Applied DelphiWebScript II Programming

DelphiWebScript Community

6th September 2003

to all members of the DWS community

This is **version 1.0** of *Applied DelphiWebScript II Programming*.

Please understand that this document still lacks serious proof-reading. Be aware that the main author of the original version of this document is no native English speaker. If you find grammar mistakes, mistakenly used words, bad style or other *bad things*, please contact him at `mailto:willi@dwscript.com`.

If you think that a specific topic you can not find should definitely go into this document, we strongly encourage you to drop us a note. (Use the forum at our sourceforge page!)

This document was created with MiKTEX and LYX on Windows. Several LATEX packages have been used in order to create this document: *framed*, *eso-pic*, *color*, *rotating*, *array*, *listings* and *hyperref*.

Date	Version of Document
09/06/2003	RC3 -> version 1.0
08/25/2003	Some layout changes.
08/17/2003	Added DBDemo doc contributed by Yeoh Ray Mond
08/12/2003	• Added HowTos contributed by Yeoh Ray Mond • Added HowTo for emulating dynamic arrays • Bugfixed TWindow class • Added StringsUnit description • Added LeftStr, RightStr and MidStr
08/11/2003	Changes contributed by Yeoh Ray Mond
08/08/2003	Release Candidate 1

Still to do:

- add explanations to COMConnector
- Finish section 2.2 -> Explain IInfo

A special "Thank you" goes out to

Yeoh Ray Mond for providing lots of corrections and helpful suggestions

Foreword

Applied DelphiWebScript II Programming aims to provide you with an in-depth coverage of the *DWS* component, the intentions we had when creating the interpreter, some outlook on future developments and of course it will provide you with the necessary examples and hints for building professional *DWS* driven applications - regardless if you are developing web applications or "old style" GUI applications.

This document has been brought to you by long standing members of the community and some of the most active *DWS* developers. We hope that it'll answer all questions you possibly might have regarding *DWS* and that it is able to remove any misconceptions that might have arisen over time.
Most probably the name Delphi*WebScript* itself is already source for such misconceptions, as it implies that *DWS* is for web scripting only. This is, however, not the case at all. In fact *DWS* is a Delphi component that takes some script (written in a subset of Delphi), compiles it into an abstract syntax tree and then runs the script. The reason for the "WebScript" in the name is the ability of DWS to run any kind of input through a so called filter before starting the compile cycle. The filter is responsible for "morphing" the input into a valid DWS script the compiler expects. Currently one filter ships with the DWS distribution: *HTMLFilter*.

The *HTMLFilter* expects a text file as input and does nothing else but wrap a "print" command around the text found in the file. Of course this description is an over simplification, as *HTMLFilter* allows you to insert DWS scripts within special marked blocks in the text file, but you get the basic idea.

Back in the year 2000 (you remember the Internet hype?), Delphi 5 was lacking some convenient way of producing server side scriptable HTML pages. Delphi only featured the well known TPageProducer. One main disadvantage of this component was that the CGI or ISAPI module had to be recompiled every time a change in the application logic (i.e. within the events of the page producer component) was made. DI Hannes Hernler by that time came across the *TScript* component written by Matthias Ackermann and saw the potential of a modified *TScript* that could parse native HTML for his web applications.

To make a long story short: Matthias Ackermann delivered a modified *TScript* component and it was agreed to make this component widely available, thus making it open source under the MPL license. By that time it was also decided to rename the component to *DWS*, as the main future was seen in developing web applications.

Since then many things happened: The Internet hype vanished, DWS was re-written from scratch (DWS II) several research projects were made for DWS (Session Server, automatic wrapper generator, Package Connector, Debugger just to name a few) and even DWS3 was started in our CVS repository at sourceforge.

Today DWS II is available for Delphi 5, Delphi 6 and Delphi 7 (it shipped with companion CD of Delphi 6 and Delphi 7), Kylix 3 and Borland C++ Builder.

DWS would not have been possible without the work of many people from all over the world! So we would like to say a special “Thank You” to all who have contributed to *DWS*.

Thanks to...

- Andreas Luleich for countless compiler improvements
- Hannes Hernler for paying the web server bill
- Mark Ericksen for IDE Demos, several other tools and improvements in lots of areas.
- Matthias Ackermann for bringing TScript, DWS and DWS II 1.0 to life!
- All other contributors (in alphabetical order):
  - Brink, Danie (South Africa)
  - Darling, Jeremy
  - Egorov, Nikita (Russia)
  - Fuchs, Manfred (Germany)
  - Grange, Eric (France)
  - Hariseno, Jagad (Hungary)
  - Krenn, Willibald (Austria)
  - Lind, Martin (Denmark)
  - Listac, John
  - Rheinheimer, Danilo Luiz (Brazil)
  - Riepp, Michael (Germany)
  - Ronzano, Juan Luis (Portugal)
  - Tosik
  - van Leijen, Wilbert (Netherlands)
  - Waldenburg, Martin (Germany)
  - Wilcox, Ken

Contents

Part I.

A Quick Introduction to DWS II

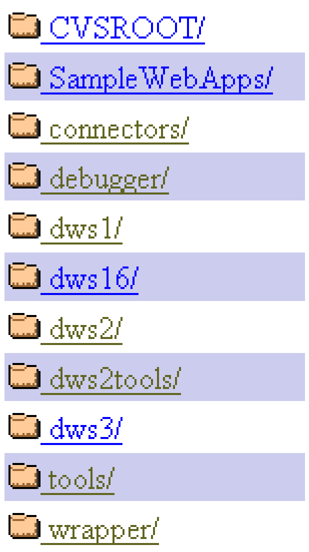
1. Installation

This chapter shows how to set up your development environment and will discuss the modules DWS II is broken into. Note that the information presented here is subject to change in future releases. It is therefore a good practice, to read the “Readme” file that comes with every DWS distribution package.

This chapter won’t cover the automatic installer that might have shipped with DWS II. Instead it’ll explain the modules we have in our CVS repository and how to install these modules by hand.

Some of the modules are not included in the DWS II distribution package because of their unfinished state.

1.1. Directory Structure



CVSROOT/
SampleWebApps/
connectors/
debugger/
dws1/
dws16/
dws2/
dws2tools/
dws3/
tools/
wrapper/

Figure 1.1.: DWS root in CVS

Browsing our CVS repository reveals the module listing from above. Although this document deals with DWS II only, we’ll briefly go over all non-dws2 modules too.

The “*SampleWebApps*”, “*connectors*”, “*dws3*” and “*tools*” modules are deprecated - respectively currently not actively maintained. *SampleWebApps* was meant to hold a few sample web applications that were developed using DWS. Unfortunately this goal was never reached. The *connectors* module is a reminiscent of some feature we had envisioned for DWS. Today the COM connector - which is the one and only connector that got implemented - resides in the *dws2/Source* directory. In the *dws1* module you can find the first version of DWS that lacked OOP, but had support for function overloading implemented. This module is not actively maintained anymore, as is the *dws16* module, that contains the 1.6 release of DWS. (This release was checked in to a separate module to ease checkout for developers who are not familiar with CVS tags or branches.)

DWS3 holds a DWS version that should have become the next release after DWS II 1.0. It's based on the DWS II 1.0 source code but was enhanced to allow concurrent execution of the same TProgram object. To put it differently, DWS3 is thread safe. Please keep in mind that DWS3 is not production quality and more or less experimental. The DWS3 module also contained some preliminary version of the "PackageConnector"[1]. Currently there is no development going on in the DWS3 module. Finally, the *tools* and *wrapper* modules contain some deprecated code.

Today, the development takes place in the *dws2* and *dws2tools* modules. Figure 1.2 on the facing page shows the directory tree of the checked out dws2 module.

The most important subdirectory in the dws2 module is probably the "Source" directory. This directory contains the complete Delphi source code for the core DWS II components. If you want to use DWS II in your applications, you have to include this directory in your Delphi (or Kylix) library path! The various Delphi<version> and Kylix<version> subdirectories contain the runtime and design-time packages for DWS II.

Obviously a good starting point to explore the power of DWS II is the "Demos" subdirectory. Please note that you'll need to have synedit (a recent snapshot!) installed if you want to try them all out. Later on we'll examine the demos anyway, as they show how to use DWS.

Thereafter we'll briefly explore the supplied libraries and finally we'll cover the dws2tools module. You can see the directory structure of the dws2tools module on page 22.

1.2. Packages

DelphiWebScript II is separated into several runtime and design-time packages. This is not only demanded by Borland, but also necessary for developers who want to ship their projects split into runtime packages[2].

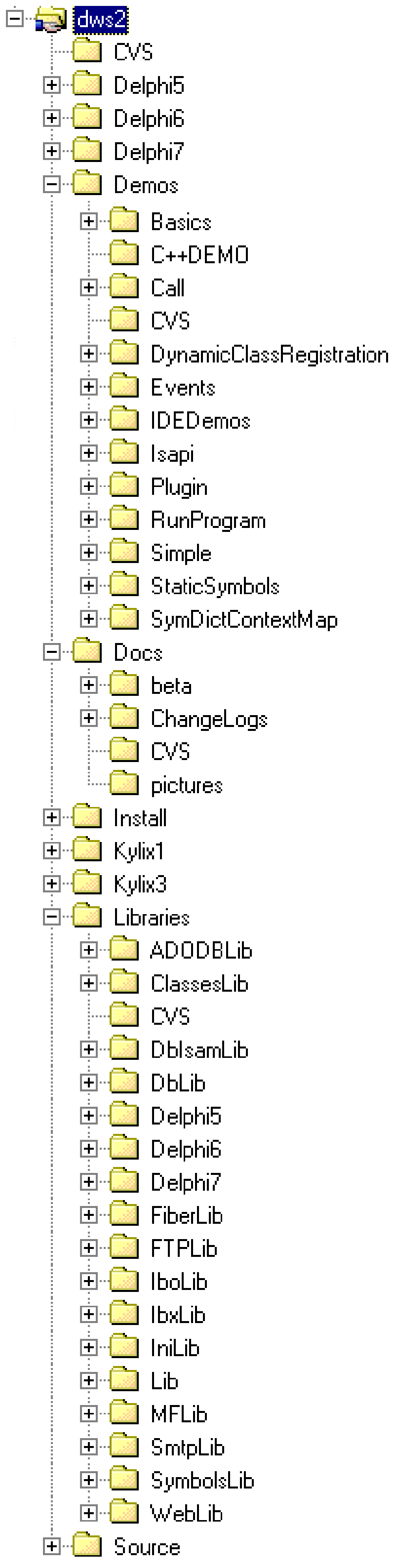
Basically DWS II is separated into:

[1]For the interested reader: Due to some difficulties with the DWS3 type system and the lack of interest in the DWS3 module itself, the PackageConnector was finally canned. The over-all idea behind the package connector was to be able to use native Delphi classes in any DWS script *without* the need to create wrapper code. This was achieved by letting the Delphi compiler do the wrapping - in other words: Let Delphi compile the code into runtime packages and call the methods found in the resulting runtime package.

The implementation got never beyond some proof of concept code. If you want to see some examples that already worked you might visit `http://www.countsandbarons.com/spass/SpassFrames.htm`

[2]We strongly suggest that readers who are not familiar with the package concept of Delphi should consult their favourite Delphi book and read up on that topic.

If anyone is interested in the name mangling schema of the Delphi/Kylix compilers (versions 5 & 6 - Delphi 7 adds something new if the mangled string is longer than 255 chars): `http://www.countsandbarons.com/spass/art/packages_en.htm`



```
dws2
  CVS
  Delphi5
  Delphi6
  Delphi7
  Demos
    Basics
    C++DEMO
    Call
    CVS
    DynamicClassRegistration
    Events
    IDEDemos
    Isapi
    Plugin
    RunProgram
    Simple
    StaticSymbols
    SymDictContextMap
  Docs
    beta
    ChangeLogs
    CVS
    pictures
  Install
  Kylix1
  Kylix3
  Libraries
    ADODBLib
    ClassesLib
    CVS
    DbIsamLib
    DbLib
    Delphi5
    Delphi6
    Delphi7
    FiberLib
    FTPLib
    IboLib
    IbxLib
    IniLib
    Lib
    MFLib
    SmtpLib
    SymbolsLib
    WebLib
  Source
```

Figure 1.2.: ”dws2” module, directory layout

- One design-time package, that is only needed by developers working with DWS II in the Delphi IDE,
- one general runtime package, that contains all non-GUI DWS II classes (all the core classes fall in this category),
- one VCL runtime package, that depends on the general runtime package and provides functions that rely on the VCL
- and last but not least one CLX runtime package, that depends on the general runtime package and provides the same functionality found in the VCL runtime package.

Depending on the Delphi version you have, you can not compile all runtime packages mentioned above. Delphi 5 (and all versions below) does not know anything about the CLX, so it's senseless trying to compile the CLX runtime. Kylix on the other hand does not ship with the VisualComponentLibrary, so you can't compile the VCL runtime.

NOTE | Note, that the Delphi Personal editions do not ship with CLX and might not even ship with all VCL classes, so there could be trouble compiling the VCL runtime.

Most of the libraries that ship with DWS II are not Linux compatible and do not have the design-time/runtime separation. That's because they only register one component in the Delphi IDE and do not depend on any Delphi IDE runtime package for IDE integration.

1.2.1. Common Runtime Package: *dws2Runtime.dpk*

Starting with the *dws2Runtime* package, we want to cover the contents of the packages and mention some versioning problems that might arise when using runtime packages.

Until Delphi 5 runtime packages had to be named along the pattern: <my-Product>50.dpk for Delphi 5, <my-Product>40.dpk for Delphi 4 and so on. This was necessary so that versions of the same runtime package compiled with different versions of Delphi could reside on the same computer.

Borland decided to "improve" upon this situation and from Kylix 1 / Delphi 6 on developers should use the LIBSUFFIX option to indicate the Delphi version a package was compiled with.

CONVENTION

The DWS community uses following convention for the LIBSUFFIX option:

- Delphi 6: *60*
- Delphi X: *X0*
- Kylix X: *X0*

Beginning with DWS II 2.0 we'll also set the LIBVERSION option:

- DWS II 2.0: *20*
- DWS II X.Y: *XY*

The Delphi 5 packages, which have no support for the options mentioned above, will be named like *dws2Runtime50_<DWSIIVersion>.dpk* (e.g. dws2Runtime50_20.dpk).

Following components are located in the common runtime package:

1. TDelphiWebScriptII
2. Tdws2HtmlFilter
3. Tdws2SimpleDebugger
4. Tdws2FileFunctions
5. Tdws2GlobalVarsFunctions
6. Tdws2StringResultType
7. Tdws2Unit
8. Tdws2HTMLUnit,
9. Tdws2StringsUnit

1.2.2. VCL Runtime Package: *dws2VCLRuntime.dpk*

As already mentioned, all VCL dependant classes are located in the VCL runtime package. This package provides following components:

1. dws2ComConnector
2. dws2VclGuiFunctions

HINT
You can only use these components when developing a VCL based application! Delphi will automatically hide all dws2 VCL based components if the currently active project is CLX based.

1.2.3. CLX Runtime Package: *dws2CLXRuntime.dpk*

The CLX runtime package is mostly considered for cross-platform development. Therefore no DWS II COM support was included in this package.

Currently, the only component that is contained in dws2CLXRuntime is the “dws2ClxGuiFunctions” component, that can be considered as a 1:1 copy of the VCL GUI functions component. Of course dws2ClxGuiFunctions bind to CLX instead of the VCL.

ATTENTION
It is not possible to use the COM connector in CLX based projects.

1.3. Installation of the Dws2 Core Components

Installation of DWS II by hand is pretty straight forward. First, locate the Delphi subdirectory matching your Delphi version in the dws2 directory. Load the dws2Runtime package and compile it. Next load the other runtime packages (VCL,CLX) and compile again.

Now you are ready to install the design-time package: Load the dcldws2.dpk package and install.

HINT

Do not forget to include the dws2/Source directory in your library path, so that Delphi can find all dws2 source files.
If you are using Linux, you might want to modify your LD_LIBRARY_PATH variable to point to the directory where all bpl*.so.* files were put by the Delphi compiler.
If it might happen, that you have to convert Delphi 5 projects that are using DWS to Delphi 6 (or 7), you should modify the Delphi.upg file found in your Delphi 6 (Delphi 7) "bin" directory and append following lines:

```
dws2Runtime50_<Version>=dws2Runtime
dws2VCLRuntime50_<Version>=dws2VCLRuntime
dws2CLXRuntime50_<Version>=dws2CLXRuntime

dcldws250=dcldws2
```

Your component palette should now have a tab labeled "DWS2". By selecting this tab, you should see all components from figure 1.4 on the following page.

CONVENTION

The component's background color indicates the type of the component:

- Yellow means *internal functions* (usually fast)
- Blue stands for *debugger*
- Orange means *filter*
- Gray indicates *result types* and *TDelphiWebScriptII*
- Pink stands for *dws2units*

Congratulations: By now you should have a working DWS2 installation running. Using this as a basis, we'll introduce different DWS II libraries and tools in the next few chapters. But before doing so, we'll examine the basic DWS II capabilities by looking at a few examples that run without additional libraries.

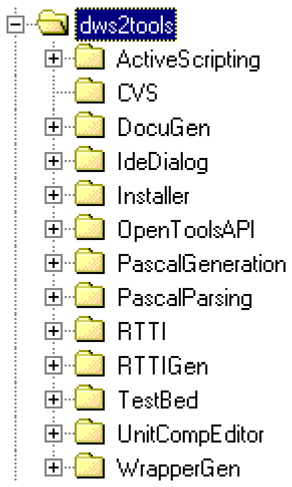


Figure 1.3.: "dws2tools" module - directory tree

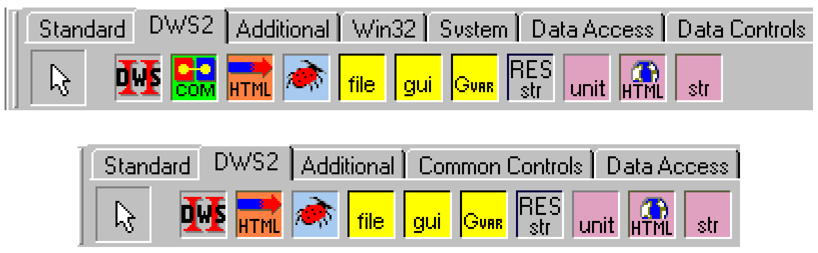


Figure 1.4.: dws2 core (above: VCL; below: CLX)

2. DWS II - the Basics

We've decided to cover a few easy examples that ship with DWS right here, so that you get an idea of what can be done with DWS.

2.1. SimpleDemo.dpr

In the "dws2/Demos/Simple" directory you'll find the SimpleDemo that demonstrates the minimum requirements DWS II needs in oder to compile and run scripts.

If you open the Delphi project, you'll see a main form that has two memos, one button and a TDelphiWebScript2 component dropped onto it. If you press F9 and then the "Compile&Execute" button, you'll get the output seen in figure 2.1.

By looking at the event handler source code of the TButton you'll see the minimum code that is needed for embedding the DWS II component.

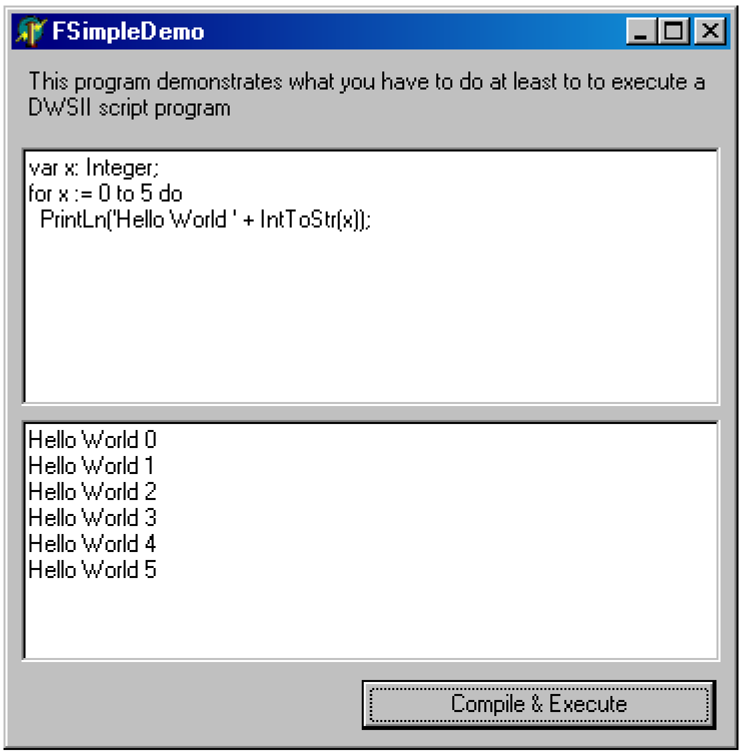
Listing 2.1: SimpleDemo EventHandler code

```

uses
  dws2Exprs, dws2Compiler;

procedure TFSimpleDemo.BNCompileAndExecuteClick(Sender:
    TObject);
var
  x: Integer;
  Prog: TProgram;
begin
  // Compile the script program
  Prog := DelphiWebScriptII1.Compile(MSource.Text);
  // Display error messages (if any)
  if Prog.Msgs.HasCompilerErrors then
    for x := 0 to Prog.Msgs.Count - 1 do
      ShowMessage(Prog.Msgs[x].AsInfo);
  // Execute the script program
  Prog.Execute;
  // Display the output
  MResult.Text := Tdws2DefaultResult(Prog.Result).Text;

  // Display error messages (if any)
  for x := 0 to Prog.Msgs.Count - 1 do
    ShowMessage(Prog.Msgs[x].AsInfo);
```



This program demonstrates what you have to do at least to to execute a DWSII script program

```
var x: Integer;
for x := 0 to 5 do
  PrintLn('Hello World ' + IntToStr(x));
```

```
Hello World 0
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
```

Figure 2.1.: SimpleDemo Output

```
end;
```

Several things in this example are worth to mention: If you look at the implementation's uses clause, you'll notice that two units (*dws2Exprs* and *dws2Compiler*) were included by hand. We need to include *dws2Exprs* because it defines the TProgram class the DWS II compiler returns. *Dws2Compiler* is necessary because of the *Tdws2DefaultResult* cast.

