

Summary

This reference manual describes the DelphiScript language used by the Scripting Engine in Altium Designer.

This DelphiScript reference details each of the statements, functions and extensions that are supported. These are special procedures that are used to control and communicate directly with Altium Designer. It is assumed that you are familiar with basic programming concepts as well as the basic operation of Altium Designer.

The scripting system supports the DelphiScript language which is very similar to Borland Delphi (TM). The key difference is that DelphiScript is a typeless or untyped scripting language which means you cannot define records or classes and pass pointers as parameters to functions. You can still declare variables within scripts for readability.

Exploring the DelphiScript Language

This document and the [DelphiScript Keyword Reference](#) document contain reference material on interfaces, components, global routines, types, and variables that make up the DelphiScript scripting language.

Objects

An object consists of methods, and in many cases, properties, and events. Properties represent the data contained in the object. Methods are the actions the object can perform. Events are conditions the object can react to. All objects descend from the top level object of `TObject` type.

Object Interfaces

An object interface consists of methods, and in many cases, properties but cannot have data fields. An interface represents an existing object and each interface has a GUID which marks it unique. Properties represent the data contained in the object that the interface is associated with. Methods are the actions the object (in which the interface is associated with) can perform.

Components

Components are visual objects that you can manipulate at design time from the *Tool Palette* panel. All components descend from the `TComponent` class in the Borland Delphi Visual Component Library.

Routines

Global routines are the procedures and functions from the scripting system. These routines are not part of a class, but can be called either directly or from within class methods on your scripts.


Types

The variable types are used as return types and parameter types for interface methods and properties, object's methods, properties and events and for global routines. In many cases, types are documented in the Enumerated Types sections in all API documents.

For example, the Client Enumerated Types for the DXP Object Model is detailed in the [System Reference](#) document

Altium Designer and Borland Delphi Run Time Libraries

The Scripting system also supports a subset of Borland Delphi Run Time Library (RTL) and a subset of Altium Designer RTL which is covered in the [Altium Designer RTL](#) Guide.

 You can navigate to the various Altium Designer API documents via **Configuring the System » Scripting in Altium Designer » Altium Designer RTL Reference** from the *Knowledge Center* panel.

DelphiScript Source files

A script project is organized to store script documents (script units and script forms). You can execute the script from a menu item, toolbar button or from the Run Script dialog from the Altium Designer's system menu.

PRJSCR, PAS and DFM files

Scripts are organized into projects with a *.PRJSCR extension. Each project consists of files with a *.pas extension. Files can be either script units or script forms (each form has a script file with *.pas extension and a corresponding form with a *.dfm extension). A script form is a graphical window that hosts different controls that run on top of Altium Designer.

It is possible to attach scripts to different projects and it is highly recommended to organize scripts into different projects to manage the number of scripts and their procedures / functions.

About Example Scripts

The following examples illustrate the basic features of DelphiScript programming using simple scripts for use in Altium Designer. Example scripts can be found in the `Examples` folder under your Altium Designer installation. The location and purpose of some of the example scripts are mentioned below:

- `Examples\Scripts\DelphiScript Scripts\DXP` sub folder - Demonstrate Client and system API
- `Examples\Scripts\DelphiScript Scripts\PCB` subfolder - Demonstrate PCB API
- `Examples\Scripts\DelphiScript Scripts\Processes` sub folder - Demonstrate server processes
- `Examples\Scripts\DelphiScript Scripts\General` sub folder - Demonstrate DelphiScript keywords
- `Examples\Scripts\DelphiScript Scripts\Sch` subfolder - Demonstrate Schematic API
- `Examples\Scripts\DelphiScript Scripts\WSM` subfolder - Demonstrate Workspace Manager API

Writing DelphiScript Scripts

DelphiScript Naming Conventions

In general, there are no restrictions to the names you can give to procedures, functions, variables and constants as long as they adhere to the following rules:

- The name can contain the letters A to Z, a to z, the underscore character "_" and the digits 0 to 9
- The name must begin with a letter
- The name cannot be a DelphiScript keyword, directives or reserved word
- Names are case insensitive when interpreted. You may use both upper and lower case when naming a function, subroutine, variable or constant, however the interpreter will not distinguish between upper and lower case characters. Names which are identical except for case will be treated as the same name in DelphiScript.

In a DelphiScript file, the functions and procedures are declared using the `Procedure Begin End` or `Function Begin End` blocks. Both of these statement blocks require a name to be given to the procedure or function. DelphiScript allows you to create named variables and constants to hold values used in the current script.

Including Comments in scripts

In a script, comments are non-executable lines of code which are included for the benefit of the programmer. Comments can be included virtually anywhere in a script. Any text following `//` or enclosed with `(*)` or `{ }` are ignored by DelphiScript.

// Comment type example

```
//This whole line is a comment
```

{ } Comment type example

```
{This whole line is a comment}
```

(*) comment type example

```
(*
This whole line is a comment
This whole line is a comment
This whole line is a comment
*)
```

Comments can also be included on the same line as executed code. For example, everything after the semi colon in the following code line is treated as a comment.

```
ShowMessage ('Hello World'); //Display Message
```

Local and Global Variables

Since all scripts have local and global variables, it is very important to have unique variable names in your scripts within a script project. If the variables are defined outside any procedures and functions, they are global and can be accessed by any script unit in the same project.

If variables are defined inside a procedure or function, then these local variables are not accessible outside these procedures/functions.

Example of Local and Global Variables in a Script

```
// The Uses keyword is not needed.
// Variables from UnitA script are available to this Script Unit,
// as long UnitA is in the same project as this Unit Script Unit.
Const
    GlobalVariableFromThisUnit='Global Variable from this unit';
```

```

Procedure TestLocal;
var
    Local;
Begin
    // we can access a variable from UnitA without the Uses keyword
    Local := 'Local Variable';
    ShowMessage(Local);
End;

Procedure TestGlobal;
Begin
    //ShowMessage(Local); // produces an error.
    ShowMessage(GlobalVariableFromThisUnit);
    ShowMessage(GlobalVariableFromUnitA);
End;

```

UnitA script

```

Const
    GlobalVariableFromUnitA = 'Global Variable from Unit A';

```

Using Named Variables in a Script

In a script, you use named variables or constants to store values to be used during program execution. All variables in a script are always of Variant type. Typecasting is ignored. Types in variables declaration are ignored and can be skipped, so these declarations are correct:

```

Var a : integer;
Var b : integer;
Var c, d;

```

Splitting a Line of Script

Each code statement is terminated with the semi-colon ";" character to indicate the end of the statement. DelphiScript allows you to write a statement on several lines of code, splitting a long instruction on two or more lines. The only restriction in splitting programming statements on different lines is that a string literal may not span several lines.

