension.

2.3.2 Editing the script in code editor

The IDE provides you with a code editor with full syntax highlight for the script language. The main features of code editor are:

- code completion (pressing Ctrl+Space)
- syntax highlight
- line numbering
- clipboard operations
- automatic identation
- among other features
-

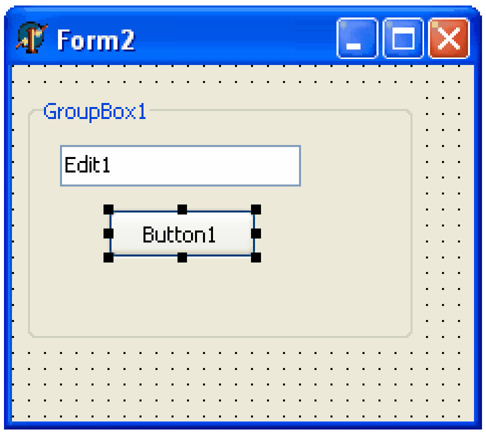


```
{$FORM TForm2, Unit2.sfm}

uses
  Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

procedure Button1Click(Sender: TObject);
begin
  ShowMessage('Hello World!');
  But
end;

begin
end;
```

2.3.3 Designing forms

When you're dealing with units that are forms, then you have two parts: the script and the form. You can switch between script and form using F12 key or by pressing the "Code" and "Design" tabs at the bottom of the screen.

The form editor looks like the picture below:

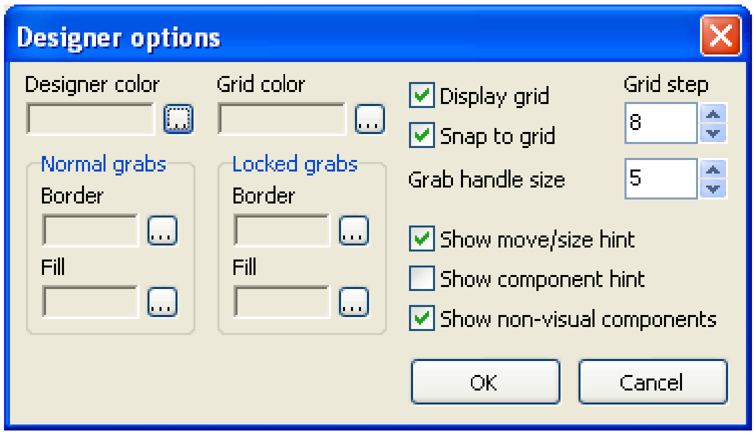


Designing forms is a similar task as designing forms in Delphi or Visual Studio. You can use the tool palette the choose a component to drop on the form, position the component using the mouse or keyboard (resize, move, etc.) and change the properties using the object inspector.

The main features of the form designer are:

- Multi-selection
- Clipboard operations
- Alignment palette (menu "Edit | Align")
- Bring to front / Send to back
- Tab order dialog
- Size dialog
- Locking/unlocking controls
- Grid and Snap to Grid
- among other features

You can change some properties of the form designer by opening the Designer Options dialog. This is available under the menu "Tools | Designer options":


Designer options

Designer color

Grid color

☑ Display grid

☑ Snap to grid

Grid step: 8

Grab handle size: 5

Normal grabs: Border, Fill

Locked grabs: Border, Fill

☑ Show move/size hint

☐ Show component hint

☑ Show non-visual components

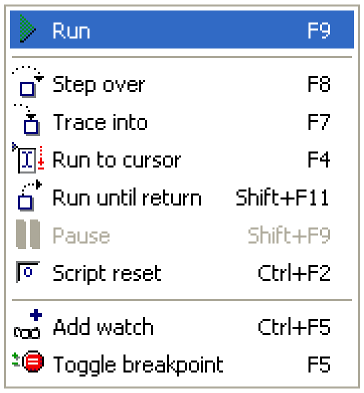
You can customize the look and feel of the designer choosing colors, hints and grid options.

2.3.4 Running and debugging scripts

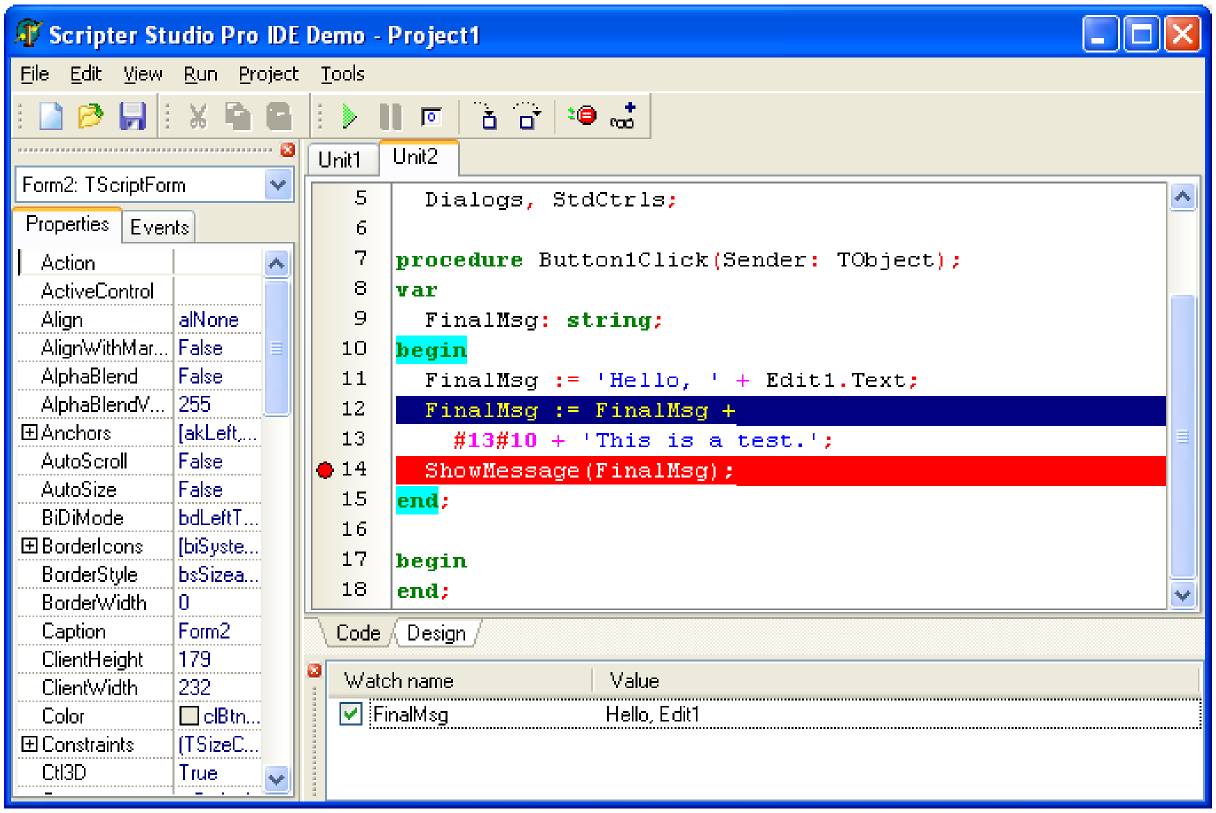
You can run and debug scripts from the TMS Scripter IDE. The main features of the debugger are:

- Breakpoints
- Watches
- Step over/Trace into
- Run to cursor/Run until return
- Pause/Reset
- and more...