Also note the pattern we used to compile and execute the script! First we compile, then we check for errors and only if everything worked out OK we call the execute method of the script. Please keep in mind that a script could cause different runtime errors! So we check for them too.

HINT

This example does not use any kind of *filter*, nor a custom *result type*.
If no custom result type is applied to the DWS II component, *TProgram.Result* always gives back a *Tdws2DefaultResult* object!
You can apply a custom DWS II result type by dropping e.g. the *dws2StringResultType* component onto the form that already contains a *TDelphiWebScriptII* component. Do not forget to set the Configuration.ResultType property of the TDelphiWebScriptII component to the newly added result type component!

If you drop a filter component onto a form *dws2Compiler* automatically gets added to the uses clause, so you only have to add the *dws2Exprs* unit. To enable the filter, you have to set the Configuration.Filter property of the TDelphiWebScriptII component to the newly added filter.

A filter / result type can only be assigned to one TDelphiWebScriptII component at a time!

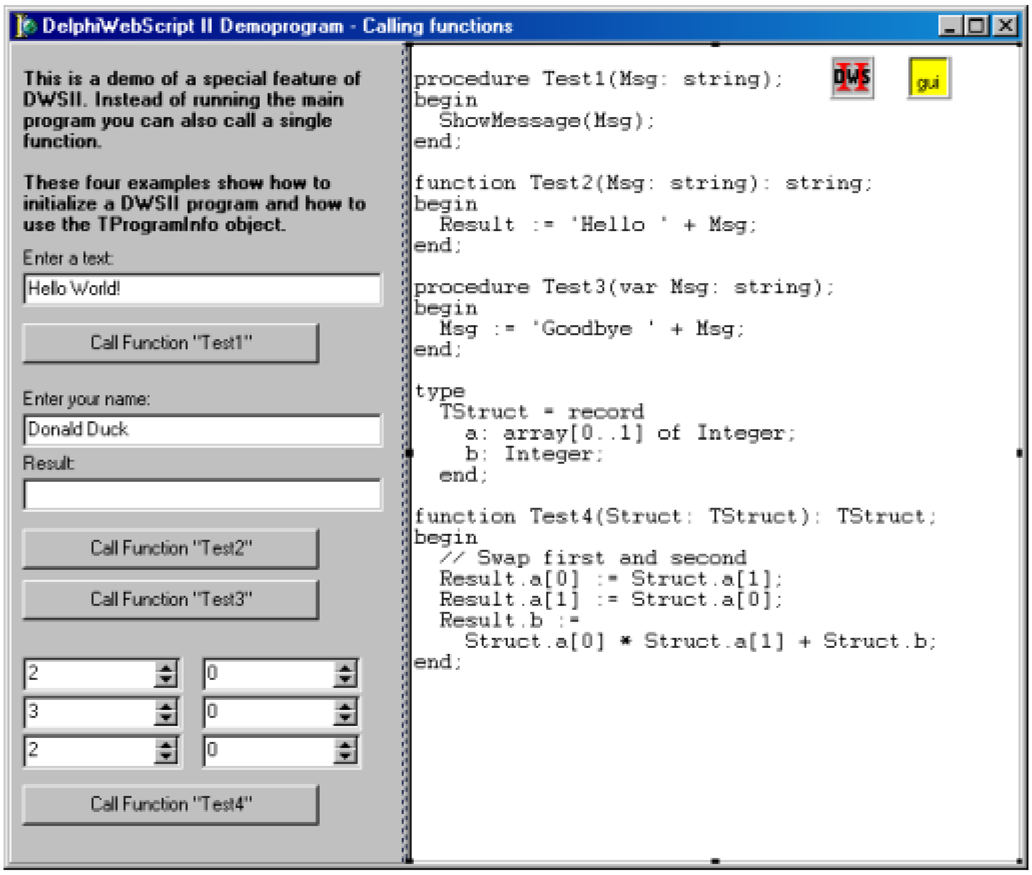
2.2. Calling Script Functions from Delphi: *Call.dpr*

We've now seen how to compile and execute a complete DWS script. Often, however, we only want to call a function in the script and do not want to execute the whole script. Indeed, it could be that the script does not have a ”main” program but only functions and procedures defined!

DWS offers you the flexibility you need: It's no problem calling a script function from Delphi code. To see how this works, load the “Call” demo that ships with DWS II 2.0. Figure 2.2 on the next page shows the design-time view of the demo's main form.

It's easy to see that the script on the right side of the form lacks a main program. It consists only of four functions/procedures and a type declaration.

The reason we need the *GUI* component here is the call to ShowMessage in procedure *Test1*. Without dropping the GUI component onto the form, DWS II would not be able to compile the



This is a demo of a special feature of DWSII. Instead of running the main program you can also call a single function.

These four examples show how to initialize a DWSII program and how to use the TProgramInfo object.

Enter a text: Hello World!

Call Function "Test1"

Enter your name: Donald Duck

Result:

Call Function "Test2"

Call Function "Test3"

2	0
3	0
2	0

Call Function "Test4"

```
procedure Test1(Msg: string);
begin
  ShowMessage(Msg);
end;

function Test2(Msg: string): string;
begin
  Result := 'Hello ' + Msg;
end;

procedure Test3(var Msg: string);
begin
  Msg := 'Goodbye ' + Msg;
end;

type
  TStruct = record
    a: array[0..1] of Integer;
    b: Integer;
  end;

function Test4(Struct: TStruct): TStruct;
begin
  // Swap first and second
  Result.a[0] := Struct.a[1];
  Result.a[1] := Struct.a[0];
  Result.b :=
    Struct.a[0] * Struct.a[1] + Struct.b;
end;
```

Figure 2.2.: Call Demo - Design-time

script! For a complete list of functions defined by the GUI component please refer to to chapter 3.

So, what's necessary to call a script procedure?

Listing 2.2: CallDemo: calling Test1

```
procedure TForm1.Button1Click(Sender: TObject);
var
  prog: TProgram;
begin
  prog := DelphiWebScriptII.Compile(Memo1.Text);
  prog.BeginProgram;
  prog.Info.Func['Test1'].Call([Edit1.Text]);
  prog.EndProgram;
  prog.Free;
end;
```

As you can see, it's pretty easy to invoke a script procedure!

First, the script is compiled into a TProgram. As a next step the *BeginProgram* function is called.
It's utterly important that you call this method *before* trying to invoke a script function! You also have to call *EndProgram* before destroying the TProgram object.

NOTE

Please keep in mind that the source found in listing 2.2 is nowhere near production quality! There are several points that "smell":

- No try-finally around the creation of the TProgram object and the call to *free*
- No check for DWS II compile time errors

Before going into details about the invocation call itself, let's look at the Delphi source code necessary to call the *Test4* script function:

Listing 2.3: CallDemo: calling Test4

```
procedure TForm1.Button4Click(Sender: TObject);
var
  prog: TProgram;
  funcInf, resultInf: IInfo;
begin
```

```
  prog := DelphiWebScriptII.Compile(Memo1.Text);
  prog.BeginProgram;
  funcInf := prog.Info.Func['Test4'];
  funcInf.Parameter['Struct'].Member['a'].Element([0]).
      Value := SpinEdit1.Value;
  funcInf.Parameter['Struct'].Member['a'].Element([1]).
      Value := SpinEdit2.Value;
  funcInf.Parameter['Struct'].Member['b'].Value :=
      SpinEdit3.Value;
  resultInf := funcInf.Call;
  SpinEdit4.Value := resultInf.Member['a'].Element([0]).
      Value;
  SpinEdit5.Value := resultInf.Member['a'].Element([1]).
      Value;
  SpinEdit6.Value := resultInf.Member['b'].Value;
  prog.EndProgram;
  prog.Free;
end;
```

Clearly, Calling *Test4* demands more work to be done by the programmer.

HINT Just in case you wonder where TStruct was defined, have a look at the script in figure 2.2 on page 26!

ToDo: Explain IInfo here!

2.3. DWS II Language - a Subset of Delphi Pascal

By now we've only covered the Delphi - side of DWS II. This section will give you an idea of how the Pascal subset of DWS II looks like.

The syntax of DWS II script is similar to Delphi, making the learning curve not very steep. Perhaps we should start with a list of what is *not* supported by DWS II.

ATTENTION

Following Delphi language features are NOT supported by DWS II

- sets
- with clause
- pointer
- interfaces
- function overloading
- goto, label, public, protected, private, published
- other low level Delphi instructions (packed, in/out, cdecl/.., etc..)
- call DLL functions from script directly
- save a compiled DWS II script to a file[a]

[a]Lots of developers have already demanded such a feature. Therefore we'll briefly discuss why we will **not** implement it.

First, DWS II lets you define custom filters that enables you to encrypt your scripts. Second, DWS II has a very fast compiler. It's no performance problem to compile scripts instead of loading a binary representation that has to be de-serialized. Third, DWS II very strongly depends on libraries providing functions that can be called. When loading a binary representation of a DWS II script we would have to make sure all libraries are compiled into the application that loads and wants to execute the script. When compiling the script this is automatically done by the DWS II compiler. Fourth, it would be tremendous work without gaining any significant feature:

If you want something compiled into a binary representation for speed reasons, *USE DELPHI*!

Perhaps it's more important to say what actually IS supported (Bold printed Items are not supported by Delphi):

- Flow-Control:
  - If-then-else
  - For, While, Repeat-until
    - Break, Continue
  - Case
    - **Supports non-ordinal data types**

    - Try-except, Try-finally

- Functions, Procedures
    - Recursion
    - Mutual Recursion
    - **Complex Return Types**
    - Default Parameters
- Classes
    - Fields
    - Methods
        - Virtual
        - Static
    - Properties
        - Array Properties
        - Default Properties
- Exceptions
    - native Delphi Exceptions (are mapped to 'EDelphi')
    - DWS script Exceptions
- Data types
    - Basic Data types:
        - String
        - Integer
        - Boolean
        - Float (= Double)
        - Variant
        - Every *Connector* may define other basic Data types (e.g.: COM Connector introduces COMVariant.)
    - Structured Data types:
        - Records
        - Arrays, Dynamic Arrays
    - Enumerations

- Compiler
  - Constant Folding (and/or other custom optimizations)
  - **All Variables are initialized**
  - **Supports declarations anywhere in script**
- ....

2.3.1. How to Design DWS II Scripts

In order to get the maximum DWS performance, you should know how to design your scripts. We find it very important that you stick to the design rules given in this document.

1. Scripts should be:
   a) As short as possible
   b) Must not contain calculation intense algorithms
   c) Do not excessively use loops - especially with lots of iterations
2. "Outsource" as many functions as possible to Delphi and call them. (Use the dws2Unit component!)
3. If possible, cache your compiled TPrograms - do not compile every time again.

HINT

DWS II has an option for optimization when compiling the script. The main work done by the optimizer is constant folding. Note, however, that only *internal* functions can take advantage of this by default. All other functions have to support this explicitly.
Internal functions are so to speak built into the TDelphiWebScriptII component. So most internal functions are always available.
You can find a list of all internal functions in section 3.2 on page 52.

2.3.2. DWS II Script - Syntax

The main difference of the syntax used by DWS II and Delphi is that DWS II allows you to define types/variables almost everywhere in script. This ability is necessary for working with HTML files, as there is no such structure present that we find in Delphi units or programs.

This has one important implication, namely DWS II does not like to have types defined where you would define them in Delphi: After the function / procedure header but before the 'begin' keyword of the function's / procedure's implementation part. See listing 2.4 for an example.

HINT All listings within a shadowbox are considered to be DWS II scripts. All other listings contain Delphi code.

Listing 2.4: DWSIISyntax: Differences to Delphi

```
procedure test;
  //for compatibility reasons you can declare
  // variables here
  var x:variant;

  procedure y;  // declare nested procedure - ERROR
  begin
    // DWSII does not allow you to define
    // a nested procedure here!!
  end;

begin
  procedure z; // ok
  var
    a:float;
    b:integer;
  begin
    // normal DWSII syntax: declare everything
    // you need within begin / end pairs
    var c:boolean;
  end;
end;
```

It is highly recommended that you load the *Basics* demo and experiment with the available Demos yourself! For convenience we'll reprint here the *NestedDeclarations* example. It demonstrates the lifetime of a declaration.

HINT

You might have noticed by now that there is no explicit ’main’ program in DWS II scripts. This observation is partly correct. Ideally you should think that the whole script you write is encapsulated between a *begin - end* pair! As a consequence the whole script *is* the “main” program.

Another important twist comes in here: When defining variables or types within a begin - end pair, you have to begin the declaration every time with the appropriate keyword! So the following script is not valid and will produce a compile time error.

```
//[...]
begin
var
x: integer;
y: bool; // error!
end;
//[...]
```

Below you’ll find the source of the “NestedDeclarations” example. Please note how *x* and *s* are defined the first time: Every declaration starts with the **var** keyword. (Following declaration would also have been valid: “**var** x, s: Integer;”.)

Listing 2.5: DWSIISyntax: Nested Declaration

```
{
Special DWSII Syntax:
It's possible to declare variables everywhere in
the script code. The declaration is valid inside the
active block (after the declaration) and its sub-blocks.
}
var x: Integer;
var s: Integer; // you need the 'var' here!!
for x := 0 to 10 do
begin
  var s: string; // overrides previous declaration
                 // of "s" (only inside this loop)
                 // s now a STRING
  s := IntToStr(x);
  PrintLn(s);
end;
for x := 0 to 10 do
begin
  var s: Float; // overrides previous declaration
                // of "s" (only inside this loop)
                // s now a FLOAT!
  s := Sqrt(x);
  PrintLn(s);
end;
for x := 0 to 10 do
begin
  s := Round(Sqr(x)); // No redeclaration of "s" so
                      // the initial declaration is used
  PrintLn(s);
end;
```

We recommend that you look through all examples the *Basic* demo provides.

2.4. Working with Delphi Objects in Scripts: *dws2Unit*

As a last introducery topic, we'll briefly cover the mechanisms DWS II uses to work with Delphi objects. We'll discuss the examples found in the *Basic* demo here.

The *dws2Unit* component is a cornerstone in DWS II: It's the link between native, compiled Delphi code and DWS II scripts. As a consequence it's therefore important to know how to work with this component.

NOTE

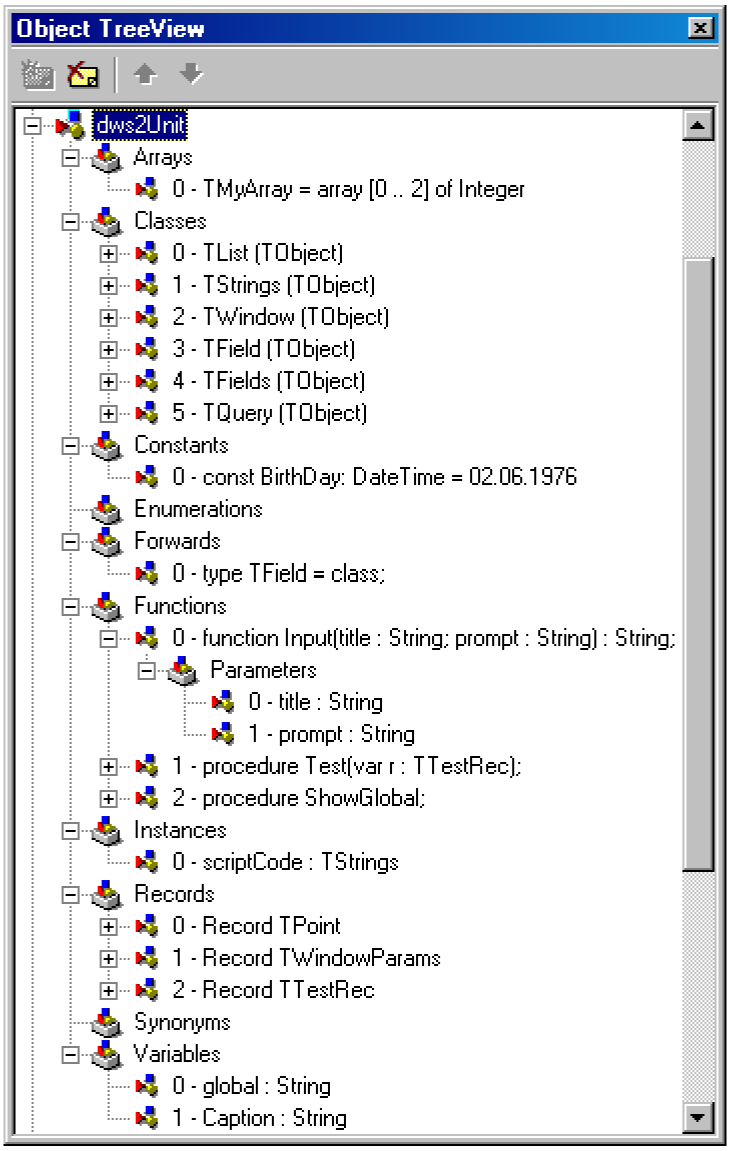
Although *dws2Unit* made it somewhat easier to create "wrapper code" for DWS II, it's still a bit cumbersome to work with it. In future, however, there will be some improvement to this situation as there is a dws2UnitEditor in development. (You can find the sources in the dws2tools module).

Figure 2.3 on the facing page shows all types and functions that are defined in the dws2Unit component of the Basic demo.

Modifying the state of a native Delphi object in DWS II script is a very common task. Unlike .NET and Java, Delphi does not offer reflection over all types. As you might know, only published parts of a class declaration offer meta-data in Delphi. DWS II therefore needs your help to figure out what methods, fields and properties a Delphi class/object exposes.[1] In the following chapters we'll refer to this process of providing the information as *writing wrapper code*.

If you open the *Using Delphi in DWS -> Dws2AccessInDelphi.dws* script in the Basic demo, you should see following source:

[1]Delphi runtime packages offer far more information about the members they contain. Unfortunately it's not possible to determine the return type of a method/function found in a runtime package. Therefore even if it would be possible to call functions in runtime packages, one would have to provide the complete interface declaration of the functions one wants to call.



```
dws2Unit
  Arrays
    0 - TMyArray = array [0 .. 2] of Integer
  Classes
    0 - TList (TObject)
    1 - TStrings (TObject)
    2 - TWindow (TObject)
    3 - TField (TObject)
    4 - TFields (TObject)
    5 - TQuery (TObject)
  Constants
    0 - const BirthDay: DateTime = 02.06.1976
  Enumerations
  Forwards
    0 - type TField = class;
  Functions
    0 - function Input(title : String; prompt : String) : String;
      Parameters
        0 - title : String
        1 - prompt : String
    1 - procedure Test(var r : TTestRec);
    2 - procedure ShowGlobal;
  Instances
    0 - scriptCode : TStrings
  Records
    0 - Record TPoint
    1 - Record TWindowParams
    2 - Record TTestRec
  Synonyms
  Variables
    0 - global : String
    1 - Caption : String
```

Figure 2.3.: Object TreeView of the dws2Unit component found in Basic demo

Listing 2.6: Working with Delphi objects in DWSII: TWindow

```
 2 {
     Demonstrates how to call methods from native
 4   Delphi code and how to access the datastructures
     of DWS.
 6   Have a look at the Delphi source code of
     the DWSII class "TWindow"!
 8 }
   var w: TWindow;
10 var i, j, k: integer;

12 w := TWindow.Create(10, 10, 'Hello World');

14 for i := 10 to 100 do
   begin
16   Pause(10); // Sleep
     w.SetPosition(i, i);
18   RedrawDwsDemo;
   end;
20


22 var p: TWindowParams;
   type test = array [1..2] of record a, b: integer end;
24 var t: Test;
   p.Left := 200;
26 p.Top := 200;
   p.Width := 300;
28
   p.Height := 50;
30 p.Caption := '';
   w.SetParams(p);
32 w.UseVarParamTest;
   var v: TWindow;
34 v := TWindow(w.NewInstance);

36 Pause(3000);
```

If you compile and run the demo, you should see three windows created: At first a little window that moves from top left 100 steps to bottom right. This is done by the *for* loop starting at line 14. If the loop has completed, the script will create two windows and assign different captions and positions to them.

ATTENTION

If you look at the script, you'll notice that there is no call to a destructor of any of the objects created. If you look at figure 2.4 on the next page, you'll see that the TWindow class *does* have a destructor defined.
So what's going on here?
The answer is pretty simple: DWS II 2.x features *Garbage Collection*!

You do not need to call any destructor in a DWS II script!

HINT

It's important to realize that a type that can be used in a DWS II script is NOT the native type defined by Delphi - even if the DWS II type has the same name as the Delphi type and tries to provide the same interface as the Delphi type.

Everything that you can use in a DWS II script is some wrapper around some native code.

The COM connector might be - partly - an exception to this rule.

To fully understand the script from listing 2.6, we must have a look at the definitions and the wrapper code provided by the dws2Unit component for TWindow. See figure 2.4 on the following page for the 'interface' definition of the TWindow class.