For example:

```

X.AddPoint( 25, 100);
X.AddPoint( 0, 75);
// is equivalent to:
X.AddPoint( 25, 100); X.AddPoint( 0, 75);

```

But

```
'Hello World!'
```

is not equivalent to

```
'Hello
World!'
```

DelphiScript does not put any practical limit on the length of a single line of code in a script, however, for the sake of readability and ease of debugging it is good practice to limit the length of code lines so that they can easily be read on screen or in printed form.

If a line of code is very long, you can break this line into multiple lines and this code will be treated by the DelphiScript interpreter as if it were written on a single line.

Unformatted Code Example

```
If Not (PcbApi_ChooseRectangleByCorners(BoardHandle, 'Choose first corner', 'Choose final
corner', x1, y1, x2, y2)) Then Exit;
```

Formatted Code Example

```
If Not (PcbApi_ChooseRectangleByCorners(BoardHandle,
                                         'Choose first corner',
                                         'Choose final corner',
                                         x1, y1, x2, y2)) Then Exit;
```

Case Sensitivity

The DelphiScript language used in writing scripts is not case sensitive, i.e. all keywords, statements, variable names, function and procedure names can be written without regard to using capital or lower case letters. Both upper and lower case characters are considered equivalent. For example, the variable name `myVar` is equivalent to `myvar` and `MYVAR`. DelphiScript treats all of these names as the same variable.

The only exception to this is in literal strings, such as the title string of a dialog definition or the value of a string variable, these strings retain case differences.

The Space Character

A space is used to separate keywords in a script statement. However, DelphiScript ignores any additional white spaces in a statement.

For example:

```
X = 5
```

is equivalent to

```
X      =          5
```

You may use white spaces to make your script more readable.

Functions and Procedures in a Script

The DelphiScript interpreter allows two kinds of procedures: Procedures and Functions. The only difference between a function and a procedure is that a function returns a value.

A script can have at least one procedure which defines the main program code. You can, however, define other procedures and functions that can be called by your code. As with Borland Delphi, procedures and functions are defined within a `Begin End` statement block. To invoke or call a function or procedure, include the name of the function or procedure in a statement in the same way that you would use the built-in DelphiScript functions and procedures. If the function or procedure requires parameters, then you must include these in the calling statement. Both functions and procedures can be defined to accept parameters, but only functions can be defined to return a value to the calling statement.

You may assign any name to functions and procedures that you define, as long as it conforms to the standard DelphiScript naming conventions.

Typical DelphiScript Procedure

```
Procedure CreateSchObjects;
Begin
    If SchServer = Nil Then Exit;
    SchDoc := SchServer.GetCurrentSchDocument;

    If SchDoc = Nil Then Exit;
    PlaceSchematicObjects;
    SchDoc.GraphicallyInvalidate;
End;
```

Typical DelphiScript Function

```
Function BooleanToString(AValue : Boolean) : String;  
Begin  
    If (AValue) Then Result := 'True'  
    Else          Result := 'False';  
End;
```

The name of a function cannot be used to set its return value. The Result keyword must be used instead.

Var Begin End Global Block

```
Var  
    A, B, C;  
Begin  
    B := 10;  
    C := 20;  
    A := B + C;  
    ShowMessage(IntToStr(A));  
End;
```

Tips on Writing Scripts

Referencing Scripts in a Script Project

You can have code in one script to call a procedure in another script in the same script project. You can also access any global variable in any script within the same project.

Local and Global Variables

Since all scripts have local and global variables, it is important to have unique variable names in your scripts within a script project. If the variables are defined outside any procedures and functions, they are global and can be accessed by any unit in the same project.

If variables are defined inside a procedure or function, then these local variables are not accessible outside these procedures/functions.

It is recommended to put scripts of similar nature in a project and to keep the number of scripts in a project at a manageable size. Keeping track of global variables in many scripts becomes an issue. It is not mandatory to store scripts in a script project, you can put scripts in other project types.

Unique Identifiers and Variables

With script forms, ensure that all script forms have unique form names. It is possible to have all script forms named `form1` in the same script project. The scripting system gets confused when trying to display which form when a script form is executed.

Change the script form name by using the Object Inspector, the name will be changed in the script unit and in the script form files automatically.

Parameter-less Procedures and Functions

Design your scripts so that the procedures required to be invoked to run the script will only appear in the *Select Items to Run* dialog. To prevent other procedures/functions from appearing in the *Select Items to Run* dialog, you can insert a (Dummy : Integer) parameter next to the method name. See the example below.

Example

```
Function TSineWaveform.CreateShape(Dummy : Integer) : TShape;
Begin
    // do something
End;
{.....}
{.....}
Procedure TSineWaveform.bCloseClick(Sender: TObject);
var
    I : integer;
Begin
    // doing something
    Close;
End;
{.....}
{.....}
procedure DrawSine;
Begin
    SineWaveform.showmodal;
End;
```

Differences between DelphiScript and Object Pascal

In this section, the differences between DelphiScript and Object Pascal of Borland Delphi 32 bit versions will be covered in detail. The scripting system uses untyped DelphiScript language therefore there are no data types in scripts.

Although you can declare variables and their types and specify the types for functions/procedures or methods' parameters for readability, DelphiScript converts undeclared variables when a script is being executed.

For example, you cannot define records or classes.