The image below shows the options under the menu item "Run",



You can use the shortcuts above or use the menu/toolbar buttons to perform running/ debugging actions, like run, pause, step over, trace into, etc..
You can also toggle a breakpoint on/off by clicking on the left gutter in the code editor.
The image below shows a script being debugged in the IDE. A watch has been added in this example to inspect the value of variable "FinalMsg"


Scripter Studio Pro IDE Demo - Project1
File Edit View Run Project Tools
Form2: TScriptForm
Properties Events
Unit1 Unit2
Code Design

Action	
ActiveControl	
Align	alNone
AlignWithMar...	False
AlphaBlend	False
AlphaBlendV...	255
⊞Anchors	[akLeft,...
AutoScroll	False
AutoSize	False
BiDiMode	bdLeftT...
⊞BorderIcons	[biSyste...
BorderStyle	bsSizea...
BorderWidth	0
Caption	Form2
ClientHeight	179
ClientWidth	232
Color	☐ clBtn...
⊞Constraints	(TSizeC...
Ctl3D	True

```
  Dialogs, StdCtrls;

procedure Button1Click(Sender: TObject);
var
  FinalMsg: string;
begin
  FinalMsg := 'Hello, ' + Edit1.Text;
  FinalMsg := FinalMsg +
    #13#10 + 'This is a test.';
● ShowMessage(FinalMsg);
end;

begin
end;
```

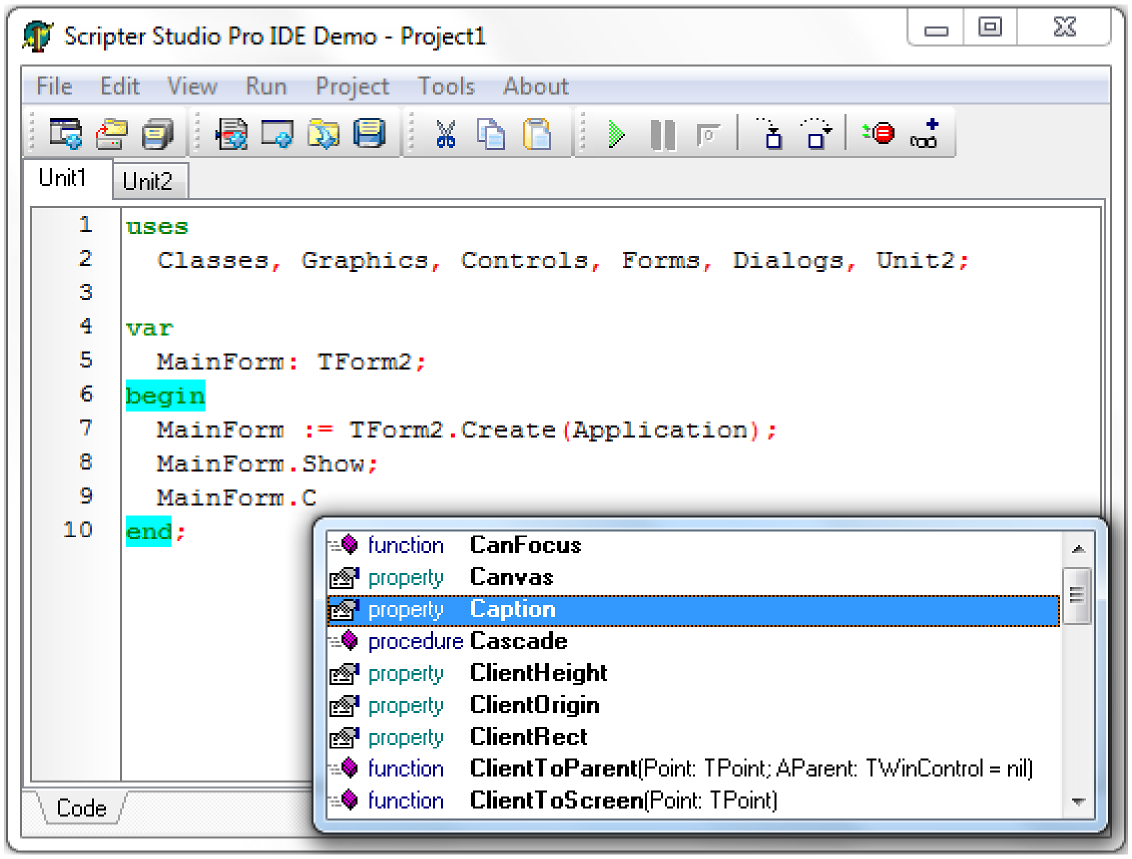
Watch name	Value
✓ FinalMsg	Hello, Edit1

2.4 Code Insight features

TMS Scripter comes with code insight features, meaning that in the IDE editor you can have fully automatic code completion and parameter hints

2.4.1 Code Completion

Code completion is a feature activated by Ctrl+<Space> or when you type an identifier name followed by a "." (dot).
A list appears at the cursor position, displaying all the available options (methods, properties, functions, variables) for the given context.


Scripter Studio Pro IDE Demo - Project1
File Edit View Run Project Tools About
Unit1 Unit2
Code

```
uses
  Classes, Graphics, Controls, Forms, Dialogs, Unit2;

var
  MainForm: TForm2;
begin
  MainForm := TForm2.Create(Application);
  MainForm.Show;
  MainForm.C
end;
```

```
function   CanFocus
property   Canvas
property   Caption
procedure  Cascade
property   ClientHeight
property   ClientOrigin
property   ClientRect
function   ClientToParent(Point: TPoint; AParent: TWinControl = nil)
function   ClientToScreen(Point: TPoint)
```

Smart code completion
When a code completion list appears, it will automatically preselect the item which was previously chosen by you. The item selected is specific to the context.
For example, you might be dealing with a TDataset and retrieving several field values from it, using FieldByName method. You follow this steps to use code completion for the first line:

1. Invoke code completion by typing Ctrl+<Space>
2. Start type the naming of your TDataset object, for example, type "Dat" and then you get the "Dataset1" item selected in the completion list.
3. Press "." to insert "Dataset1." text in the editor.
4. A new code completion list will appear listing the methods and properties of the dataset.
5. You start typing "FieldB" to select the item "FieldByName" from the completion list
6. Press "(" to insert "FieldByName(", type the name of field, type ")." to close the parameters and invoke the list again.
7. Type "AsStr" to find AsString property and then press ";" to finally complete the line.

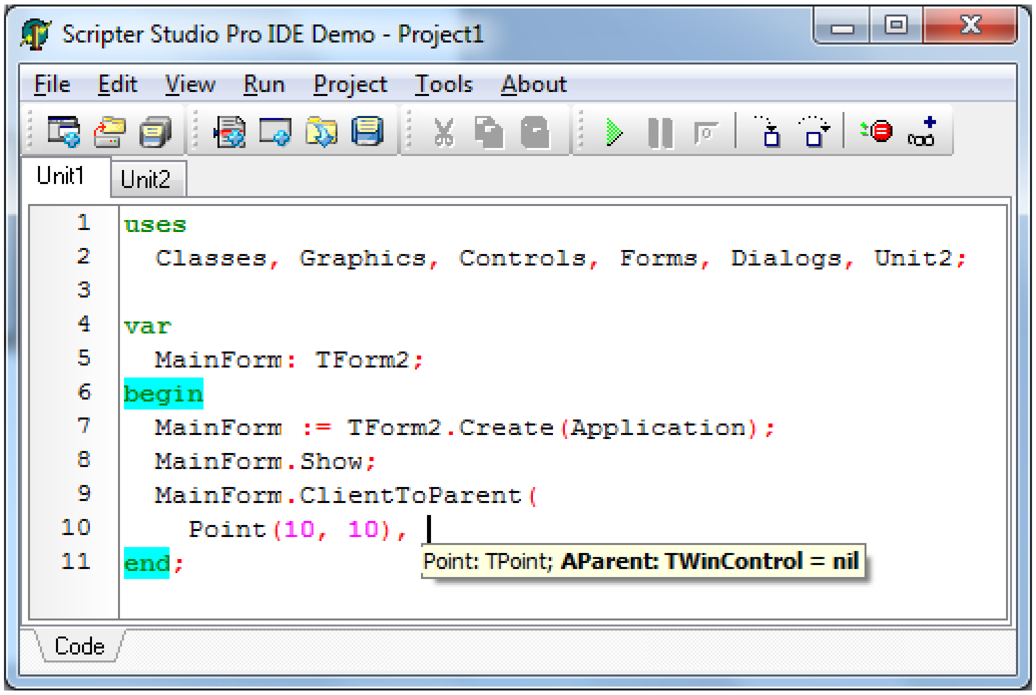
Now, you want to start a second line with the same code for another field. Smart code completion will remember your last options, and this is what you would need to type:

1. Invoke code completion by typing Ctrl+<Space>
2. "Dataset1" will come preselected in the list. Just press "." to insert text and invoke a new list for Dataset1 members.
3. "FieldByName" will come preselected in the list. Just press "(" to insert text and type the field name
4. When close FieldByName parameters and press "." again, "AsString" will also come preselected, and you can just type ";" to finish typing.

Easy navigation
When you have the desired item selected in code completion list, you can click Enter to make the selection be typed in the text editor, so you don't have to type it.
You can type other keyboard keys in order to complete the text and also insert the character. For example, if you press "." (dot), the selected item will be inserted in the text, followed by a dot, and a new completion list will be displayed for the selected context.