2.4.1. Constructor and ExternalObject

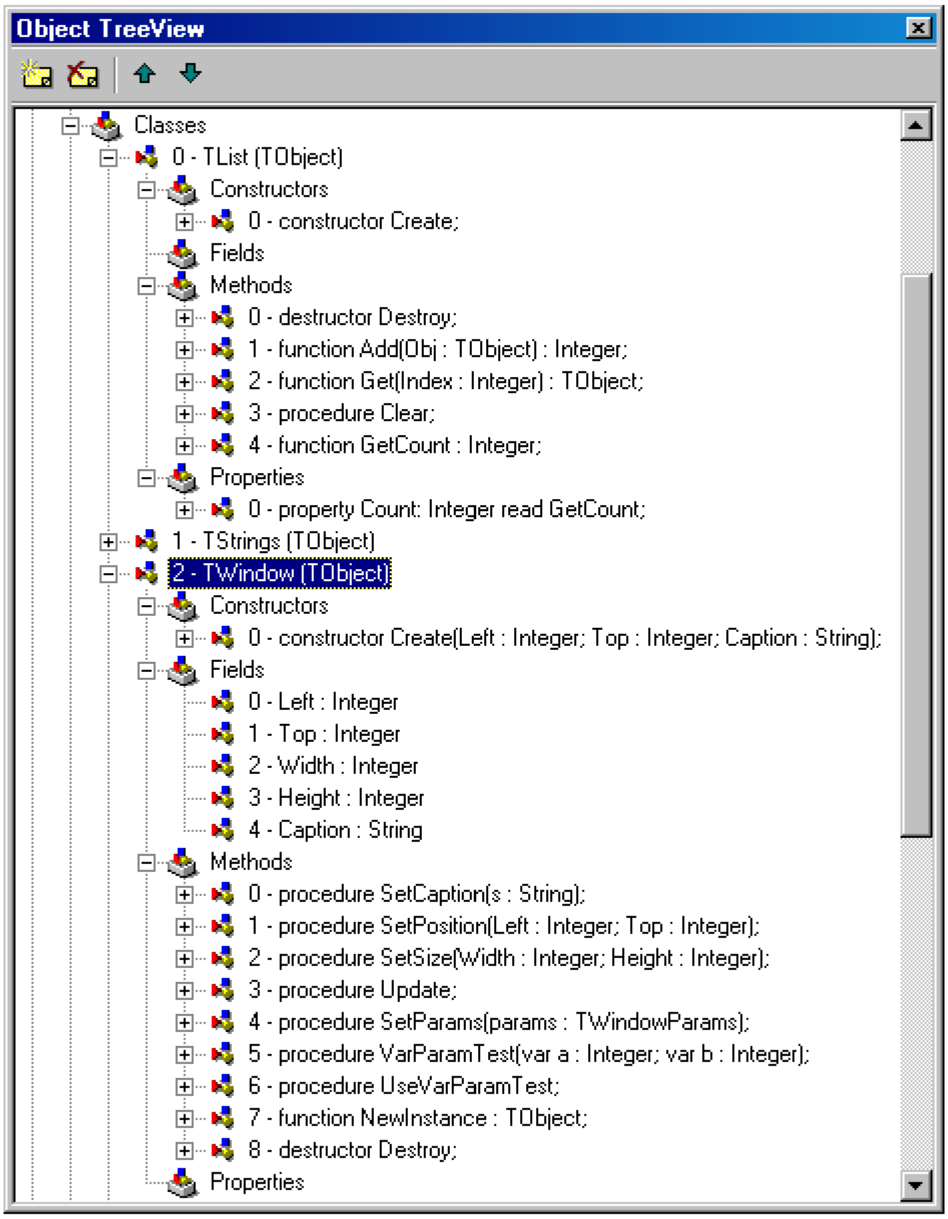
Let's start dissecting the example and begin with line 12, where the TWindow constructor gets called the first time.

Figure 2.5 on page 39 shows a screen shot of the Object Inspector's settings for TWindow's constructor. Most notably is the *OnAssignExternalObject* event. Listing 2.7 shows the code behind that event.

Listing 2.7: Working with Delphi objects in DWSII: TWindow Constructor

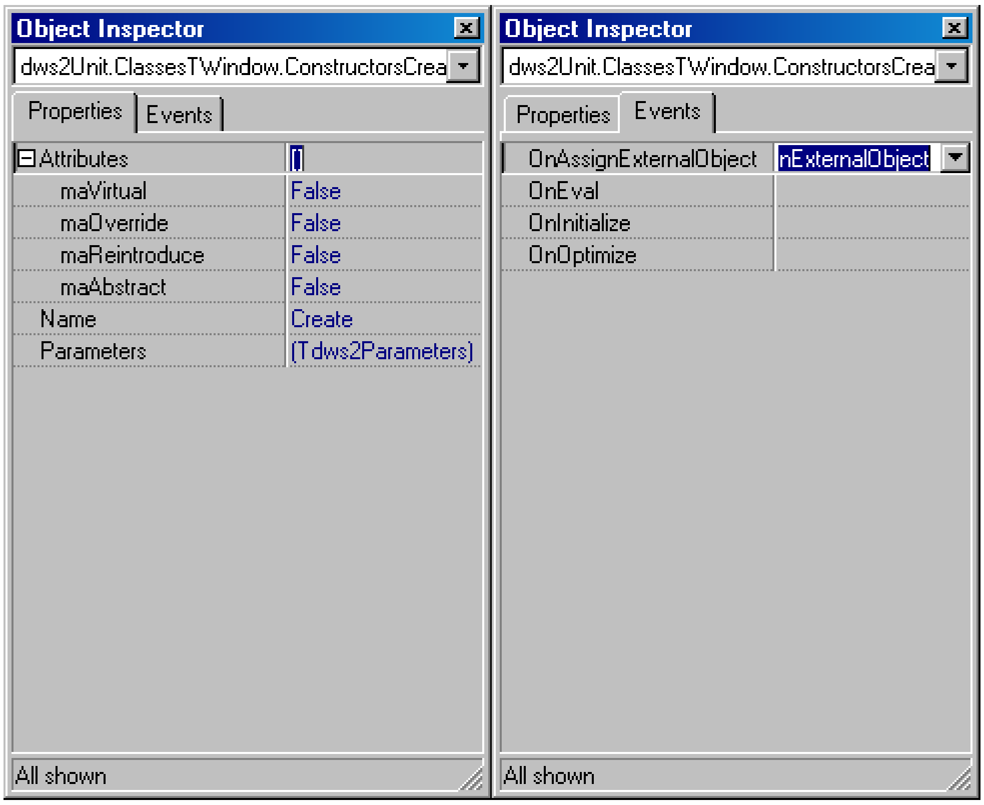
```

procedure TFDwsDemo.
    dws2UnitClassesTWindowConstructorsCreate
  AssignExternalObject(Info: TProgramInfo; var ExtObject:
    TObject);
begin
  ExtObject := TFTest.Create(nil);
```



```
Object TreeView
Classes
  0 - TList (TObject)
    Constructors
      0 - constructor Create;
    Fields
    Methods
      0 - destructor Destroy;
      1 - function Add(Obj : TObject) : Integer;
      2 - function Get(Index : Integer) : TObject;
      3 - procedure Clear;
      4 - function GetCount : Integer;
    Properties
      0 - property Count: Integer read GetCount;
  1 - TStrings (TObject)
  2 - TWindow (TObject)
    Constructors
      0 - constructor Create(Left : Integer; Top : Integer; Caption : String);
    Fields
      0 - Left : Integer
      1 - Top : Integer
      2 - Width : Integer
      3 - Height : Integer
      4 - Caption : String
    Methods
      0 - procedure SetCaption(s : String);
      1 - procedure SetPosition(Left : Integer; Top : Integer);
      2 - procedure SetSize(Width : Integer; Height : Integer);
      3 - procedure Update;
      4 - procedure SetParams(params : TWindowParams);
      5 - procedure VarParamTest(var a : Integer; var b : Integer);
      6 - procedure UseVarParamTest;
      7 - function NewInstance : TObject;
      8 - destructor Destroy;
    Properties
```

Figure 2.4.: Object TreeView of the TWindow class defined by dws2Unit



Object Inspector

dws2Unit.ClassesTWindow.ConstructorsCrea

Properties | Events

⊟ Attributes	[]
maVirtual	False
maOverride	False
maReintroduce	False
maAbstract	False
Name	Create
Parameters	(Tdws2Parameters)

All shown

Object Inspector

dws2Unit.ClassesTWindow.ConstructorsCrea

Properties | Events

OnAssignExternalObject	nExternalObject
OnEval	
OnInitialize	
OnOptimize	

All shown

Figure 2.5.: Object Inspector: TWindow Constructor

```
   TFTest(ExtObject).Show;
 end;
```

As you can see, the *ExtObject* parameter is declared using the *var* directive. This indicates that *ExtObject* can already have a value assigned when entering *OnAssignExternalObject*! (Otherwise we would have used the *out* directive.)

There is only one case where this can happen: If you call a constructor via *Info.Vars['TMyClass'].-GetConstructor('Create', delphiObject).Call.Value*, OnAssignExternalObject gets called with *ExtObject* assigned to *delphiObject*.

The most important part of this listing is found in line 5. In this line a *Delphi* object gets created and assigned to the ExtObject variable.

ATTENTION

It's important that you understand what function the *ExtObject* fulfills! Remember: We are writing *wrapper code* here. In other words, we have to create the glue that is necessary for manipulating native Delphi objects. The ExtObject hereby holds a reference to the native Delphi object we want to control within the DWS II script. You have to create it in the code that implements the *OnAssignExternalObject* event and you have to destroy it later in a destructor. (The destructor gets automatically called by DWS II)
Any method you define and implement later for your DWS II class will get exactly the same ExtObject you created here passed in as function parameter!

HINT

The dws2Unit component offers a feature called *Instances* that is useful for wrapping variables / properties if they contain / return an object. The *Using Delphi in DWS* -> *Tdws2UnitVariablesAutoinstantiate.dws* script shows how to use that feature.
Basically it allows you to assign the ExtObject on the first read operation of the "*instance unit variable*".

2.4.2. Simple Method

The first real method call in listing 2.6 can be found in line 17.

Listing 2.8: Working with Delphi objects in DWSII: TWindow SetPosition

```
2 procedure TFDwsDemo.
      dws2UnitClassesTWindowMethodsSetPositionEval(Info:
      TProgramInfo; ExtObject: TObject);
  begin
4   Info.Vars['Self'].Member['Left'].Value := Info['Left'];
    Info.Vars['Self'].Member['Top'].Value := Info['Top'];
6   Info.Func['Update'].Call;
  end;
```

As you can see in figure 2.4 on page 38, this method has two parameters: *Left* and *Top*. The TWindow class itself also has two members (fields) named *Left* and *Top*.

The code in listing 2.8 does nothing else, than assign the values of the method's parameters to the internal fields. (Lines 4 and 5.)

Line 6 should look somewhat familiar, if you've already read section 2.2 on page 25. To put it in a nutshell: The *Update* method of TWindow gets called.

Update is very interesting to us, because it's there, where the linking between DWS II Script Class and native Delphi class happens:

Why not “Info. Vars['Self'] .Func['Update'] ..”?! -ASK-

Listing 2.9: Working with Delphi objects in DWSII: TWindow Update

```
procedure TFDwsDemo.dws2UnitClassesTWindowMethodsUpdateEval
    (Info: TProgramInfo; ExtObject: TObject);
begin
  TFTest(ExtObject).Left := Info['Left'];
  TFTest(ExtObject).Top := Info['Top'];
  TFTest(ExtObject).Width := Info['Width'];
  TFTest(ExtObject).Height := Info['Height'];
end;
```

2.4.3. More Advanced Techniques

Another common situation is that your script functions get a script object (say a TSomeScriptObj) as parameter. In the *OnEval* event, however, you need to modify the underlying native Delphi object.

So how do you get the native Delphi Object? Listing 2.10 shows you how to do that.

Listing 2.10: Advanced Techniques: Get ScriptObject in OnEval

```
{ Declaration of the script procedure: procedure(SomeObject
    :TSomeScriptObj)
```

```
  Note: TSomeScriptObj is a script class that has a
      TSomeObject Delphi object assigned.. It's a wrapper
      around TSomeObject in other words.}
var
  ScriptObj: IScriptObj;
  NativeObject: TSomeObject;
begin
  // get the script object
  ScriptObj := IScriptObj(IUnknown(Info['SomeObject']));
  if ScriptObj = nil then
    NativeObject := nil
  else
    // get the native Object
    NativeObject := TSomeObject(ScriptObj.ExternalObject);
end
```

Here is the declaration of the IScriptObj interface that represents a DWS II script object:

Listing 2.11: Advanced Techniques: IScriptObj definition

```
  IScriptObj = interface
    ['{8D534D1E-4C6B-11D5-8DCB-0000216D9E86}']
    function GetClassSym: TClassSymbol;
    function GetData: TData;
    procedure SetData(Dat: TData);
    function GetExternalObject: TObject;
    procedure SetExternalObject(Value: TObject);
    property ClassSym: TClassSymbol read GetClassSym;
    property Data: TData read GetData write SetData;
    property ExternalObject: TObject read GetExternalObject
         write SetExternalObject;
  end;
```

Other advanced topics:

- If you want to know, how you can call a constructor of a script class and what to return, look at the *NewInstance* method of TWindow.
- There are plenty of other examples available in the basic demo - so have a look at them!

2.5. Case Study on DWS$DBDemos Script

by Yeoh Ray Mond

This example shows how we wrap Delphi's TQuery, TFields and TField classes. Corresponding DWS classes were created to wrap each of this object. So, for every DWS TQuery, we will expect to hold a reference to a Delphi TQuery instance. In the DWS TQuery class, there is a property named FFields, which is of the DWS TFields type, which in turn wraps around a Delphi TFields object (or a helper class in this instance). Finally, the DWS TFields object has a GetField method, which is used to return a specific DWS TField instance, given the field name.

So, bear in mind the following as we work through the example:

DWS object	corresponding Delphi object
TQuery	TQuery
TFields	TFieldsLookup (helper class for TFields)
TField	TField

DWS script:

Listing 2.12: DWS DBDemos

```
   var q: TQuery;
 2 q := TQuery.Create('DBDEMOS', 'select␣*␣from␣customer');
   q.First;
 4 while not q.Eof do
   begin
 6   Print(q.FieldByName('CustNo').AsInteger);
     Print(',␣');
 8   PrintLn(q.FieldByName('Company').AsString);
    q.Next;
10 end;
```

In line 2, a DWS TQuery instance is constructed with 2 parameters, a database alias and a SQL statement. This causes the DWS constructor for the DWS TQuery class to be called. The OnAssignExternalObject will be called first, followed by OnEval.

In the OnAssignExternalObject method, you can see the following Delphi code:

Listing 2.13: DWS DBDemos - TQuery.Create

```
procedure TFDwsDemo.dws2UnitClassesTQueryConstructorsCreate
      AssignExternalObject( Info: TProgramInfo; var ExtObject
    : TObject);
begin
  ExtObject := TQuery.Create(nil);
end;
```

A Delphi TQuery instance is created, and assigned to the ExtObject field of the script object. In other words, the DWS 'q' variable will hold a reference to this Delphi TQuery instance via the DWS TQuery instance.

Next, the OnEval method of the DWS TQuery constructor is called:

Listing 2.14: DWS DBDemos - TQuery.Create 2

```
procedure TFDwsDemo.
    dws2UnitClassesTQueryConstructorsCreateEval( Info:
    TProgramInfo; ExtObject: TObject);
var
  q: TQuery;
  fieldsLookup: TFieldsLookup;
begin
  q := TQuery(ExtObject);
  try
    q.DatabaseName := Info['db'];
    q.SQL.Text := Info['query'];
    q.Prepare;
    q.Open;
    fieldsLookup := TFieldsLookup.Create(q.Fields);
    Info['FFields'] := Info.Vars['TFields'].GetConstructor(
        'Create', FieldsLookup).Call.Value;
  except
    q.Free;
    raise;
  end;
end;
```

We can use the Delphi TQuery object we created earlier by typecasting the ExtObject variable. We then fill in the database name and SQL text to execute, and then run the query. If an exception is raised, we immediately free the Delphi TQuery object.

If the query was successfully ran, we now need to store a reference to each Delphi TField object that is returned from the script. To do this, we use a helper class named TFieldsLookup (for performance reasons. We will discuss this further at the end). The definition of TFieldsLookup is as follows:

Listing 2.15: DWS DBDemos - TFieldsLookup Definition

```
type
  TFieldsLookup = class
  private
    FFields: TFields;
    FDwsFields: TInterfaceList;
  public
```

```
    constructor Create(Fields: TFields);
    destructor Destroy; override;
    property Fields: TFields read FFields;
    property DwsFields: TInterfaceList read FDwsFields;
  end;
```

We can see that the constructor accepts a TFields object, and is defined as follows:

Listing 2.16: DWS DBDemos - TFieldsLookup.Create

```
constructor TFieldsLookup.Create(Fields: TFields);
begin
  FFields := Fields;
  FDwsFields := TInterfaceList.Create;
end;
```

Basically, a reference to the TFields object is stored in the FFields field and a TInterfaceList is created.

Once we have set up a TFieldsLookup instance, we have the following statement:

`Info['FFields']:=Info.Vars['TFields'].GetConstructor('Create',fieldsLookup).Call.Value;`

Briefly, what it does is assign the result of the right hand side expression to the DWS FFields field i.e. Info['FFields']. Let's analyse the right hand expression in manageable bits:

- *Info.Vars['TFields']* - refers to the DWS TFields object
- *Info.Vars['TFields'].GetConstructor('Create', fieldsLookup)* - returns the DWS constructor in an IInfo interface
- *Info.Vars['TFields'].GetConstructor('Create', fieldsLookup).Call* - calls the constructor of the DWS TFields class, passing in the Delphi fieldsLookup instance as the ExtObject in the constructor. In this case, ExtObject is already assigned to the DWS object. That is why in the constructor for the DWS TFields class, there is no code in the OnAssignExternalObject method.

In the OnEval method of the DWS TFields object, we see the following code:

Listing 2.17: DWS DBDemos - TFields.Create

```
procedure TFDwsDemo.
    dws2UnitClassesTFieldsConstructorsCreateEval( Info:
    TProgramInfo; ExtObject: TObject);
var
  x: Integer;
  fieldsLookup: TFieldsLookup;
  dwsField: IUnknown;
begin
  fieldsLookup := (ExtObject as TFieldsLookup);
  // Create a DWS-TField object for every Delphi-TField
  for x := 0 to fieldsLookup.Fields.Count - 1 do
  begin
    dwsField := Info.Vars['TField'].GetConstructor('Create'
        , fieldsLookup.Fields[x]).Call.Value;
    fieldsLookup.DwsFields.Add(dwsField);
  end;
end;
```

Since ExtObject contains a reference to our TFieldsLookup instance (fieldsLookup), we can just typecast it and use it here. Remember, our fieldsLookup instance also contains a reference to the Delphi TFields object (in its FFields field) we obtained earlier from the Delphi TQuery instance. Now, for each Delphi TField instance, we create a corresponding DWS TField instance. Remember, each DWS object implements the IScriptObj interface, so we can hold the reference using an IUnknown type.

To link a DWS TField object to a Delphi TField object, we call the constructor of the DWS TField class via the following:

dwsField:=Info.Vars['TField'].GetConstructor('Create',fieldsLookup.Fields[x]).Call.Value;

If you look at the constructor for the DWS TField object, you do not see any code at all. This is because the most important thing, which is the linking of a Delphi TField instance to a DWS TField instance, is already done for us by the GetContructor method.

At this point, we have a DWS TQuery that references a Delphi TQuery object, a DWS TFields object that references a Delphi TFieldsLookup object (that acts as a wrapper around a Delphi TFields object) and many DWS TField objects that references many Delphi TField objects. How do we make use of this?

Let's look at the FieldByName function defined for the DWS TQuery class, which returns a DWS TField object.

Listing 2.18: DWS DBDemos - TQuery.FieldByName

```
procedure TFDwsDemo.dws2UnitClassesTQueryFieldByNameEval(
    Info: TProgramInfo; ExtObject: TObject);
begin
  Info['Result'] := Info.Vars['FFields'].Method['GetField'
      ].Call([Info['FieldName']]).Value;
end;
```

What this does is call the DWS GetField method of the FFields class in the DWS TQuery instance. This call also passes in one parameter, which is the name of the field we want to use.

Listing 2.19: DWS DBDemos - TFields.GetField

```
procedure TFDwsDemo.
    dws2UnitClassesTFieldsMethodsGetFieldEval( Info:
    TProgramInfo; ExtObject: TObject);
var
  fieldsLookup: TFieldsLookup;
  fieldIndex: Integer;
begin
  fieldsLookup := (ExtObject as TFieldsLookup);
  fieldIndex := fieldsLookup.Fields.FieldByName(Info['
     FieldName']).Index;
  Info['Result'] := fieldsLookup.DwsFields[fieldIndex];
end;
```

Here, we make use of our helper class to find the index of the field we are interested in, and store in Info['Result'] the IScriptObj reference.

We then return the result as a DWS TField object. With this, we can then access the various AsString and AsInteger methods of the DWS TField class. This allows you to write the following in DWS:

```
Print(q.FieldByName('CustNo').AsInteger);
PrintLn(q.FieldByName('Company').AsString);
```

Wrapping the other methods of the Delphi TQuery object is quite straightforward, so I will not explain them here.

When the script ends, all the destructors for the used DWS object instances will be called. In the DWS TQuery destructor, we see the following:

Listing 2.20: DWS DBDemos - TQuery.Destroy

```
procedure TFDwsDemo.dws2UnitClassesTQueryDestroyEval(Info:
    TProgramInfo; ExtObject: TObject);
begin
  ExtObject.Free;
end;
```

The Delphi TQuery instance is freed here, together with its TFields field and the corresponding TField instances. We do not have to do this ourselves, since we never created any Delphi TFields or TField instances. We only assigned references to these Delphi objects.

We do need to free objects we created explicitly i.e. the instance of TFieldsLookup that we created. This is done in the destructor for the DWS TFields class:

Listing 2.21: DWS DBDemos - TFields.Destroy

```
procedure TFDwsDemo.
    dws2UnitClassesTFieldsMethodsDestroyEval( Info :
    TProgramInfo; ExtObject: TObject);
begin
  ExtObject.Free;
end;
```

The Delphi destructor for the TFieldsLookup class has to also free the TInterfaceList what we created.

Listing 2.22: DWS DBDemos - TFieldsLookup.Destroy

```
destructor TFieldsLookup.Destroy;
begin
  FDwsFields.Free;
  inherited;
end;
```

Note that since all DWS script objects are interfaces in Delphi, they are automatically garbage collected when they go out of scope. That is why in the script, you do not see a statement to free the DWS TQuery instance anywhere i.e. there is no q.Free anywhere.

Additional note:

In this example, we are using a help class to manage the Delphi TFields instance. We could just have easily done the following in the FieldByName method of the DWS TQuery class:

Info['Result'] := Info.Vars['TField'].GetConstructor('Create', TQuery(ExtObject).FieldByName (Info['FieldName'])).Call.Value;

What this does is create a DWS TField instance for the field that we want, everytime we call the FieldByName method. This is less efficient if we were to call FieldByName frequently, as a DWS script object would need to be repeatedly created and freed.

Part II.

