Contents dCG (draft) Code Generator v0.07a

About dCG (draft) Code Generator v0.07a

A general purpose code generator for Microsoft Windows

dCG was written by: **rncbc** a.k.a. **Rui Nuno Capela**
Email: **rncbc@mail.telepac.pt**
CompuServe: **200256,2502** or **rncbc@compuserve.com**
http://ourworld.compuserve.com/homepages/rncbc/dCG1.htm

dCG is the code name for a draft code generator for Microsoft Windows. It is still under development and intended to be a script-based general purpose code generator. It is implemented using my own dCWindow C++ class library, so it will be available for the whole Windows environment: Windows 3.x, Windows 95 and Windows NT.

dCG was initially designed to address the need of generating base application code for the dCWindow class library. But intentionally, the whole code generation process is script-driven, so that it can be applied to any class library, framework or even any programming language, not only C/C++.

The basic idea, is that you have a set of source code templates that are tailored to your own specific needs, framework or environment. These templates are source files in their own nature, written in a procedural high-level scripting language, that will be fed into the dCG generator. dCG will parse those source-code templates, make the proper substitutions, and voilá, you'll end up with a new custom set of source files of a brand new application to start with.

The net result may be that you generate code exactly the way you want.

Early main features of dCG:

1. High-level procedural scripting language that control every aspect of code generation.

2. Open data-type (object) definition, management and storage via an application programming interface - dCG API.

3. User code protection. Your custom user code in the generated files can be preserved along several regenerations.

Introduction

The dCG scripting language is a procedural high-level interpreted language that can be used to create template driven text files. The dCG scripts are considered source code templates in its own nature. These templates - the source scripts - consist in interpreted statements and expressions that are embedded in literal text portions.

The aim is to produce one or more output text files that have the same image of that of the literal text portions and tailored by the interpretation and substitution of the variable portions - driven by language statements, variable and function expressions.

Statements have crucial role in flow control, I/O and user interaction. There can be only one statement per line. Statements dont produce any direct output (with the unique exception of the PRINT statement in special cases) but do control how output is to be performed. However, can be more than one expression multiple within one line and expressions are considered the main output agents of the dCG scripting. They are ruled by the dCG substitution mechanism.

Substitution Mechanism and Variables

The essential feature of dCG is the scripting substitution mechanism. All scripting statements and expressions are inscribed within the output text (just like a template) and can be recognized by its delimiter markers. By default, the script delimiter markers are **$[** and **]$**, respectively for the start and end of the script statement or expression portion. Therefore, every text that isn't delimited between these markers is considered literal text, and will be transferred to output without modification. These markers can be changed by the Options command (View Menu) dialog settings.

All source text that is found between the scripting markers are to be processed by the script interpreter and will result in some action depending if it's a statement or a simple expression. In this later case, the expression is evaluated and if the result is a character string or a numeric value, it will be transferred to current output in the same position occupied by the marked up script portion. If one or more expressions in one single line leads to an empty or blank output line, no output is made, that is, the output line is suppressed.

For example, the following script line:

```
$[UserName = "John Doe"]$
```

is an assignment statement, where a variable called `UserName` is assigned a character string. No declaration is needed to the variable. Variables are typeless, that is, they can hold whatever data type in one time. Further in the source we could write the line:

```
Authorized user: $[UserName]$
```

where there's some literal text with an embedded script portion. The script portion contains the variable name that has been assigned before. It is treated as an expression that leads to a character string result. After processing, the corresponding output should be:

```
Authorized user: John Doe
```

Variable names can have any reasonable length. Can be codified with any alphanumeric (`A`-`Z`; `0`-`9`) or the underscore character (_). However, variable names cannot start with a numeric digit, they must begin with an alphabetic (`A`-`Z`) or the underscore character (_). As the whole language, including statements and function names, variable names identifiers are not case sensitive.