2.4.2 Parameter Hints

Parameter hints is a feature activated by Ctrl+Shift+<Space>, or when you type a method/function name followed by "(".
This will display the list of parameters for the specified method, so you can properly type the parameters.
The current parameter being typed is highlighted in the hint



```
uses
  Classes, Graphics, Controls, Forms, Dialogs, Unit2;

var
  MainForm: TForm2;
begin
  MainForm := TForm2.Create(Application);
  MainForm.Show;
  MainForm.ClientToParent(
    Point(10, 10), 
end;
```

Point: TPoint; **AParent: TWinControl = nil**

Enabling parameter hints
Parameter hints features is enabled by default, but you have to provide info to it. For each method, you must provide the names and types of parameters so scripter can show them. This is done with UpdateParameterHints method of TatMethod object. You would

usually do this when you register a new method in scripter.
You can use DefineMethod method and pass it as the last parameter:

```
DefineClass(TSomeClass).DefineMethod('MyMethod', 2, TkInteger, nil,
MyMethodProc, true, 0,
  'Name:string;Value:integer');
```

or you can just call UpdateParameterHints:

```
with DefineMethod('MyMethod', 2, TkInteger, nil, MyMethodProc) do
  UpdateParameterHints('Name:string;Value:integer');
```

Parameter hint syntax
The parameter hint has a very simple and specific syntax, which is:

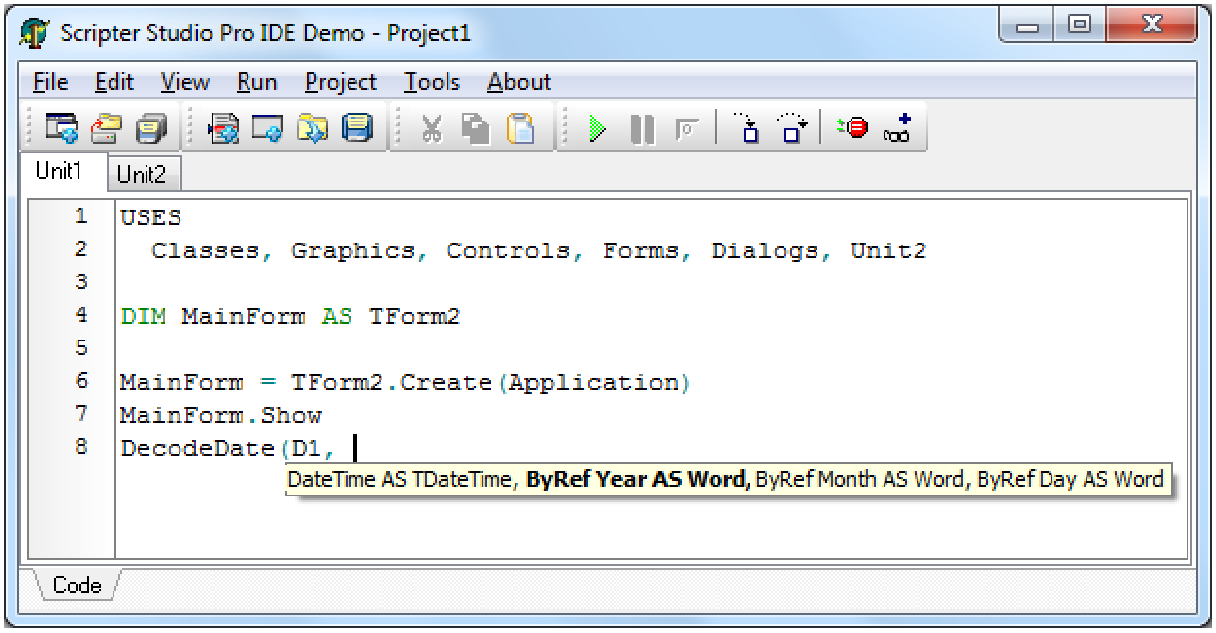
ParamName[:ParamType][=DefaultValue]

Parts between brackets are optional. If there are more than one parameter, you must separate then with semicommas (;). Some examples:

```
'Param1:String;Param2:Integer'
'Param1; Param2; Param3 = 0'
'Param1; Param2: TButton; Param3: boolean = false'
```

You can have spaces between the characters, and you must not include any parameter modifier (var, const, etc.), this will be used automatically by the scripter.
Also be aware that the parameter hints do NOT affect any information in the registered method itself. For example, if you build the hint with a different number of parameters than the specified for the method, the remaining parameters will be ignored. This is also valid for default values and param types, they are only used for hinting purposes.

Cross-language parameter hints
Scripter will automatically translate the parameter hints to the proper script language, so you don't need to register a parameter hint for each language syntax. The hint will be displayed according to the current script syntax. Even if you use script-based libraries, written in Pascal language, for example, when you call those methods from a Basic script, parameter hints will be displayed in Basic syntax.


Scripter Studio Pro IDE Demo - Project1
File Edit View Run Project Tools About
Unit1 Unit2
Code

```
USES
  Classes, Graphics, Controls, Forms, Dialogs, Unit2

DIM MainForm AS TForm2

MainForm = TForm2.Create(Application)
MainForm.Show
DecodeDate(D1, 
```

DateTime AS TDateTime, **ByRef Year AS Word,** ByRef Month AS Word, ByRef Day AS Word

Import tool
The scripter import tool properly generates the DefineMethod call including the correct parameter hint for the method being registered.

Enchanced RTTI
If you use Delphi 2010 and up, and register your classes using the new enchanced RTTI, parameter hint are retrieved automatically with the RTTI and are available in the editor with no need for extra code.

2.5 Building your own IDE

TMS Scripter provides you several components to make it easy to build your own IDE. All elements in the IDE like the code editor, object inspector, tool palette, etc., are available for stand-alone or integrated use.

And more, you don't need to use all components, you can use only three, two, or even one single component!

The "magic" here is that all components are grouped together under a TIDEEngine. If you want one component to work in sync with another one, just use a TIDEEngine component to group them. The following sections will provide more information about the available components and the engine.

2.5.1 IDE Components available

The "pieces" of the IDE available as componentes are:

TIDEMemo
Stand-alone syntax-highlighting memo for editing script source code. It is inherited from TAdvMemo component.

TIDEFormDesignControl
Stand-alone form designer control to allow designing forms and its child controls.

TIDEPaletteToolbar
TIDEPaletteButtons
Component palette controls. The TIDEPaletteToolbar is a Delphi7-like component toolbar, while TIDEPaletteButtons is a tool palette which looks like the Delphi 2005-2007 component palette. The TIDEPaletteButtons component is not available for Delphi 7 and previous versions.

TIDEInspector
Stand-alone object inspector for viewing/changing properties of components.

TIDEComponentComboBox
A combo box which lists all the components available in the form, and selects the control when the user chooses an item from the combo box. To be used in conjunction with TIDEFormDesignControl

TIDEWatchListView
A stand-alone list view which shows the watches defined in the IDE, for debugging purposes.

2.5.2 The TIDEEngine component

The TIDEEngine component is the code behind the IDE. In other words, it has all the code which makes the IDE work and integrates all IDE components together. All IDE components provide feedback to the engine in order to synchronize other components. For example, when a component is selected in the form designer, the form designer notifies the TIDEEngine so that the engine can notify the inspector to update itself and show the properties of the selected component.

All IDE components have an Engine property which points to a TIDEEngine component. And the TIDEEngine component also have properties which points to the component pieces that builds an IDE. The Engine property in the components are public, and the properties in the TIDEEngine are published, so at design-time you use the TIDEEngine component properties to link everything together. The key properties of the TIDEEngine component are:

Scripter
Points to an TIDEScripter component. The scripter is used to hold the scripts belonging to a project, to retrieve the name of the available event handlers, to refactor, among other functions.

ComponentCombo

Points to a TIDEComponentComboBox component. This component is optional, but if you associated it to the engine, then the engine will update the combo automatically and no extra code is needed to make it work.

DesignControl
Points to a TIDEFormDesignControl component. This component is used to design the form components. The engine synchronizes this component with the inspector, the component combo and the component palette toolbar (or buttons).

Inspector
Points to a TIDEInspector component. This component is used to inspect the properties and events of the component(s) selected in the designer. The engine synchronizes the inspector and the designer accordingly.

Memo
Points to a TIDEMemo component. The engine automatically updates the memo source code with the currently selected unit in the project, and also automatically provides code completion and other features

PaletteToolbar or **PaletteButtons**
Points to a TIDEPaletteToolbar or TIDEPaletteButtons (from Delphi 2005 or above) component. They display at runtime all the components that are available to be dropped in the form designer. Several components are already available, and you can register more components in the IDE if you want to. The engine synchronizes the component palette and the designer, so that a component selected in the toolbar can be dropped in the designer.

TabControl
Points to a regular TTabControl component. This component is used to display the available units in the project, and also to select the desired unit when the end-user clicks a tab.

WatchList
Points to a TIDEWatchListView component. This component shows all active watches in the debugging environment, and the engine automatically updates the watches while debugging.

2.5.3 Basic steps to build a custom IDE

The following steps are a quick start guide to build a custom IDE. With these basic steps you can get a custom IDE running with minimum functionality.

