HTML Scripter

Table of Contents

1 Introduction

HTML Scripter is a cross-platform and 100% native scripting library for Delphi. It supports most of Object Pascal language features including modern anonymous functions and for-in statement. Scripter engine is GUI-independent and threadsafe, so can be used in both desktop and server applications.
Library is optimized for both execution and parsing/compiling performance and can be used in high-loaded applications. It also support special features, like JQuery $() function for using inside HTML Component Library THtDocument class.

Supported platforms:

- Win32/VCL
- Win64/VCL
- Win32/FMX
- Win64/FMX
- OSX
- Android
- iOS
- Linux

Supported Delphi versions

Delphi 5 - Delphi 10.3 Rio.

Main features

- Extremely **fast** parsing and compilation.
- Using Delphi methods and properties via **RTTI** (2010+).
- Easy registration of Delphi functions (no need for special intermediate functions, just pass function address and description).
- **Anonymous** functions.
- **for .. in ..** statement.
- **DLL** functions.
- Ordinal types **helpers**
- Using script functions as Delphi **event handlers**.
- **Debugging** and logging support.
- Profiling
- **Set of** and **array** parameters.
- **Asynchronous** and delayed execution
- **HTML** documents integration (**JQuery** $ operator)

2 Getting started

Simplest example of how to use THtScriptParser:

```
uses htscriptparse, htscriptgui;
...

procedure TForm1.ButtonClick(Sender: TObject);
var SP: THtScriptParser;
begin
  SP := THtScriptParser.Create('ShowMessage(''test'');');
  try
    SP.Run;
  finally
    SP.Free
  end;
end;
```

3 Script Language

3.1 Overview

Scripter language syntax is almost identical to Object pascal syntax except the following:

- Variables declaration is possible but not required (if soRequireVarDeclaration is not set in Options).
- Classes declaration is not supported

3.2 Script structure

Script has the following structure: (all sections are optional)

[program <name>; | unit <name>;]
[interface]
[uses <unit list>]
[const and functions declarations]
[main code block | implementation]

[initialization]

Example:

```
const
  MyText = 'Sample text';

procedure Sample(s: string);
begin
  ShowMessage(s);
end;

Sample(MyText);
```

Example of unit:

```
unit MyTest;

interface

function MyInttoStr(Value: integer): string;

implementation
```

```
function MyInttoStr(Value: integer): string;
begin
  if Value = 0 then
    Result := ''
  else
    Result := InttoStr(Value)
end;

end.
```

3.3 Expressions

Following operations are allowed in expressions
* , / , and , + , - , or , <> , >=, <= , = , > , < , div , mod , xor , shl , shr, ^, @, is, not, in, not in

"in" operation can be used with arrays:

```
  if s in ['abc', 'def'] then ...
```

sets or ranges

```
    if n in [1..10] then
```

and classes with enumerators (see Registering for-in enumerators) 18
Example:

```
  L := TStringList.Create();
  L.Add('abc');
  L.Add('def');
  if s in L then ...
```

is operator supports simple types:

```
  if k is string then ...;
```

Compound assignment operators:

```
  k += 2;
  k -= 2;
  k *= 2;
```

Ternary operator:

```
a := if b = 1 then 2 else 3;
```

case operator:

```
a := case b of 1: 2; 1+1: 4-1; else 4 end;
```

3.4 Comments

Following comment styles can be used:

```
{ multiline comment }

(* multiline comment 2 *)

// .. single line comment
```

3.5 Statements

List of supported statements:

Assignment:

```
<variable or property> := <expression>;
(including compound assignments +=, -=, *=)

if <expression> then
  <statement>
[else <statement>]

for <variable> := <from expression> to <to expression> do
  <statement>

for <variable> in <expression> [index <variable>] do
  <statement>

"for in" statement has optional index variable which is set to current loop iteration
Note that when using with string starting with 1 (ZEROBASEDSTRINGS OFF) loop will sta
```

loop variable can be declared inside **for** statement, with or without type:

```
for var i: integer := 1 to N do
for var i := 1 to N do

while <expression> do
  <statament>

repeat
 <statement>
until <expression>

case <expression> of
  <expression>: <statement>;
  ..
  <expression>: <statement>;
 [ else <statement> ]
end;
```

case expressions can be of any type.

```
raise <exception object>;

try
  <statement>
except
  [on <variable>: <type> do ]
  <statement>
end;

try
  <statement>
finally
  <statement>
end;
```

3.6 Variables

Variables can be declared inside function body, before **begin**.
All variables declared inside function, function parameters and Result variable is treated as local and does not affect script global variables.
For example:

```
procedure Test();
var a: integer;
begin
  a := 100;
end;

a := 200;
Test();
// a is still 200

variable declaration can contain initial value:

var a: integer = 0;
```

Variables used as **for** statement iterator in local functions are treated as local variables.

Variables ordinal types:

- array
- ansistring
- boolean
- byte
- cardinal
- char
- currency

- double
- extended
- integer
- int64
- longint
- pchar
- pointer
- rawbytestring
- string
- single
- set
- smallint
- TDatetime
- variant
- widestring
- word

3.7 Arrays

Library supports single-dimensional arrays.
Arrays can be passed as function parameters, for example

```
s := Format('Value is %s', [value]);
```

and assigned in code:

```
a := [1, 2, 3];
```

Following functions can be used on arrays:
High() - array high bound
Length() - array length
SetLength() - set array length
[**index**] - get or set array element

Arrays supports **in** operator, f.e.

```
if 5 in a then ...
```

and can be used in **for-in** statements:

```
for x in a do ...
```

3.8 Numbers

Supported number formats:

123 // integer
123.45 //float
123.45e2 //float
$A1B2 //Hex
%1110101 //Binary

3.9 Function declaration

Script function has the following structure:

```
procedure|function <name>( [parameters] ) [: <result type>]);
[ var <variables>; ]
begin
 <function body>
end;
```

Example:

```
function Substring(s: string; Start, Len: integer): string;
begin
  Result := copy(s, Start, Len)
end;
```

functions can have default parameters.
Example:

```
function Substring(s: string; Start: integer = 1; Len: integer = MaxInt): string;
begin
  Result := copy(s, Start, Len)
end;
```