A little more elaborated script (and almost as useless) can be exemplified by the following source lines:

```
$[OUTPUT "colors.out"]$
$[Colors = LIST("Red", "Green", "Blue")]$
$[Intensities = LIST("Light", "Dark")]$
$[Count = 0]$
Available colors:
$[FOR EACH Color IN Colors]$
    $[FOR EACH Intensity IN Intensities]$
        $[Count = Count + 1]$
        Color $[Color]$: $[Intensity + "-" + Color]$
    $[END FOR]$
$[END FOR]$
There are $[Count]$ colors available.
```

this will create an output file named **colors.out** with the following contents:

```
Available colors:
    Color 1: Light-Red
    Color 2: Dark-Red
    Color 3: Light-Green
    Color 4: Dark-Green
    Color 5: Light-Blue
    Color 6: Dark-Blue
There are 6 colors available.
```

Data Types and Expressions

The dCG Scripting language supports four basic data types:

Integer	A scalar 32-bit numeric integer value.
String	A general character string (<64K).
List	A container data type that can hold a unordered set of any other data type items, including lists and objects.
Object	A structured data type that is implemented as an associative array. Each data item, an object member, has a name identifier and a value that can be of any data type, including lists and objects.

An expression is every script entity that isnt considered a statement. An expression is evaluated and lead to a result of a certain data type. If this result is of string or integer type it is transferred to current output. However, one can use expressions to feed almost every statement parameter or function argument.

Expressions are constructed by applying standard operators to one or more operands. Operands can be any literal value, variable, object member (that is treated just like a special case of a variable) or function return values.

The defined operators are shown below, in precedence order:

Operator	Description
`.` (dot)	Object member accessor.
`[ ]`	List item accessor (by index expression)
`~`	Bit-wise negation
`!`	Logical negation
`*`	Integer multiplication
`/`	Integer division
`%`	Integer remainder
`+`	Integer addition; String concatenation
`-`	Integer Subtraction
`<<`	Bit-wise integer shift left
`>>`	Bit-wise integer shift right
`==` or `=`	Logical equality (*equal to*); Integer; String
`!=` or `<>`	Logical inequality (*not equal to*) ; Integer; String
`<=`	Logical *less or equal than*; Integer; String
`<`	Logical *less than*; Integer; String
`>=`	Logical *greater or equal than*; Integer; String
`>`	Logical *greater than*; Integer; String
`&`	Bit-wise integer AND
`^`	Bit-wise integer XOR
`|`	Bit-wise integer OR
`&&`	Logical AND
`||`	Logical OR

User defined Objects

Objects are just static data structures. The dCG script language is not an object-oriented language so don't start looking for any inheritance mechanism or any methods associated to dCG objects. It's just a way to aggregate data items that have a common relationship. Nothing more simple than that.

An example of an object variable declaration within a script is shown below:

```
$[OBJECT MyApplication]$
    $[Name   = "MyApp"]$
    $[Dir    = "C:\USR\MYAPP"]$
    $[System = LIST("Windows 3.x", "Win32")]$
$[END OBJECT]$
```

This will result in declaring an object variable named `MyApplication` with three data members. The first member, entitled `Name`, is defined as a character string with default value "MyApp". The second member, named `Dir`, is also a string with initial value "C:\USR\MYAPP". The third and last member, identified by `System`, is defined as a list of two strings. After definition the `MyApplication` object and any of its members can be used as individual variables.

Object member names have the same restrictions applicable to stand-alone variable names, and are not case sensitive. An object member is accessed by using a dot character followed by the member name identifier. They can also be the target (left hand operand) of an assignment statement. For example, for the following script lines:

```
Application Name.....: $[MyApplication.Name]$
Application Directory: $[MyApplication.Dir]$
Application Platform.: $[MyApplication.System[0]]$
```

the corresponding output should be:

```
Application Name.....: MyApp
Application Directory: C:\USR\MYAPP
Application Platform.: Windows 3.x
```

Notice the way one can access a list item by index, as it is done in `MyApplication.System[0]` to retrieve the first list item. Object members can be of any data type, determined by the initial value expression on the declaration (right side of the equal character). Assigning one object variable to another is just a way to make an object clone (member-wise copy).