Standard DWS II Library Reference

3. DWS II Standard Libraries

The main part of Applied DWS II Programming is the documentation of the libraries that ship with DWS II. Consider the information found here as "standard".

Figure 3.1 shows a screen shot of the DWS2 tab of the Delphi component palette with the DWS II Common Libraries installed.

CONVENTION

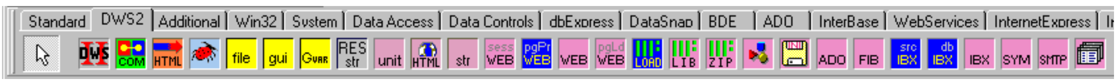
If you want to implement a library for DWS, please search this chapter for libraries already providing a similar service and consider the API they provide as *must implement*. (Note: Before raising some sort of "ENotImplemented" exceptions in your library, skip the method alltogether - even if this method is demanded by the standard.)
For example: If you want to implement a new database connectivity library for - say - SAP DB, then look at the DB libraries found here. They define the minimum subset of functions and methods your library has to provide in oder to be *DWS II Standard Library Compliant*!
Note, that only compliant libraries will be included in the main DWS II release package in future.

3.0.1. How To Manually Install a DWS II Library

Installing a DWS II Library by hand is fairly easy. Just select File->Open Project (*.dpk as file type) in the Delphi main menu. Delphi now shows an open file dialog box. Browse to your DWS II installation directory and double click on the "Libraries" subdirectory. The open file dialog now shows you the contents of the *dws2\Libraries* directory. Search for a subdirectory according to your Delphi version: *Delphi6* when you are using Delphi version 6, *Kylix3* if you are running Kylix version 3, etc.

Open that subdirectory and open any Delphi package of your choice. After you've opened the package, click *Install* in the Delphi package dialog window to install the selected Library.

Figure 3.1.: DWS II tab with Common Libraries installed



HINT

Do not forget to update your Library-Path settings!

3.1. COM - Connector

Listing 3.1: Connectors: COM

```
{ COM connector defines two datatypes: "ComVariant" and "
    ComVariantArray"}

const ComOpt: Variant = [varError];

{ Function to create a new COM-Object: Creates an COM
    object and returns a pointer to the IDispatch interface
    . See Delphi help for CreateOleObject.}
function CreateOleObject(ClassName: String): ComVariant;
```

3.2. Internal Functions

Most of the internal functions are always available in DWS II scripts. Only the GUI, GlobalVar and File functions have to be manually included by dropping the appropriate components onto the form.

NOTE

The procedures *print* and *println* are built into the TDelphiWebScriptII component.

- **procedure** Print(Output: String);
  Adds Output without adding CRLF to TProgram.Result
- **procedure** PrintLn(Output: String);
  Adds Output + CRLF to TProgram.Result

Note that you can not use PrintLn if you want to add one - in the web browser visible - line feed to a HTML page!

3.2.1. Maths Functions

Listing 3.2: Internal Functions: Maths

```
RegisterInternalFunction(TSinFunc, 'Sin', ['a', cFloat], cFloat);
RegisterInternalFunction(TSinhFunc, 'Sinh', ['a', cFloat], cFloat);
RegisterInternalFunction(TCosFunc, 'Cos', ['a', cFloat], cFloat);
RegisterInternalFunction(TCoshFunc, 'Cosh', ['a', cFloat], cFloat);
RegisterInternalFunction(TTanFunc, 'Tan', ['a', cFloat], cFloat);
RegisterInternalFunction(TTanhFunc, 'Tanh', ['a', cFloat], cFloat);
RegisterInternalFunction(TArcSinFunc, 'ArcSin', ['v', cFloat], cFloat);
RegisterInternalFunction(TArcSinhFunc, 'ArcSinh', ['v', cFloat], cFloat);
RegisterInternalFunction(TArcCosFunc, 'ArcCos', ['v', cFloat], cFloat);
RegisterInternalFunction(TArcCoshFunc, 'ArcCosh', ['v', cFloat], cFloat);
RegisterInternalFunction(TArcTanFunc, 'ArcTan', ['v', cFloat], cFloat);
RegisterInternalFunction(TArcTanhFunc, 'ArcTanh', ['v', cFloat], cFloat);
RegisterInternalFunction(TCotanFunc, 'Cotan', ['a', cFloat], cFloat);
RegisterInternalFunction(THypotFunc, 'Hypot', ['x', cFloat, 'y', cFloat], cFloat);
RegisterInternalFunction(TIncFunc, 'Inc', ['@a', cInteger, 'b', cInteger], cInteger);
RegisterInternalFunction(TAbsFunc, 'Abs', ['v', cFloat], cFloat);
RegisterInternalFunction(TExpFunc, 'Exp', ['v', cFloat], cFloat);
RegisterInternalFunction(TLnFunc, 'Ln', ['v', cFloat], cFloat);
RegisterInternalFunction(TLog2Func, 'Log2', ['v', cFloat], cFloat);
RegisterInternalFunction(TLog10Func, 'Log10', ['v', cFloat], cFloat);
RegisterInternalFunction(TLogNFunc, 'LogN', ['n', cFloat, 'x', cFloat], cFloat);
RegisterInternalFunction(TSqrtFunc, 'Sqrt', ['v', cFloat], 
    cFloat);
RegisterInternalFunction(TSqrFunc, 'Sqr', ['v', cFloat], 
    cFloat);
RegisterInternalFunction(TIntFunc, 'Int', ['v', cFloat], 
    cFloat);
RegisterInternalFunction(TFracFunc, 'Frac', ['v', cFloat], 
    cFloat);
RegisterInternalFunction(TTruncFunc, 'Trunc', ['v', cFloat
    ], cInteger);
RegisterInternalFunction(TRoundFunc, 'Round', ['v', cFloat
    ], cInteger);
RegisterInternalFunction(TPowerFunc, 'Power', ['base', 
    cFloat, 'exponent', cFloat], cFloat);
RegisterInternalFunction(TDegToRadFunc, 'DegToRad', ['a', 
    cFloat], cFloat);
RegisterInternalFunction(TRadToDegFunc, 'RadToDeg', ['a', 
    cFloat], cFloat);
RegisterInternalFunction(TMaxFunc, 'Max', ['v1', cFloat, '
    v2', cFloat], cFloat);
RegisterInternalFunction(TMinFunc, 'Min', ['v1', cFloat, '
    v2', cFloat], cFloat);
RegisterInternalFunction(TPiFunc, 'Pi', [], cFloat);
RegisterInternalFunction(TRandomFunc, 'Random', [], cFloat)
    ;
RegisterInternalFunction(TRandomIntFunc, 'RandomInt', ['
    range', cInteger], cInteger);
RegisterInternalFunction(TRandomizeFunc, 'Randomize', [], '
    ');
RegisterInternalFunction(TRandGFunc, 'RandG', ['mean', 
    cFloat, 'stdDev', cFloat], cFloat);
RegisterInternalFunction(TRandSeedFunc, 'RandSeed', [], 
    cInteger);
RegisterInternalFunction(TSetRandSeedFunc, 'SetRandSeed', [
    'seed', cInteger],'');
```

3.2.1.1. Math Trigonometric Utilities - DWS II Reference

Sin, Sinh

Returns the (hyperbolic) sine of the angle in radians.

Declaration

function Sin(X: Float): Float;
function SinH(X: Float): Float;

Description

The Sin function returns the sine of the argument.
X is a real-type expression. Sin returns the sine of the angle X in radians.

Sinh calculates the hyperbolic sine of X.

Cos, Cosh

Returns the (hyperbolic) cosine of the angle in radians.

Declaration

function Cos(X: Float): Float;
function CosH(X: Float): Float;

Description

The cosine function returns the cosine of the argument.
X is a real-type expression. Cos returns the sine of the angle X in radians.

Cosh calculates the hyperbolic cosine of X.

Tan, Tanh

Returns the (hyperbolic) tangent of the angle in radians.

Declaration

function Tan(X: Float): Float;
function TanH(X: Float): Float;

Description

The Tan function returns the tangent of the argument. (Tan(x) = Sin(x) / Cos(x))
X is a real-type expression. Tan returns the tangent of the angle X in radians.

Tanh calculates the hyperbolic tangent of X.

ArcSin, ArcSinh

Calculates the inverse (hyperbolic) sine of a given number.

Declaration

function ArcSin(X: Float): Float;
function ArcSinH(X: Float): Float;

Description

ArcSin returns the inverse sine of X. X must be between -1 and 1. The return value will be in the range [-Pi/2..Pi/2], in radians.
ArcSinh returns the inverse hyperbolic sine of X.

ArcCos, ArcCosh

Calculates the inverse (hyperbolic) cosine of a given number.

Declaration

function ArcCos(X: Float): Float;
function ArcCosH(X: Float): Float;

Description

ArcCos returns the inverse cosine of X. X must be between -1 and 1. The return value will be in the range [0..Pi], in radians.

ArcCosh returns the inverse hyperbolic cosine of X.

ArcTan, ArcTanh

Calculates the inverse (hyperbolic) tangent of a given number.

Declaration

function ArcTan(X: Float): Float;
function ArcTanH(X: Float): Float;

Description

ArcTan returns the arctangent of X.

ArcTanh returns the inverse hyperbolic tangent of X. The value of X must be between -1 and 1 (inclusive).

Cotan

Calculates the cotangent of an angle.

Declaration

function CoTan(X: Float): Float;

Description

Call Cotan to obtain the cotangent of X. The cotangent is calculated using the formula

1 / Tan(X)

Do not call Cotan with X = 0!

Pi

Returns 3.1415926535897932385.

Declaration

function Pi: Float;

Description

Use Pi in mathematical calculations that require pi

DegToRad

Returns the value of a degree measurement expressed in radians.

Declaration

function DegToRad(x: Float): Float;

RadToDeg

Converts radians to degrees.

Declaration

function RadToDeg(x:Float): Float;

3.2.1.2. Math Float Utilities - DWS II Reference

Abs

Abs returns the absolute value of the argument, X.
X is a float-type expression

Declaration

function Abs(x: Float): Float;

Exp

Returns the value of a degree measurement expressed in radians.

Declaration

function Exp(x: Float): Float;

Ln

Returns the value of a degree measurement expressed in radians.

Declaration

function Ln(x: Float): Float;

Log2

Log2 returns the log base 2 of X.

Declaration

function Log2(x: Float): Float;

Log10

Log10 returns the log base 10 of X.

Declaration

function Log10(x: Float): Float;

LogN

LogN returns the log base Base of X.

Declaration

function LogN(Base, x: Float): Float;

Sqrt

X is a floating-point expression. The result is the square root of X.

Declaration

function Sqrt(x: Float): Float;

Sqr

The Sqr function returns the square of the argument.
X is a floating-point expression. The result, of the same type as X, is the square of X, or X*X.

Declaration

function Sqr(x: Float): Float;

Int

Int returns the integer part of X; that is, X rounded toward zero. X is a real-type expression.

Declaration

function Int(x: Float): Float;

Frac

The Frac function returns the fractional part of the argument X.
X is a real-type expression. The result is the fractional part of X; that is, Frac(X) = X - Int(X).

Declaration

function Frac(x: Float): Float;

Trunc

The Trunc function truncates a real-type value to an integer-type value. X is a real-type expression. Trunc returns an Integer value that is the value of X rounded toward zero.

Declaration

function Trunc(x: Float): Integer;

Round

The Round function rounds a real-type value to an integer-type value.

X is a real-type expression. Round returns an Integer value that is the value of X rounded to the nearest whole number. If X is exactly halfway between two whole numbers, the result is always the even number. This method of rounding is often called Bankers Rounding.

Note: The behavior of Round can be affected by the Set8087CW procedure or SetRoundMode function.

Declaration

function Round(x: Float): Integer;

Power

Power raises Base to any power. For fractional exponents or exponents greater than MaxInt, Base must be greater than 0.

Declaration

function Power(Base, Exponent: Float): Float;

Max

Call Max to compare two numeric values. Max returns the greater value of the two.

Declaration

function Max(v1,v2: Float): Float;

Min

Call Min to compare two numeric values. Min returns the smaller value of the two.

Declaration

function Min(v1,v2: Float): Float;

3.2.1.3. Math Random Utilities - DWS II Reference

Random

Returns a random number within the range 0 < = X < 1

Declaration

function Random: Float;

RandomInt

Returns a random number within the range 0 < = X < Range; x e IN

Declaration

function RandomInt(Range: Integer): Integer;

Randomize

Randomize initializes the built-in random number generator with a random value (obtained from the system clock). The random number generator should be initialized by making a call to Randomize, or by assigning a value to RandSeed.

Do not combine the call to Randomize in a loop with calls to the Random function. Typically, Randomize is called only once, before all calls to Random.

Declaration

procedure Randomize;

RandG

RandG produces random numbers with Gaussian distribution about the Mean. This is useful for simulating data with sampling errors and expected deviations from the Mean.

Declaration

function RandG(Mean, StdDev: Float): Float;

RandSeed

Get the value of the Delphi-RandSeed variable.

Declaration

function RandSeed: Integer;

SetRandSeed

Set the Delphi RandSeed variable.

Declaration

procedure SetRandSeed(val: Integer);

3.2.1.4. Math Misc. Utilities - DWS II Reference

Inc

Increments a by b.

Declaration

procedure Inc(a,b: Integer);

Hypot

Calculates the length of the hypotenuse.

Declaration

function Hypot(x, y: Float): Float;

3.2.2. String Functions

Listing 3.3: Internal Functions: String Functions

```

RegisterInternalFunction(TIntToStrFunc, 'IntToStr', ['i', cInteger],cString);
RegisterInternalFunction(TStrToIntFunc, 'StrToInt', ['str', cString], cInteger);
RegisterInternalFunction(TStrToIntDefFunc, 'StrToIntDef', ['str', cString, 'def', cInteger], cInteger);
RegisterInternalFunction(TIntToHexFunc, 'IntToHex', ['v', cInteger, 'digits', cInteger], cString);
RegisterInternalFunction(TFloatToStrFunc, 'FloatToStr', ['f', cFloat], cString);
RegisterInternalFunction(TStrToFloatFunc, 'StrToFloat', ['str', cString], cFloat);
RegisterInternalFunction(TStrToFloatDefFunc, 'StrToFloatDef', ['str', cString, 'def', cFloat], cFloat);
RegisterInternalFunction(TChrFunc, 'Chr', ['x', cInteger], cString);
RegisterInternalFunction(TOrdFunc, 'Ord', ['s', cString], cInteger);
RegisterInternalFunction(TCharAtFunc, 'CharAt', ['s', cString, 'x', cInteger], cString);
RegisterInternalFunction(TSetCharAtFunc, 'SetCharAt', ['@s', cString, 'x', cInteger, 'c', cString], '');
RegisterInternalFunction(TDeleteFunc, 'Delete', ['@S', cString, 'Index', cInteger, 'Len', cInteger], '');
RegisterInternalFunction(TInsertFunc, 'Insert', ['src', cString, '@S', cString, 'Index', cInteger], '');
RegisterInternalFunction(TLowerCaseFunc, 'LowerCase', ['str', cString], cString);
RegisterInternalFunction(TAnsiLowerCaseFunc, 'AnsiLowerCase', ['str', cString], cString);
RegisterInternalFunction(TUpperCaseFunc, 'UpperCase', ['str', cString], cString);
RegisterInternalFunction(TAnsiUpperCaseFunc, 'AnsiUpperCase', ['str', cString], cString);
RegisterInternalFunction(TPosFunc, 'Pos', ['subStr', cString, 'str', cString], cInteger);
RegisterInternalFunction(TLengthFunc, 'Length', ['str', cString], cInteger);
RegisterInternalFunction(TSetLengthFunc, 'SetLength', ['@S', cString, 'NewLength', cInteger], '');
RegisterInternalFunction(TTrimLeftFunc, 'TrimLeft', ['str', cString], cString);
RegisterInternalFunction(TTrimRightFunc, 'TrimRight', ['str', cString], cString);
RegisterInternalFunction(TTrimFunc, 'Trim', ['str', cString], cString);
RegisterInternalFunction(TCompareTextFunc, 'CompareText', ['str1', cString, 'str2', cString], cInteger);
RegisterInternalFunction(TAnsiCompareTextFunc, 'AnsiCompareText', ['str1', cString, 'str2', cString], cInteger);
RegisterInternalFunction(TCompareStrFunc, 'CompareStr', ['str1', cString, 'str2', cString], cInteger);
RegisterInternalFunction(TAnsiCompareStrFunc, 'AnsiCompareStr', ['str1', cString, 'str2', cString], cInteger);
RegisterInternalFunction(TIsDelimiterFunc, 'IsDelimiter', ['delims', cString, 's', cString, 'index', cInteger], cBoolean);
RegisterInternalFunction(TLastDelimiterFunc, 'LastDelimiter', ['delims', cString, 's', cString], cBoolean);
RegisterInternalFunction(TQuotedStrFunc, 'QuotedStr', ['str', cString], cString);
RegisterInternalFunction(TCopyFunc, 'Copy', ['str', cString, 'Index', cInteger, 'Len', cInteger], cString);
RegisterInternalFunction(TLeftStrFunc, 'LeftStr', ['AText', cString, 'ACount', cInteger], cString);
RegisterInternalFunction(TRightStrFunc, 'RightStr', ['AText', cString, 'ACount', cInteger], cString);
RegisterInternalFunction(TMidStrFunc, 'MidStr', ['AText', cString,'AStart', cInteger, 'ACount', cInteger], cString);
RegisterInternalFunction(TStringOfCharFunc, 'StringOfChar', ['Ch', cString, 'Count', cInteger], cString);
```

Please note that most of the Delphi function descriptions below are taken from the Delphi help. For your convenience we chose to replicate them here.

3.2.2.1. String Utilities - DWS II Reference

IntToStr

IntToStr converts an integer into a string containing the decimal representation of that number.

Declaration

function IntToStr(i: Integer): String;

StrToInt

StrToInt converts the string S, which represents an integer-type number in either decimal or hexadecimal notation, into a number. If S does not represent a valid number, StrToInt raises an EDelphi exception.

Declaration

function StrToInt(S: String): Integer;

StrToIntDef

StrToIntDef converts the string S, which represents an integer-type number in either decimal or hexadecimal notation, into a number. If S does not represent a valid number, StrToIntDef returns the number passed in Default.

Declaration

function StrToIntDef(const S: string; Default: Integer): Integer;

IntToHex

IntToHex converts a number into a string containing the number's hexadecimal (base 16) representation. Value is the number to convert. Digits indicates the minimum number of hexadecimal digits to return.

Declaration

function IntToHex(Value: Integer; Digits: Integer): string;

FloatToStr

FloatToStr converts the floating-point value given by Value to its string representation. The conversion uses general number format with 15 significant digits.

For greater control over the formatting of the string, use the FloatToStrF function.

Declaration

function FloatToStr(Value: Float): string;

StrToFloat

Use StrToFloat to convert a string, S, to a floating-point value. S must consist of an optional sign (+ or -), a string of digits with an optional decimal point, and an optional mantissa. The mantissa consists of 'E' or 'e' followed by an optional sign (+ or -) and a whole number. Leading and trailing blanks are ignored.

The DecimalSeparator global variable defines the character that must be used as a decimal point. Thousand separators and currency symbols are not allowed in the string. If S doesn't contain a valid value, StrToFloat raises an EConvertError (=EDelphi) exception.

Declaration

function StrToFloat(const S: string): float;

StrToFloatDef

Same as StrToFloat but if the conversion fails the result value is parameter def .

Declaration

function StrToFloatDef(const S: string; def: float): float;

Chr

Chr returns the character with the ordinal value (ASCII value) of the byte-type expression, X.

Declaration

function Chr(x: Integer): String;

Ord

Ord returns the ASCII value of char s.

Declaration

function Ord(s: String): Integer;

CharAt

Returns the char at position i from string x.
Replacement for Delphi style *result := x[i]*

Declaration

function CharAt(x: String; i: Integer): String;

SetCharAt

Sets the char at position i in string x to value.
Replacement for Delphi style *x[i] := value*

Declaration

procedure SetCharAt(**var** x:string; i: integer; value: string);

Delete

Delete removes a substring of Count characters from string S starting with S[Index]. S is a string-type variable. Index and Count are integer-type expressions.

If index is larger than the length of the S or less than 1, no characters are deleted.

If count specifies more characters than remain starting at the index, Delete removes the rest of the string. If count is less than 0, no characters are deleted.

Declaration

procedure Delete(**var** S: string; Index, Count:Integer);

Insert

Insert merges Source into S at the position S[index].

Source is a string-type expression. S is a string-type variable of any length. Index is an integer-type expression. It is a character index and not a byte index.

If Index is less than 1, it is mapped to a 1. If it is past the end of the string, it is set to the length of the string, turning the operation into an append.

If the Source parameter is an empty string, Insert does nothing.

Insert throws an EOutOfMemory (=EDelphi) exception if it is unable to allocate enough memory to accommodate the new returned string.

Declaration

procedure Insert(Source: string; **var** S: string; Index: Integer);

LowerCase

LowerCase returns a string with the same text as the string passed in S, but with all letters converted to lowercase. The conversion affects only 7-bit ASCII characters between ’A’ and ’Z’. To convert 8-bit international characters, use AnsiLowerCase.

Declaration

function LowerCase(S: string): string;

AnsiLowerCase

AnsiLowerCase returns a string that is a copy of the given string converted to lower case. The conversion uses the current locale. This function supports multi-byte character sets (MBCS).

Declaration

function AnsiLowerCase(S: string): string;

UpperCase

UpperCase returns a copy of the string S, with the same text but with all 7-bit ASCII characters between ’a’ and ’z’ converted to uppercase. To convert 8-bit international characters, use AnsiUpperCase instead.

Declaration

function UpperCase(S: string): string;

AnsiUpperCase

AnsiUpperCase returns a string that is a copy of S, converted to upper case. The conversion uses the current locale.

Note: This function supports multi-byte character sets (MBCS).

Declaration

function AnsiUpperCase(S: string): string;

Pos

Pos searches for a substring, Substr, in a string, S. Substr and S are string-type expressions.

Pos searches for Substr within S and returns an integer value that is the index of the first character of Substr within S. Pos is case-sensitive. If Substr is not found, Pos returns zero

Declaration

function Pos(Substr: string; S: string): Integer;

Length

Length returns the number of characters actually used in the string.