3.10 Anonymous functions

Library supports declaration of anonymous functions. Anonymous functions can be passed as parameter or assigned to variable.
Example:

```
t := function(s: string): string;
     begin
       Result := copy(s, 2, MaxInt)
```

```
    end);
ShowMessage(t('123'));
```

3.11 Reference to function

Variables can contain reference to any function defined in script or registered from Delphi code.
Example:

```
t := @ShowMessage;
t('test');
```

3.12 Ordinal types helpers

Ordinal types helpers are available for Delphi XE4+.
Helpers (from SysUtils unit) are available for the following types:

- string
- single
- double
- integer

For example, you can write

```
ShowMessage('abc'.ToUpper());
s := 'test';
s1 := s.substring(1, 2);
```

Overloaded helper methods

Scripter can distinct between overloaded helper methods with different number of parameters.
For example, **TStringHelper** has two **Substring** methods:

```
    function Substring(StartIndex: Integer): string; overload;
    function Substring(StartIndex: Integer; Length: Integer): string; overload;
```

You can use both:

`a.SubString(1)` and `a.SubString(1, 3)`

3.13 Asynchronous functions

Library has several build-in functions providing asynchronous code execution.

Asynchronous execution

```
procedure Async(AsyncFunction, AfterFunction);

AsyncFunction: function: variant;
AfterFunction: procedure(Value: variant);
```

AsyncFunction is executed in separate thread. When execution is finished, **AfterFuncion** is executed in a main thread context, and result of AsyncFunction is passed as AfterFunction parameter.

Passing parameters

Asynchronous function can access global variables, but they can be changed in a main thread while asynchronous function is executed. Values can be passed directly to both AsyncFunction and AfterFunction using third parameter of Async. Example:

```
Async(
  function(n: integer) begin Result := n + 1 end,
  function(n, res: integer) begin ... end,
  [123]
);
```

Both functions should have same set of parameters, but AfterFunc also have additional parameter for passing AsyncFunction result.

Delayed execution

Similarly to Javascript SetTimeout, there is

```
procedure SetTimeout(AFunction: procedure; Timeout: integer);

AFunction is executed in a main thread context after Timeout milliseconds.

Timer can be reset using ClearInterval function. Example:

t := SetTimeout(@MyFunc, 500);
ClearInterval(t);
```

4 Using Delphi classes and functions

4.1 Registeging Delphi classes and methods

Registration of Delphi class is necessary (for Delphi 2009+) only in following cases

- object of this class will be created in script via constructor
- class is used in 'is' operator
- object of this class is used in for-in statement.
- RTTI is disabled for the class.

Classes can be registered for global usage (in any script) or for single Scripter instance. To register global class use **HtScriptGlobal** instance for registration.

Delphi class registration:

Scripter.**RegisterClass**(<constructor declaration>, <constructor>, <class>);

Example:

```
HtScriptGlobal.RegisterClass('Create()', @TStringList.Create,
 TStringList);
```

Registering methods and properties:

For Delphi 2009+ and classes with RTTI enabled, registering properties and methods is optional. All properties and methods that has RTTI or Extended RTTI will be available in script without registration.

Example of registering class with properties and methods:

```
TStringsHack = class(TStrings);

 with HtScriptGlobal.RegisterClass('Create()', @TStrings.Create, TStrings) do
 begin
  RegisterProperty('Count', 'integer', @TStringsHack.GetCount, nil);
  RegisterProperty('Items', 'string', @TStringsHack.Get, @TStringsHack.Put,
    'integer', true);
  RegisterProperty('Objects', 'TObject', @TStringsHack.GetObject,
    @TStringsHack.PutObject, 'integer');
  RegisterProperty('Text', 'string', @TStringsHack.GetTextStr,
    @TStringsHack.SetTextStr);
  RegisterMethod('Add(const s: string)', @TStrings.Add);
  RegisterMethod('AddObject(const s: string; O: TObject)',
    @TStrings.AddObject);
  RegisterMethod('Delete(Index: integer)',
```

```
    @TStrings.Delete);
  RegisterMethod('Exchange(Index1, Index2: integer)',
    @TStrings.Exchange);
  RegisterMethod('Insert(Index: Integer; const s: string)',
    @TStrings.Insert);
end;
```

4.2 Default properties

For Delphi versions with extended RTTI support (2010+) default properties are determined automatically.
Simply call them with standard syntax:

```
StringList[i] := 'abc';
```

4.3 Generic method handler

Generic method handlers are intended for processing several class methods or properties in one handler. When class has registered generic method handler, all calls to class methods or object instance methods and properties are first processed by generic handler.
For example it can be used for implementing SOAP/REST client class where all methods calls will be passed directly to a server.
To register generic method handlers, declare procedure of the following type:

```
function(const Sender: TScriptParser; const Instance: TObject;
  const MethodName: string; var Params: TScriptStack; StackTop,
  AParamCount: integer; IsSetter: boolean; var Res: Variant): boolean;
```

and set GenericHandler property of the class:

```
  HtScriptGlobal.RegisterClass('create()', @TTestClass.Create,
    TTestClass).GenericHandler := MyGeneric;
```

When IsSetter is true, method is property setter and new value is passed in Res variable, in other cases method should return value in Res.
Handler should return **true** if method is processed successfully, otherwise standard method/property processing will be used.
Generic handler will be used even when it is registered to object class ancestor, but only first found handler will be executed. For example:

```
type
  C1 = class;
  C2 = class(C1);
  C3 = class(C2);
```

if generic handlers are registered for C1 and C2 and instance is of class C3, only handler for C2 will be called.
For class methods (including constructor) Instance parameter contains nil.