User defined Functions

Other than the pre-defined functions, the definition of new functions is supported at script level. The following example is a simple example of the definition of an user function named `MyFunc` that calculates the mean value of its two arguments:

```
$[PROCEDURE MyFunc ( Arg1, Arg2 )]$
    $[Result = (Arg1 + Arg2) / 2]$
    $[RETURN Result]$
$[END PROC]$
```

After this definition, the script:

```
Half-way value: $[MyFunc(100,200)]$
```

will have the following output:

```
Half-way value: 150
```

Functions have a single return value that can be of any defined data type and may be part of any consistent expression. A function must be defined before it is called and may have any number of arguments.

Every function that accept one or more parameter (arguments), including predefined ones, may be called in two different forms. The usual way is passing all arguments between parenthesis and separated by commas, like `MyFunc(X,200)`.

The second way is by attaching the function name to a variable identifier, using the dot character, like `X.MyFunc(200)`. This means that the variable value is to be passed as if it were the first argument in regular form. If the affected variable is of the same type as the function return value, it will be assigned the later. This is the only way one can mimic a parameter passing by reference. In all other cases, all parameter passing is done by value (except for the internal functions).

This means that `MyFunc(X,200)` and `X.MyFunc(200)` are equivalent function calls, but the later will also set the value of `X` to the functions return value.

In the following examples both forms of calling a function are shown.

Regular calling form:

```
$[X = MyFunc(100,200)]$
The result is: $[X]$
```

Dot calling form:

```
$[X = 100]$
The result is: $[X.MyFunc(200)]$
```

Both ways are functionally equivalent and would result in the same output:

```
The result is: 150
```

User defined Dialogs

The dCG Scripting language supports the definition of dynamic dialog boxes to let the user interact with the interpretation and generation process. Each in-line defined dialog is mapped to an object variable whose members can be accessed as any other data value in a statement or expression.

This is an example of an in-line dialog definition, that would be mapped to an object variable named `MyDialog`:

```
$[DIALOG MyDialog, "Name Entry Example", 30, 40, 140, 40]$
    $[CONTROL Prompt, Text, "Enter Your Name:", 4,  4, 80,  8]$
    $[CONTROL Name,   Edit,         "",         4, 14, 80, 12]$
    $[CONTROL Ok,     OkButton,     "Ok",      90,  4, 46, 14]$
    $[CONTROL Cancel, CancelButton, "Cancel",  90, 22, 46, 14]$
$[END DIALOG]$
```

In this dialog there is declared four child window controls: one static text control (`Prompt`), one edit box (`Name`) and two standard push-buttons (`Ok` and `Cancel`). The following script lines will display the dialog and process the data the user has entered in it:

```
$[DIALOG MyDialog]$
$[IF MyDialog.Result]$
    My name is $[MyDialog.Name.Text]$!
$[ELSE]$
    I have no name.
$[END IF]$
```

Supposing the user had typed "John Doe" in the dialog edit box control (which contents referenced as `MyDialog.Name.Text`) and chose the Ok button (yielding the dialog's object member `MyDialog.Result` to a non-zero value) the output would be:

```
My name is John Doe!
```

If the Cancel button had been chosen (or the user had closed the dialog by selecting the control menu or hitting the Escape key) the script output would certainly be:

```
I have no name.
```

That is, `MyDialog.Result` now has a 0 (zero) value.

User Code Protection

Each time a script is interpreted any output files are completely overwritten. However there is the possibility to mark a code block as being protected, that is, not overwritten upon script interpretation, preserving any modification the user had made after a previous generation. This is called user code protection and the following example shows how a protect block can be defined within a script:

```
// $[PROTECT "ProtectTag1"]$
// Some first-time generation text...
pstrText = "Default text";
// $[END PROTECT]$
```

After generation, the corresponding output file would look like:

```
// %PROTECT ProtectTag1
// Some first-time generation text...
pstrText = "Default text";
// %ENDPROTECT
```

Any text that falls between the `%PROTECT`/`%ENDPROTECT` pair may be freely modified and is known to be a protected code block. Next time the same output file is generated all modifications made within this block will stay intact. Normally the `%PROTECT`/`%ENDPROTECT` marks should be coded as part of a comment area. In the example above, C/C++ is the target language; if it were a COBOL example one would use an asterisk '`*`' on 7th column; in PASCAL the curly brace `{`...`}` pair would lead the same purpose.