Declaration

function Length(S): Integer;

SetLength

Sets the length of string S to NewLength.

Declaration

procedure SetLength(var S : String; NewLength : Integer);

TrimLeft

TrimLeft returns a copy of the string S with leading spaces and control characters removed.

Declaration

function TrimLeft(S: string): string;

TrimRight

TrimRight returns a copy of the string S with trailing spaces and control characters removed.

Declaration

function TrimRight(S: string): string;

Trim

Trim removes leading and trailing spaces and control characters from the given string S.

Declaration

function Trim(S: string): string;

CompareText

CompareText compares S1 and S2 and returns 0 if they are equal. If S1 is greater than S2, CompareText returns an integer greater than 0. If S1 is less than S2, CompareText returns an integer less than 0. CompareText is not case sensitive and is not affected by the current locale.

Declaration

function CompareText(S1, S2: string): Integer;

AnsiCompareText

AnsiCompareText compares S1 to S2, without case sensitivity. The compare operation is controlled by the current locale. AnsiCompareText returns a value less than 0 if S1 < S2, a value greater than 0 if S1 > S2, and returns 0 if S1 = S2.

Declaration

function AnsiCompareText(S1, S2: string): Integer;

CompareStr

CompareStr compares S1 to S2, with case sensitivity. The return value is less than 0 if S1 is less than S2, 0 if S1 equals S2, or greater than 0 if S1 is greater than S2. The compare operation is based on the 8-bit ordinal value of each character and is not affected by the current locale.

Declaration

function CompareStr(S1, S2: string): Integer;

AnsiCompareStr

AnsiCompareStr compares S1 to S2, with case sensitivity. The compare operation is controlled by the current locale. The return value is:

Condition Return Value

S1 > S2 > 0

S1 < S2 < 0

S1 = S2 = 0

Note: Most locales consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Thus, AnsiCompareStr('a','A') returns a value less than zero, while CompareStr('a','A') returns a value greater than zero.

Declaration

function AnsiCompareStr(S1, S2: string): Integer;

IsDelimiter

Call IsDelimiter to determine whether the character at byte offset Index in the string S is one of the delimiters in the string Delimiters. Index is the 0-based index of the byte in question, where 0 is the first byte of the string, 1 is the second byte, and so on.

When working with a multi-byte character system (MBCS), IsDelimiter checks to make sure the indicated byte is not part of a double byte character. The delimiters in the Delimeters parameter must all be single byte characters.

Declaration

function IsDelimiter(Delimiters, S: string; Index: Integer): Boolean;

LastDelimiter

Call LastDelimiter to locate the last delimiter in S. For example, the line

MyIndex := LastDelimiter('\.:','c:\filename.ext');

sets MyIndex to 12.

When working with multi-byte character sets (MBCS), S may contain double byte characters, but the delimiters listed in the Delimeters parameter must all be single byte non-null characters.

Declaration

function LastDelimiter(Delimiters, S: string): Integer;

QuotedStr

Use QuotedStr to convert the string S to a quoted string. A single quote character (’) is inserted at the beginning and end of S, and each single quote character in the string is repeated.

Declaration

function QuotedStr(S: string): string;

Copy

S is an expression of a string. Index and Count are integer-type expressions. Copy returns a substring containing Count characters or elements starting at S[Index].

If Index is larger than the length of S, Copy returns an empty string or array.

If Count specifies more characters than are available, only the characters from S[Index] to the end of S are returned.

Declaration

function Copy(S: String; Index, Count: Integer): String;

LeftStr

Does a Copy(Info[’AText’], 1, integer(info[’ACount’]));

Declaration

function LeftStr(AText: String; ACount: Integer): String;

RightStr

Does a Copy(Info[’AText’], Length(Info[’AText’])
+ 1 - integer(info[’ACount’]), integer(info[’ACount’]));

Declaration

function RightStr(AText: String; ACount: Integer): String;

MidStr

Does a Copy(Info[’AText’], integer(info[’AStart’]), integer(info[’ACount’]));

Declaration

function MidStr(AText: String; AStart, ACount: Integer): string;

StringOfChar

StringOfChar returns a string containing Count characters with the character value given by Ch. For example,
S := StringOfChar(’A’, 10);
sets S to the string ’AAAAAAAAAA’.

Declaration

function StringOfChar(Ch : String; Count : Integer) : String;

3.2.3. Time Functions

Listing 3.4: Internal Functions: DateTime Functions

```

RegisterInternalFunction(TNowFunc, 'Now', [], cDateTime);
RegisterInternalFunction(TDateFunc, 'Date', [], cDateTime);
RegisterInternalFunction(TTimeFunc, 'Time', [], cDateTime);
RegisterInternalFunction(TDateTimeToStrFunc, 'DateTimeToStr
    ', ['dt', cDateTime], cString);
RegisterInternalFunction(TStrToDateTimeFunc, 'StrToDateTime
    ', ['str', cString], cDateTime);
RegisterInternalFunction(TDateToStrFunc, 'DateToStr', ['dt'
    , cDateTime], cString);
RegisterInternalFunction(TStrToDateFunc, 'StrToDate', ['str
    ', cString], cDateTime);
RegisterInternalFunction(TTimeToStrFunc, 'TimeToStr', ['dt'
    , cDateTime], cString);
RegisterInternalFunction(TStrToTimeFunc, 'StrToTime', ['str
    ', cString], cDateTime);
RegisterInternalFunction(TDayOfWeekFunc, 'DayOfWeek', ['dt'
    , cDateTime], cInteger);
RegisterInternalFunction(TFormatDateTimeFunc, '
    FormatDateTime', ['frm', cString, 'dt', cDateTime],
    cString);
RegisterInternalFunction(TIsLeapYearFunc, 'IsLeapYear', ['
    year', cInteger], cBoolean);
RegisterInternalFunction(TIncMonthFunc, 'IncMonth', ['dt',
    cDateTime, 'nb', cInteger], cDateTime);
RegisterInternalFunction(TDecodeDateFunc, 'DecodeDate', ['
    dt', cDateTime, '@y', cInteger, '@m', cInteger, '@d',
    cInteger], '');
RegisterInternalFunction(TEncodeDateFunc, 'EncodeDate', ['y
    ', cInteger, 'm', cInteger, 'd', cInteger], cDateTime);
RegisterInternalFunction(TDecodeTimeFunc, 'DecodeTime', ['
    dt', cDateTime, '@h', cInteger, '@m', cInteger, '@s',
    cInteger, '@ms', cInteger], '');
RegisterInternalFunction(TEncodeTimeFunc, 'EncodeTime', ['h
    ', cInteger, 'm', cInteger, 's', cInteger, 'ms',
    cInteger], cDateTime);
```

3.2.3.1. Time Utilities - DWS II Reference

Now

Returns the current date and time, corresponding to Date + Time.

Note: Although TDateTime values can represent milliseconds, Now only returns the time to the closest second.

Declaration

function Now: TDateTime;

Date

Use Date to obtain the current local date as a TDateTime value. The time portion of the value is 0 (midnight).

Declaration

function Date: TDateTime;

Time

Time returns the current time as a TDateTime value.

Declaration

function Time: TDateTime;

DateTimeToStr

The function DateTimeToString converts the TDateTime value given by DateTime using the format given by the ShortDateFormat global Delphi variable, followed by the time using the format given by the LongTimeFormat global Delphi variable. The time is not displayed if the fractional part of the DateTime value is zero.

To change how the string is formatted, change ShortDateFormat and LongTimeFormat global date time formatting variables in Delphi.

Declaration

function DateTimeToStr(DateTime: TDateTime): string;

StrToDateTime

Call StrToDate to parse a string that specifies a date and time value. If S does not contain a valid date, StrToDate raises an EConvertError (=EDelphi) exception.

The S parameter must use the current locales date/time format. In the US, this is commonly MM/DD/YY HH:MM:SS format. Specifying AM or PM as part of the time is optional, as are the seconds. Use 24-hour time (7:45 PM is entered as 19:45, for example) if AM or PM is not specified.

Y2K issue: The conversion of two-digit year values is determined by the TwoDigitYearCenturyWindow variable. For more information, see StrToDate.

Note: The format of the date and time string varies when the values of date/time formatting variables are changed.

Declaration

function StrToDateTime(const S: string): TDateTime;

DateToStr

Use DateToStr to obtain a string representation of a date value that can be used for display purposes. The conversion uses the format specified by the ShortDateFormat global variable.

Declaration

function DateToStr(Date: TDateTime): string;

StrToDate

Call StrToDate to parse a string that specifies a date. If S does not contain a valid date, StrToDate raises an EConvertError (=EDelphi) exception.

S must consist of two or three numbers, separated by the character defined by the DateSeparator global Delphi variable. The order for month, day, and year is determined by the ShortDateFormat global Delphi variable - possible combinations are m/d/y, d/m/y, and y/m/d.

If S contains only two numbers, it is interpreted as a date (m/d or d/m) in the current year.

Year values between 0 and 99 are converted using the TwoDigitYearCenturyWindow global Delphi variable. If TwoDigitYearCenturyWindow is 0, year values between 0 and 99 are assumed to be in the current century. If TwoDigitYearCenturyWindow is greater than 0, its value is subtracted from the current year to determine the pivot; years on or after the pivot are kept in the current century, while years prior to the pivot are moved to the next century. For example:

Current year	TwoDigit-Year-Century-Window	Pivot	date = mm/dd/03	date = mm/dd/50	date = mm/dd/68
1998	0	1900	1903	1950	1968
2002	0	2000	2003	2050	2068
1998	50	1948	2003	1950	1968
2000	50	1950	2003	1950	1968
2002	50	1952	2003	2050	1968
2020	50	1970	2003	2050	2068
2020	10	2010	2103	2050	2068

Note: The format of the date string varies when the values of date/time formatting variables are changed.

Declaration

function StrToDate(const S: string): TDateTime;

TimeToStr

TimeToStr converts the Time parameter, a TDateTime value, to a string. The conversion uses the format specified by the LongTimeFormat global Delphi variable. Change the format of the string by changing the values of some of the date and time Delphi variables.

Declaration

function TimeToStr(Time: TDateTime): string;

StrToTime

Call StrToTime to parse a string that specifies a time value. If S does not contain a valid time, StrToTime raises an EConvertError (=EDelphi) exception.

The S parameter must consist of two or three numbers, separated by the character defined by the TimeSeparator global variable, optionally followed by an AM or PM indicator. The numbers represent hour, minute, and (optionally) second, in that order. If the time is followed by AM or PM, it is assumed to be in 12-hour clock format. If no AM or PM indicator is included, the time is assumed to be in 24-hour clock format.

Note: The format of the date and time string varies when the values of date/time formatting Delphi variables are changed.

Declaration

function StrToTime(const S: string): TDateTime;

DayOfWeek

DayOfWeek returns the day of the week of the specified date as an integer between 1 and 7, where Sunday is the first day of the week and Saturday is the seventh.

Note: DayOfWeek is not compliant with the ISO 8601 standard, which defines Monday as the first day of the week.

Declaration

function DayOfWeek(Date: TDateTime): Integer;

FormatDateTime

FormatDateTime formats the TDateTime value given by DateTime using the format given by Format. See Date-Time format strings in the Delphi help for more information.

If the string specified by the Format parameter is empty, the TDateTime value is formatted as if a ’c’ format specifier had been given.

Declaration

function FormatDateTime(Format: string; DateTime: TDateTime): string;

IsLeapYear

Call IsLeapYear to determine whether the year specified by the Year parameter is a leap year. Year specifies the calendar year.

Declaration

function IsLeapYear(Year: Integer): Boolean;

IncMonth

IncMonth returns the value of the Date parameter, incremented by NumberOfMonths months. NumberOfMonths can be negative, to return a date N months previous.

If the input day of month is greater than the last day of the resulting month, the day is set to the last day of the resulting month. The time of day specified by the Date parameter is copied to the result.

Declaration

function IncMonth(Date: TDateTime; NumberOfMonths: Integer): TDateTime;

DecodeDate

The DecodeDate procedure breaks the value specified as the Date parameter into Year, Month, and Day values. If the given TDateTime value has a negative (BC) year, the year, month, and day return parameters are all set to zero.

Declaration

procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Integer);

EncodeDate

EncodeDate returns a TDateTime value from the values specified as the Year, Month, and Day parameters.

The year must be between 1 and 9999.

Valid Month values are 1 through 12.

Valid Day values are 1 through 28, 29, 30, or 31, depending on the Month value. For example, the possible Day values for month 2 (February) are 1 through 28 or 1 through 29, depending on whether or not the Year value specifies a leap year.

If the specified values are not within range, EncodeDate raises an EConvertError (= EDelphi) exception.

Declaration

function EncodeDate(Year, Month, Day: Integer): TDateTime;

DecodeTime

DecodeTime breaks the object specified as the Time parameter into hours, minutes, seconds, and milliseconds.

Declaration

procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Integer);

EncodeTime

EncodeTime encodes the given hour, minute, second, and millisecond into a TDateTime value.

Valid Hour values are 0 through 24. If Hour is 24, Min, Sec, and MSec must all be 0, and the resulting TDateTime value represents midnight (12:00:00:000 AM) of the following day.

Valid Min and Sec values are 0 through 59.

Valid MSec values are 0 through 999.

If the specified values are not within range, EncodeTime raises an EConvertError exception.

The resulting value is a number between 0 and 1 (inclusive) that indicates the fractional part of a day given by the specified time or (if 1.0) midnight on the following day. The value 0 corresponds to midnight, 0.5 corresponds to noon, 0.75 corresponds to 6:00 pm, and so on.

Declaration

function EncodeTime(Hour, Min, Sec, MSec: Integer): TDateTime;

3.2.4. Variant Functions

Listing 3.5: Internal Functions: Variant Functions

```
procedure InitVariants(SystemTable, UnitSyms, UnitTable :
    TSymbolTable);
var
  T, E : TTypeSymbol;
begin
  T := SystemTable.FindSymbol('Integer') as TTypeSymbol;
  E := TEnumerationSymbol.Create('TVarType',T);
  UnitTable.AddSymbol(E);
  UnitTable.AddSymbol(TElementSymbol.Create('varEmpty', E,
      varEmpty, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varNull', E,
      varNull, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varSmallint',
      E, varSmallint, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varInteger', E
      , varInteger, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varSingle', E
      , varSingle, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varDouble', E
      , varDouble, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varCurrency',
      E, varCurrency, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varDate', E,
      varDate, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varOleStr', E
      , varOleStr, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varDispatch',
      E, varDispatch, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varError', E,
      varError, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varBoolean', E
      , varBoolean, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varVariant', E
      , varVariant, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varUnknown', E
      , varUnknown, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varShortInt',
      E, varShortInt, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varByte', E,
      varByte, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varWord', E,
      varWord, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varLongWord',
      E, varLongWord, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varInt64', E,
      varInt64, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varStrArg', E
      , varStrArg, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varString', E
      , varString, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varAny', E,
      varAny, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varTypeMask',
      E, varTypeMask, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varArray', E,
      varArray, True));
  UnitTable.AddSymbol(TElementSymbol.Create('varByRef', E,
      varByRef, True));

  T := SystemTable.FindSymbol('Variant') as TTypeSymbol;
  UnitTable.AddSymbol(TConstSymbol.Create('Null', T, Null))
      ;
  UnitTable.AddSymbol(TConstSymbol.Create('Unassigned', T,
      Unassigned));
end;
initialization
  RegisterInternalInitProc(@InitVariants);
  RegisterInternalFunction(TVarClearFunc, 'VarClear', ['@v'
      , cVariant], '');
  RegisterInternalFunction(TVarIsNullFunc, 'VarIsNull', ['v
      ', cVariant], cBoolean);
  RegisterInternalFunction(TVarIsEmptyFunc, 'VarIsEmpty', [
      'v', cVariant], cBoolean);
  RegisterInternalFunction(TVarAsTypeFunc, 'VarAsType', ['v
      ', cVariant, 'VarType', cInteger], cVariant);
  RegisterInternalFunction(TVarToStrFunc, 'VarToStr', ['v'
      , cVariant], cString);

end.
```

3.2.4.1. Variant Utilities - DWS II Reference

Null

Use Null to obtain a Null Variant that can indicate unknown or missing data. Null Variants can be assigned to variables in an application that must contain a null value. Assigning Null to a variant variable does not cause an error, and Null can be returned from any function with a variant return value.

Expressions involving Null Variants always result in a Null Variant. Thus, Null is said to propagate through intrinsic functions that return variant data types. If any part of the expression evaluates to Null, the entire expression evaluates to Null.

Declaration

const Null: Variant;

Unassigned

The Unassigned constant is used to indicate that a variant has not yet been assigned a value. The initial value of any variant is Unassigned. The Unassigned value disappears as soon as a variant is assigned any other value, including the value 0, a zero-length string, and the Null value.

Use the VarIsEmpty internal function to test whether a variant is Unassigned. When used on an Unassigned variant, the VarType internal function returns varEmpty. An EVariantError (=EDelphi) exception is raised if an Unassigned variant appears in any other expression or an attempt is made to convert an Unassigned variant to another type (using VarAsType).

You can make a variant unassigned by assigning the Unassigned return value to it..

Declaration

const Unassigend: Variant;

VarClear

Calling VarClear is equivalent to assigning the Unassigned value to the Variant. V can be either a Variant or an OleVariant, but it must be possible to assign a value to it (it must be an lvalue).

After calling VarClear, the VarIsEmpty function returns True. Using an unassigned variant in an expression causes an EVariantError (=EDelphi) exception to be thrown. Likewise, if you attempt to convert an unassigned Variant to another type (using VarAsType), an EVariantError (=EDelphi) exception is thrown.

Note: Do not confuse clearing a Variant, which leaves it unassigned, with assigning a Null value. A Null Variant is still assigned, but has the value Null. Unlike unassigned Variants, Null Variants can be used in expressions and can be converted to other types of Variants.

Declaration

procedure VarClear(V: Variant);

VarIsNull

VarIsNull returns True if the given variant contains the value Null. If the variant contains any other value, the function result is False.

Note: Do not confuse a Null variant with an unassigned variant. A Null variant is still assigned, but has the value Null. Unlike unassigned variants, Null variants can be used in expressions and can be converted to other types of variants.

Declaration

function VarIsNull(const V: Variant): Boolean;

VarIsEmpty

VarIsEmpty returns True if the given variant contains the value Unassigned. If the variant contains any other value, the function result is False.

Note: Do not confuse an unassigned variant with a Null variant. A Null variant is still assigned, but has the value Null. Unlike unassigned variants, Null variants can be used in expressions and can be converted to other types of variants.

Declaration

function VarIsEmpty(const V: Variant): Boolean;

VarAsType

VarAsType converts a variant to a specified type and returns a new Variant that has the specified type.

V is the Variant to convert.

VarType is a variant type code that indicates the type to which V should be converted. This can be one of the constants defined by the dws2VariantFunctions Delphi unit, or it can be the integer type code for a custom variant type. It cannot include the varArray or varByRef bits.

VarAsType throws an EVariantError (= EDelphi) exception if the variant cannot be converted to the given type.

Following VarTypes are available: *varEmpty*, *varNull*, *varSmallint*, *varInteger*, *varSingle*, *varDouble*, *varCurrency*, *varDate*, *varOleStr*, *varDispatch*, *varError*, *varBoolean*, *varVariant*, *varUnknown*, *varShortInt*, *varByte*, *varWord*, *varLongWord*, *varInt64*, *varStrArg*, *varString*, *varAny*, *varTypeMask*, *varArray*, *varByRef*.

Declaration

function VarAsType(V: Variant; VarType: Integer): Variant;

VarToStr

VarToStr converts the data in the variant V to a string and returns the result. If the variant has a null value, VarToStr returns an empty string.

Declaration

function VarToStr(const V: Variant): String;

3.2.5. FileFunctions (Component)

Due to security reasons the file functions are NOT included by default! If you want to allow your scripts access to the local file system, you have to drop the FileFunctions component onto the form that contains the TDelphiWebScriptII component.

ATTENTION

Including the file functions may lead to security problems!
For web development we strongly discourage the use of this component, but suggest that you write your own access functions that provide only access to the files you absolutely need.

Listing 3.6: Internal Functions: File Functions

```

RegisterInternalFunction(TSaveStringToFileFunc, 'SaveStringToFile', ['fileName', cString, 'data', cString], '');
RegisterInternalFunction(TLoadStringFromFileFunc, 'LoadStringFromFile', ['fileName', cString], cString);
RegisterInternalFunction(TAppendStringToFileFunc, 'AppendStringToFile', ['fileName', cString, 'data', cString], '');
RegisterInternalFunction(TFileExistsFunc, 'FileExists', ['fileName', cString], cBoolean);
RegisterInternalFunction(TDeleteFileFunc, 'DeleteFile', ['fileName', cString], cBoolean);
RegisterInternalFunction(TRenameFileFunc, 'RenameFile', ['oldName', cString, 'newName', cString], cBoolean);
RegisterInternalFunction(TChDirFunc, 'ChDir', ['s', cString], '');
RegisterInternalFunction(TCreateDirFunc, 'CreateDir', ['dir', cString], cBoolean);
RegisterInternalFunction(TRemoveDirFunc, 'RemoveDir', ['dir', cString], cBoolean);
RegisterInternalFunction(TGetCurrentDirFunc, 'GetCurrentDir', [], cString);
RegisterInternalFunction(TSetCurrentDirFunc, 'SetCurrentDir', ['dir', cString], cBoolean);
RegisterInternalFunction(TFileSearchFunc, 'FileSearch', ['name', cString, 'dirList', cString], cString);
RegisterInternalFunction(TExtractFileDriveFunc, 'ExtractFileDrive', ['fName', cString], cString);
RegisterInternalFunction(TExtractFileDirFunc, 'ExtractFileDir', ['fName', cString], cString);
RegisterInternalFunction(TExtractFileNameFunc, '
    ExtractFileName', ['fName', cString], cString);
RegisterInternalFunction(TExtractFilePathFunc, '
    ExtractFilePath', ['fName', cString], cString);
RegisterInternalFunction(TExtractFileExtFunc, '
    ExtractFileExt', ['fName', cString], cString);
RegisterInternalFunction(TChangeFileExtFunc, 'ChangeFileExt
    ', ['fName', cString, 'ext', cString], cString);
```

3.2.5.1. File Utilities - DWS II Reference

SaveStringToFile

Saves string "data" to file "filename".