DelphiScript Variables

All variables in a script are always of Variant type, Types in variables declaration are ignored and can be skipped, so these declarations are correct:

```
Var
a : Integer;
```

```
Var
b : Integer;
```

```
Var
    c, d;
```

Types of parameters in procedure/function declaration are ignored and can be skipped. For example, this code is correct:

```
Function Sum(a, b) : Integer;
Begin
    Result := a + b;
End;
```

In general, you can use variants to store any data type and perform numerous operations and type conversions. A variant is type-checked and computed at run time. The compiler won't warn you of possible errors in the code, which can be caught only with extensive testing. On the whole, you can consider the code portions that use variants to be interpreted code, because many operations cannot be resolved until run time. This can affect the speed of the code.

Now that you are aware of the use of the Variant type, it is time to look at what it can do. Once you have declared a variant variable such as the following and you have the variant value, you can copy it to any compatible-or incompatible-data type:


```
Var
    V;
Begin
    // you can assign to it values of several different types:
    V := 10;
    V := 'Hello, World';
    V := 45.55;
End;
```

If you assign a value to an incompatible data type, DelphiScript interpreter performs a conversion where possible. Otherwise, a run-time error is issued. In fact, a variant stores type information along with the data so DelphiScript is slower than a Borland Delphi compiled code.

Sets in DelphiScript Scripts

DelphiScript does not have Set types and does not support Set operators unlike the Object Pascal language which does have Set types and supports Set operators.

To use sets in DelphiScript scripts, use the built-in functions which enable you to manipulate sets in a DelphiScript script.

 Navigate to the [Using Sets in DelphiScript](#) section for more information on using sets in DelphiScript scripts.

Functions / Procedures inside a Function or Procedure

It is recommended that you write standalone functions or procedures, (recursive procedures/functions are permitted although). This function snippet is not recommended.

```
Function A
    Function B
    Begin
        // blah
    End;
Begin
    B;
End;
```

Recommended Function Structure

```
Function B
Begin
    // blah
End;
```

```
Function A
Begin
    B;
End;
```

Result Keyword

Use the Result keyword to set the return value within a function block.

Example

```
Function Foo : String;
Begin
    Result := 'Foo Foo';
End;
```

Nested Routines

Nested routines are supported but you can't use variables of top level function from the nested one.

Array Elements

Type of array elements is ignored and can be skipped so these declarations are equal:

```
Var
    x : array [1..2] of double;
Var
    x : array [1..2];
```

You cannot declare array types but you can declare arrays to variables

Illegal example

```
Type
    TVertices = Array [1..50] Of TLocation;
Var
    NewVertices : TVertices;
```

Legal example

```
Var
    NewVertices : Array [1..50] of TLocation;
```

Open Array Declaration

The Open Array Declaration is not supported.

Case Keyword

The `case` keyword can be used for any type. So you can write:

```
Case UserName of
    'Alex', 'John' : IsAdministrator := true;
    'Peter' : IsAdministrator := false;
Else
    Raise('Unknown user');
End;
```

Class Declarations

You cannot define new classes, but you can use existing DelphiScript classes and instantiate them. For example `TList` and `TStringList` classes can be created and used in your scripts.

See also

`TList`
`TStringList`

CreateObject Function

The `CreateObject` function can be used to create objects that will be implicitly freed when no longer used. So instead of:

```
Procedure Proc;
Var
    l;
Begin
    l := TList.Create;
    Try
        // do something with l
    Finally
        L.Free;
    End;
End;
```

you can write:

```
Procedure Proc;
Var
    l;
Begin
    l := CreateObject(TList);
    // Do something with l
End;
```

See also

`CreateObject` keyword

Raise Exceptions

The `Raise` keyword can be used without parameters to re-raise the last exception. You can also use `Raise` with string parameter to raise the exception with the specified message string. The Exception objects are not supported, because the `On` keyword is not supported.

Example

```
Raise(Format('Invalid value : %d', [Height]));
```

See also

`Try` keyword

`Finally` keyword

`Raise` keyword

ThreadVar

The `Threadvar` keyword is treated as `Var`. Note that in Object Pascal, the variables declared using `Threadvar` keyword have distinct values in each thread.

Set Operators

The Set operator `In` is not supported. You can use `InSet` to check whether a value is a member of set.

For example,

```
If InSet(fsBold, Font.Style) then
    ShowMessage('Bold');
```

Note, that set operators `+`, `-`, `*`, `<=`, `>=` don't work correctly. You have to use the built in functions; `MkSet`, `MkSetRange`, `SetUnion`, `SetDifference`, `SetIntersection`, `SubSet` and `InSet`.

Example

`ASet := BSet + CSet;` should be changed to

```
ASet := SetUnion(BSet, CSet);
```

The [...] set constructors are not supported. You can use the `MkSet` function to create a set.

Example

```
Font.Style := MkSet(fsBold, fsItalic);
```

See also

`MkSet` keyword

`MkSetRange` keyword

`InSet` keyword

`SetDifference` keyword

`SetIntersection` keyword

`SetUnion` keyword

`SubSet` keyword

`InSet` keyword

Operators

`^` and `@` operators are not supported.

Directives

The following directives are not supported (note that some of them are obsolete and aren't supported by Borland Delphi too):

`absolute`, `abstract`, `assembler`, `automated`, `cdecl`, `contains`, `default`, `dispid`, `dynamic`, `export`, `external`, `far`, `implements`, `index`, `message`, `name`, `near`, `nodefault`, `overload`, `override`, `package`, `pascal`, `private` `protected`, `public`, `published`, `read`, `readonly`, `register`, `reintroduce`, `requires`, `resident`, `safecall`, `stdcall`, `stored`, `virtual`, `write`, `writeonly`.

Note, the "in" directive in the `Uses` clause is ignored.

Ignored Keywords

The `interface`, `implementation`, `program` and `unit` keywords are ignored in DelphiScript. The scripts can contain them but they have no effect, these keywords can enhance the readability of scripts.

Unsupported Keywords

The following keywords are not supported in DelphiScript:

- `as`, `asm`, `class`, `dispinterface`, `exports`, `finalization`, `inherited`, `initialization`, `inline`, `interface`, `is`, `library`, `object`, `out`, `property`, `record`, `resourcestring`, `set`, `supports`, `type`.