Statements dCG Scripting Reference

See also: Functions

Conditional Statements

IF
SELECT
CASE

Loop Statements

FOR	Integer iteration
FOREACH	List iteration
WHILE	General iteration
BREAK	Exit current loop
CONTINUE	Skip to next iteration

I/O Statements

INCLUDE	Start processing another script
OUTPUT	Name current output file (create)
APPEND	Name current output file (append)
PRINT	Display expression result

Procedure Statements

PROCEDURE	User function definition
RETURN	Exit current user function or included script

Object Statements

OBJECT	User object definition
INSPECT	Inspect object definition

Dialog Statements

DIALOG	User dialog definition
CONTROL	Dialog item definition

User Code Protection Statements

PROTECT	User protected block definition

APPEND Statement

Usage:

APPEND *sFilename*

Description:

Opens the output file with filename specified by *sFilename* for appending and closes any output file currently active. All subsequent output will be written and appended to this file, until one of the following conditions occur: another APPEND statement is encountered, an OUTPUT statement or the end of input scripting file has been reached.

See Also:

INCLUDE
OUTPUT

BREAK Statement

Usage:

BREAK

Description:

Passes control to the next statement following the end of the current loop construct, such as FOR, FOREACH and WHILE statements. Should be only used within a loop construct and has complementary behavior of that of the CONTINUE statement.

See Also:

CONTINUE
FOR
FOREACH
WHILE

CASE Statement

Usage:

CASE *iCondition*

Description:

Is the branch condition of a SELECT statement construct. If the expression *iCondition* evaluates to a non-zero value the statements up to the next CASE or end of SELECT will be executed. Only one CASE condition within a SELECT construct should be true. If none evaluates to a non-zero value the statements following the OTHERWISE statement, if any, are executed.

See Also:

IF
SELECT

CONTINUE Statement

Usage:

CONTINUE

Description:

Passes control to the next statement following the start of the current loop construct, after the loop-condition is re-evaluated. Should be only used within a loop construct, such as FOR, FOREACH and WHILE statements. Has complementary behavior of that of the BREAK statement.

See Also:

BREAK
FOR
FOREACH
WHILE

CONTROL Statement

Usage:

CONTROL *CtlName*, *CtlType*, *sText*, *iX*, *iY*, *iWidth*, *iHeight*

Description:

Defines a control item within a DIALOG definition construct. A control is a child window that is displayed within the client area of the dialog. *CtlName* is the control literal member identifier. This is how the control will be referenced in the dialog object as a member. *CtlType* is the control class type literal. This determines the control type and behavior, such as if it is a text label or a text entry box. See below for a list of implemented control types. *sText* is the control caption text character string. *iX* and *iY* are the top-left position coordinates within the dialog client area and *iWidth* and *iHeight* are the horizontal and vertical extents of the control window, respectively.

Control coordinates are relative to the dialog window and are expressed in dialog units, which are based on increments of 1/8 (for horizontal coordinates) and 1/12 (for vertical coordinates) of the dialog system font (MS Sans Serif 8pt).

Each control item is represented as an object member within the dialog object, with name given by *CtlName*. The control itself is an object structure with the following members:

Member	**Type**	**Description**
`Type`	String	Control type (see below).
`Text`	String	Control caption text.
`X`	Integer	Horizontal coordinate of controls left position.
`Y`	Integer	Vertical coordinate of controls top position.
`Width`	Integer	Horizontal extent.
`Height`	Integer	Vertical extent.
`ID`	Integer	Control internal window id-number.
`Select`	Integer/List	Selection state, index or selection list.
`Items`	List	Control list contents.