Note: SaveStringToFile creates the file if it does not exist. Otherwise the file is opened in write mode.

The function might throw an EDelphi exception if the file can not be opened / created.

Declaration

procedure SaveStringToFile(FileName, Data: String);

LoadStringFromFile

Returns contents of file "filename" as string.

File “FileName” is opened using the *fmShareDenyNone* flag.

Note that this function might throw an EDelphi exception, if the file can not be opened.

Declaration

function LoadStringFromFile(FileName: string): String;

AppendStringToFile

Adds string "Data" to file "FileName"

Declaration

procedure AppendStringToFile(FileName, Data: String);

FileExists

FileExists returns True if the file specified by FileName exists. If the file does not exist, FileExists returns False.

Declaration

function FileExists(FileName: string): Boolean;

DeleteFile

DeleteFile deletes the file named by FileName from the disk. If the file cannot be deleted or does not exist, the function returns False.

Declaration

function DeleteFile(FileName: string): Boolean;

RenameFile

RenameFile attempts to change the name of the file specified by OldFile to NewFile. If the operation succeeds, RenameFile returns True. If it cannot rename the file (for example, if a file called NewName already exists), it returns False.

Declaration

function RenameFile(OldName, NewName: string): Boolean;

ChDir

ChDir changes the current directory to the path specified by S.

You may get an EDelphi exception when calling this function and an IOError occurred.

Declaration

procedure ChDir(S: string);

CreateDir

CreateDir creates a new directory. The return value is True if a new directory was successfully created, or False if an error occurred.

Declaration

function CreateDir(Dir: string): Boolean;

RemoveDir

Call RemoveDir to remove the directory specified by the Dir parameter. The return value is True if a new directory was successfully deleted, False if an error occurred. The directory must be empty before it can be successfully deleted.

Declaration

function RemoveDir(Dir: string): Boolean;

GetCurrentDir

GetCurrentDir returns the fully qualified name of the current directory.

Declaration

function GetCurrentDir: string;

SetCurrentDir

The SetCurrentDir function sets the current directory. The return value is True if the current directory was successfully changed, or False if an error occurred.

Declaration

function SetCurrentDir(Dir: string): Boolean;

FileSearch

FileSearch searches through the directories passed in DirList for a file named Name. DirList is a list of path names delimited by semicolons on Windows (colons on Linux). If FileSearch locates a file matching Name, it returns a string specifying a path name for that file. If no matching file exists, FileSearch returns an empty string.

Declaration

function FileSearch(Name, DirList: string): string;

ExtractFileDrive

ExtractFileDrive returns a string containing the drive portion of a fully qualified path name for the file passed in the FileName. For file names with drive letters, the result is in the form '<drive>'. For file names with a UNC path the result is in the form '. If the given path contains neither style of path prefix, the result is an empty string.

Declaration

function ExtractFileDrive(FileName: string): string;

ExtractFileDir

The resulting string is a directory name suitable for passing to the CreateDir, RemoveDir, and SetCurrentDir functions. This string is empty if FileName contains no drive and directory parts.

Declaration

function ExtractFileDir(const FileName: string): string;

ExtractFileName

The resulting string is the rightmost characters of FileName, starting with the first character after the colon or backslash that separates the path information from the name and extension. The resulting string is equal to FileName if FileName contains no drive and directory parts.

Declaration

function ExtractFileName(FileName: string): string;

ExtractFilePath

The resulting string is the leftmost characters of FileName, up to and including the colon or backslash that separates the path information from the name and extension. The resulting string is empty if FileName contains no drive and directory parts.

Declaration

function ExtractFilePath(FileName: string): string;

ExtractFileExt

Use ExtractFileExt to obtain the extension from a file name. For example, the following code returns the extension of the file name specified by a variable named MyFileName:

MyFilesExtension := ExtractFileExt(MyFileName);

The resulting string includes the period character that separates the name and extension parts. This string is empty if the given file name has no extension.

Declaration

function ExtractFileExt(FileName: string): string;

ChangeFileExt

ChangeFileExt takes the file name passed in FileName and changes the extension of the file name to the extension passed in Extension. Extension specifies the new extension, including the initial dot character.

ChangeFileExt does not rename the actual file, it just creates a new file name string.

Declaration

function ChangeFileExt(FileName, Extension: string): string;

3.2.6. GUI Functions (Component)

Often the GUI functions are not needed in script. Due to this fact and the separation of the runtime packages into a version that depends on VCL and one that depends on CLX, the GUI functions are provided in a separate component and not included by default. If you want to use them, simply drop the GUI Functions component onto the form containing the TDelphiWebScriptII component.

HINT

If you use this component:

- You have to redistribute the *dws2VCLRuntime* package - if you want to ship your VCL program using runtime packages.
- Redistribute the *dws2CLXRuntime* package, if you are using the CLX and ship your program with runtime packages.

Of course you also need to ship the *dws2Runtime* package in addition to the package mentioned above.

Listing 3.7: Internal Functions: GUI Functions

```
RegisterInternalFunction(TShowMessageFunc, 'ShowMessage', [
    'msg', cString], '');
RegisterInternalFunction(TInputBoxFunc, 'InputBox', ['
    aCaption', cString, 'aPrompt', cString, 'aDefault',
    cString], cString);
RegisterInternalFunction(TErrorDlgFunc, 'ErrorDlg', ['msg'
    , cString], '');
RegisterInternalFunction(TInformationDlgFunc, '
    InformationDlg', ['msg', cString],'');
RegisterInternalFunction(TQuestionDlgFunc, 'QuestionDlg', [
    'msg', cString], cBoolean);
RegisterInternalFunction(TOkCancelDlgFunc, 'OkCancelDlg', [
    'msg', cString], cBoolean);
```

3.2.6.1. GUI Utilities - DWS II Reference

ShowMessage

Call ShowMessage to display a simple message box with an OK button. The name of the application's executable file appears as the caption of the message box.

Msg parameter is the message string that appears in the message box.

Note: If the user types Ctrl+C in the message box, the text of the message is copied to the clipboard.

Declaration

procedure ShowMessage(Msg: string);

InputBox

Call InputBox to bring up an input dialog box ready for the user to enter a string in its edit box.

ACaption is the caption of the dialog box.

APrompt is the text that prompts the user to enter input in the edit box.

ADefault is the string that appears in the edit box when the dialog box first appears.

If the user chooses the Cancel button, InputBox returns the default string. If the user chooses the OK button, InputBox returns the string in the edit box.

Use the InputBox function when there is a default value that should be used when the user chooses the Cancel button (or presses Esc) to exit the dialog.

Declaration

function InputBox(ACaption, APrompt, ADefault: string): string;

ErrorDlg

Call ErrorDlg to display a simple message box with an OK button. The name of the application's executable file appears as the caption of the message box. The message box contains an error icon.

Msg parameter is the message string that appears in the message box.

Note: If the user types Ctrl+C in the message box, the text of the message is copied to the clipboard.

Declaration

procedure ErrorDlg(Msg: string);

InformationDlg

Call InformationDlg to display a simple message box with an OK button. The name of the application's executable file appears as the caption of the message box. The message box contains an icon showing an exclamation sign.

Msg parameter is the message string that appears in the message box.

Note: If the user types Ctrl+C in the message box, the text of the message is copied to the clipboard.

Declaration

procedure InformationDlg(Msg: string);

QuestionDlg

The message box contains an icon showing a question mark and two buttons "Yes" and "No".

Msg parameter is the message string that appears in the message box.

The return value is true if the "Yes" button was pressed and false otherwise.

Declaration

function QuestionDlg(Msg: string): Boolean;

OkCancelDlg

The message box contains an icon showing a question mark and two buttons "Yes" and "Cancel".

Msg parameter is the message string that appears in the message box.

The return value is true if the "Yes" button was pressed and false otherwise.

Declaration

function OkCancelDlg(Msg: string): Boolean;

3.2.7. Global Variable Functions (Component)

This component implements global variables functions, that allow scripts to read and write to variables across a script's context. Details:

- Variables can be declared and read from any script, or from Delphi code
- Read/Write access is thread-safe
- Variables names are **case sensitive**
- Fast, (reads/writes per second on a 800MHz Duron and a handful of variables: 100k/sec, 1000k with optimization using D5)

The global variables can be saved/restored as a whole only from Delphi code (Delphi code only as of now, mainly for security reasons) to a file, string or stream. Be aware DWS II will require special care to run in a multi-threaded environment.

Listing 3.8: Internal Functions: Global Variable Functions

```

{=> Following functions are defined in DWSII script:}

RegisterInternalFunction(TReadGlobalVarFunc, 'ReadGlobalVar
    ', ['n', cString], cVariant);
RegisterInternalFunction(TReadGlobalVarDefFunc, '
    ReadGlobalVarDef', ['n', cString, 'd', cVariant],
    cVariant);
RegisterInternalFunction(TWriteGlobalVarFunc, '
    WriteGlobalVar', ['n', cString, 'v', cVariant], '');
RegisterInternalFunction(TCleanupGlobalVarsFunc, '
    CleanupGlobalVars', [], '');





{=> Following DELPHI functions are available to access the
    stored variables: }

{Directly write a global var. }
procedure WriteGlobalVar(const aName: string; const aValue
    : Variant);

{Directly read a global var. }
function ReadGlobalVar(const aName: string): Variant;

{Directly read a global var, using a default value if
    variable does not exists.}
```

```
function ReadGlobalVarDef(const aName: string; const
    aDefault: Variant): Variant;

{Resets all global vars.}
procedure CleanupGlobalVars;

{Save current global vars and their values to a file. }
function SaveGlobalVarsToString: string;

{Load global vars and their values to a file. }
procedure LoadGlobalVarsFromString(const srcString: string)
    ;

{Save current global vars and their values to a file. }
function SaveGlobalVarsToFile(const destFileName: string):
    Boolean;

{Load global vars and their values to a file. }
function LoadGlobalVarsFromFile(const srcFileName: string)
    : Boolean;

{Save current global vars and their values to a file. }
procedure SaveGlobalVarsToStream(destStream: TStream);

{Load global vars and their values to a file. }
procedure LoadGlobalVarsFromStream(srcStream: TStream);
```

3.2.7.1. GVar Utilities - DWS II Reference

ReadGlobalVar

Returns the value of the global variable *n*.
This function returns an empty variant, if the variable can not be found.

Declaration

function ReadGlobalVar(n: String): Variant;

ReadGlobalVarDef

Similar to ReadGlobalVar, except that *d* is given back in case the variable specified by n could not be found.

Declaration

function ReadGlobalVarDef(n: String; d: Variant): Variant;

WriteGlobalVar

Assigns value of v to the global variable named *n*.
WriteGlobalVar creates a new global variable (with name *n*), if it can not find an existing global variable with the name specified by n.

Declaration

procedure WriteGlobalVar(n: String; v: Variant);

CleanupGlobalVars

Deletes and frees all global variables.

Declaration

procedure CleanupGlobalVars;

3.3. Strings Unit

str

dws2StringsUnit1

ATTENTION

Dws2StringsUnit demands that TDelphiWebScriptII's resulttype is set to *dws2StringResultType*!
In other words: To use this component, you also have to drop the "RES str" labeled *dws2StringResultType* component onto the form that contains the TDelphiWebScriptII component **and** you have to set the *TDelphiWebScriptII.Config.ResultType* property to the *dws2StringResultType* component.

Listing 3.9: Strings Unit: Functions

```
procedure Tdws2StringsUnit.AddUnitSymbols(SymbolTable:
    TSymbolTable);
var
  emptyArg: array of string;
begin
  TWriteFunction.Create(SymbolTable, 'WriteStr', ['Str',
      SYS_VARIANT], '');
  TWriteLnFunction.Create(SymbolTable, 'WriteLn', ['Str',
      SYS_VARIANT], '');
  TWriteAllFunction.Create(SymbolTable, 'WriteAll', ['Str'
      , SYS_VARIANT], '');
  SetLength(emptyArg, 0);
  TReadCharFunction.Create(SymbolTable, 'ReadChar',
      emptyArg, SYS_STRING);
  TReadLnFunction.Create(SymbolTable, 'ReadLn', emptyArg,
      SYS_STRING);
  TReadAllFunction.Create(SymbolTable, 'ReadAll', emptyArg
      , SYS_STRING);
end;
```

The string unit introduces following procedures:

- WriteStr(Str: Variant);
- WriteLn(Str: Variant);

- WriteAll(Str: Variant);

Additionally following functions are available:

- ReadChar: String;
- ReadLn: String;
- ReadAll: String;

3.4. Classes Library

dws2ClassesLib1

The classes library defines wrappers for commonly used Delphi classes. Listing 3.10 shows all types that are defined by this library.

NOTE

The listing below was created with the help of the *dws2UnitEditor*. You can find this utility in the *dws2tools/UnitCompEditor* module.
Note: This component is still in development, thus not 100% stable.

Listing 3.10: Standard Library: Classes Library

```
{ Forward Declarations }
type TStringList = class;

{ Constants }
const dupAccept : Integer = 2;
const dupError : Integer = 1;
const dupIgnore : Integer = 0;

{ Classes }
type
  TList = class(TObject)
    constructor Create;
    function Add(Obj : TObject) : Integer;
    procedure Clear;
    function Count : Integer;
    procedure Delete(Index : Integer);
    destructor Destroy; override;
    function GetItems(Index : Integer) : TObject;
    function IndexOf(Obj : TObject) : Integer;
    procedure Insert(Index : Integer; Obj : TObject);
    function Remove(Obj : TObject) : Integer;
    procedure SetItems(Index : Integer; Value : TObject);
    property Items[x : Integer]: TObject read GetItems
        write SetItems; default;
  end;

type
  TStrings = class(TObject)
    constructor Create;
    function Add(Str : String) : Integer;
    function AddObject(S : String; AObject : TObject) :
        Integer;
    procedure AddStrings(Strings : TStringList);
    procedure Clear;
    procedure Delete(Index : Integer);
    destructor Destroy; override;
    procedure Exchange(Index1 : Integer; Index2 : Integer);
    function Get(Index : Integer) : String;
    function GetCommaText : String;
    function GetCount : Integer;
    function GetNames(s : String) : String;
    function GetObjects(Index : Integer) : TObject;
    function GetStrings(Index : Integer) : String;
    function GetText : String;
    function GetValues(Str : String) : String;
    function IndexOf(Str : String) : Integer;
    function IndexOfName(Str : String) : Integer;
    function IndexOfObject(AObject : TObject) : Integer;
    procedure Insert(Index : Integer; Str : String);
    procedure InsertObject(Index : Integer; S : String;
        AObject : TObject);
    procedure LoadFromFile(FileName : String);
    procedure Move(CurIndex : Integer; NewIndex : Integer);
    procedure SaveToFile(FileName : String);
    procedure SetCommaText(Value : String);
    procedure SetObjects(Index : Integer; Value : TObject);
    procedure SetStrings(Index : Integer; Value : String);
    procedure SetText(Value : String);
    procedure SetValues(Str : String; Value : String);
    property CommaText: String read GetCommaText write
        SetCommaText;
    property Count: Integer read GetCount;
    property Names[s : String]: String read GetNames;
    property Objects[x : Integer]: TObject read GetObjects
        write SetObjects;
    property Strings[x : Integer]: String read GetStrings
        write SetStrings; default;
    property Text: String read GetText write SetText;
    property Values[s : String]: String read GetValues
        write SetValues;
  end;

type
  TStringList = class(TStrings)
    function Find(S : String; var Index : Integer) :
        Boolean;
    function GetDuplicates : Integer;
    function GetSorted : Boolean;
    procedure SetDuplicates(Value : Integer);
    procedure SetSorted(Value : Boolean);
    procedure Sort;
    property Duplicates: Integer read GetDuplicates write
        SetDuplicates;
    property Sorted: Boolean read GetSorted write SetSorted
        ;
  end;

type
  THashtable = class(TObject)
    function Capacity : Integer;
    procedure Clear;
    function Size : Integer;
  end;

type
  TIntegerHashtable = class(THashtable)
    constructor Create;
    destructor Destroy; override;
    function Get(Key : Integer) : TObject;
    function HasKey(Key : Integer) : Boolean;
    procedure Put(Key : Integer; Value : TObject);
    function RemoveKey(Key : Integer) : TObject;
  end;

type
  TStringHashtable = class(THashtable)
    constructor Create;
    destructor Destroy; override;
    function Get(Key : String) : TObject;
    function HasKey(Key : String) : Boolean;
    procedure Put(Key : String; Value : TObject);
    function RemoveKey(Key : String) : TObject;
  end;

type
  TStack = class(TObject)
    constructor Create;
    function Count : Integer;
    destructor Destroy; override;
    function Peek : TObject;
    function Pop : TObject;
    procedure Push(Obj : TObject);
  end;

type
  TQueue = class(TObject)
    constructor Create;
    function Count : Integer;
    destructor Destroy; override;
    function Peek : TObject;
    function Pop : TObject;
    procedure Push(Obj : TObject);
  end;
```

We won't cover most of the types and methods here, because they are commonly used in Delphi and rely on the behavior of the equivalent native Delphi classes. Please consult your Delphi help, if you have questions regarding these types.

THashTable, *TIntegerHashTable* and *TStringHashTable* provide script access to the types defined in the *dws2Hashtables* DWS II Delphi unit. You can find this file in the "Source" directory of your DWS II installation.

THashTable

THashTable is the base class for all DWS II HashTable classes.
Following methods are defined:

function *Size: Integer;*
Returns the number of stored items.

function *Capacity: Integer;*
Returns the number of items that may be stored without having to re-allocate memory.

procedure *Clear;*
Clears the HashTable.

TIntegerHashTable

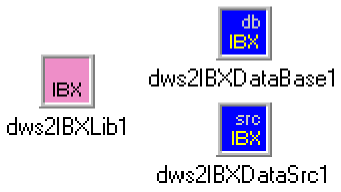
Derived from THashTable, TIntegerHashTable can store any kind of objects that use an integer value as hash key.

All methods should be self explanatory, so we do not cover them here.

TStringHashTable

Use TStringHashTable to store objects that have strings as hash keys.

3.5. Database Libraries (IBX, IBO)



Especially for web applications it is crucial to have fast, reliable and flexible access to databases. The IBX and IBO Libraries shipping with DWS II implement several features that enable DWS II to fulfill this demand.

NOTE Consider the functionality IBXLib offers as DWS II database connectivity standard.
The IBO Library sticks to that standard, so whenever you are reading "IBX" here, you can think of "IBX and IBO".

3.5.1. dws2IBXDataBase, dws2IBODataBase

Component to add more TIB_Connections if you use more than one database. DWS Queries and Statements can be created with database as parameter.

3.5.2. dws2IBXDataSrc, dws2IBODataSrc

Component to add a predefined TIB_Query, TIB_Cursor or TIB_Statement as script object that can be used in the script without creation or prepare.

3.5.3. dws2IBXLib, dws2IBOLib

Listing 3.11: Standard Library: IBX / IBO Library

```
{ Forward Declarations }
type TDataset = class;

{ Classes }
type
  TDatabase = class(TObject)
    constructor create(database : String; user : String;
       pwd : String);
    procedure connect;
    procedure disconnect;
    function getcharset : String;
    function getdialect : Integer;
    procedure setcharset(sCharset : String);
    procedure setdialect(iDialect : Integer);
    property charset: String read getcharset write
        setcharset;
    property dialect: Integer read getdialect write
        setdialect;
  end;

type
  TCustomDBField = class(TObject)
    function GetValue : Variant; virtual;
    function GetValueStr : String; virtual;
    procedure SetValue(Value : Variant); virtual;
    procedure SetValueStr(Value : String); virtual;
    property AsString: String read GetValueStr write
        SetValueStr;
  end;

type
  TDBField = class(TCustomDBField)
    function GetDateTime : DateTime;
    function GetFloat : Float;
    function GetInteger : Integer;
    procedure SetDateTime(Value : DateTime);
    procedure SetFloat(Value : Float);
    procedure SetInteger(Value : Integer);
    property AsDateTime: DateTime read GetDateTime write
        SetDateTime;
    property AsFloat: Float read GetFloat write SetFloat;
    property AsInteger: Integer read GetInteger write
        SetInteger;
    property AsString: String read GetValueStr write
        SetValueStr;
    property Value: Variant read GetValue write SetValue;
  end;

type
  TLUField = class(TCustomDBField)
    function GetDateTime : DateTime;
    function GetFloat : Float;
    function GetInteger : Integer;
    function GetValue : Variant; override;
    function GetValueStr : String; override;
    procedure SetDateTime(Value : DateTime);
    procedure SetFloat(Value : Float);
    procedure SetInteger(Value : Integer);
    procedure SetValue(Value : Variant); override;
    procedure SetValueStr(Value : String); override;
    property AsDateTime: DateTime read GetDateTime write
        SetDateTime;
    property AsFloat: Float read GetFloat write SetFloat;
    property AsInteger: Integer read GetInteger write
        SetInteger;
    property AsString: String read GetValueStr write
        SetValueStr;
    property Value: Variant read GetValue write SetValue;
  end;

type
  TMLUField = class
    constructor create(DataSet : TDataset; MLUFieldName :
        String);
    destructor free;
    function GetValue(Name : String) : String;
    procedure SetNameValue(Name : String; Value : String);
  end;

type
  TStatement = class
    constructor Create;
    constructor CreateFromDB(database : TDatabase); virtual
        ;
    procedure Execute;
    function Field(FieldName : String) : Variant;
    function FieldByName(FieldName : String) : TDBField;
    function FieldIsNull(FieldName : String) : Boolean;
    destructor Free;
    function GetSQL : String;
    procedure Open; virtual;
    function ParamByName(ParamName : String) : TDBField;
    procedure SetParam(ParamName : String; Value : Variant)
        ;
    procedure SetSQL(sSQL : String);
    property SQL: String read GetSQL write SetSQL;
  end;

type
  TDataset = class(TStatement)
    constructor Create;
    constructor CreateFromDB(Database : TDatabase);
        override;
    procedure Cancel;
    procedure Close;
    procedure Delete;
    procedure Edit;
    function Eof : Boolean;
    function First : Boolean;
    procedure Insert;
    function Next : Boolean;
    procedure Open; override;
    procedure Post;
  end;

type
  TQuery = class(TDataset)
    constructor Create;
    constructor CreateFromDB(DataBase : TDatabase);
        override;
    function GetFilter : String;
    function GetFiltered : Boolean;
    function GetSortOrder : Integer;
    function LookUp(KeyFieldValue : String) :
        TCustomDBField;
    function Prior : Boolean;
    procedure SetFilter(FilterStr : String);
    procedure SetFiltered(Filtered : Boolean);
    procedure SetLookUpFields(KeyFieldName : String;
        LUfieldName : String);
    procedure SetSortOrder(SortOrder : Integer);
    property Filter: String read GetFilter write SetFilter;
    property Filtered: Boolean read GetFiltered write
        SetFiltered;
    property SortOrder: Integer read GetSortOrder write
        SetSortOrder;
  end;

type
  TDataSetGrp = class
    constructor Create(dataSet : TDataset; groupFieldName
         : String);
    procedure AddRow;
    procedure AddSumField(FieldName : String);
    function Changed : Boolean;
    function Count : Integer;
    destructor Free;
    procedure Reset;
    procedure Restart;
    function SumOfField(FieldName : String) : Float;
  end;
```

- LookUpField : special for web apps - define a master-detail table relation, you can lookup details belonging to one master dataline (like with name=value in stringlists) when "name" field is string.
- MLUField : MemoLookUpField .. special for web apps - define a MemoField, you can lookup values with name=value from stringlist
- DataSetGrp : DataSetGroup, special for reports - if you iterate through a dataset you can use DataSetGrp to watch changing of single field values, calculate group sums and total sums (like all incomes listed per person and totals per person....).