The following Delphi RTL functions aren't supported in DelphiScript:

- `Abort`, `Addr`, `Assert`, `Dec`, `FillChar`, `Finalize`, `Hi`, `High`, `Inc`, `Initialize`, `Lo`, `Low`, `New`, `Ptr`, `SetString`, `SizeOf`, `Str`, `UniqueString`, `VarArrayRedim`, `VarArrayRef`, `VarCast`, `VarClear`, `VarCopy`.

The functions from the Borland Delphi's Windows unit (`windows.pas` file) are not supported (for example the `RGB` function is not supported).

Using Altium Designer RTL in DelphiScript scripts

You cannot create your own records or classes types and instantiate them in a script, however you can use certain classes from the Altium Designer Run Time Library (RTL). For example, `TStringList` and `TList` classes can be instantiated and used as containers of data storage (usually of the same type) for your scripting needs.

The Object Interfaces representing Altium Designer objects are available to use in scripts. For example, you have the ability to update design objects on Schematic and PCB documents through the use of Schematic object interfaces and PCB objects interfaces.

Interface names as a convention have an `I` added in front of the name for example `IPCB_Board` represents an interface for an existing PCB document. An example of PCB interfaces in use is shown next.

PCB Interfaces in a script example

```

Procedure ViaCreation;
Var
    Board : IPCB_Board;
    Via    : IPCB_Via;
Begin
    Board := PCBServer.GetCurrentPCBBoard;
    If Board = Nil Then Exit;
    (* Create a Via object *)
    Via      := PCBServer.PCBObjectFactory(eViaObject, eNoDimension, eCreate_Default);
    Via.X     := MilsToCoord(7500);
    Via.Y     := MilsToCoord(7500);
    Via.Size  := MilsToCoord(50);
    Via.HoleSize := MilsToCoord(20);
    Via.LowLayer := eTopLayer;
    Via.HighLayer := eBottomLayer;
    (* Put this via in the Board object*)
    Board.AddPCBObject(Via);
End;
```

The following APIs can be used in your Scripts:

- Certain Borland Delphi™ functions and classes and DelphiScript extensions
- Client API
- PCB Server API
- Schematic Server API
- Work Space Manager Server API
- Nexus API
- Altium Designer RTL functions
- Parametric processes

Examine the scripts in the `\Examples\Scripts\` folder of your installation to see Altium Designer Object Interfaces and functions, Borland Delphi objects and functions being used in scripts.



Refer to [Getting Started with Scripting](#) and [Building Script Projects](#) tutorials.



Refer to the [Using the Altium Designer RTL](#) guide for details on how to use design objects and their interfaces in your scripts.



You can navigate to the various Altium Designer API documents via **Configuring the System » Scripting in Altium Designer » Altium Designer RTL Reference** from the *Knowledge Center* panel.

DelphiScript Keywords

The scripting system supports the DelphiScript language. The DelphiScript keywords reference is covered in a different document.



Consult the [DelphiScript Keyword Reference](#) document for more information on keywords.

Expressions and Operators

An expression is a valid combination of constants, variables, literal values, operators and function results. Expressions are used to determine the value to assign to a variable, to compute the parameter of a function or to test for a condition. Expressions can include function calls.

DelphiScript has a number of logical, arithmetic, Boolean and relational operators. These operators are grouped by the order of precedence, which is different to the precedence orders used by Basic, C etc. For example, the `AND` and `OR` operators have precedence compared to the relational one.

If you write `a<b` and `c<d`, the DelphiScript will do the `AND` operation first, resulting in an error. To fix this problem, you have to enclose each of the `<` expression in parentheses: `(a<b) and (c<d);`

These operators listed below are the operators supported by DelphiScript.

Operators Grouped by Precedence

Unary operators have the highest precedence

Not	Boolean or bitwise NOT.
-----	-------------------------

Multiplicative and Bitwise Operators

*	Arithmetic multiplication.
/	Floating point division.
div	Integer division.
mod	modulus (remainder of integer division).
and	Boolean or bitwise AND.
shl	Bitwise left shift.
shr	Bitwise right shift.

Additive Operators

+	Arithmetic addition, string concatenation.
-	Arithmetic subtraction.
or	Boolean or bitwise OR
xor	Boolean or bitwise EXCLUSIVE OR.

Relational and Comparison Operators (lowest precedence)

=	Test whether equal or not.
<>	Test whether not equal or not.
<	Test whether less than or not.
>	Test whether greater than or not.
<=	Test whether less than or equal to or not.
>=	Test whether greater than or equal to or not.

Note

The ^ and @ operators are not supported by DelphiScript.

DelphiScript Functions

Some of the statements used by the DelphiScript language are covered here. A range of functions are covered in the FileIO routines, Math Routines, String Routines and Extension routines. In this DelphiScript Functions Section:

- [Calculating Expressions with the Evaluate Function](#)
- [Passing Parameters to Functions and Procedures](#)
- [Exiting from a Procedure](#)
- [File IO Routines](#)
- [Math Routines](#)
- [String Routines](#)
- [Extension Routines](#)
- [Using Sets in DelphiScript](#)
- [Using Exception Handlers](#)

Calculating Expressions with the Evaluate Function

The built in function, Evaluate interprets a string containing a valid mathematical expression during runtime and a result is returned. For example, you can write a script such as `Evaluate(ProcNames[ProcIndex])`; and the procedure specified in `ProcNames[ProcIndex]` will be called.

To calculate such an expression you can use `Evaluate` method where expression is specified by `Expr` parameter. For example, you can calculate expressions like the following:

```
Evaluate('2+5');
Evaluate('( (10+15)-5)/2*5');
Evaluate('sin(3.1415926/2)*10');
Evaluate('2.5*log(3)');
```

Passing Parameters to Functions and Procedures

Both functions and procedures defined in a script can be declared to accept parameters. Additionally, functions are defined to return a value. Types of parameters in procedure/function declaration are ignored and can be skipped. For example, this code is correct:

```
Function sum(a, b) : integer;
Begin
    Result := a + b;
End;
```

Exiting from a Procedure

DelphiScript provides `Exit` and `Break` statements should you want to exit from a procedure before the procedure would terminate naturally. For example; if the value of a parameter is not suitable, you might want to issue a warning to the user and exit, as example below shows.

```
Procedure DisplayName (s);
Begin
    If s = '' Then
        Begin
            ShowMessage('Please enter a name');
            Exit;
```

```

End;
ShowMessage(S + ' is shown');
End;

```

File IO Routines

DelphiScript has the following IO routines:

Append	AssignFile	ChDir
CloseFile	Eof	Eoln
Erase	GetDir	MkDir
Read	Readln	Reset
Rewrite	RmDir	Write
Writeln		

DelphiScript gives you the ability to write information to a text file and since DelphiScript is an untyped language, you must convert all values to strings first. Read/ReadLn routines are equivalent, they read a line up to but not including the next line. A Writeln(String) routine is equivalent to a Write(S) and a Write(LineFeed + CarriageReturn) routine.