Example of generic handler for TDataset.Fields property.
Currently this property cannot be used properly inside script because it is declared as
Fields: TFields
instead of
Fields[Index: integer]: TField.

```
function TDatasetGenericHandler(const Sender: THtScriptParser; const Instance: TObject
  const MethodName: string; var Params: TScriptStack; StackTop, AParamCount: integer;
begin
  Result := false;
  if SameText(MethodName, 'fields') then
  begin
    Result := true;
    Res := Object2Variant((Instance as TDataSet).Fields[Params[StackTop].Value]);
  end;
end;

initialization

 HtScriptGlobal.RegisterControlClass(@TDataSet.Create, TDataSet).GenericHandler := TDa
```

4.4 Registering for-in enumerators

To use an object of some class in **for-in** statement it is necessary to register a class enumerator.
Class enumerator is a descendant of abstract **THtScriptEnumerator** class:

```
  THtScriptEnumerator = class
  public
    ///<summary> Create enumerator for Instance object</summary>
    constructor Create(const Instance: TObject); virtual; abstract;
    ///<summary> Move to next item. Return false if end of list is reached </summary>
    function Next: boolean; virtual; abstract;
    ///<summary> Get current item </summary>
    procedure GetCurrent(var Value: variant); virtual; abstract;
  end;
```

Example of enumerator implementation:

```
constructor TStringListEnumerator.Create(const Instance: TObject);
begin
  fIndex := -1;
  fList := Instance as TStringList;
```

```
end;

procedure TStringListEnumerator.GetCurrent(var Value: variant);
begin
  Value := fList[fIndex]
end;

function TStringListEnumerator.Next: boolean;
begin
  inc(fIndex);
  Result := fIndex < fList.Count;
end;
```

Registering class enumerator:

```
  with HtScriptGlobal.RegisterClass('Create()', @TList.Create, TList) do
  begin
    EnumeratorClass := TListEnumerator;
  end;
```

4.5 Registering Delphi functions

Functions can be registered for global usage (in any script) or for Scripter instance.
To register global function use **HtScriptGlobal** instance for registration.

To register common Delphi function pass function declaration and pointer to function:

Example:

```
 HtScriptGlobal.RegisterFunc('WeekOf(AValue: TDateTime): Word', @WeekOf);
```

Important note: functions should use "Register" calling convention.

Registering class function (non-static):

```
 HtScriptGlobal.RegisterFunc('Error(Msg: string; Data: integer)',
@TList.Error, nil, TList);
```

Registering object function (method):

```
 MyScript.RegisterFunc('Sample()', @MyObject.Sample, MyObject);
```

Object function call looks in script code like normal function call:

```
Sample();
```

but it will execute **TMyObject.Sample()** method of **MyObject** object instance.

Parameters default values

functions can have parameters with default values.
Example:

```
HtScriptGlobal.RegisterFunc('MyFunc(s: string; Index: integer = 1)', @MyFunc);
```

4.6 "Set of" parameters

Function parameter with "set of.." type should be registered as "set".
Example:

```
HtScriptGlobal.RegisterFunc('MessageDlg(Msg: string; DlgType: integer;
  Buttons: Set; HelpCtx: integer): Integer', @MessageDlg);
```

Calling in script:

```
MessageDlg('Test', mtWarning,  [mbYes, mbNo, mbCancel], 0);
```

4.7 Array parameters

Array parameters are supported with standard syntax:

function([a1, a2, a3])

4.8 Registering DLL functions

Example of DLL function declaration:

```
function MessageBox(Handle: integer; Text, Caption: pchar; uType: integer): Integer;
  external 'user32.dll' name 'MessageBoxW';
```

4.9 Magic functions

Magic function is a special function used in following cases
- number of parameters can vary
- Parameter types are not known
- parameter cannot be passed in a standard way (for example - array)

- function needs Scripter instance to work.
- function has var parameters (for example Insert or Delete)

Magic function has following declaration:

```
function MagicFunction(const Sender: TScriptParser; var Params: TScriptStack;
  StackTop, ParamCount: integer): Variant;
```

Sender is Scripter instance in which function is called.
Params is pointer to current runtime stack (array of TExVariable)
StackTop is index of top stack variable
ParamCount - number of parameters in call.

Example of magic function implementation:

```
function MagicInc(const Sender: THtScriptParser; var Params: TScriptStack;
  StackTop, ParamCount: integer): Variant;
begin
  if ParamCount = 1 then
  begin
    if Params[StackTop].IsTemp then
      Sender.Tokenizer.Error(sVarRequiredforInc);
    Result := Params[StackTop].Value + 1;
    Params[StackTop].Value := Result;
  end else begin
    if Params[StackTop - 1].IsTemp then
      Sender.Tokenizer.Error(sVarRequiredforInc);
    Result := Params[StackTop - 1].Value + Params[StackTop].Value;
    Params[StackTop - 1].Value := Result;
  end;
end;
```

Registering magic function:

```
  HtScriptGlobal.RegisterMagicFunc('Inc', MagicInc);
```

Getting function name.

When magic function is registered with different names, current (called) function name can be obtained via call stack:

```
 Name := TScriptFunc(Sender.DebugStack[Sender.DebugStack.Count - 1]).Name
```

4.10 Registering constants

Constants can be registered for global usage (in any script) or for Scripter instance.
To register global constant use **HtScriptGlobal** instance for registration.

Example:

```
HtScriptGlobal.RegisterConst('MaxInt', MaxInt);
```

4.11 Registering enumerations

To register enumeration type use **TScriptParser.RegisterEnum** method.
Example:

```
HtScriptGlobal.RegisterEnum(TypeInfo(TMsgDlgType));
```

Every element of enumeration is registered as constant. Maximum enumeration size is 32 for 32-bit target.

4.12 Creating and destroying objects

Objects created in script using constructors, for example

```
L := TStringList.Create();
```

will be destroyed automatically with scripter instance. Also you can destroy it manually via Free() call:

```
L.Free();
```

After calling the destructor, object variable is set to null.

If an object created in script should be available after script is destroyed, release it using

```
ReleaseObject(O: TObject): TObject
```

function.
If you create an object in Delphi code (for using in script) and want this object to be added into auto-free list use

```
Scripter.CreateObject(MyObject);
```

method.

Also you can set AutoFreeResult flag when registering a function:

```
HtScriptGlobal.RegisterFunc('CreateSpecialList: TObject',
 @CreateSpecialList).AutoFreeResult := true;
```

so all objects created via this function will be destroyed automatically.

Creating and destroying Components and Controls.

Since TComponent has Owner/Components mechanism for destroying, there is special soAutoFreeComponents flag in Script.Options.
When set to false (by default) all objects of TComponent type created by constructor and having Owner will not be destroyed by script.