1. Drop a TIDEEngine component in the form.

2. Drop a TIDEScripter component in the form.

3. Drop a TTabControl component in the form.

4. Drop a TIDEMemo component in the TTabControl. You can set Align property to alClient to make it look better.

5. Drop a TIDEFormDesignControl component in the TTabControl. You can set Align property to alClient to make it look better.

6. Drop a TIDEInspector component in the form

7. Drop a TIDEPaletteToolbar (or TIDEPaletteButtons) component in the form

8. Select the TIDEEngine component and set the following properties, pointing to the respective components: Scripter (link to the TIDEScripter component), DesignControl (link to the TIDEFormDesignControl component), Inspector (link to TIDEInspector component), Memo (link to the TIDEMemo component), PaletteToolbar (or PaletteButtons, linking to the TIDEPaletteToolbar or TIDEPaletteButtons component), TabControl (link to the TTabControl component).

That's it, you have the IDE running already. Of course, you need to add several actions to create unit, create form, save, load, etc., and you do that by using the TIDEDialog component programatically. So, as an example, you can perform these extra 9 and 10 steps here to have a project running:

9. Drop a TButton in the form, change the Caption property to "Start" and in the OnClick event add the following code:

```
IDEEngine1.CreateMainUnits(slPascal);
```

10. Drop a TButton in the form, change the Caption property to "Run" and in the OnClick event add the following code:

```
IDEEngine.RunProject;
```

2.5.4 Using ready-to-use inspector and palette forms

As an alternative to using TIDEInspector and TIDEPaletteButtons component, you can use some already built forms which contain those components. The advantage of using the forms is that they add some extra functionality (for example, the inspector form has the tabset which displays the tabs "properties" and "events", while the palette buttons form adds filtering functionality).

TfmObjectInspector form
The form with the inspector is available in the fObjectInspector.pas unit. Just create an instance of the TfmObjectInspector form and set its Engine property to a valid TIDEEngine component.

TfmToolPalette form
The form with the palette buttons is available in the fToolPalette.pas unit. Just create an instance of the TfmToolPalette form and set its Engine property to a valid TIDEEngine component.

2.5.5 Using ready-to-use actions

TMS Scripter also provides a TDataModule which contains several actions that can be used in your custom IDE.
Just add the dIDEActions.pas unit to your project. Link your IDE form to this unit by adding it to the uses clause, create an instance of the TdmIDEActions data module and use the actions as you want. These actions are used by the default IDE provided by the

TIDEDialog component, so you don't need to add extra code to perform basic operations like new project, open project, save file, create unit, copy to clipboard, etc..

2.6 Using TIDEEngine component programatically

The TIDEEngine component is the core component of an IDE in TMS Scripter. It provides several methods and properties to work with the IDE programatically. This topic shows some basic operations you can do with the component in either situation, and in all examples the name of the TIDEEngine component will be *IDEEngine1*.

2.6.1 Creating a new project

Use NewProject method. This will clear all existing files in the project and creating a new blank project:

```
IDEEngine1.NewProject;
```

Optionally, you can ask the engine to create the main units for a very basic project. This would be a blank form, and a separated unit (which will be the main unit) that creates an instance of the form and show it. To do that, call this method (you must pass the language used to create the units):

```
IDEEngine1.CreateMainUnits(slPascal);
```

2.6.2 Adding/removing units (scripts and forms) to the project

You can add new blank units and forms to the project using these methods:

```
var
  ANewUnit: TIDEProjectFile;
  ANewForm: TIDEProjectFile;
begin
  {Creates a blank unit in Basic}
  ANewUnit := IDEEngine1.NewUnit(slBasic);

  {Creates a blank form in Pascal}
  ANewForm := IDEEngine1.NewFormUnit(slPascal);
end;
```

To remove a unit from the project, just destroy the TIDEProjectFile object inside the collection:

```
//Remove Unit1 from project
var
  AUnit: TIDEProjectFile;
begin
  AUnit := IDEEngine1.Files.FindByUnitName('Unit1');
  if AUnit <> nil then
    AUnit.Free;
end;
```

C++Builder example

2.6.3 Executing a project programatically

The example below creates a new project, add a unit with a script source code, and execute it.

```
procedure TForm1.RunSampleProject;
var
  AUnit: TIDEProjectFile;
  AEngine: TIDEEngine;
  AScripter: TIDEScripter;
begin
  AEngine := TIDEEngine.Create(nil);
  AScripter := TIDEScripter.Create(nil);
  AEngine.Scripter := AScripter;
  AEngine.NewProject;
  AUnit := AEngine.NewUnit(slPascal);
  AUnit.Script.SourceCode.Text := 'ShowMessage(''Hello world!'');';
  AEngine.RunProject;
  AEngine.Free;
  AScripter.Free;
end;
```

This example does the same, but instead of executing the code, it opens the IDE with the current unit:

```
procedure TForm1.ShowIDEWithSimpleUnit;
var
  AUnit: TIDEProjectFile;
  ADialog: TIDEDialog;
  AEngine: TIDEEngine;
  AScripter: TIDEScripter;
begin
  ADialog := TIDEDialog.Create(nil);
  AEngine := TIDEEngine.Create(nil);
  AScripter := TIDEScripter.Create(nil);
  ADialog.Engine := AEngine;
  AEngine.Scripter := AScripter;
  AEngine.NewProject;
  AUnit := AEngine.NewUnit(slPascal);
  AUnit.Script.SourceCode.Text := 'ShowMessage(''Hello world!'');';
  ADialog.Execute;
  ADialog.Free;
  AEngine.Free;
  AScripter.Free;
end;
```

C++Builder example

2.6.4 Managing units and changing its properties

All units in a project are kept in a collection named Files (IDEEngine1.Files). Each unit (file) is a TIDEProjectFile object. So, for example, to iterate through all units in a project:

```
var
  AUnit: TIDEProjectFile;
begin
  for c := 0 to IDEEngine1.Files.Count - 1 do
  begin
    AUnit := IDEEngine1.Files[c];
    //Do something with AUnit
  end;
end;
```

C++Builder example

The TIDEProjectFile class has several properties and we list here the main ones (see full component reference for all properties):

Script
Points to the TatScript object inside the scripter component. When a unit is created, it also creates a TatScript object in the Scripter component. They are in sync (the file and the script). Use this to change source code, for example:

```
AUnit.Script.SourceCode.Text := 'ShowMessage(''Hello world!'')';
```

IsForm
Use this function to check if the unit has a form associated with it:

```
HasForm := AUnit.IsForm;
```

2.6.5 Setting the active unit in the IDE

Use ActiveFile property to specify which file is the one selected in the IDE:

```
AMyUnit := IDEEngine1.Files.FindByUnitName('Unit1');
IDEEngine1.ActiveFile := AMyUnit;
```

C++Builder example

2.6.6 Running and debugging a project

To run a project, use RunProject method:

```
IDEEngine1.RunProject;
```

the main unit will be executed. The main unit is the unit specified by IDEEngine1.MainUnit property. There are several methods for debugging the script, and all of them start with "Debug" in method name.

C++Builder example

Here is a list with the main methods:

```
{Pauses the script execution, for IDE debugging purposes}
procedure DebugPause;

{Perform debug step over action in the current active script}
procedure DebugStepOver(RunMode: TIDERunMode = rmMainUnit);

{Perform debug step into action in the current active script}
procedure DebugTraceInto(RunMode: TIDERunMode = rmMainUnit);

{Perform debug action "run to line": run the active script until the
selected line in memo}
procedure DebugRunToLine(RunMode: TIDERunMode = rmMainUnit);

{Perform debug action "run until return": run the active script until the
routine exists}
procedure DebugUntilReturn;

{Halts script execution}
procedure DebugReset;

{Toggle breakpoint on/off in the memo and script.
 If ALine is -1 then current line in memo will be toggled for breakpoint}
procedure DebugToggleBreak(ALine: integer = -1);
```