To write out a text file, you need to employ AssignFile, ReWrite, Writeln and CloseFile procedures. To read in a text file, you need to employ the AssignFile, Reset, Readln and CloseFile procedures. This example writes to a text file and adds an end-of-line marker.

Use of Try / Finally / End block is recommended to make scripts secure in the event of an IO failure.

Example

```

Var
    InputFile  : TextFile;
    OutputFile : TextFile;
    I          : Integer;
    Line       : String;
Begin
    AssignFile(OutputFile,eConvertedFile.Text);
    Rewrite(OutputFile);
    AssignFile(InputFile,eOriginalFile.Text);
    Reset(InputFile);
    Try
        While not EOF(InputFile) do
            Begin
                Readln(InputFile,Line);
                For I := 1 to Length(Line) Do
                    Line[I] := UpperCase(Line[I]);
                Writeln(Outputfile, Line);
            End;
        Finally
            CloseFile(InputFile);
            CloseFile(OutputFile);
        End;
    End;
End;

```


Math Routines

DelphiScript has the following math routines:

Abs	ArcTan	Cos
Exp	Frac	Int
Random	Randomize	Round
Sin	Sqr	Sqrt
Trunc		

String Routines

DelphiScript has the following string routines which can manipulate strings or characters. Only a subset of string routines (most of the string routines used by DelphiScript are imported from the Borland Delphi's SysUtils unit) are shown in this table below:

AnsiCompareStr	AnsiCompareText	AnsiDequotedStr
AnsiExtractQuotedStr	AnsiLowerCase	AnsiPos
AnsiQuotedStr	AnsiSameStr	AnsiSameText
AnsiUpperCase	CompareStr	CompareText
Copy	Delete	FloatToStr
FloatToStrF	Format	Frac
Insert	Int	IsDelimiter
LastDelimiter	Length	LowerCase
Ord	Pos	Pred
QuotedStr	Round	SameText
SetLength	StringOfChar	StringToGUID
Succ	Trim	TrimLeft
TrimRight	UpCase	UpperCase
WideLowerCase	WideSameStr	WideSameText
WideUpperCase		

These string routines that are not supported by DelphiScript are outlined in the table below:

AnsiContainsStr	AnsiContainsText	AnsiEndsStr
AnsiEndsText	AnsiIndexText	AnsiMatchStr
AnsiMatchText	AnsiReplaceStr	AnsiReplaceText
AnsiResemblesProc	AnsiResemblesText	AnsiStartsStr
AnsiStartsText	Concat	DecodeSoundExInt
DecodeSoundExWord	DupeString	LeftStr
MidStr	NullStr	ReverseString
RightStr	SetString	SoundEx
SoundExCompare	SoundExInt	SoundExProc
SoundExSimilar	SoundExWord	Str
StuffString	Val	

Extension Routines

The extension routines are used when you are dealing with server processes (a command is a packaged server process with parameters if any) in your scripts, especially if you need to extract or set strings for the parameters of processes. Some of the routines are listed below.

To execute parameters of processes in your script, you may need following functions:

AddColorParameter	AddIntegerParameter	AddLongIntParameter
AddSingleParameter	AddWordParameter	GetIntegerParameter
GetStringParameter	ResetParameters	RunProcess

Useful functions

SetCursorBusy	ResetCursor	CheckActiveServer
GetActiveServerName	GetCurrentDocumentFileName	RunApplication
SaveCurrentDocument		

Useful Dialogs

ConfirmNoYes	ConfirmNoYesCancel	ShowError
ShowInfo	ShowWarning	



Refer to the [Altium Designer RTL](#) and [Server Process](#) references for more details.

Using Sets in DelphiScript

Object Pascal's Set and In keywords and the set operators +, -, *, <= and >= are not supported in DelphiScript. The equivalent Set Operator keywords are tabulated below.

DelphiScript has no set types. To use sets in DelphiScript scripts, use the built-in functions; MkSet, MkSetRange, InSet, SetUnion, SetDifference, SetIntersection and SubSet functions which enable you to manipulate sets in a DelphiScript script. These are described in more detail below.

Set Operators

Object Pascal Set Operator keyword	Equivalent DelphiScript Set Operator keyword
+	SetUnion
-	SetDifference
*	SetIntersection
<=	SubSet
=	=
<>	<>
In	InSet

MkSet function

The MkSet function is a set constructor and has a variable number of arguments. For example:

```
Font.Style = MkSet(fsBold,fsItalic);
```

The MkSet(fsBold,fsItalic) denotes two set elements fsBold and fsItalic only.

MkSetRange function

The MkSetRange function is a set constructor and has a range of arguments. For example:

```
LayerSet := MkSetRange(eTopLayer,eBottomLayer);
```

The MkSetRange(eTopLayer,eBottomLayer) function denotes a range of layers from eTopLayer to eBottomLayer.

InSet function

This `InSet` function is used as substitution of Object Pascal's `In` operator. `A in B` is equal to `InSet(A, B)`.

```
If InSet(A,B) then
    ShowMessage('A is in B set')
Else
    ShowMessage('A not in B set');
```

SetUnion function

The `SetUnion` function is used as substitution of Object Pascal's `+` operator. `A + B` is equal to `SetUnion(A, B)`.

SetDifference function

The `SetDifference` function is used as substitution of Object Pascal's `-` operator. `A - B` is equal to `SetDifference(A, B)`.