4.13 Using script functions as Delphi event handler

Script functions can be used as Delphi components event handler. Currently library supports only events of following types:
TNotifyEvent
TCloseQuery
TMouseDown
TMouseMove
TMouseWheel

To use script function as event handler simply assign a function to event property in a script code:

```
procedure TestEvent(Sender: TObject);
begin
  ShowMessage(Sender.Name);
end;

Form.OnDblClick := @TestEvent;
```

Setting event handler from Delphi code

To set script function as event handler use TScriptFunc.AsNotifyEvent property.
Example:

```
 Form1.OnDblClick := Script.CreateAnonymousFunction('MessageDlg(''Test'',
  mtWarning, [mbYes, mbNo, mbCancel], 0)').AsNotifyEvent;

Using existing script function:

 Form1.OnDblClick := Script.FindFunction('MyEventHandler').AsNotifyEvent;
```

4.14 Using script functions for callback

Script functions can be used as callback functions in Delphi Code.
For example, there is a class containing list of objects, and we want to sort it from script with custom compare procedure.

In Delphi code we declare compare procedure:

```
function TMyList.ScriptCompare(Func: TScriptFunc;
  const E1, E2: TListElement): integer;
begin
    Result := Func.Owner.RunFunction(Func, [Object2Variant(E1), Object2Variant(E2)])
end;
```

and Sort procedure

```
procedure TMyList.SortElements(Compare: TScriptFunc; Asc: boolean);
begin
  ...
end;
```

Sort procedure is called from script code:

```
 MyList.SortElements(
   function(E1, E2: TObject);
   begin
    ...
   end,
  true);
```

4.15 Changing or disabling standard functions and classes set

Sometimes it is necessary to reduce set of standard functions/classes or replace it with custom function set.
To use custom functions/classes set instead of standard, create a **THtScriptGlobal** class descendant.
THtScriptGlobal contains several method for functions and classes registration accordingly to Delphi units:

```
  THtScriptGlobal = class(THtScriptParser)
  protected
    // TObject.class
    fTObject: TScriptClass;
    // TObject.Free method
    fFreeMethod: TScriptFunc;
    StringHelper, IntegerHelper, SingleHelper, DoubleHelper: pointer;
    procedure RegisterInternals; virtual;
    procedure RegisterOrdinalHelpers; virtual;
    procedure RegisterMagicFunctions; virtual;
    procedure RegisterHtFunctions; virtual;
    procedure RegisterMathFunctions; virtual;
    procedure RegisterSystemFunctions; virtual;
    procedure RegisterSysUtilsFunctions; virtual;
```

```
    procedure RegisterClassesFunctions; virtual;
    procedure RegisterDateUtilsFunctions; virtual;
  public
    procedure RegisterAll; virtual;
  end;
```

By overriding **RegisterAll** method you can control what functions/classes will be registered.
Default **RegisterAll** code:

```
fScriptGlobal := Self;
RegisterInternals;
RegisterOrdinalHelpers;
RegisterMagicFunctions;
RegisterSystemFunctions;
RegisterSysUtilsFunctions;
RegisterClassesFunctions;
RegisterMathFunctions;
RegisterDateUtilsFunctions;
RegisterHtFunctions;
```

functions/classes registered by **RegisterInternals** are required, so this method should be called by any T
RegisterOrdinalHelpers is optional, like other methods, but is necessary for ordinal type helpers to work.

To use new **THtScriptGlobal** descendant class, create an object instance of this class and set ScriptParser.**ScriptGlobal** property to this instance.

5 Expressions

TScriptExpression class is designed for evaluating expressions of any type, including function calling. This class can be used separately from TScriptParser class to evaluate expression and obtain result without creating ScriptParser instance.

5.1 Expression evaluation

For expressions without parameters simply call class function **TScriptExpression.Evaluate**. Example:

```
 t := TScriptExpression.Evaluate('Now() + 1');
```

For expression that contain variables, TScriptExpression instance should be created:

```
E := TScriptExpression.CreateandParse('s + IntToStr(t)');
E['t'] := 1;
E['s] := 'test';
a := E.Calc;
```

5.2 Passing parameters

Use TScriptExpression.Variables property to set expression variable value.
Variables object can be accessed via **Expression.Parser.Vars** property.

5.3 Using custom variables getter/setter

In some cases it is useful to have custom getter and setter for expression variables. For example, expression is used to calculate field value in Dataset, and all variables should be mapped to Dataset fields:

```
procedure GetDatasetVar(const V: TExVariable);
var F: TField;
begin
  F := Dataset.FindField(V.Name);
  if Assigned(F) then
    V.fValue := F.AsVariant
end;

procedure SetDatasetVar(const V: TExVariable);
var F: TField;
begin
  F := Dataset.FindField(V.Name);
  if Assigned(F) then
    F.Value := V.fValue;
```

```
end;

E := TScriptExpression.CreateandParse('MyField1:=MyField2+MyField3');
E.OnGetVar := GetDatasetVar;
E.OnSetVar := SetDatasetVar;
E.Calc;
```

5.4 Evaluating expression inside script

Similarly to javascript **eval** function, library supports runtime expression evaluation. Example:

```
t := 100;
k := 200;
Result := eval('t+k');
```

6 Executing script

6.1 Executing script

To execute script main block call **TScriptParser.Run**.
Example:

```
uses htscriptgui;

  SP := THtScriptParser.Create('ShowMessage(''test'');');
  SP.Run;
```

6.2 Accesing global variables

To get or set value of global script variable use **TScriptParser.Variables** (or **TScriptParser.Objects** for objects) property.
Example:

```
t := Script.Variables['myvar'];
Script.Variables['myvar'] := 100;
```

To check is variable already registered and get variable object use **TScriptParser.Vars** property.