2.6.7 Methods for end-user interaction - open, save dialogs, etc.

The TIDEEngine component provides several high-level methods for user interaction. All of those methods begin with "Dlg" in the method name, and are used to open/save project and units, closing units, etc.. The difference from the regular methods for saving/loading (or removing units) is that they perform more higher level operations, like displaying the open/save dialogs, checking if the file was saved or not, asking for saving if the file was modified, checking if the unit name exist, etc.. These are the main methods:

```
{Creates a new project. Returns true if the new project is created
sucessfully}
function DlgNewProject: boolean;

{DlgProjectFile opens a dialog for choosing a project file and then open
the project
 file, clearing all units and loading the units belonging to that
project. It returns true if the project
 is opened successfully}
function DlgOpenProject: boolean;

{Call DlgOpenFile to open an existing file in the IDE interface. It will
open a dialog
 for choosing the file, and if confirmed, the new file will be added to
the project
 and opened in the IDE. This method returns the newly created
TIDEProjectFile which contains the opened file.}
function DlgOpenFile: TIDEProjectFile;

{Call DlgSaveFile method to save the file specified by AFile. It
automatically opens the "Save as..."
 dialog if the file was not yet saved for the first time.
 This method returns true if the file was saved succesfully}
function DlgSaveFile(AFile: TIDEProjectFile): boolean;

{Same for DlgSaveFile method, except it automatically saves the currently
active file in the project.
This method returns true if the file was saved succesfully}
function DlgSaveActiveFile: boolean;

{Open the "Save as..." dialog for saving an unit. It performs extra
operations like checking if the unit name
 already exists, and update the script source code (directive "$FORM")
with the correct file name, in case the file name was changed.
 This method returns true if the file was saved succesfully}
function DlgSaveFileAs(AFile: TIDEProjectFile): boolean;

{Same as DlgSaveFileAs, except that it automatically saves the currently
active file.
This method returns true if the file was saved succesfully}
function DlgSaveActiveFileAs: boolean;

{Save all files in the project at once. For each file, if the file is not
saved, it opens a "Save as..." dialog.
 If the dialog is canceled at some point, the remaining files will not be
saved. This function returns true if
 all files were saved sucessfully}
function DlgSaveAll: boolean;

{Closes the file specified by AFile. If the file was already saved, then
it is not removed from project, just made invisible in the IDE.
```

```
 If the file is a new file that was not saved yet, then it's removed. If
the file was modified, the engine asks the user if the file
 must be saved or not. The result of the closing operation is returned in
the TIDECloseFileResult}
function DlgCloseFile(AFile: TIDEProjectFile): TIDECloseFileResult;

{Same as DlgCloseFile, except that it automatically closes the currently
active file}
function DlgCloseActiveFile: TIDECloseFileResult;

{Close all files in the project. It calls DlgCloseFile for each file in
the project. It returns true if all files were closed
 succesfully.}
function DlgCloseAll: boolean;

{Same as DlgRemoveFile, except it removes the currently active file.}
function DlgRemoveActiveFile: boolean;

{Remove the file specified by AFile from the project. If the file was not
saved, it asks for saving it. The method
 returns true if the file was successfully removed}
function DlgRemoveFile(AFile: TIDEProjectFile): boolean;

{Opens a save dialog to save the project. Returns true if the project was
saved sucessfully.}
function DlgSaveProjectAs: boolean;

{Save the current project. If the project was not saved yet, it calls
DlgSaveProjectAs to choose the file name for the project.}
function DlgSaveProject: boolean;

{Calls the Add Watch dialog to add a new watch while debugging. Returns
nil if no watch is added, otherwise returns the newly
 created TatDebugWatch object. There is no need to destroy this object
later, the engine takes care of it automatically.}
function DlgAddWatch: TatDebugWatch;
```

2.7 Registering components in the IDE

This topic covers some tasks that you can do to register (or unregister) components in the IDE system.

2.7.1 Retrieving existing registered components

All the components already registered in the IDE system are available in the TIDEEngine.RegisteredComps property. It is a collection of TIDERegisteredComp objects which holds information for each registered component. As an example, the code below retrieves information about all registered components:

```
var
  ARegComp: TIDERegisteredComp;
  c: integer;

  ACompClass: TComponentClass;
  AUnits: string;
  APage: string;
begin
  for c := 0 to IDEEngine1.RegisteredComps.Count - 1 do
  begin
    ARegComp := IDEEngine1.RegisteredComps[c];

    {Contains the class registered, for example, TButton}
    ACompClass := ARegComp.CompClass;

    {Contains the name of units (separated by commas) that will be added
to the script when the component is dropped in a form.
     For example, 'ComCtrls,ExtCtrls'}
    AUnits := ARegComp.Units;     property CompClass: TComponentClass
read FCompClass write FCompClass;

    {Contains the name of the page (category, tab) where the component
will be displayed. For example, 'Standard'}
    APage := ARegComp.Page;
  end;
end;
```

C++Builder example

2.7.2 Registering/Unregistering standard tabs

The TIDEEngine component provides some methods which register/unregister automatically some components that are commonly used. The methods available are:

```
{Register the following components in the tab "Standard": TMainMenu,
TPopupMenu, TLabel,
 TEdit, TMemo, TButton, TCheckBox, TRadioButton, TListBox, TComboBox,
TGroupBox, TPanel, TRadioGroup}
procedure RegisterStandardTab;

{Register the following components in the tab "Additional": TBitBtn,
TSpeedButton, TMaskEdit,
 TImage, TShape, TBevel, TStaticText, TSplitter}
procedure RegisterAdditionalTab;

{Register the following components in the tab "Dialogs": TOpenDialog,
TSaveDialog, TFontDialog,
 TColorDialog, TPrintDialog, TPrinterSetupDialog}
procedure RegisterDialogsTab;

{Register the following components in the tab "Win32":
 TTabControl, TPageControl, TProgressBar, TTreeView, TListView,
TDateTimePicker}
procedure RegisterWin32Tab;
```

To unregister a tab from the palette, just call UnregisterTab method. Example:

```
IDEEngine1.UnregisterTab('Win32');
```

C++Builder example

2.7.3 Register new components

To register a new component in the component palette, just call RegisterComponent method. For example:

```
{Register the new component TMyComponent in the tab "Custom". When the
user drops this
 component in the form, the units ComCtrls, ExtCtrls and MyComponentUnit
are added to the script.
 These units must be registered in scripter in order to give access to
them in the script environment.
 This registration can be done manually (check "Accessing Delphi objects"
chapter) or using the ImportTool. }
IDEEngine1.RegisterComponent('Custom', TMyComponent,
'ComCtrls,ExtCtrls,MyComponentUnit');
```

C++Builder example

To set the image used to display the component in the palette, use the TIDEEngine.OnGetComponentImage event.

2.8 Storing units in a database (alternative to files)

By default the IDE in TMS Scripter saves projects and units to regular files. It displays open/save dialogs and then open/save the files. But you can also change this behaviour and make the IDE save/load the files in the place you want. The most common use for it is databases. You can also replace the open/save dialogs to display your own dialogs for the end-user to choose the available files to open, or choose the file name to be saved.
To do that, you must add code to some special events of TIDEDialog component. This topic covers those events and how to use them.

2.8.1 Replacing save/load operations

You must add event handler code to two events: OnLoadFile and OnSaveFile.

Declaration:

```
type
  TIDELoadFileEvent = procedure(Sender: TObject; IDEFileType:
TIDEFileType; AFileName: string;
    var AContent: string; AFile: TIDEProjectFile; var Handled: boolean)
of object;
  TIDESaveFileEvent = procedure(Sender: TObject; IDEFileType:
TIDEFileType; AFileName: string;
    AContent: string; AFile: TIDEProjectFile; var Handled: boolean) of
object;

property OnLoadFile: TIDELoadFileEvent read FOnLoadFile write
FOnLoadFile;
property OnSaveFile: TIDESaveFileEvent read FOnSaveFile write
FOnSaveFile;
```

Example:

```
procedure TForm1.IDEEngine1SaveFile(Sender: TObject;
  IDEFileType: TIDEFileType; AFileName, AContent: String;
  AFile: TIDEProjectFile; var Handled: Boolean);
begin
  {The IDEFileType parameter tells you if the file to be saved is a
project file, a script file, or a form file.
   Valid values are: iftScript, iftProject, iftForm}

  {The AFileName string contains the name of the file that was chosed in
the save dialog. Remember that
   you can replace the save dialog by your own, so the AFileName will
depend on the value returned by
   the save dialog}

  {The AContent parameter contains the file content in string format}

  {The AFile parameter points to the TIDEProjectFile object that is being
saved. You will probably not need
   to use this parameter, it's passed only in case you need additional
information for the file}

  {If you save the file yourself, you need to set Handled parameter to
true. If Handled is false, then the IDE engine
   will try to save the file normally}

  {So, as an example, the code below saves the file in a table which
contains the fields FileName and Content. Remember
   that AContent string might be a big string, since it has all the
content of the file (specially for form files)}

  MyTable.Close;
  Case IDEFileType of
    iftScript: MyTable.TableName := 'CustomScripts';
    iftForm: MyTable.TableName := 'CustomForms';
    iftProject: MyTable.TableName := 'CustomProjects';
  end;
  MyTable.Open;
  if MyTable.Locate('FileName', AFileName, [loCaseInsensitive]) then
    MyTable.Edit
  else
  begin
    MyTable.Append;
    MyTable.FieldByName('FileName').AsString := AFileName;
  end;
  MyTable.FieldByName('Content').AsString := AContent;
  MyTable.Post;
  Handled := true;
end;
```