SetIntersection function

The `SetIntersection` function is used as substitution of Object Pascal's `*` operator. `A * B` is equal to `SetIntersection(A, B)`.

SubSet function

The `SubSet` function is used as substitution of Object Pascal's `<=` operator. `A <= B` is equal to `SubSet(A, B)`.

Examples:

```
ASet := BSet + CSet;
```

should be changed to:

```
ASet := SetUnion(BSet,CSet);
```

in order to achieve the desired result in your script.

Using Exception Handlers

The `try` keyword introduces a `try-except` statement or a `try-finally` statement. These two statements are related but serve different purposes.

Try Finally

The statements in the `finally` block are always executed, regardless of whether an exception from the `try` block, `exit` or `break` occurs. Use `try-finally` block to free temporary objects and other resources and to perform clean up activities. Typically you do not need more than one `try-finally` statement in a subroutine.

Example

```
Reset(F);
Try
    // process file F
Finally
    CloseFile(F);
End;
```

Try Except

Use `try-except` to handle exceptional cases, for example, to catch specific exceptions and do something useful with them, such as log them in an error log or create a friendly dialog box. Since the `On` keyword is not supported in DelphiScript, so you can use the `Raise` statement inside the `Except` block.

Example

```
Try
    X := Y/Z;
Except
    Raise('A divide by zero error!');
End;
```

Raise

The `Raise` keyword is related to the `Try` keyword. The `Raise` keyword can be used without parameters to re-raise the last exception. It can also be used with a string parameter to raise an exception using a specific message.

Example

```
Raise(Format('Invalid Value Entered : %d', [Height]));
```

Note, the `On` keyword is not supported so you cannot use `Exception` objects as in Borland Delphi.

Working with Script Forms and Graphical Components

Introduction to Components

The scripting system handles two types of components: Visual and Non-visual components. Visual components are used to build the user interface and the non-visual components are used for different tasks such as these Timer, OpenFileDialog and MainMenu components. The Timer non-visual component can be used to activate specific code at scheduled intervals and it is never seen by the user. The Button, Edit and Memo components are visual components.

Both types of components appear at design time but non visual components are not visible at runtime. Components from the *Tool Palette* panel are object oriented and have the three following items:

- Properties
- Events
- Methods

A property is a characteristic of an object that influences either the visible behavior or the operations of this object. For example, the Visible property determines whether this object can be seen or not on a script form.

An event is an action or occurrence detected by the script. In a script, the programmer writes code for each event handler designed to capture a specific event such as a mouse click.

A method is a procedure that is always associated with an object and defines the behavior of an object.

All script forms have one or more components. Components usually display information or allow the user to perform an action. For example, a Label is used to display static text, an Edit box is used to allow user to input some data, a Button can be used to initiate actions.

Any combination of components can be placed on a form and while your script is running, a user can interact with any component on a form. It is your task, as a programmer to decide what happens when a user clicks a button or changes text in an Edit box.

The Scripting system supplies a number of components for you to create complex user interfaces for your scripts. You can find all of the components you can place on a form from the *Tool Palette*.

To place a component on a form, locate its icon on the *Tool Palette* panel and double-click it. This action places a component on the active form. Visual representation of most components is set with their set of properties. When you first place a component on a form, it is placed in a default position, with default width and height however you can resize or re-position this component. You can also change the size and position later using the *Object Inspector*.

When you drop a component onto a form, the Scripting system automatically generates code necessary to use the component and updates the script form. You only need to set properties, put code in event handlers and use methods as necessary to get the component on the form working.

Designing Script Forms

A script form is designed to interact with the user within the environment. Designing script forms is the core of visual development. Every component you place on a script form and every property you set is stored in a file describing the form (a *.DFM file) and has a relationship with the associated script code (the *.PAS file). For every script form, there is the .PAS file and the corresponding .DFM file.

When you are working with a script form and its components, you can operate on its properties using the *Object Inspector* panel. You can select more than one component by shift clicking on the components or by dragging a selection rectangle around the components on this script form. A script form has a title (the Caption property on the Object Inspector panel).

Creating a New Script Form

With a script project open, right click on a project in the *Projects* panel, and a pop up menu appears, click on the **Add New to Project** item, and choose a Delphi Script Form item. A new script form appears with the `EditScript1.pas` name as the default name.

Displaying a Script Form

In a script, you will need to have a procedure that displays the form when the script form is executed. Within this procedure, you invoke the `ShowModal` method for the form. The `Visible` property of the form needs to be false if the `ShowModal` method of the script form is to work properly.

ShowModal example

```
Procedure RunDialog;
Begin
    DialogForm.ShowModal;
End;
```

The `ShowModal` example is a very simple example of displaying the script form when the `RunDialog` from the script is invoked. Note, you can assign values to the components of the `DialogForm` object before the `DialogForm.ShowModal` is invoked.

ModalResult Example

```
Procedure TForm.OKButtonClick(Sender: TObject);
Begin
    ModalResult := mrOK;
End;
```

```
Procedure TForm.CancelButtonClick(Sender: TObject);
Begin
    ModalResult := mrCancel;
End;
```

```
Procedure RunShowModalExample;
Begin
    // Form's Visible property must be false for ShowModal to work properly.
    If Form.ShowModal = mrOk      Then ShowMessage('mrOk');
    If Form.ShowModal = mrCancel Then ShowMessage('mrCancel');
End;
```

The `ModalResult` property example here is a bit more complex. The following methods are used for buttons in a script form. The methods cause the dialog to terminate when the user clicks either the **OK** or **Cancel** button, returning `mrOk` or `mrCancel` from the `ShowModal` method respectively.

You could also set the `ModalResult` value to `mrOk` for the **OK** button and `mrCancel` for the button in their event handlers to accomplish the same thing. When the user clicks either button, the dialog box closes. There is no need to call the `Close` method, because when you set the `ModalResult` method, the script engine closes the script form for you automatically.

Note, if you wish to set the form's `ModalResult` to `Cancel` when user presses the `Escape` key, enable the `Cancel` property to `True` for the `Cancel` button in the *Object Inspector* panel or insert `Sender.Cancel := True` in the form's button cancel click event handler.