Is variable registered:

```
if Script.Vars.GetVariable('MyVar') then ...
```

Enumerate all variables:

```
for i := Script.Vars.Count - 1 do
  List.Add(Script.Vars[i].Name);
```

6.3 Calling script function

Any script function can be called from Delphi code.
Example:

```
t := Script.RunFunction('MyFunc', [1, 'test']);
```

For faster calling you can get reference to a function and call it via reference:

```
var SF: TScriptFunc;
begin
```

```
  SF := Script.FindFunction('MyFunc');
  if Assigned(SF) then
    t := Script.RunFunction(SF, [1, 'test'])
  else
   ShowMessage('Function not found');
end;
```

6.4 Calling script function with var (out) parameters

To call script function with var or out parameters use RunVarFunction method.
Example:

```
var SP: THtScriptParser;
    PA: array of variant;
begin
  SP := THtScriptParser.Create('function x(var a: integer): string; begin a:=a+100; Re
  SP.Parse;
  SetLength(PA, 1);
  PA[0] := 100;
  SP.RunVarFunction('x', PA);
  ShowMessage(PA[0]);
```

For faster calling you can get reference to function and call it via reference:

```
  SF := Script.FindFunction('x');
  if Assigned(SF) then
    t := Script.RunVarFunction(SF, PA)
```

6.5 Executing code block

It is possible to execute additional script (code block) in context of current script without changing and recompiling original script.
At first step the script is converted into anonymous function and then executed.

Example:

```
var SF: TScriptFunc;
begin
  SF := Script.CreateAnonymousFunction('MyGlobalVar := MyScriptFunc(100, ''test'')');
  Script.RunFunction(SF, [])
end;

Code block can contain function with parameters.
Example:

var SF: TScriptFunc;
begin
```

```
  SF := Script.CreateAnonymousFunction('function(p: integer; begin MyGlobalVar := MySc
  Script.RunFunction(SF, [100]);
end;
```

6.6 Predefined variables

Following global variables are defined by script engine:

- **Scripter**: THtScriptParser - ScriptParser instance
- **Console**: THtConsole - debug console

Custom global variables can be defined using **HtScriptGlobal** class, f.e.

```
HtScriptGlobal.Objects['application'] := Application;
HtScriptGlobal.Variables['appname'] := 'My Application';
```

6.7 Executing script from script

Script can create and run another scripter instance.
Example:

```
P := THtScriptParser.Create('Result:=x+100;');
P.Variables['x']:=100;
P.Run();
ShowMessage(P.Variables['Result']);
```

6.8 Units/Uses

Script can be divided into separate units. To use external units in script create THtScriptParser.**OnGetUnit** event handler.
Example:

```
uses htutils;

function TForm1.OnGetUnit(Sender: THtScriptParser; const UnitName: string;
  var UnitText: string): boolean;
begin
  Result := false;
 if FileExists(ScriptsPath + UnitName + '.pas') then
  begin
    UnitText := FiletoStr(ScriptsPath + UnitName + '.pas');
    Result := true
  end;
end;
```

To use this event handler for all Scripter instances assign it to **HtScriptGlobal.OnGetUnit**.

6.9 Error address and callstack

When exception is raised during script execution, call stack and source line are saved into Parser.ExceptionStack and ExceptionLine properties.
Exceptions raised by scripter engine itself are of EHtScriptException type and contains the following fields:

- **Sender**: THtScriptParser - Scripter
- **SourcePos**: integer - absolute position in source code
- **Line**: integer - line in source code
- **ScriptUnit**: string - name of unit (empty for main program).

7 Debugging

7.1 ScriptDebugger class

For debugging a script it is necessary to implement a THtScriptDebugger abstract class. It has following methods:

```
THtScriptDebugger = class
public

  /// Executed before script run
  procedure OnRun(Sender: THtScriptParser); virtual;

  /// Executed after script run
  procedure OnStop(Sender: THtScriptParser); virtual;

  /// Executed when script is paused
  procedure OnPause(Sender: THtScriptParser); virtual;

  /// Executed when script continue running
  procedure OnResume(Sender: THtScriptParser); virtual;

  /// Used in Paused mode to process application messages. Simply call Application.
  procedure ProcessMessages(Sender: THtScriptParser); virtual;

  /// Triggered by Console.Log method in script
  procedure OnConsoleLog(MessageType: THtConsoleMessageType; const s: string); virtu
end;
```

After implementing a class set **HtScriptDebuggerGlobal** global variable.

7.2 Conrolling script execution

Following methods can be used to control script execution:

```
  /// Execute scipt to next line without tracing into functions
  procedure StepOver;

  /// Execute scipt to next line with tracing into functions
  procedure TraceInto;

  /// Stop script execution on ALine line.
  procedure RuntoCursor(ALine: integer);

  /// Stop script execution on ALine line.
  procedure RunUntilReturn;

  /// Continue script execution if script was paused.
  procedure Continue;
```

```
/// Break execution
procedure Halt;
```

7.3 Getting variables

To get variable value in current context *(taking into account current function stack),* for example, to display variable value in debug watch window, use

```
function GetDebugVariable(const Name: string): TExVariable;
```

7.4 Console and logging

Console object (similarly to Javascript console https://developer.mozilla.org/en/docs/Web/API/Console) is intended for logging and profiling purposes.
Following **Console** object methods are available:

Log(Message: string) - output message to console window
Info(Message: string) - output info message to console window
Warn(Message: string) - output warning message to console window
Error(Message: string) - output error message to console window
Assert(Condition: boolean; Message: string) - check condition and output message into console window (also exception is raised).

7.5 Profiling

Script execution can be profiled using Console **Time** and **TimeEnd** methods.
Call **Time** before profiled code and **TimeEnd** after, with the same text key.
Example:

```
Console.Time('MyExecution time is');
MyFunction();
Console.TimeEnd('MyExecution time is');
```

Result will be passed to the script debugger object via **OnConsoleLog** method.

7.6 Breakpoints

Use following methods to manage script breakpoints:

```
procedure AddBreakpoint(ALine: integer; APassCount : integer = 0; const ACondition:
procedure RemoveBreakpoint(ALine: integer);
procedure ToggleBreakpoint(ALine: integer);
function HasBreakpointAt(ALine: integer): boolean;
```

7.7 Expression evaluation

To evaluate exppression in a current context, use Evaluate function.
Example:

```
s := Script.Evaluate('inttostr(a)');
```

8 Using script in HTML document

8.1 Introduction

To use scripting library in HTML documents add **htdefscriptadapter** unit to uses list.
Scripts are defined in HTML document similarly to standard Javascript scripts, but with type attribute set to **passcript** or **text/passcript.**
Example:

```
<script type="passcript">
 procedure Test();
 begin
  document.getElementbyId('myid').innerHTML := 'test';
 end;
</script>
```

Note, that <script> tag with type set to passcript is necessary event if its content is empty, otherwise script adapter will not process elements and document events. This is done to prevent attempts of executing pascal script on normal HTML documents containing Javascript in elements events.