Sample code for loading the file:

```
procedure TForm1.IDEEngine1LoadFile(Sender: TObject;
  IDEFileType: TIDEFileType; AFileName: String; var AContent: String;
  AFile: TIDEProjectFile; var Handled: Boolean);
begin
  {The IDEFileType parameter tells you if the file to be saved is a
project file, a script file, or a form file.
   Valid values are: iftScript, iftProject, iftForm}

  {The AFileName string contains the name of the file that was chosed in
the save dialog. Remember that
   you can replace the save dialog by your own, so the AFileName will
depend on the value returned by
   the save dialog}

  {The AContent parameter contains the file content in string format. You
must return the content in this parameter}

  {The AFile parameter points to the TIDEProjectFile object that is being
saved. You will probably not need
   to use this parameter, it's passed only in case you need additional
information for the file}

  {If you save the file yourself, you need to set Handled parameter to
true. If Handled is false, then the IDE engine
   will try to save the file normally}

  {So, as an example, the code below saves the file in a table which
contains the fields FileName and Content. Remember
   that AContent string might be a big string, since it has all the
content of the file (specially for form files)}

  MyTable.Close;
  Case IDEFileType of
    iftScript: MyTable.TableName := 'CustomScripts';
    iftForm: MyTable.TableName := 'CustomForms';
    iftProject: MyTable.TableName := 'CustomProjects';
  end;
  MyTable.Open;
  if MyTable.Locate('FileName', AFileName, [loCaseInsensitive]) then
  begin
    AContent := MyTable.FieldByName('Content').AsString;
  end else
    raise Exception.Create(Format('File %s not found!', [AFileName]));
  Handled := true;
end;
```

C++Builder example

2.8.2 Replacing open/save dialogs

You must add event handler code to two events: OnOpenDialog and OnSaveDialog. The parameters are similar to the OnLoadFile and OnSaveFile. You must build your own windows to replace the default ones. Remember that in FileName parameter you can also return a path structure like '\MyFiles\MyFileName.psc'. Then you must handle this structure yourself in the OnLoadFile and OnSaveFile events.

Declaration:

```
type
  TIDEOpenDialogEvent = procedure(Sender: TObject; IDEFileType:
TIDEFileType; var AFileName: string;
    var ResultOk, Handled: boolean) of object;
  TIDESaveDialogEvent = procedure(Sender: TObject; IDEFileType:
TIDEFileType; var AFileName: string;
    AFile: TIDEProjectFile; var ResultOk, Handled: boolean) of object;

    property OnSaveDialog: TIDESaveDialogEvent read FOnSaveDialog write
FOnSaveDialog;
    property OnOpenDialog: TIDEOpenDialogEvent read FOnOpenDialog write
FOnOpenDialog;
```

Example:

```
procedure TForm1.IDEEngine1SaveDialog(Sender: TObject;
  IDEFileType: TIDEFileType; var AFileName: String; AFile:
TIDEProjectFile;
  var ResultOk, Handled: Boolean);
begin
  {The IDEFileType parameter tells you if the file to be saved is a
project file, a script file, or a form file.
   Valid values are: iftScript, iftProject. itForm is not used for open/
save dialogs}

  {The AFileName string contains the name of the file that was chosed in
the save dialog. You must return the
   name of the file to be saved here}

  {The AFile parameter points to the TIDEProjectFile object that is being
saved. You will probably not need
   to use this parameter, it's passed only in case you need additional
information for the file}

  {You must set ResultOk to true if the end-user effectively has chosen a
file name. If the end-user canceled the
   operation, set ResultOk to false so that save process is canceled}

  {If you display the save dialog yourself, you need to set Handled
parameter to true. If Handled is false, then the IDE engine
   will open the default save dialog}

  {So, as an example, the code below shows a very rudimentar save dialog
(InputQuery) in replacement to the
   regular save dialog. Note that this example doesn't check if the file
is a
   project or a script. You must consider this parameter in your
application}

  AResultOk := InputQuery('Save unit', 'Choose a file name', AFileName);
  Handled := true;
end;
```

Sample code for replacing open dialog:

```
procedure TForm1.IDEEngine1OpenDialog(Sender: TObject;
  IDEFileType: TIDEFileType; var AFileName: String; var ResultOk,
  Handled: Boolean);
var
  AMyOpenDlg: TMyOpenDlgForm;
begin
  {The IDEFileType parameter tells you if the file to be saved is a
project file, a script file, or a form file.
   Valid values are: iftScript and iftProject. itForm is not used for
open/save dialogs}

  {The AFileName string contains the name of the file that was chosed in
the save dialog. You must return the
   name of the file to be saved here}

  {You must set ResultOk to true if the end-user effectively has chosen a
file name. If the end-user canceled the
   operation, set ResultOk to false so that save process is canceled}

  {If you display the save dialog yourself, you need to set Handled
parameter to true. If Handled is false, then the IDE engine
   will open the default save dialog}

  {So, as an example, the code below shows an open dialog in replacement
to the
   regular save dialog. It considers that the form TMyOpenDlgForm lists
all available units
   from a database table or something similar. Note that this example
doesn't check if the file is a
   project or a script. You must consider this parameter in your
application}

  AMyOpenDlg := TMyOpenDlgForm.Create(Application);
  AResultOk := (AMyOpenDlg.ShowModal = mrOk);
  if AResultOk then
    AFileName := AMyOpenDlg.ChosenFileName;
  AMyOpenDlg.Free;
  Handled := true;
end;
```

C++Builder example

2.8.3 Checking if a file name is valid

Another event that must have code attached is the OnCheckValidFile event. This event is called just after an open dialog is called, and before the file is opened. It is used to check if the file name provided by the open dialog is a valid file name, before effectively opening the file.

Attention: this event is also important for the engine to know if there is a form file associated with a script. When using regular files, the engine tests if the file "UnitName.XFM" exists in order to know if the script has a form or not. So, you must return the correct information for the event so everything works fine.

```
type
  TCheckValidFileEvent = procedure(Sender: TObject; AFileName: string;
var AValid: boolean) of object;
property OnCheckValidFile: TCheckValidFileEvent read FOnCheckValidFile
write FOnCheckValidFile;

procedure TForm1.IDEEngine1CheckValidFile(Sender: TObject; IDEFileType:
TIDEFileType;
  AFileName: String; var AValid: Boolean);
begin
  {The IDEFileType parameter tells you if the file to be checked is a
form, script or project.
   Valid values are: iftScript, iftProject}

  {The AFileName is the file name to be tested}

  {the AValid parameter must be set to true if the file name is valid.}

  {The code below is an example of how to use this event}
  MyTable.Close;
  Case IDEFileType of
    iftScript: MyTable.TableName := 'CustomScripts';
    iftForm: MyTable.TableName := 'CustomForms';
    iftProject: MyTable.TableName := 'CustomProjects';
  end;
  MyTable.Open;
  AValid := MyTable.Locate('FileName', AFileName, [loCaseInsensitive]);
end;
```

C++Builder example

Chapter III

Language Features

3 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.

3.1 Pascal syntax

3.1.1 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**)

3.1.2 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.

3.1.3 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
```

3.1.4 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.';
```

3.1.5 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';
```

3.1.6 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');
```

3.1.7 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.

3.1.8 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[I + 1] returns the character immediately after the one indexed by I. More examples:

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

3.1.9 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';
```

3.1.10 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;
```

3.1.11 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 constrol 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;
```

3.1.12 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);
```

3.1.13 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 reachs 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;
```

3.1.14 case statements

Scripter support case statements with following syntax:

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

if selectorExpression matches the result of one of caseexprn expressions, the respective statement (or block) will be execute. Otherwise, elsestatement 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;
```

3.1.15 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 implicited declared result variable. Parameters by reference can also be used, with the restriction mentioned: no need to specify variable types. Some examples:

```
procedure HelloWord;
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;
```

3.2 Basic syntax

3.2.1 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**)

3.2.2 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.

3.2.3 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
```

3.2.4 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."
```

3.2.5 New statement

TMS Scripter provides the "new" statement for Basic syntax. Since you don't provide the method name in this statement, scripter 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.