Available control types:

CtlType	**Description**
`OkButton`	Standard OK default push-button. Selecting this button control terminates the dialog, updating the dialog object with all control settings. Pressing the ENTER key yields to the same action of selecting this button. The dialog `Result` member is set with the non-zero (1) result code.
`CancelButton`	Standard Cancel push-button. A user chooses the Cancel button to close the dialog without taking any action. Selecting this button control terminates the dialog without updating the dialog object with any control settings. Pressing the ESC key yields to

	the same action of selecting this button. The dialog **`Result`** member is set with the 0 (zero) result code.
`PushButton`	User defined push-button. Selecting this button control terminates the dialog, updating the dialog object with all control settings. The control **`Select`** member and the dialog **`Result`** member are set with the non-zero (1) result code.
`RadioButton`	Option radio button. This control type is generally part of an option group where only one button is selected in one time. The control **`Select`** member will be set to a non-zero value if this control is selected.
`CheckBox`	Check box button. The control **`Select`** member will be set to a non-zero value if this control is selected (i.e. checked).
`Edit`	Single-line text edit box into which the user can enter information. The control **`Text`** member will be set to the text the user entered, and can be used to set the initial control text.
`MLEdit`	Multiline text edit box into which the user can enter information. The control **`Text`** member will be set to the text the user entered, and can be used to set the initial control text.
`Text`	Static text label. The control **`Text`** member can be used to set the control caption text.
`ListBox`	Single selection list box. The control **`Items`** member is used to set the initial control contents in the form of list object with only character string items. The item that is selected is set by the value of the control **`Select`** member as a zero based index.
`MSListBox`	Multiple selection list box. The control **`Items`** member is used to set the initial control contents in the form of list object with only character string items. The items that are selected are given by the control **`Select`** member as a list object.
`ComboBox`	Combo box, a single control that is a combination of an edit text box and a single selection list box. The control **`Text`** member is used to set the edit box text. The control **`Items`** member is used to set the initial control contents in the form of list object with only character string items. The item that is selected is set by the value of the control **`Select`** member as a zero based index.
`DropListBox`	Drop-down list box. The control **`Items`** member is used to set the initial control contents in the form of list object with only character string items. The item that is selected is set by the value of the control **`Select`** member as a zero based index.

See Also:

User defined Dialogs
DIALOG
OBJECT

DIALOG Statement

Usage:

> **DIALOG** *DlgName* **,** *sTitle*, *iX*, *iY*, *iWidth*, *iHeight*
>   **CONTROL** *CtlName*, *CtlType*, *sText*, *iX*, *iY*, *iWidth*, *iHeight*
>   [**CONTROL** *CtlName*, *CtlType*, *sText*, *iX*, *iY*, *iWidth*, *iHeight*]
>   ...
> **END**[ ]**DIALOG**

or:

> **DIALOG** *DlgName*

Description:

Encloses the control statements that define a dialog box created within the script. A dialog definition consist of the DIALOG statement and a series of CONTROL statements for each one of the dialog control elements, such as the OK button, Cancel button and so on. *sTitle* is the dialog caption title character string. *iX* and *iY* are the top-left position coordinates within the dialog client area and *iWidth* and *iHeight* are the horizontal and vertical extents of the dialog window respectively.

Dialog coordinates are expressed in dialog units, which are based on increments of 1/8 (for horizontal coordinates) and 1/12 (for vertical coordinates) of the dialog system font (MS Sans Serif 8pt).

After the dialog definition construct, each control element can be initialized and the dialog box can be displayed by means of the second form of the DIALOG statement. The dialog box is closed as soon the user chooses any push-button control (OkButton, CancelButton or PushButton control types), closes the dialog from the control menu, or presses the ESC key (both these later cases are equivalent to selecting the Cancel button). Unless the Cancel button has been chosen all control objects are updated with the dialog settings.

Each dialog is represented as an object structure with the following members:

Member	Type	Description
`Title`	String	Dialog caption title.
`X`	Integer	Horizontal coordinate of dialogs left position.
`Y`	Integer	Vertical coordinate of dialogs top position.
`Width`	Integer	Horizontal extent.
`Height`	Integer	Vertical extent.
`Result`	Integer	Dialog result code.
Control...	Object	The following members represent each dialog control item. Control members are identified by their literal names, as given by *CtlName*.

Although any statement or expression is allowed between an dialog definition (not only control declaration statements) it is highly not recommended doing so.