TDataSetGrp

- .create(DataSet: TDataset; GroupFieldName: String);
  // DataSet should be prepared;!!
- .Free;
- .Changed: boolean;
  will be false while the content of the GroupField stays the same while iterating through this dataset
- .AddSumField(FieldName: String);
  after create, before start iterating, add all fieldnames of fields which should be added in the group or total
- .SumOfField(FieldName: String): float;
  returns actual value of added field values. is set to 0 with "reset"
- .Count: integer;
  returns the number of datarecords in this group.
- .AddRow;
  adds values of all fields
- .Restart;
  reset values for a new group, DOES NOT RESET FIELD SUMS
- .Reset;
  reset values AND FIELD SUMS for a new group

Example:

Listing 3.12: Standard Library: IBX / IBO Library - DataSetGrp Example

```
{
dws2IBO : TDataSetGrp Demo
}
```

```

var qAmounts: TDataset;
var grpEmployee : TDataSetGrp;
qAmounts := TDataset.Create;
qAmounts.SQL := 'select * from empl_amounts'; // wages of
    employees
qAmounts.open;
grpEmployee := TDataSetGrp.create(qAmounts,'name');
grpEmployee.AddSumField('amount');
while not qAmounts.Eof do
begin
  Print('name: ');
  PrintLn(qAmounts.Field('name'));
  while not grpEmployee.changed do
  begin
    Print(qAmounts.Fieldbyname('bill_date').value);
    Print(', ');
    PrintLn(qAmounts.Field('amount'));
    grpEmployee.addrow;
    qAmounts.Next;
    i := i+1;
  end;
  Print(grpEmployee.count);
  Print('-----------, sum: ');
  PrintLn(grpEmployee.SumOfField('amount'));
  Println('');
  grpEmployee.reset;
end;
```

3.6. Fiber Library

FIB
DWSII_FiberLibraryExtension1

The Fiber Library enables the use of control scripts that "pause". It was intended for OpenGL (GLScene) scripting, with a *runall* being called(between rendering) each 100 ms or so...
For further details, see the included demo. You can find it in the FiberLibrary directory.

Listing 3.13: Standard Library: Fiber

```
procedure FiberMessage(MessageText: String);
function GetFiberData(Name: String): String;
procedure NextFiber();
procedure SetFiberData(Name: String; Value: String);
```

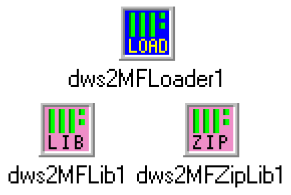
3.7. Ini Library

dws2IniLib1

Listing 3.14: Standard Library: Ini

```
{ Classes }
type
  TIniFile = class
    constructor Create(FileName : String);
    procedure DeleteKey(Section : String; Ident : String);
    destructor Destroy; override;
    procedure EraseSection(Section : String);
    function GetFilename : String;
    function ReadBool(Section : String; Ident : String;
        Default : Boolean) : Boolean;
    function ReadDate(Section : String; Name : String;
        Default : DateTime) : DateTime;
    function ReadDateTime(Section : String; Name : String;
        Default : DateTime) : DateTime;
    function ReadFloat(Section : String; Name : String;
        Default : Float) : Float;
    function ReadInteger(Section : String; Ident : String;
        Default : Integer) : Integer;
    procedure ReadSection(Section : String; Strings :
        TStringList);
    procedure ReadSections(Strings : TStringList);
    procedure ReadSectionValues(Section : String; Strings
         : TStringList);
    function ReadString(Section : String; Ident : String;
        Default : String) : String;
    function ReadTime(Section : String; Name : String;
        Default : DateTime) : DateTime;
    function SectionExists(Section : String) : Boolean;
    procedure UpdateFile;
    function ValueExists(Section : String; Ident : String)
        : Boolean;
    procedure WriteBool(Section : String; Ident : String;
        Value : Boolean);
    procedure WriteDate(Section : String; Name : String;
        Value : DateTime);
    procedure WriteDateTime(Section : String; Name : String
        ; Value : DateTime);
    procedure WriteFloat(Section : String; Name : String;
        Value : Float);
    procedure WriteInteger(Section : String; Ident : String
        ; Value : Integer);
    procedure WriteString(Section : String; Ident : String
        ; Value : String);
    procedure WriteTime(Section : String; Name : String;
        Value : DateTime);
    property Filename: String read GetFilename;
  end;
```

3.8. MF Library



There are **lots** of functions and constants defined in MF Library. Most of them, however, are only meaningful on Windows.

We provide here a listing of all functions and constants defined by this Library! Don't expect them to be documented here - this would simply blow the 200 pages border...

MFLib also comes with some demos!

HINT: Please use the documentation shipping with Delphi if you have questions regarding one of the constants/functions/classes below!

3.8.1. dws2MFLoader

This component enables you to load encrypted DWS II scripts.

3.8.2. Basic

3.8.2.1. Functions

Listing 3.15: Standard Library: MFLib - Basic

```
procedure Beep; forward;
procedure Dec(var I : Integer); forward;
procedure Dec2(var I : Integer; N : Integer); forward;
procedure Inc(var I : Integer); forward;
procedure Inc2(var I : Integer; N : Integer); forward;
function GetTickCount : Integer; forward;
procedure Sleep(mSecs : Integer); forward;
procedure WriteLn(Text : String); forward;
```

3.8.3. Connection

3.8.3.1. Functions

Listing 3.16: Standard Library: MFLib - Connection

```
{ Functions }
function Connected : Boolean; forward;
function IPAddress : String; forward;
```

3.8.4. Dialog

3.8.4.1. Functions

Listing 3.17: Standard Library: MFLib - Dialog

```
function SelectStringDialog(Title: String; Strings:
    TStringList; Selected: Integer): Integer;
```

3.8.5. File

3.8.5.1. Constants

- DRIVE_UNKNOWN
- DRIVE_NO_ROOT_DIR
- DRIVE_REMOVABLE
- DRIVE_FIXED
- DRIVE_REMOTE
- DRIVE_CDROM
- DRIVE_RAMDISK
- FILEDATE_CREATION
- FILEDATE_LASTACCESS
- FILEDATE_LASTWRITE
- MOVEFILE_REPLACE_EXISTING
- MOVEFILE_COPY_ALLOWED
- MOVEFILE_DELAY_UNTIL_REBOOT
- MOVEFILE_WRITE_THROUGH
- MOVEFILE_CREATE_HARDLINK
- MOVEFILE_FAIL_IF_NOT_TRACKABLE

3.8.5.2. Functions

Listing 3.18: Standard Library: MFLib - File

```
function DescCopy(Source, Target: String): Boolean;
function DescListCreate(Dir: String): TStringList;
function DescListRead(List: TStringList; FileName: String)
    : String;
```

```
function DescMove(Source, Target: String): Boolean;
function DescRead(Filename: String): String;
function DescWrite(FileName, Desc: String): Boolean;
function OpenDialog(FileName, InitialDir, Title, DefaultExt
    , Filter: String; FilterIndex: Integer): String;
function OpenDialogMulti(FileName, InitialDir, Title,
    DefaultExt, Filter: String; FilterIndex: Integer):
    TStringList;
function SaveDialog(FileName, InitialDir, Title, DefaultExt
    , Filter: String; FilterIndex: Integer): String;
function CDClose(Drive: Integer): Boolean;
function CDOpen(Drive: Integer): Boolean;
function GetCRC32FromFile(FileName: String): Integer;
function GetDriveName(Drive: Integer): String;
function GetDriveNum(Drive: String): Integer;
function GetDriveReady(Drive: Integer): Boolean;
function GetDriveSerial(Drive: Integer): String;
function GetDriveType(Drive: Integer): Integer;
function DirectoryExists(DirName: String): Boolean;
function DirectoryList(DirName: String; Recurse, Hidden):
    TStringList;
function CopyFile(Source, Target: String; Fail: Boolean):
    Boolean;
function FileDate(FileName: String; Flag: Integer):
    DateTime;
function FileList(FileName: String; Recurse, Hidden, Dirs:
    Boolean): TStringList;
function FileSize(FileName: String): Integer;
function MakePath(Drive, Dir, Name, Ext: String): String;
function MoveFile(Source, Target: String): Boolean;
function MoveFileEx(Source, Target: String; Flags: Integer)
    : Boolean;
function ReadOnlyPath(Path: String): Boolean;
procedure SplitPath(Path: String; var Drive, Dir, Name, Ext
    : String);
function SubdirectoryExists(Dir: String): Boolean;
```

3.8.6. Info

3.8.6.1. Constants

- VER_UNKNOWN
- VER_WIN32S
- VER_WIN95

- VER_WIN98
- VER_WIN98SE
- VER_WINME
- VER_WINNT
- VER_WINNT4
- VER_WIN2000

3.8.6.2. Functions

Listing 3.19: Standard Library: MFLib - Info

```
function GetAllUsersDesktopDirectory: String;
function GetAllUsersProgramsDirectory: String;
function GetAllUsersStartmenuDirectory: String;
function GetAllUsersStartupDirectory: String;
function GetAppdataDirectory: String;
function GetCacheDirectory: String;
function GetChannelFolderName: String;
function GetCommonFilesDirectory: String;
function GetComputerName: String;
function GetConsoleTitle: String;
function GetCookiesDirectory: String;
function GetCPUSpeed: String;
function GetDesktopDirectory: String;
function GetDevicePath: String;
function GetEnvironmentVariable: String;
function GetFavoritesDirectory: String;
function GetFontsDirectory: String;
function GetHistoryDirectory: String;
function GetLinkfolderName: String;
function GetMediaPath: String;
function GetNethoodDirectory: String;
function GetPersonalDirectory: String;
function GetPFAccessoriesName: String;
function GetPrinthoodDirectory: String;
function GetProgramfilesDirectory: String;
function GetProgramsDirectory: String;
function GetRecentDirectory: String;
function GetSendtoDirectory: String;
function GetSMAccessoriesName: String;
function GetStartmenuDirectory: String;
function GetStartupDirectory: String;
function GetSystemDirectory: String;
function GetTempDirectory: String;
function GetTemplatesDirectory: String;
function GetUserName: String;
function GetVersion: String;
function GetWallpaperDirectory: String;
function GetWindowsDirectory: String;
function GetWindowsVersion: String;
function IsWin2000: Boolean;
function IsWin9x: Boolean;
function IsWinNT: Boolean;
function IsWinNT4: Boolean;
function SetComputerName: String;
function SetConsoleTitle: String;
```

3.8.7. Registry

3.8.7.1. Constants

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_PERFORMANCE_DATA
- HKEY_CURRENT_CONFIG
- HKEY_DYN_DATA

3.8.7.2. Functions

Listing 3.20: Standard Library: MFLib - Registry

```
procedure RegCreateKey(MainKey: Integer; SubKey: String);
procedure RegDeleteKey(MainKey: Integer; SubKey: String);
procedure RegDeleteValue(MainKey: Integer; SubKey: String;
    ValName: String);
function RegReadInteger(MainKey: Integer; SubKey: String;
    ValName: String; ValDef: Integer): Integer;
function RegReadString(MainKey: Integer; SubKey: String;
    ValName: String; ValDef: String): String;
```

```
function RegGetType(MainKey: Integer; SubKey: String;
    ValName: String; var Size: Integer): Integer;
function RegKeyExists(MainKey: Integer; SubKey: String):
    Boolean;
function RegValueExists(MainKey: Integer; SubKey: String;
    ValName: String): Boolean;
procedure RegWriteInteger(MainKey: Integer; SubKey: String
    ; ValName: String; Value: Integer);
procedure RegWriteString(MainKey: Integer; SubKey: String;
    ValName: String; Value: String);
```

3.8.8. Shell

3.8.8.1. Functions

Listing 3.21: Standard Library: MFLib - Shell

```
DesktopRefresh;
```

3.8.9. String

3.8.9.1. Constants

- cdrLeft
- cdrRight

3.8.9.2. Functions

Listing 3.22: Standard Library: MFLib - String

```
function ANSI2OEM(S: String): String;
function ChangeTokenValue(S: String; Name: String; Value:
    String; Delimiter: String): String;
function Chr(Byte: Integer): String;
function CmpRE(S: String; RE: String): Boolean;
function CmpWC(S: String; WC: String; Case: Boolean):
    Boolean;
```

```
function Crop(S: String; Len: Integer; Dir: Integer;
    Delimiter: String)
function ForEach(List: TStringList; Func: String; Flag:
    Integer): Boolean;
procedure FormatColumns(List: TStringList; Delimiter:
    String; Space: String; Adjustment: Integer);
function GetCRC32FromString(S: String): Integer;
function GetStringFromList(List: TStringList; Delimiter:
    String): String;
function GetTokenList(S: String; Delimiter: String;
    Repeater: Boolean; IgnoreFirst: Boolean; IgnoreLast:
    Boolean): TStringList;
function IncWC(S: String; WC: String; Case: Boolean):
    String;
function Num2Text(Num: Integer): String;
function OEM2ANSI(S: String): String;
function Ord(Char: String): Integer;
function PosX(SubStr: String; S: String): Integer;
function StringOfChar(Ch: String; Count: Integer): String;
function TestWC(S: String): Integer;
function Translate(S: String; Out: String; In: String;
    Place: String; Case: Boolean): String;
```

3.8.10. System

3.8.10.1. Constants

- SW_HIDE
- SW_SHOWNORMAL
- SW_NORMAL
- SW_SHOWMINIMIZED
- SW_SHOWMAXIMIZED
- SW_MAXIMIZE
- SW_SHOWNOACTIVE
- SW_SHOW
- SW_MINIMIZE
- SW_SHOWMINNOACTIVE

- SW_SHOWNA
- SW_RESTORE
- SW_SHOWDEFAULT
- EWX_LOGOFF
- EWX_SHUTDOWN
- EWX_REBOOT
- EWX_FORCE
- EWX_POWEROFF
- EWX_FORCEIFHUNG
- Message-Constants for PostMessage() and SendMessage():
  - WM_MOVE
  - WM_SIZE
  - WM_ACTIVATE
  - WM_SETFOCUS
  - WM_KILLFOCUS
  - WM_ENABLE
  - WM_CLOSE
  - WM_QUIT
  - WM_SHOWWINDOW
  - WM_ACTIVATEAPP
  - WM_SETCURSOR
  - WM_MOUSEACTIVATE
  - WM_CHILDACTIVATE
  - WM_HELP
  - WM_COMMAND
  - WM_SYSCOMMAND
  - WM_HSCROLL
  - WM_VSCROLL
  - WM_MOUSEMOVE
  - WM_LBUTTONDOWN
  - WM_LBUTTONUP

- WM_LBUTTONDBLCLK
- WM_RBUTTONDOWN
- WM_RBUTTONUP
- WM_RBUTTONDBLCLK
- WM_MBUTTONDOWN
- WM_MBUTTONUP
- WM_MBUTTONDBLCLK
- WM_MOUSEWHEEL
- WM_MDITILE
- WM_MDICASCADE
- WM_CUT
- WM_COPY
- WM_PASTE
- WM_CLEAR
- WM_UNDO
- WM_USER
- BM_CLICK
- BM_GETCHECK
- BM_SETCHECK
- BST_CHECKED
- BST_INDETERMINATE
- BST_UNCHECKED
- CB_GETCOUNT
- CB_GETCURSEL
- CB_GETDROPPEDSTATE
- CB_GETEDITSEL
- CB_GETTOPINDEX
- CB_RESETCONTENT
- CB_SETCURSEL
- CB_SETEDITSEL
- CB_SETTOPINDEX
- CB_SHOWDROPDOWN
- EM_GETSEL
- EM_SETSEL

- EM_UNDO
- LB_GETCOUNT
- LB_GETCURSEL
- LB_GETSEL
- LB_GETSELCOUNT
- LB_GETTEXT
- LB_GETTEXTLEN
- LB_GETTOPINDEX
- LB_ITEMFROMPOINT
- LB_RESETCONTENT
- LB_SELITEMRANGE
- LB_SETCURSEL
- LB_SETSEL
- LB_SETTOPINDEX
- SB_BOTTOM
- SB_ENDSCROLL
- SB_LINEDOWN
- SB_LINEUP
- SB_PAGEDOWN
- SB_PAGEUP
- SB_THUMBPOSITION
- SB_THUMBTRACK
- SB_TOP

3.8.10.2. Functions

Listing 3.23: Standard Library: MFLib - System

```
function ShellExecute(Operation: String; Filename: String;
    Parameters: String; Directory: String; ShowCmd: Integer)
    : Integer;
function ShellExecuteWait(Filename: String; Parameters:
    String; Directory: String; ShowCmd: Integer): Boolean;
function ExitWindowsEx(Flags: Integer): Integer;
function WriteMailslot(Machine: String; Mailslot: String;
    Text: String): Boolean;
function GetProcessList: TStringList;
function HiWord(Value: Integer): Integer;
function IsFileActive(FileName: String): Boolean;
function KillProcess(Window: Integer; FileName: String;
    KillAll: Boolean): Boolean;
function LoWord(Value: Integer): Integer;
function MakeLong(Low, High: Integer): Integer;
function PostMessage(Window, Msg, WParam, LParam: Integer)
    : Boolean;
procedure SendKeys(Keys: String);
procedure SendKeysEx(Keys: String; Wait: Integer);
function SendKeysNamedWin(Window: String; Keys: String):
    Integer;
function SendKeysNamedWinEx(Window: String; Keys: String;
    Wait: Integer; Back: Boolean): Integer;
function SendKeysWin(Window: Integer; Keys: String):
    Integer;
function SendKeysWinEx(Window: Integer; Keys: String; Wait
    : Integer; Back: Boolean): Integer;
function SendMessage(Window, Msg, WParam, LParam: Integer)
    : Integer;
```

3.8.11. Window

3.8.11.1. Functions

Listing 3.24: Standard Library: MFLib - Window

```
function FindWindow(Class: String; Window: String): Integer
    ;
function FindWindowEx(Parent: Integer; ChildAfter: Integer
    ; Class: String; Window: String): Integer;
function GetClassName(Window: Integer): String;
function GetWindowText(Window: Integer): String;
procedure HideTaskbar;
function IsIconic(Window: Integer): Boolean;
function IsWindow(Window: Integer): Boolean;
function IsWindowEnabled(Window: Integer): Boolean;
function IsWindowVisible(Window: Integer): Boolean;
function IsZoomed(Window: Integer): Boolean;
function SearchWindow(var Class; var Window: String; ProcID
    : Integer): Integer;
function SearchWindowEx(Parent: Integer; ChildAfter:
    Integer; var Class; var Window: String; Num: Integer):
    Integer;
procedure ShowTaskbar;
```

```
function WaitForWindow(var Class; var Window: String;
    Timeout: Integer; ProcID: Integer): Integer;
function WaitForWindowClose(Window: Integer; Timeout:
    Integer): Boolean;
function WaitForWindowClose(Class: String; Window: String;
    Timeout: Integer; ProcID: Integer): Boolean;
function WaitForWindowEnabled(Window: Integer; Timeout:
    Integer): Boolean;
function WaitForWindowEnabled(Class: String; Window: String
    ; Timeout: Integer; ProcID: Integer): Boolean;
function WaitForWindowEx(Parent: Integer; var Class; var
    Window: String; Timeout: Integer; Num: Integer): Integer
    ;
procedure WindowMove(Window: Integer; X: Integer; Y:
    Integer; Abs: Boolean);
procedure WindowResize(Window: Integer; X: Integer; Y:
    Integer; Abs: Boolean);
```

3.8.12. dws2MFZipLib

3.8.12.1. Constants

- Zip_Add
- Zip_Move
- Zip_Update
- Zip_Freshen
- Zip_Extract
- Zip_Test
- Zip_OverwriteConfirm
- Zip_OverwriteAlways
- Zip_OverwriteNever
- Zip_EraseFinal
- Zip_EraseAllowUndo