3.2.6 Character strings

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

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

3.2.7 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")
```

3.2.8 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 acessible from other scripts.

Variables can be default initialized with the following syntax

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

3.2.9 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[I + 1] returns the character immediately after the one indexed by I. More examples:

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

3.2.10 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"
```

3.2.11 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
```

3.2.12 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 constrol 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
```

3.2.13 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
```

3.2.14 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 reachs 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
```

3.2.15 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
```

3.2.16 function and sub declaration

Declaration of functions and subs are similar to basic. In functions to return function values, use implicited 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 HelloWord
   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 acessible from other scripts.

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

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

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

3.3 Calling dll functions

3.3.1 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 LoadLibary and FreeLibrary from unit Windows can be used.

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

3.3.2 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).

3.3.3 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).

3.3.4 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 IV

Working with scripter

4 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

4.1 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.

4.2 Cross-language feature: TatScripter and TIDEScripter

TMS Scripter provides a single scripter component that allows cross-language and cross-platform scripting: TatScripter.

Replacing old TatPascalScripter and TatBasicScripter by the new TatScripter 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.

4.3 Common tasks

4.3.1 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.

4.3.2 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".

4.3.3 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.

4.4 Accessing Delphi objects

4.4.1 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.

4.4.2 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.

4.4.3 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.

4.4.4 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

4.4.5 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(GetInpu
tArgAsString(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

4.4.6 Acessing 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 tfield object. Note that in SetFieldValueProc method was used GetInputArg (instead of GetInputArgAsString). This is because GetInputArg returns a variant.

4.4.7 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(Memo1.Lines.Strings[3]);
Memo1.Lines.Strings[3]:=Memo1.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;
```

4.4.8 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

4.4.9 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

4.4.10 Delphi 2010 and up - Registering using new RTTI

Taking advantage of new features related to RTTI and available from Delphi 2010, TMS Scripter 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.

4.4.10.1 Registering a class in scripter

To register a class in Scripter, usually we use TatCustomScripter.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 Scripter.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 TatCustomScripter.DefineClassByRTTI method to register the class in scripter, and automatically all their methods and properties:

```
Scripter.DefineClassByRTTI(TMyClass);
```

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

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

- AClass: class to be registered in scripter;
- 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, scripter 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.

4.4.10.2 Registering a record in scripter

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

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

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

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

Records registered in scripter 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 scripter instantiates automatically). Example:

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

4.4.10.3 What is not supported

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

- Indexed properties (arrays) are not automatically registered in scripter. To define them use DefineProp method, passing the dimension of array property in AIndexCount argument.
- Scripter 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, scripter will only define fields and properties declared in these clauses.
- If a class method has overloads, scripter will register only the first method overload declared in that class.
- Methods having parameters with default values, when automatically defined in scripter, 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 scripter. 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 scripter, since Delphi does not provide enough information about these methods.

4.5 Acessing Delphi functions, variables and constants

4.5.1 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.

4.5.2 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.

4.5.3 Acessing 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));
```

4.5.4 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:=Memo1.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

4.6 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 Studo, 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 TatScript 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.

4.7 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 folowing 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.

4.8 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 scripter 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 scripter component.

Limitations
Since scripter 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.

4.9 Using the Refactor

Every TatScript object in Scritper.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 TMS Scripter are 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. Basicaly is 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;
```

4.10 Using libraries

4.10.1 Overview

Libraries are just a concept of extending scripter 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.

4.10.2 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(TatScripterLibrary)
   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:=Memo1.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:=Memo1.Lines;
   Scripter.Execute;
end;
```

C++Builder example

Both codes do the same: add CurrToStr procedure to script. Note that scripter 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, scripter preparation is done adding TExampleLibrary class to scripter, 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 confortable. 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 TatScripterLibrary class descendant:
- Pros: No need to check if class is instantiated – scripter does it automatically. It is easy to port code – all methods are inside a class library, so you can add it in any scripter 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 RegisterScripterLibrary procedure. For example:

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

RegisterScripterLibrary is a global procedure that registers the library in a global list, so all scripter components are aware of that library. The second parameter of RegisterScripterLibrary 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 scripter 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.

4.10.3 The TatSystemLibrary library

There is a library that is added by default to all scripter components, it is the TatSystemLibrary. This library is declared in the uSystemLibrary unit. It adds commonly used routines and functions to scripter, 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]);
```

4.10.4 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

4.10.5 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

4.11 Debugging scripts

4.11.1 Overview

TMS Scripter 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.

4.11.2 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.

4.11.3 Using debug components

TMS Scripter has specific component for debugging (only for VCL applications). 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

4.12 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.

4.13 C++ Builder issues

4.13.1 Overview

Since TMS Scripter 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 TMS Scripter.

4.13.2 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 V

The syntax highlighting memo

5 The syntax highlighting memo

5.1 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:

```
AvdMemo1.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 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 Integrated Development Environment

6.1.1 Using TIDEEngine component programatically

6.1.1.1 Adding/removing units (scripts and forms) to the project

```
TIDEProjectFile *ANewUnit, *ANewForm;

// Creates a blank unit in Basic
ANewUnit = IDEEngine1->NewUnit(slBasic);

// Creates a blank form in Pascal
ANewForm = IDEEngine1->NewFormUnit(slPascal);

// Remove Unit1 from project
TIDEProjectFile *AUnit = IDEEngine1->Files->FindByUnitName("Unit1");
if(AUnit != NULL)
  delete AUnit;
```

Original topic

6.1.1.2 Executing a project programatically

```
void __fastcall TForm1::RunSampleProject()
{
  TIDEProjectFile *AUnit;
  TIDEEngine *AEngine;
  TIDEScripter *AScripter;

  AEngine = new TIDEEngine(NULL);
  AScripter = new TIDEScripter(NULL);

  AEngine->Scripter = AScripter;
  AEngine->NewProject();
  AUnit = AEngine->NewUnit(slPascal);
  AUnit->Script->SourceCode->Text = "ShowMessage('Hello world!');";

  AEngine->RunProject();
  delete AEngine;
  delete AScripter;
}

void __fastcall TForm1::ShowIDEWithSimpleUnit()
{
  TIDEProjectFile *AUnit;
  TIDEDialog *ADialog;
  TIDEEngine *AEngine;
  TIDEScripter *AScripter;

  ADialog = new TIDEDialog(NULL);
  AEngine = new TIDEEngine(NULL);
  AScripter = new TIDEScripter(NULL);

  ADialog->Engine = AEngine;
  AEngine->Scripter = AScripter;
  AEngine->NewProject();
  AUnit = AEngine->NewUnit(slPascal);
  AUnit->Script->SourceCode->Text = "ShowMessage('Hello world!');";
  ADialog->Execute();
  delete ADialog;
  delete AEngine;
  delete AScripter;
}
```

Original topic

6.1.1.3 Managing units and changing its properties

```
TIDEProjectFile *AUnit;

for(int c = 0; c < IDEEngine1->Files->Count; c++)
{
  AUnit = IDEEngine1->Files->Items[c];
  // Do something with AUnit
}
```

Original topic

6.1.1.4 Setting the active unit in the IDE

```
TIDEProjectFile *AMyUnit;

AMyUnit = IDEEngine1->Files->FindByUnitName("Unit1");
IDEEngine1->ActiveFile = AMyUnit;
```

Original topic

6.1.1.5 Running and debugging a project

```
IDEEngine1->RunProject();
```

Original topic

6.1.2 Registering components in the IDE

6.1.2.1 Retrieving existing registered components

```
TIDERegisteredComp *ARegComp;
TComponentClass ACompClass;
AnsiString AUnits, APage;

for(int c = 0; c < IDEEngine1->RegisteredComps->Count; c++)
{
  ARegComp = IDEEngine1->RegisteredComps->Items[c];

  // Contains the class registered, for example, TButton
  ACompClass = ARegComp->CompClass;

  // Contains the name of units (separated by commas) that will be
  // added to the script when the component is dropped in a form.
  // For example, 'ComCtrls,ExtCtrls'
  AUnits = ARegComp->Units;

  // Contains the name of the page (category, tab) where the component
  // will be displayed. For example, 'Standard'
  APage = ARegComp->Page;
}
```

Original topic

6.1.2.2 Registering/Unregistering standard tabs

```
IDEEngine1->UnregisterTab("Win32");
```

Original topic

6.1.2.3 Register new components

```
// Register the new component TMyComponent in the tab "Custom". When the
user
// drops this component in the form, the units ComCtrls, ExtCtrls and
// MyComponentUnit are added to the script.
// These units must be registered in scripter in order to give access to
them in
// the script environment.
// This registration can be done manually (check "Accessing Delphi
objects"
// chapter) or using the ImportTool.
IDEEngine1->RegisterComponent("Custom", __classid(TMyComponent),
  "ComCtrls,ExtCtrls,MyComponentUnit");
```

Original topic

6.1.3 Storing units in a database (alternative to files)

6.1.3.1 Replacing save/load operations

```
void __fastcall TForm1::IDEEngine1SaveFile(TObject *Sender,
  TIDEFileType IDEFileType, AnsiString AFileName, AnsiString AContent,
  TIDEProjectFile *AFile, bool &Handled)
{
  // The IDEFileType parameter tells you if the file to be saved is a
project
  // file, a script file, or a form file.
  // Valid values are: iftScript, iftProject, iftForm}