Accepting Input from the User

One of the common components that can accept input from the user is the **EditBox** component. This **EditBox** component has a field where the user can type in a string of characters. There are other components such as masked edit component which is an edit component with an input mask stored in a string. This controls or filters the input.

The example below illustrates what is happening when user clicks on the button after typing something in the edit box. That is, if the user did not type anything in the edit component, the event handler responds with a warning message.

```
Procedure TScriptForm.ButtonClick(Sender : TObject);
Begin
    If Edit1.Text = '' Then
    Begin
        ShowMessage('Warning - empty input!');
        Exit;
    End;
    // do something else for the input
End;
```

Note, a user can move the input focus by using the **Tab** key or by clicking with the mouse on another control on the form.

Responding to Events

When you press the mouse button on a form or a component, Altium Designer sends a message and the Scripting System responds by receiving an event notification and calling the appropriate event handler method.

See also

HelloWorld project from the `\Examples\Scripts\DelphiScript Scripts\General\` folder of your installation.

ShowModal example script from the `\Examples\Scripts\DelphiScript Scripts\General\` folder of your installation.

Writing Event Handlers

Each component has a set of event names. You as the programmer, decide how a script will react to user actions in Altium Designer. For instance, when a user clicks a button on a form, Altium Designer sends a message to the script and the script reacts to this new event. If the `OnClick` event for a button is specified it gets executed.

The code to respond to events is contained in DelphiScript event handlers. All components have a set of events that they can react on. For example, all clickable components have an `OnClick` event that gets fired if a user clicks a component with a mouse. All such components have an event for getting and losing the focus, too. However, if you do not specify the code for `OnEnter` and `OnExit` (`OnEnter` - the control has focus; `OnExit` - the control loses focus) the event will be ignored by your script.

Your script may need to respond to events that might occur to a component at run time. An event is a link between an occurrence in Altium Designer such as clicking a button, and a piece of code that responds to that occurrence. The responding code is an event handler. This code modifies property values and calls methods.

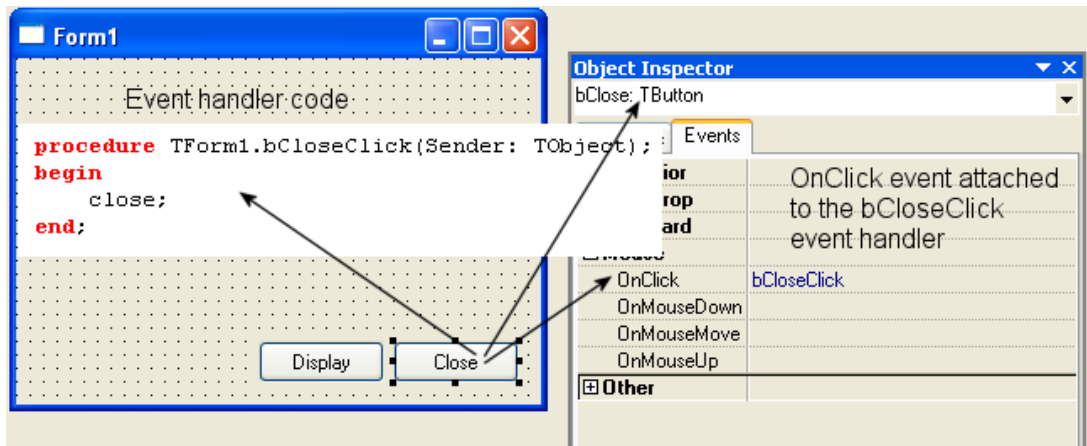
Component's Properties

To see a list of properties for a component, select a component and in the *Object Inspector* and activate the **Properties** tab.

Component's Events

To see a list of events a component can react on, select a component and in the *Object Inspector* panel, activate the **Events** tab. To create an event handling procedure, decide on what event you want your component to react to and double click the event name.

For example, select the `TButton` component from the *Tool Palette* panel and drop it on the script form, double click next to the `OnClick` event name in the *Object Inspector*. The scripting system will bring the Code Editor in focus and the skeleton code for the `OnClick` event will be created.



For example, a button has a `Close` method in the `CloseClick` event handler. When the button is clicked, the button event handler captures the on click event and the code inside the event handler gets executed. That is, the `Close` method closes the script form.

In a nutshell, you select a button component either on the form or by using the *Object Inspector* panel, select the **Events** page and double click on the right side of the `OnClick` event, a new event handler will appear on the script. Alternatively, double click on the button and the scripting system will add a handler for this `OnClick` event. Other types of components will have completely different default actions.

Component's Methods

 To see a list of methods for a component, see the [Component Reference](#) document.

Dropping Components on a Script Form

To use components from the *Tool Palette* panel in your scripts, you need to have a script form first before you can drop components on the form. Normally when you drop components on a script form, you do not need to create or destroy these objects, the script form does them for you automatically.

The scripting system automatically generates code necessary to use the component and updates the script form. You then only need to set properties, put code in event handlers and use methods as necessary to get the script form working.

Creating Components from a Script

You can directly create and destroy components in a script by passing a `Nil` parameter to the `Constructor` of a component. Normally you don't need to pass in the handle of the form because the script form takes care of it automatically for you.

For example, you can create and destroy Open and Save Dialogs (`TOpenDialog` and `TSaveDialog` classes as part of Borland Delphi Run Time Library).

Customizing Script Forms

- To make the form stay on top of other open panels for example, set the `FormStyle` property to `fsStayOnTop`
- To define the default behavior of a form, set the `FormKind` to one of the following values; `fkNone`, `fkNormal`, `fkServerPanel` or `fkModal`
- If `fkModal` is closed, then the form will be a modal form i.e. waiting for user input before proceeding such as closing the form. If `fkServerPanel` then the form will be shown as a Server panel. If `fkNormal` then the form acts as a normal non-modal form
- To remove the form's default scroll bars, change the value of the `HorzScrollBar` and `VertScrollBar` properties
- To make the form a MDI frame or MDI child, use the `FormStyle` property
- To change the form's border style, use the `BorderIcons` and `BorderStyle` properties. (The results are visible at runtime.)
- To change the icon for the minimized form, use the `Icon` property
- To specify the initial position of a form in the application window, use the `Position` property
- To specify the initial state of the form, (e.g., minimized, maximized or normal) use the `WindowState` property

- To define the working area of the form at runtime, use the `ClientHeight` and `ClientWidth` properties. (Note that `ClientHeight` and `ClientWidth` represent the area within the form's border; `Height` and `Width` represent the entire area of the form.)
- To specify which control has initial focus in the form at runtime, use the `ActiveControl` property
- To pass all keyboard events to form, regardless of the selected control, use the `KeyPreview` property
- To specify a particular menu, if your form contains more than one menu, use the `Menu` property.