Predefined variables

Following variables are defined when script in running inside an HTML document:

- **document**: THtDocument - current document
- **window** : TControl - control that owns document (for example THtPanel)
- **this**: TElement - current element
- **event**: THtMouseEvent - processed event
- **dragged**: TElement - dragged element (in drag events)
- **form**: TForm - current form (if script is running inside HTML control placed on a form)

Custom variables

Any object variable can be registered in script adapter for use in script:

```
HtPanel.Doc.ScriptAdapter.RegisterObject(const Name: string; Value: TObject);
```

8.2 JQuery support

Subset of JQuery functionality is supported via $() function when script is executed inside HTML Document.
Syntax:
$(Selector, [Context]).Operation

Selector is any valid CSS selector, for example

```
$('#id') - elements with id="id".
$('.myclass') - elements having CSS class myclass.
$('div') - all div elemets.
```

Context is optional context element. If context is passed, JQuery will start searching from this element.
Result is list of found elements. Operation is executed on each element in the list.
Some of operations returns element list to allow chaining.

Following operations are available:

```
property Attr[Name]
```
Get or set elements attribute value. When list contains several elements result is list of attribute values separated by comma.

```
property HTML: string
```
Get (for first element) or set inner HTML.

```
property CSS[name: string]
```
Add CSS value.

```
function Each(Proc: procedure(Element): THtNodeList
```
Execute procedure Proc on each element.

```
function AddClass(ClassName: string): THtNodeList
```
Add CSS class

```
function RemoveClass(ClassName: string): THtNodeList
```
Remove CSS class

```
function ToggleClass(ClassName: string): THtNodeList
```
Toggle CSS class

```
function On(EventName: string; Code: string): THtNodeList
```
Set event handler for event EventName

```
function First: THtNodeList
      return list containing first element only

function Last: THtNodeList
      return list containing last element only

function Parent: THtNodeList
      return list containing parent elements of all elements

function Children: THtNodeList;
```

```
      return list containing chilren of all elements

procedure Sort(Compare: function(E1, E2: TElement): integer, Asc: boolean)
```
sort list using Compare function

Example

Make rows of table #table selectable (user can click on row to select or deselect it)

```
$('#table tbody tr').On('click', 'this.ToggleClass(''selected'')');
```

8.3 Events

Supported element events:

onmousemove - mouse is moved over an element
onclick - element is clicked
ondblclick - element is double-clicked
onmouseover - mouse enters an element
onmouseout - mouse leaves an element
onmousedown - left button pressed
onmouseup - left button released
onchange - for input elements
onblur - element lose focus (for input element)
onscroll - element was scrolled
onresize - element size is changing
onresizeend - element size was changed

Events fired on the draggable target (the source element):

ondragstart - occurs when the user starts to drag an element
ondrag - occurs when an element is being dragged
ondragend - occurs when the user has finished dragging the element

Events fired on the drop target:

ondragenter - mouse enters an element while dragging
ondragleave - mouse leaves and element while dragging
ondrop - other element is dropped on element

Example

Set div semitransparent when other element is dragged over it.

```
$('mydiv').on('dragenter', 'if Event.Target.tag=''div'' then Event.Target.css(''opacit
$('mydiv').on('dragleave', 'if Event.Target.tag=''div'' then Event.Target.css(''opacit
```

8.4 AJAX

** HTML Report Library is required*

Script inside HTML page can execute HTML Report using RunHtReport function.

```
function RunHTReport(Report, ContexXML: string): string;
```

Report parameter contains report text and ContextXML can be used for passing additional parameters to report.
Example:

```
s:='<report-objects><object name="cust" '+
 'sql="select * from customer where upper(company) like ''%%{{SEARCH}}%%'' order by
  '</report-objects>'+
'{{#cust.ROWDATA}}'+
'<p><i class="fa fa-user"/> <a href="#">{{COMPANY}}</a></p>'+
'{{/cust.ROWDATA}}';

Res := RunHtReport(s, '<CONTEXT SEARCH="ab"/>');
```

8.5 Interactive hints

When script is running inside **HtPanel**, it is possible to show interactive hint window with HTML document.
Example:

```
<script type="passcript">

procedure showhint;
begin
  document.control.ShowFloatHint(this, '<h3>This is hint</h3>', true);
end;

</script>

<a onclick="showhint()">Click me</a>

function ShowFloatHint(Element: TElement; Hint: string; Animated: boolean);
```

Element: hint element - hint will be hidden when mouse leaves this element.
Hint: hint text.

Animated - animated show.

8.6 Usage examples

8.6.1 Make table sortable

```
<script type="passcript">
$('#tableid th').on('click', 'this.upto(''table'').SortColumn:=this.Col;document.repa
</script>

<table id="tableid">
 <tr><th>header</th></tr>
 <tr><td>3</td></tr>
 <tr><td>2</td></tr>
 <tr><td>1</td></tr>
</table>
```

8.6.2 Highlight list items starting with 'A'

```
procedure TestA(E: TElement);
begin
  if AnsiStartsWith(E.Attr['name'], 'A') then
    E.css('font-weight: bold;');
end;

procedure Highlight();
begin
  $('#ul li').Each(@TestA);
end;
```

8.6.3 Convert nested list into expandable tree

```
<style>
li>ul {display: none}
li.show>ul {display: block}
</style>

<script type="passcript">

 $('li a').on('click', 'this.parent.toggleClass(''show'')');

</script>

<ul>
 <li><a href="#">Item1</a>
  <ul>
    <li>Subitem1</li>
```

```
    <li><a href="#">Subitem2</a>
      <ul>
        <li>Sub-Sub-Item1</li>
      </ul>
    </li>
    <li>Subitem3</li>
  </ul>
 </li>
</ul>
```