  // The AFileName string contains the name of the file that was chosed
```

```
in the
  // save dialog.
  // Remember that you can replace the save dialog by your own, so the
AFileName
  // will depend on the value returned by the save dialog

  // The AContent parameter contains the file content in string format

  // The AFile parameter points to the TIDEProjectFile object that is
being
  // saved. You will probably not need to use this parameter, it's passed
only
  // in case you need additional information for the file

  // If you save the file yourself, you need to set Handled parameter to
true.
  // If Handled is false, then the IDE engine will try to save the file
normally

  // So, as an example, the code below saves the file in a table which
contains
  // the fields FileName and Content. Remember that AContent string might
be a
  // big string, since it has all the content of the file (specially for
form
  // files)

  MyTable->Close();
  switch(IDEFileType)
  {
    case iftScript:
      MyTable->TableName = "CustomScripts";
      break;
    case iftForm:
      MyTable->TableName = "CustomForms";
      break;
    case iftProject:
      MyTable->TableName = "CustomProjects";
      break;
  }
  MyTable->Open();
  if(MyTable->Locate("FileName",AFileName, TLocateOptions()
<<loCaseInsensitive))
    MyTable->Edit();
  else
  {
    MyTable->Append();
    MyTable->FieldByName("FileName")->AsString = AFileName;
  }
  MyTable->FieldByName("Content")->AsString = AContent;
  MyTable->Post();
  Handled = true;
}

void __fastcall TForm1::IDEEngine1LoadFile(TObject *Sender,
  TIDEFileType IDEFileType, AnsiString AFileName, AnsiString &AContent,
  TIDEProjectFile *AFile, bool &Handled)
{
  // The IDEFileType parameter tells you if the file to be saved is a
project
  // file, a script file, or a form file.
  // Valid values are: iftScript, iftProject, iftForm

  // The AFileName string contains the name of the file that was chosed
in the
  // save dialog. Remember that you can replace the save dialog by your
own, so
  // the AFileName will depend on the value returned by the save dialog

  // The AContent parameter contains the file content in string format.
You must
  // return the content in this parameter

  // The AFile parameter points to the TIDEProjectFile object that is
being
  // saved. You will probably not need to use this parameter, it's passed
only
  // in case you need additional information for the file

  // If you save the file yourself, you need to set Handled parameter to
true.
  // If Handled is false, then the IDE engine will try to save the file
normally

  // So, as an example, the code below saves the file in a table which
contains
  // the fields FileName and Content. Remember that AContent string might
be a
  // big string, since it has all the content of the file (specially for
form
  // files)
  MyTable->Close();
  switch(IDEFileType)
  {
    case iftScript:
      MyTable->TableName = "CustomScripts";
      break;
    case iftForm:
      MyTable->TableName = "CustomForms";
      break;
    case iftProject:
      MyTable->TableName = "CustomProjects";
      break;
  }
  MyTable->Open();
  if(MyTable->Locate("FileName",AFileName, TLocateOptions()
<<loCaseInsensitive))
    AContent = MyTable->FieldByName("Content")->AsString;
  else
    throw Exception(Format("File %s not found!",
      OPENARRAY(TVarRec, (AFileName))));
  Handled = true;
}
```

Original topic

6.1.3.2 Replacing open/save dialogs

```
void __fastcall TForm1::IDEEngine1SaveDialog(TObject *Sender,
  TIDEFileType IDEFileType, AnsiString &AFileName,
  TIDEProjectFile *AFile, bool &ResultOk, bool &Handled)
{
  // The IDEFileType parameter tells you if the file to be saved is a
project
  // file, a script file, or a form file.
  // Valid values are: iftScript, iftProject. itForm is not used for
open/save
  // dialogs

  // The AFileName string contains the name of the file that was chosed
in the
  // save dialog. You must return the name of the file to be saved here
  // The AFile parameter points to the TIDEProjectFile object that is
being
  // saved. You will probably not need to use this parameter, it's passed
only
  // in case you need additional information for the file
  // You must set ResultOk to true if the end-user effectively has chosen
a file
  // name. If the end-user canceled the operation, set ResultOk to false
so that
  // save process is canceled
  // If you display the save dialog yourself, you need to set Handled
parameter
  // to true. If Handled is false, then the IDE engine will open the
default
  // save dialog

  // So, as an example, the code below shows a very rudimentar save
dialog
  // (InputQuery) in replacement to the regular save dialog. Note that
this
  // example doesn't check if the file is a project or a script. You must
  // consider this parameter in your application
  ResultOk = InputQuery("Save unit", "Choose a file name", AFileName);
  Handled = true;
}

void __fastcall TForm1::IDEEngine1OpenDialog(TObject *Sender,
  TIDEFileType IDEFileType, AnsiString &AFileName, bool &ResultOk,
  bool &Handled)
{
  // The IDEFileType parameter tells you if the file to be saved is a
project
  // file, a script file, or a form file.
  // Valid values are: iftScript and iftProject. itForm is not used for
  // open/save dialogs

  // The AFileName string contains the name of the file that was chosed
in the
  // save dialog. You must return the name of the file to be saved here
  // You must set ResultOk to true if the end-user effectively has chosen
a file
  // name. If the end-user canceled the operation, set ResultOk to false
so that
  // save process is canceled
  // If you display the save dialog yourself, you need to set Handled
parameter
  // to true. If Handled is false, then the IDE engine will open the
default
  // save dialog

  // So, as an example, the code below shows an open dialog in
replacement to
  // the regular save dialog. It considers that the form TMyOpenDlgForm
lists
  // all available units from a database table or something similar. Note
that
  // this example doesn't check if the file is a project or a script. You
must
  // consider this parameter in your application

  TMyOpenDlgForm *AMyOpenDlg;

  AMyOpenDlg = new TMyOpenDlgForm(Application);
  ResultOk = AMyOpenDlg->ShowModal() == mrOk;
  if(ResultOk)
    AFileName = AMyOpenDlg->ChosenFileName;
  delete AMyOpenDlg;
  Handled = true;
}
```

Original topic

6.1.3.3 Checking if a file name is valid

```
void __fastcall TForm1::IDEEngine1CheckValidFile(TObject *Sender,
  TIDEFileType IDEFileType, AnsiString AFileName, bool &AValid)
{
  // The IDEFileType parameter tells you if the file to be checked is a
form,
  // script or project.
  // Valid values are: iftScript, iftProject

  // The AFileName is the file name to be tested

  // The AValid parameter must be set to true if the file name is valid.

  // The code below is an example of how to use this event

  MyTable->Close();
  switch(IDEFileType)
  {
    case iftScript:
      MyTable->TableName = "CustomScripts";
      break;
    case iftForm:
      MyTable->TableName = "CustomForms";
      break;
    case iftProject:
      MyTable->TableName = "CustomProjects";
      break;
  }
  MyTable->Open();
  AValid = MyTable->Locate("FileName", AFileName,
    TLocateOptions() << loCaseInsensitive);
}
```

Original topic

6.2 Working with scripter

6.2.1 Getting started

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

Scripter->SourceCode->Text = AdvMemo1->Lines->Text;
Scripter->Execute();
```

Original topic

6.2.2 Cross-language feature: TatScripter and TIDEScripter

```
TatScript *MyPascalScript, *MyBasicScript;

MyPascalScript = atScripter1->AddScript(slPascal);
```

```
MyBasicScript = atScripter1->AddScript(slBasic);
```

Original topic

6.2.3 Common tasks

6.2.3.1 Calling a subroutine in script

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

Original topic

6.2.3.2 Returning a value from script

```
Variant FunctionValue;

FunctionValue = Scripter->ExecuteSubroutine("Calculate");

FunctionValue = Scripter->Execute();
```

Original topic

6.2.3.3 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

6.2.4 Accessing Delphi objects

6.2.4.1 Registering Delphi components

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

Original topic

6.2.4.2 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

6.2.4.3 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

6.2.4.4 Acessing 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

6.2.4.5 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(Memo1);
  TatClass *stringsClass = Scripter->DefineClass(__classid(TStrings));
  stringsClass->DefineProp("Strings", Atscript::tkString, GetStringsProc,
    SetStringsProc, NULL, false, 1);
}
```

Original topic

6.2.4.6 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

6.2.4.7 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

6.2.5 Acessing Delphi functions, variables and constants

6.2.5.1 Registering global constants

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

Original topic

6.2.5.2 Acessing 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 = Memo1->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

6.2.5.3 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 = Memo1->Lines;
  Scripter->Execute();
}
```

Original topic

6.2.6 Using libraries

6.2.6.1 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 = Memo1->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 = Memo1->Lines;
  Scripter->Execute();
}
```

Original topic

6.2.6.2 Removing functions from the System library

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

Original topic