3.8.12.2. Classes

Listing 3.25: Standard Library: MFLib - Zip

```
{ Classes }
type
  TZip = class
    function Add(Action : Integer; ZipFile : String;
       FileName : String) : Integer;
    function AddList(Action : Integer; ZipFile : String;
       FileNames : TStringList) : Integer;
    constructor Create;
    function Delete(ZipFile : String; FileName : String) :
       Integer;
    function DeleteList(ZipFile : String; FileNames :
       TStringList) : Integer;
    destructor Destroy;
    function Extract(Action : Integer; ZipFile : String;
       FileName : String; BaseDir : String) : Integer;
    function ExtractList(Action : Integer; ZipFile : String
       ; FileNames : TStringList; BaseDir : String) :
       Integer;
    function List(ZipFile : String) : TStringList;
    function Message(ZipFile : String) : String;
    function ReadAddCompLevel : Integer;
    function ReadAddDirNames : Boolean;
    function ReadAddDiskSpan : Boolean;
    function ReadAddDiskSpanErase : Boolean;
    function ReadAddEncrypt : Boolean;
    function ReadAddHiddenFiles : Boolean;
    function ReadAddRecurseDirs : Boolean;
    function ReadAddSeparateDirs : Boolean;
    function ReadAddZipTime : Boolean;
    function ReadConfirmErase : Boolean;
    function ReadExtrDirNames : Boolean;
    function ReadExtrOverwrite : Boolean;
    function ReadHowToDelete : Integer;
    function ReadKeepFreeOnDisk1 : Integer;
    function ReadMaxVolumeSize : Integer;
    function ReadMinFreeVolumeSize : Integer;
    function ReadPassword : String;
    function ReadSFXAskCmdLine : Boolean;
    function ReadSFXAskFiles : Boolean;
    function ReadSFXCaption : String;
    function ReadSFXCommandLine : String;
    function ReadSFXDefaultDir : String;
    function ReadSFXHideOverwriteBox : Boolean;
    function ReadSFXOverwriteMode : Integer;
    function ReadSpan(SpanFile : String; var ZipFile :
        String) : Integer;
    function ReadTemp : String;
    function SFX2ZIP(ZipFile : String) : Integer;
    procedure WriteAddCompLevel(Value : Integer);
    procedure WriteAddDirNames(Value : Boolean);
    procedure WriteAddDiskSpan(Value : Boolean);
    procedure WriteAddDiskSpanErase(Value : Boolean);
    procedure WriteAddEncrypt(Value : Boolean);
    procedure WriteAddHiddenFiles(Value : Boolean);
    procedure WriteAddRecurseDirs(Value : Boolean);
    procedure WriteAddSeparateDirs(Value : Boolean);
    procedure WriteAddZipTime(Value : Boolean);
    procedure WriteConfirmErase(Value : Boolean);
    procedure WriteExtrDirNames(Value : Boolean);
    procedure WriteExtrOverwrite(Value : Boolean);
    procedure WriteHowToDelete(Value : Integer);
    procedure WriteKeepFreeOnDisk1(Value : Integer);
    procedure WriteMaxVolumeSize(Value : Integer);
    procedure WriteMinFreeVolumeSize(Value : Integer);
    procedure WritePassword(Value : String);
    procedure WriteSFXAskCmdLine(Value : Boolean);
    procedure WriteSFXAskFiles(Value : Boolean);
    procedure WriteSFXCaption(Value : String);
    procedure WriteSFXCommandLine(Value : String);
    procedure WriteSFXDefaultDir(Value : String);
    procedure WriteSFXHideOverwriteBox(Value : Boolean);
    procedure WriteSFXOverwriteMode(Value : Integer);
    function WriteSpan(ZipFile : String; SpanFile : String)
         : Integer;
    procedure WriteTemp(Value : String);
    function ZIP2SFX(ZipFile : String) : Integer;
    property AddCompLevel: Integer read ReadAddCompLevel
        write WriteAddCompLevel;
    property AddDirNames: Boolean read ReadAddDirNames
        write WriteAddDirNames;
    property AddDiskSpan: Boolean read ReadAddDiskSpan
        write WriteAddDiskSpan;
    property AddDiskSpanErase: Boolean read
        ReadAddDiskSpanErase write WriteAddDiskSpanErase;
    property AddEncrypt: Boolean read ReadAddEncrypt write
        WriteAddEncrypt;
    property AddHiddenFiles: Boolean read
        ReadAddHiddenFiles write WriteAddHiddenFiles;
    property AddRecurseDirs: Boolean read
        ReadAddRecurseDirs write WriteAddRecurseDirs;
    property AddSeparateDirs: Boolean read
        ReadAddSeparateDirs write WriteAddSeparateDirs;
    property AddZipTime: Boolean read ReadAddZipTime write
        WriteAddZipTime;
    property ConfirmErase: Boolean read ReadConfirmErase
        write WriteConfirmErase;
    property ExtrDirNames: Boolean read ReadExtrDirNames
        write WriteExtrDirNames;
    property ExtrOverwrite: Boolean read ReadExtrOverwrite
        write WriteExtrOverwrite;
    property HowToDelete: Integer read ReadHowToDelete
        write WriteHowToDelete;
    property KeepFreeOnDisk1: Integer read
        ReadKeepFreeOnDisk1 write WriteKeepFreeOnDisk1;
    property MaxVolumeSize: Integer read ReadMaxVolumeSize
        write WriteMaxVolumeSize;
    property MinFreeVolumeSize: Integer read
        ReadMinFreeVolumeSize write WriteMinFreeVolumeSize;
    property Password: String read ReadPassword write
        WritePassword;
    property SFXAskCmdLine: Boolean read ReadSFXAskCmdLine
        write WriteSFXAskCmdLine;
    property SFXAskFiles: Boolean read ReadSFXAskFiles
        write WriteSFXAskFiles;
    property SFXCaption: String read ReadSFXCaption write
        WriteSFXCaption;
    property SFXCommandLine: String read ReadSFXCommandLine
          write WriteSFXCommandLine;
    property SFXDefaultDir: String read ReadSFXDefaultDir
        write WriteSFXDefaultDir;
    property SFXHideOverwriteBox: Boolean read
        ReadSFXHideOverwriteBox write
        WriteSFXHideOverwriteBox;
    property SFXOverwriteMode: Integer read
        ReadSFXOverwriteMode write WriteSFXOverwriteMode;
    property Temp: String read ReadTemp write WriteTemp;
  end;
```

3.9. Symbols Library

SYM
dws2SymbolsLib1

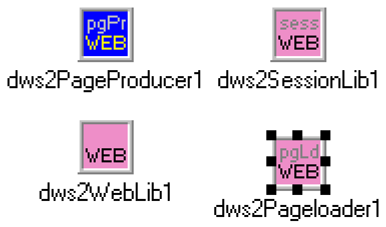
Listing 3.26: Standard Library: Symbols

```
{ Forward Declarations }
type TSymbols = class;

{ Enumerations }
type TSymbolType = (stUnknown, stArray, stAlias, stClass,
    stConstant, stField, stFunction, stMember, stParam,
    stProperty, stRecord, stUnit, stVariable);

{ Classes }
type
  TSymbols = class
    constructor CreateMain;
    constructor CreateUid(Uid : String);
    constructor CreateUnit(Name : String);
    function Caption : String;
    function Description : String;
    destructor Destroy; override;
    function Eof : Boolean;
    procedure First;
    function GetMembers : TSymbols;
    function GetParameters : TSymbols;
    function GetSuperClass : TSymbols;
    procedure Last;
    function Locate(Name : String) : Boolean;
    function Name : String;
    procedure Next;
    procedure Previous;
    function SymbolType : TSymbolType;
  end;
```

3.10. Web Library



The DWS2-Web-Library brings you 3 components for building web applications with DWS2 and 1 for desktop script handling (e.g. in reports...), namely *dws2Pageloader*.

NOTE

The Web Library has the ability to automatically check access to a web script. This is accomplished by special naming of the script files:

_S<name>.dws means that a user must have a valid user session to be able to access the script

_L<name>.dws means that a user must have a valid Login state to access the script

_M<name>.dws same as _L<name>.dws, but disallows access with GUEST login state.

3.10.1. dws2PageProducer

The *dws2PageProducer* is based on CustomContentProducer, with standard interface to be called by webactions and return content from scriptfiles.

3.10.2. dws2PageLoader

This component is similar to the *dws2PageProducer* but without interface to webbroker.

3.10.3. dws2SessionLib

Dws2SessionLib provides HTTP - usertracking and the ability to store sessiondata.

HINT

You can find a *very* simple **Session Tracking Server** in the *dws2\Libraries\WebLib\SimpleSessionServer* directory. It's a CLX based demo that shows how you can add external session tracking to DWS II enabled CGI executables.
In normal operational mode, *dws2SessionLib* does an **internal** user session tracking. This technique only works in ISAPI modules, because the module is loaded once and if a request hits the module, just a new thread is created that has access to a global internal session list.
If you plan to use DWS II enabled CGI executables and you need session tracking, then you need to install an external Session Tracking Server, as the CGI executable terminates after each web-request, thereby clearing the global session list.

Listing 3.27: Standard Library: WebLib - SessionLib

```
{ Constants }
const dwstateGLI : Integer = 1;
const dwstateLOFF : Integer = -1;
const dwstateNAT : Integer = 12;
const dwstateNLI : Integer = 0;
const dwstatePRF : Integer = 9;
const dwstateSU : Integer = 100;
const dwstateTOUT : integer = -2;
const dwstateULI : Integer = 10;
const dwstateVIP : Integer = 20;
const Version : String = '2.1.2001.6.20';

{ Classes }
type
  Session = class(TObject)
    class function GetActiveUsers : Integer;
    class function GetCState : Integer;
    class function GetIpAddr : String;
    class function GetSBrand : String;
    class function GetTLastAction : DateTime;
    class function GetTLogin : DateTime;
    class function Param(Name : string) : variant;
    class procedure SetCState(Value : Integer);
    class procedure SetIpAddr(Value : String);
    class procedure SetSBrand(Value : String);
    class procedure SetTLastAction(Value : DateTime);
    class procedure SetTLogin(Value : DateTime);
    class procedure Store(name : string; value : variant);
    property ActiveUsers: Integer read GetActiveUsers;
    property ClientState: Integer read GetCState write
        SetCState;
    property IpAddr: String read GetIpAddr write SetIpAddr;
    property SBrand: String read GetSbrand write SetSbrand;
    property TLastAction: DateTime read GetTLastAction
        write SetTLastAction;
    property TLogin: DateTime read GetTlogin write
        SetTlogin;
  end;

{ Functions }
function TIDfunc : String; forward;
function ActiveSession : boolean; forward;
function URL(AnURL : String) : String; forward;
```

3.10.4. dws2WebLib

Dws2WebLib introduces some classes to handle the ISAPI calls of Borland's webbroker (request, response, cookie), the scriptdoc class to store information about the scriptfile, and TFormVarGroup to handle multiple input fields of database tables in html forms. There are also some "functions" useful for webapps.

Listing 3.28: Standard Library: WebLib

```
{ Forward Declarations }
type TCookie = class;

{ Classes }
type
  Request = class(TObject)
    class function Accept : String;
    class function Authorization : string;
    class function Content : String;
    class function ContentLength : Integer;
    class function ContentType : String;
    class function Cookie(Name : String) : String;
    class function CookieCount : Integer;
    class function CookieName(Index : Integer) : String;
    class function CookieValue(Index : Integer) : String;
    class function Date : DateTime;
    class function From : String;
    class function Host : String;
    class function LogContent : String;
    class function Method : String;
    class function Param(Name : String) : String;
    class function ParamCount : Integer;
    class function ParamName(Index : Integer) : String;
    class function Params(Name : String) : String;
    class function ParamValue(Index : Integer) : string;
    class function PathInfo : String;
    class function Referer : String;
    class function RemoteAddr : String;
    class function RemoteHost : String;
    class function ScriptName : String;
    class function Title : String;
    class function Url : String;
    class function UserAgent : String;
  end;

type
  Response = class(TObject)
    class function Cookie(Index : Integer) : TCookie;
    class function CookieByName(Name : String) : TCookie;
    class function CookieCount : Integer;
    class function GetAllow : String;
    class function GetContent : String;
    class function GetContentEncoding : String;
    class function GetContentLength : Integer;
    class function GetContentType : String;
    class function GetContentVersion : String;
    class function GetDate : DateTime;
    class function GetDerivedFrom : String;
    class function GetExpires : DateTime;
    class function GetLastModified : DateTime;
    class function GetLocation : String;
    class function GetLogMessage : String;
    class function GetRealm : String;
    class function GetReasonString : String;
    class function GetServer : String;
    class function GetStatusCode : Integer;
    class function GetTitle : String;
    class function GetVersion : String;
    class function NewCookie : TCookie;
    class procedure SendRedirect(Uri : String);
    class procedure SetAllow(Value : String);
    class procedure SetContent(Value : String);
    class procedure SetContentEncoding(Value : String);
    class procedure SetContentLength(Value : Integer);
    class procedure SetContentType(Value : String);
    class procedure SetContentVersion(Value : String);
    class procedure SetDate(Value : DateTime);
    class procedure SetDerivedFrom(Value : String);
    class procedure SetExpires(Value : DateTime);
    class procedure SetLastModified(Value : DateTime);
    class procedure SetLocation(Value : String);
    class procedure SetLogMessage(Value : String);
    class procedure SetRealm(Value : String);
    class procedure SetReasonString(Value : String);
    class procedure SetServer(Value : String);
    class procedure SetStatusCode(Value : Integer);
    class procedure SetTitle(Value : String);
    class procedure SetVersion(Value : String);
    property Allow: String read GetAllow write SetAllow;
    property Content: String read GetContent write
        SetContent;
    property ContentEncoding: String read
        GetContentEncoding write SetContentEncoding;
    property ContentLength: Integer read GetContentLength
        write SetContentLength;
    property ContentType: String read GetContentType write
        SetContentType;
    property ContentVersion: String read GetContentVersion
        write SetContentVersion;
    property Date: DateTime read GetDate write SetDate;
    property DerivedFrom: String read GetDerivedFrom write SetDerivedFrom;
    property Expires: DateTime read GetExpires write SetExpires;
    property LastModified: DateTime read GetLastModified write SetLastModified;
    property Location: String read GetLocation write SetLocation;
    property LogMessage: String read GetLogMessage write SetLogMessage;
    property Realm: String read GetRealm write SetRealm;
    property ReasonString: String read GetReasonString write SetReasonString;
    property Server: String read GetServer write SetServer;
    property StatusCode: Integer read GetStatusCode write SetStatusCode;
    property Title: String read GetTitle write SetTitle;
    property Version: String read GetVersion write SetVersion;
  end;

type
  ScriptDoc = class(TObject)
    class function Date : DateTime;
    class function FileName : String;
    class function Path : String;
    class function Size : Integer;
  end;

type
  TCookie = class(TObject)
    function GetDomain : String;
    function GetExpires : DateTime;
    function GetHeaderValue : String;
    function GetName : String;
    function GetPath : String;
    function GetSecure : Boolean;
    function GetValue : String;
    procedure SetDomain(Value : String);
    procedure SetExpires(Value : DateTime);
    procedure SetName(Value : String);
    procedure SetPath(Value : String);
    procedure SetSecure(Value : Boolean);
    procedure SetValue(Value : String);
    property Domain: String read GetDomain write SetDomain;
    property Expires: DateTime read GetExpires write SetExpires;
    property HeaderValue: String read GetHeaderValue;
    property Name: String read GetName write SetName;
    property Path: String read GetPath write SetPath;
    property Secure: Boolean read GetSecure write SetSecure
        ;
    property Value: String read GetValue write SetValue;
  end;

type
  TFormVarGroup = class(TObject)
    constructor Create;
    function count : integer;
    function Eof : boolean;
    function Ext : String;
    procedure First;
    function GetAddNullFields : boolean;
    function GetPrefix : String;
    procedure Next;
    function RecNr : integer;
    procedure SetAddNullFields(Value : boolean);
    procedure SetPrefix(Prefix : String);
    function Value : String;
    property AddNullFields: boolean read GetAddNullFields
        write SetAddNullFields;
    property Prefix: String read GetPrefix write SetPrefix;
  end;

{ Functions }
function FormVar(ParamName : String) : String; forward;
function RtfToHtml(RtfStr : String) : String; forward;
function TrimURL(sURL : String) : String; forward;
function AcceptWML : boolean; forward;
function URLencode(sURL : String) : String; forward;
function URLdecode(sURL : String) : String; forward;
procedure _forward(NewFile : String); forward;
```

4. Non DWS II Standard Libraries

The DWS II installation package also ships some non standard libraries. These libraries will not be covered by this document. Just that you get an idea which libraries are currently considered “non standard”, we’ll list some of them here.

4.1. ADO Library

For more information regarding this library (especially the functions and classes the library exports) please have a look at the HTML documentation.

4.2. DB Library

Consider this library as deprecated.

4.3. SMTP Library (D5, D6 only)

Since Delphi 7, Borland does not ship the FastNet components with Delphi. This library is deprecated and no longer maintained.

Part III.

Advanced Topics

5. DWS II Tools

Future versions of this document may fill up this chapter. For now it can be considered as a placeholder.

6. Creating Web Applications Using DWS II 2.0

Figure 6.1 on the next page shows the WebModule of the ISAPI Demo. We'll use this example to show how a DWS II ISAPI module has to be installed and how you can use it afterwards.

We won't cover topics as 'How to setup IIS for debugging' here.

6.1. Installation of the ISAPI Demo

If you've compiled the demo, and run IIS on your computer, follow the steps given in section 6.1.1. In case you've installed Apache 2.x (Windows), follow the steps in section 6.1.2.

> **NOTE** You can not run this demo under Linux, because it's an ISAPI module.

6.1.1. Using IIS

Copy the "dws2.dll" file to the scripts directory of your server. Open the *dws2\Demos* directory in explorer, select the ISAPI subdirectory and press right mouse key. In the context menu, you'll find a menu item that allows you to publish the directory via the IIS. If you chose "IsapiDemo" as the virtual directory name on the server, you should be able to run the *index.dws* script by typing following URL in your browser: `http://localhost/scripts/dws2.dll/IsapiDemo/index.dws`

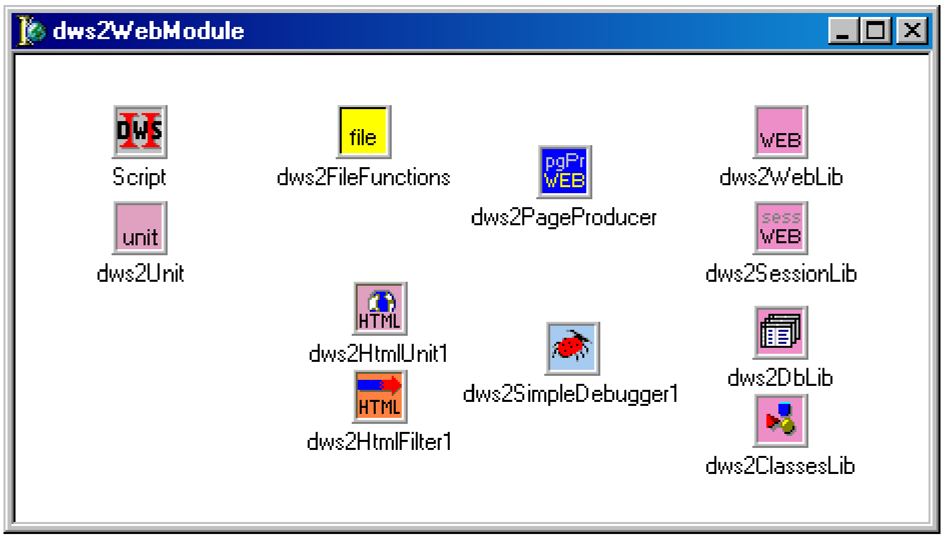
6.1.2. Using Apache 2.0

Only for Win32 Platform! (Provided by Kenneth Wilcox <k3nx@hotmail.com>)

Copy the dws2.dll to the cgi-bin directory on the server and make the following changes to the httpd.conf file:

```
AddType application/x-httpd-dws .dws
Action application/x-httpd-dws "/cgi-bin/dws2.dll"
AddHandler isapi-isa .dll
```

Figure 6.1.: ISAPI Demo - WebModule



```
  ISAPICacheFile "C:/Program Files/Apache
Group/Apache2/cgi-bin/dws2.dll"
```

If you place the dll in any other directory make sure that it is aliased and that it has ExecCGI permissions on it.

Line 1 tells Apache to set all .dws files to application/x-httpd-dws MIME type

Line 2 tells Apache that all application/x-httpd-dws MIME files are to be sent to dws2.dll for processing

Line 3 tells Apache that .dll files are ISAPI extentions

Line 4 tells Apache to load dws2.dll and keep it loaded (if this line is omitted then Apache will load/unload the ISAPI with each request)

6.2. Other demos..

Demos for Linux may follow in another version of the document.

7. How-To

The authors of this document would be glad, if some DWS II community members could provide short How-Tos that could be included here.

TIA!

7.1. Working with Objects

7.1.1. In Delphi, Get the Reference to a Delphi Object that is Wrapped by a Script Object

Use the GetExternalObjForID function defined in dws2comp.pas. It has the following definition:

function GetExternalObjForID(Info : TProgramInfo; **const** AVarName: **string**) : TObject;

7.2. Working with DWS2Unit

7.2.1. In DWS, Use Functions/Procedures/Objects Defined in other Dws2Units

Use the Dependencies property of the Tdws2Unit component (newline-separated names of used units).

7.3. Working with Scripts

7.3.1. In DWS, Emulate Dynamic Arrays

The dynamic arrays of DWSII 2.0 are in fact quite static. The only way to create a dynamic array is to declare an array constant. The compiler allocates space on the stack for this constants right below the regular stack. Because of this method there can't be a SetLength because it would mean that the regular stack would have to be moved at runtime. As long as there is not memory manager in DWSII we don't see a better way to handle this.

There are mainly two work-arounds:

1. Use a TList

2. If you can use the COMConnector, try:

Listing 7.1: HowTo: Emulate Dynmic Arrays

```
var A : ComVariantArray;
A.High := 3; // set dimension
A[0] := 10;
A[1] := 20;
A[2] := 'DWS';
A[3] := 'Array';
var i : Integer;
for i := A.Low to A.High do
begin
   println(A[i]);
end;
println(A.Low);
println(A.High);
println(A.Length);
```

7.3.2. In Delphi, Detect if a Variable is Used During the Running of the Script

In DWS, all variables are initialized i.e. a variable is defined immediately after the var-statement is executed. One workaround is to assign a never-used value during the variable assignment, and test for that instead e.g.

Listing 7.2: HowTo: Detect Variable Usage

```
var I: Integer = -9999;
if I = -9999 then
  // undefined or something
```

8. Appendices

Listings

List of Figures

Index