8.6.4 Directory tree with background loading

**HTML Report Library is required.*

```
<style>
 body {font-family: Verdana; font-size: 14px}
 ul {list-style-type: none; transition: height 0.3s;
  overflow: hidden}
 a {text-decoration: none}
 a:hover {text-decoration: underline}
 li>ul {height: 0px}
 li[type="DIRECTORY"]>img {display: none}
 li.show>ul {height: auto}
 li {padding: 3px 3px}
 .fa {color: green}
</style>

<script type="passcript">

procedure ShowFiles();
begin
 { Disable onclick event for parent li nodes}
 if Assigned(Event) then
   event.StopPropagation();
 { file or processed directory }
 if this.hasclass('processed') or
  (this.Attr['type'] <> 'DIRECTORY') then
 begin
   { open / close }
   if this.Attr['type'] = 'DIRECTORY' then
     this.toggleClass('show');
   { update folder icons }
   $('li[type="DIRECTORY"]>i').removeClass('fa-folder-open-o').
     addClass('fa').addClass('fa-folder-o');
   $('li.show>i').removeClass('fa-folder-o').
     addClass('fa-folder-open-o');
   document.Refresh();
   exit;
 end;
 { Report code }
 s := '<report-objects><object name="files" type="directory" '+
```

```
        ' sql="{{DIR}}\*.*"/></report-objects>'+
    '<ul>'+
    '{{#files.ROWDATA}}'+
    '<li dirname="{{PATH}}{{NAME}}" type="{{FILETYPE}}">'+
    '<i/> <img src="_shellsmallicons/{{EXT}}"> <a href="#">{{NAME}}</a></li>'+
    '{{/files.ROWDATA}}'+
    '</ul>';

   Async(
    function(E: TElement): string;
    begin
     Result := RunHtReport(s, '<CONTEXT DIR="'+E.attr['dirname']+'"/>');
    end,
    procedure(E: TElement; s: string);
    begin
      E.innerhtml := E.innerhtml + s;
      $('li').on('click', 'ShowFiles()');
      E.addClass('processed');
      document.Refresh();
      $('li[type="DIRECTORY"]>i').addClass('fa').
        addClass('fa-folder-o');
      E.addClass('show');
      $('li.show>i').removeClass('fa-folder-o').
         addClass('fa-folder-open-o');
      { we need second Refresh for open/close animation.
        First refresh calculate zero <ul> height, second
       (after setting .show class) calculates 'auto' height and starts transition }
      document.Refresh();
    end,
    [this]
  );
end;

{ set onclick handler for list items }
$('li').on('click', 'ShowFiles()');

this := document.getElementbyId('root');
ShowFiles();

</script>

<ul>
 <li id ="root" dirname="c:" type="DIRECTORY"><i/> <a>Root</a></li>
</ul>
```

8.6.5 Incremental search

**HTML Report Library is required.*
When edit is changed, **SetTimeout** function adds **ShowCustomers** to queue with 500ms delay. Next change of the edit, reset queued function and starts new.
ShowCustomers executes report via **RunHtReport** and passes result to **#cust** div element.

```
<style>
 a {text-decoration: none}
 a:hover {text-decoration: underline}
</style>

<script type="passcript">

procedure showcustomers();
begin
 { Report code }
 s:='<report-objects><object name="cust" '+
   'sql="select * from customer where upper(company) like ''%%{{SEARCH}}%%'' order by
    '</report-objects>'+
  '{{#cust.ROWDATA}}'+
  '<p><i class="fa fa-user"/> <a href="#">{{COMPANY}}</a></p>'+
  '{{/cust.ROWDATA}}';

  Async(
   function(value: string): string;
   begin
    Result := RunHtReport(s, '<CONTEXT SEARCH="' + AnsiUpperCase(value) + '"/>');
   end,
   procedure(value, s: string);
   begin
    $('#cust').html := s;
    document.refresh();
  end,
  this.value
 );
end;

</script>

<input type="text" onchange="if Assigned(custsearch) then ClearInterval(custsearch);
   custsearch:=SetTimeout(@showcustomers, 300);">
<div id="cust"></div>
```

8.6.6 Infinite page

New portion is loaded when page is scrolled to bottom.

```
<style>
 body {font-family: Verdana; font-size: 10px}
 .fa {color: orange}
 .item {padding: 5px 5px}
</style>

<script type="passcript">

procedure showcustomers();
begin
 { Report code }
 s := '<report-objects><object name="cust" sql="select * from customer"/></report-obje
```

```
    '{{#cust.ROWDATA}}'+
    '<div class="item"><i class="fa fa-user"/> {{COMPANY}}</div>'+
    '{{/cust.ROWDATA}}<div id="cust"></div>';
    s1 := RunHtReport(s, '<CONTEXT/>');
    $('#cust').html := s1;
    { clear id for old placeholder }
    $('#cust').first.Attr['id'] := '';
    document.refresh();
end;

procedure OnScroll();
begin
  if document.innerheight - document.scrolltop - 50 < window.innerheight then
    showcustomers();
end;

document.AddEventListener('scroll', @onscroll);

</script>

<a onclick="showcustomers()">Show Customers</a>
<div id="cust"></div>
```

8.6.7 Calling script function from Delphi

Res := (HtPanel1.Doc.ScriptAdapter as THtDefScriptAdapter).SP.RunFunctionIfExists('MyScriptFunction', [Param1]);

9 Standard functions

Magic functions

```
Async(Proc, After: procedure);
Decode(value1, result1, [valueN, resultN], valueElse: variant): variant;
ExceptionMessage() : string;
Iif(condition: boolean; IfTrue, IfFalse: variant): variant;
IfThen(condition: boolean; IfTrue, IfFalse: variant): variant;
InRange(Value: variant; Min, Max: variant): boolean;
ReleaseObject(A: TObject): TObject;
StrIn(s: string; v1, v2, [..vN] : string): boolean;
SetLength(Value: string or array; Length: integer);
SetTimeout(Proc: procedure; Timeout: integer);
```