Refreshing Script Forms and Components

When the surface of a script form gets out of date, for example, the controls are not updated or repainted then these controls can look frozen or corrupted due to intensive background processing from that script.

The `Update` method of the script form and many script components from the *Tool Palette* provides a way to refresh the graphical contents of the form or the specific control(s). The `Update` method has been highlighted in blue for ease of identification as shown in the example below.

StatusBar component and its Update method Example


```
Procedure TConverterForm.loadbuttonClick(Sender: TObject);
Begin
    If OpenPictureDialog1.Execute then
    Begin
        XPPProgressBar1.Position := 0;
        XStatusBar1.SimpleText := ' Loading...';
        XStatusBar1.Update;

        // loading a monochrome bitmap only
        Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);

        // Check if image is monochrome, otherwise prompt a warning
        If Image1.Picture.Bitmap.PixelFormat <> pflbit Then
        Begin
            ShowWarning('The image is not a monochrome!');
            Close;
        End;

        lImageSize.Caption := IntToStr(Image1.Picture.Width) + ' x ' +
                               IntToStr(Image1.Picture.Height) + ' mils';

        convertbutton.Enabled := True;
        LoadButton.Enabled := False;
        XStatusBar1.SimpleText := ' Ready...';
        XStatusBar1.Update;
    End;
End;
```

 Refer to the **PCB Logo Creator** script project located in the `\Examples\Scripts\Delphiscript Scripts\Pcb\PCB Logo Creator` folder of your installation.

DelphiScript Error Codes

Error	Description
%s expected but %s found	Wrong string used in the script.
%s or %s expected	Wrong string used in the script.
Function %s is already defined	Multiple instances of functions with same name in the code are not permitted. Rename the other functions that have the same name.
Unknown identifier: %s	Unknown identifier. Need to declare this identifier first before using this identifier.
Unknown variable type during writing program	The script has a Variable type which is not valid or unknown.
Unit %s already defined	Multiple instances of same unit names are not permitted. Ensure script unit names are unique.
Unit declaration error	The unit declaration is not properly defined.
Function %s not found	Missing function in the script.
Link Error	DelphiScript is unable to link the script to the required internal components.
Label <%s> already defined	Multiple instances of the same label exist in the script. Ensure labels are unique in the script.
Error in declaration block	The declaration block is not defined correctly.
Label <%s> not defined	The Goto label is not defined.
Variable <%s> already defined	Multiple instances of the same variables exist in the script. Ensure variables are unique.
Error in variable declaration block	Error exists in the variable declaration block. Wrong declarations or declarations not recognized by the scripting system.
Variable <%s> not defined	Variable was not defined, so the scripting system cannot define this variable.
Method declaration error	Method signature is illegal.
Method parameters declaration error	Wrong parameters used for the method.
Properties are not supported	Properties of an object not recognized by the scripting system.
Only class declarations allowed	Declarations other than classes were attempted to be declared.
%s declaration error	Declaration error exists in the script.
Syntax error at Line: %d Char: %d'#13#10'%s	A syntax error has occurred on the script - illegal statement, missing character or unrecognized keyword.
Bad identifier name <%s>	Invalid identifier name such as duplicated identifier name. Redefine the identifier name.
Bad identifier <%s>	Invalid identifier. Redefine a new identifier
Invalid function usage	Function not used correctly in the script - such as invalid parameters.
Invalid procedure usage	Procedure not used correctly in the script - such as invalid parameters.
Hex constant declaration error	Hex constant value not declared correctly.
Compile before run	The script needs to be compiled first before it can be executed. An internal error.

Real constant declaration error	Real type constant declaration error.
String constant declaration error	String type constant declaration error.
Unsupported parameter type	Unknown parameter type as reported by the scripting system.
Variable Result not found for %s	Variable value result not found for the specified string in the script.
Procedure %s not found	Missing procedure in the script.
Parameter %S not found	Missing parameter in the script.
Unknown reader type	An internal error.
Wrong number of params	The same procedure or function declared and implemented differently. Check the parameters between the two.
One of the params is not a valid variant type	One of the parameters of a method, function or procedure does not have a correct variant type.
Property does not exist or is readonly	An attempt to set a value to the read only property or a property does not exist.
Named arguments are not supported	Arguments used for the procedure or function not valid for the script.
Parameter not found	Missing parameter value.
Parameter type mismatch	Wrong parameter type used.
Unknown interface	This interface is not declared or defined.
A required parameter was omitted	Missing parameter required for the method, function or procedure.
Unknown error	DelphiScript has detected an unknown script error that is not defined in the internal errors table.
Invalid operation code	DelphiScript has detected an invalid operation code.

Revision History

Date	Version No.	Revision
01-Dec-2004	1.0	New product release
26-Apr-2005	1.1	Updated for Altium Designer
09-Dec-2005	1.2	Updated for Altium Designer 6
03-Mar-2006	1.3	Formatting and Images revised for Altium Designer 6
17-Mar-2006	1.4	Removed DelphiScript keywords and statements.
15-Jan-2008	1.5	Updated for Altium Designer 6.9
21-Apr-2008	1.6	New A4 size and a section on automatic updates of Script Form/Controls.
14-Jul-2008	1.7	The section on String functions expanded to show those that work and those that don't work in DelphiScript.
21-Mar-2011	-	Updated template.

Software, hardware, documentation and related materials:

Copyright © 2011 Altium Limited.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium Limited. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment.

Altium, Altium Designer, Board Insight, DXP, Innovation Station, LiveDesign, NanoBoard, NanoTalk, OpenBus, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.