System unit

```
Assigned(var v): boolean;
Abs(X: double): double
ArcTan(X: Extended): Extended
Cos(X: Extended): Extended
Char(c: integer): char
Copy(s: string; Index, Count: integer): string
Dec(var value: integer; decrement: integer = 1);
Delete(var s: string; Index, count: integer);
Exp(X: Extended): Extended
Frac(X: Extended): Extended
High(A: array): integer;
Inc(var value: integer; increment: integer = 1);
Insert(Substring: string; var Target: string; Index: integer);
Length(s: string): integer
Ln(X: Extended): Extended
MkDir(s: string)
Odd(X: integer): boolean
Pred(x: integer): integer
ParamCount()
ParamStr(Index: integer): string
Pos(const Substr, Str: string): integer
PosEx(const Substr, Str: string; Offset: integer): integer
Randomize()
Random(ARange: integer): integer', @_Random);
RmDir(s: string)
Round(X: double): integer
Sqr(X: double): double
Sqrt(X: Extended): Extended;', @Sqrt);
Sin(X: Extended): Extended
Succ(x: integer): integer
Swap(x: integer): integer
```

```
Sleep(x: integer)
Tangent(X: Extended): Extended
Trunc(X: double): integer
UTF8Encode(s: string): ansistring
```

Windows unit

```
GetTickCount(): integer
```

SysUtils unit

```
Abort()
AnsiLowerCase(s: string): string
AnsiSameStr(s1, s2: string): boolean
AnsiSameText(s1, s2: string): boolean
AnsiUpperCase(s: string): string
AnsiQuotedStr(s: string): string
ChangeFileExt(FileName, Extension: string): string
CompareText(s1, s2: string): integer
CurrentYear: integer
DirectoryExists(Path: string): boolean
Date(): TDateTime
DayOfWeek(Date: TDateTime): integer
DeleteFile(FileName: string): boolean
EncodeDate(Year, Month, Day: integer): TDateTime
EncodeTime(Hour, Min, Sec, s100: integer): TDateTime
ExtractFilePath(FileName: string): string
ExtractFileName(FileName: string): string
ExtractFileExt(FileName: string): string
FileExists(FileName: string): boolean
FileCreate(FileName: string): integer
FileOpen(FileName: string; Mode: integer): integer
FileClose(Handle: integer)
FileWrite(Handle, Buffer, Count: integer): integer
ForceDirectories(Dir: string)
Format(s: string; Param: array of const): string;
FormatDateTime(Format: string; Value: TDateTime): string
FormatFloat(Format: string; Value: extended): string
FloatToStr(Value: extended): string
FloatToStrF(Value: Extended; Format: integer; Precision, Digits: Integer): string
IncMonth(Date: TDateTime; NumberOfMonths: Integer): TDateTime
IntToStr(n: integer): string
IntToStr64(n: cardinal): string
IntToHex(Value: Integer; Digits: Integer): string;
LowerCase(s: string): string
Now(): TDateTime
QuotedStr(s: string): string
RenameFile(OldName, NewName: string): Boolean
ReplaceStr(S, OldPattern, NewPattern: string): string
StringReplace(S, OldPattern, NewPattern: string): string
StrToInt(s: string): integer
```

```
StrToIntDef(S: string; Default: Integer): Integer
StrToFloat(S: string): Extended;
StrToFloatDef(S: string; Default: Extended): Extended;
StrToBool(s: string): boolean
StrToTime(s: string): TDateTime
StrToDate(s: string): TDateTime
StrToDateTime(s: string): TDateTime
Trim(s: string): string
TrimLeft(s: string): string
TrimRight(s: string): string
TimetoStr(t: TDateTime): string
SetCurrentDir(Dir: string): Boolean
SameStr(s1, s2: string): boolean
SameText(s1, s2: string): boolean
UpperCase(s: string): string
```

DateUtils unit

```
YearOf(AValue: TDateTime): Word
MonthOf(AValue: TDateTime): Word
WeekOf(AValue: TDateTime): Word
DayOf(AValue: TDateTime): Word
HourOf(AValue: TDateTime): Word
MinuteOf(AValue: TDateTime): Word
SecondOf(AValue: TDateTime): Word
MillisecondOf(AValue: TDateTime): Word;
StartOfTheYear(const AValue: TDateTime): TDateTime
EndOfTheYear(const AValue: TDateTime): TDateTime
StartOfTheMonth(const AValue: TDateTime): TDateTime
EndOfTheMonth(const AValue: TDateTime): TDateTime
StartOfTheWeek(const AValue: TDateTime): TDateTime
EndOfTheWeek(const AValue: TDateTime): TDateTime
```

Math unit

```
Ceil(X: single): integer
Floor(X: single): integer
Power(Base, Exponent: Extended): Extended;
Sign(AValue: double): integer;
```

HTML functions

```
AnsiStartsWith(s, start: string): boolean
AnsiEndsWith(s, start: string): boolean
StartsWith(s, start: string): boolean
EndsWith(s, start: string): boolean
FindChar(c: char; s: string; Start: integer = 1): integer
BlankString(s: string): boolean
CalcStrCrc32(s: string): cardinal
HTMLEncode(s: string; NewLength : integer = -1): string
HTMLEncodeAttr(s: string; NewLength : integer = -1): string
```

```
HTMLColortoHex(Color: cardinal): string
HTMLColortoStr(Color: cardinal): string
iStrToFloat(s: string): single
ObjecttoXML(O: TObject; ClassProp: boolean = false; Node: THtXMLNode = nil; UpperCas
StrToDateFmt(s: string; format : string =''dd.mm.yyyy hh:nn:ss''): TDateTime
```

10 Standard constants

```
MaxInt
MaxWord
MaxCurrency
Pi
```

11 Standard classes

TObject
Exception
TStrings
TStringList
TList
TBits
TCollection
TComponent
THtXMLNode
THtInetClient

htscriptgui unit
TOpenDialog
TSaveDialog
TFileOpenDialog
TFileSaveDialog
TFindDialog
TForm
TButton
TLabel
TGroupBox
TMemo
TComboBox
TCheckBox
TRadioButton
TListBox
TShape
TImage
TTimer
TPanel
TSplitter
TLabeledEdit
TButtonedEdit