See Also:

User defined Dialogs
CONTROL
OBJECT

FOR Statement

Usage:

FOR *Identifier* **=** *iStart* **TO** *iEnd* [**STEP** *iStep*]
    ...
END[ ]**FOR**

Description:

The statements within this loop construct are executed repeatedly, incrementing the value of the integer *Identifier* from *iStart* until *iEnd*, by *iStep* increments. If the *iStep* parameter is omitted it is assumed an unit increment (*iStep* = 1). After the value of *Identifier* is incremented past the value of *iEnd* the loop terminates and execution control passes to the statement following ENDFOR. If *iStart* is greater than *iEnd* then no iteration takes place.

See Also:

BREAK
CONTINUE
FOREACH
WHILE

FOR EACH Statement

Usage:

FOR[ ]**EACH** *Identifier* **IN** *list* [**WHERE** *iCondition*]
    ...
END[ ]**FOR**

Description:

The statements within this loop construct are executed repeatedly, iterating the *list* contents. The item *Identifier* is assigned to each list item of *list* that evaluates the *iCodition* expression to a non-zero value. If the *iCondition* expression is omitted the loop will iterate for every item of *list* without filtering. If *list* is empty then no iteration takes place. The *list* contents can be of any item type and mixed types are allowed (but not recommended).

See Also:

BREAK
CONTINUE
FOR
WHILE

IF Statement

Usage:

IF *iCondition* [**THEN**]
    ...
[**ELSE**]
    ...
END[ ]**IF**

Description:

Executes statements conditionally, being *iCondition* the conditional expression. If *iCondition* evaluates to a non-zero value the execution continues to the immediately following statements until an ELSE or ENDIF statement. If iCondition evaluates to 0 (zero) the execution control passes to the statements following the ELSE or ENDIF statements.

See Also:

SELECT
CASE

INCLUDE Statement

Usage:

INCLUDE *sFilename*

Description:

Opens a input scripting file with filename specified by *sFilename* for immediate processing. The execution control is passed to the first statement on the input file and returns after its end has been reached or a RETURN statement has been found.

See Also:

APPEND
OUTPUT

INSPECT Statement

Usage:

INSPECT *oObject*

Description:

Displays an inspector dialog for the object variable *oObject*. This dialog displays all member names and corresponding values of the given object variable. Member values can be edited and navigation within compound objects is permitted.

See Also:

OBJECT

OBJECT Statement

Usage:

OBJECT *ObjName*
  MemberName **=** *expr*
 [*MemberName* **=** *expr*]
  ...
END[ ]**OBJECT**

Description:

Declares and defines an object variable. The new object variable is created with the literal name *ObjName*. An object definition consists of an OBJECT statement followed by a series of member definition statements. These statements are similar to regular assignment statements where the left operand designates the new member name and the left operand the initial member value and type. This can be any valid expression with a special restriction: it cannot call an user function where another object definition is made.

Although any statement or expression is allowed between an object definition (not only member declaration statements) it is highly not recommended doing so.

See Also:

User defined Objects
DIALOG
CONTROL
INSPECT

OUTPUT Statement

Usage:

OUTPUT *sFilename*

Description:

Creates a new output file with filename specified by *sFilename* and closes any output file currently active. All subsequent output will be written and appended to this file, until one of the following conditions occur: another OUTPUT statement is encountered, an APPEND statement or the end of input scripting file has been reached.

See Also:

APPEND
INCLUDE

PRINT Statement

Usage:

PRINT *expr* [**;** ...]

Description:

Displays the value of one or more expressions, given as semicolon or comma separated parameters. Depending on the Options command (View Menu) dialog settings the value can be displayed on the Messages window pane or written to the current output file. For the later, only strings and integers are meaningful.

See Also:

MSGBOX

PROCEDURE Statement

Usage:

PROC[**EDURE**] *ProcName* **(** [*ArgName* [**,** ...]] **)**
...
END[ ]**PROC**[**EDURE**]

Description:

Defines a function procedure with the *ProcName* literal name. A function is a series of statements with a single return value that can be called repeatedly from the main script or even from any other function procedures. The defined procedure is called just like any intrinsic function and can be part of any expression.

Function procedures must be defined before they are called the first time. A function procedure is called by using *ProcName* as a function identifier, followed by its argument list enclosed in parenthesis. The argument list (*ArgName*, ...) stands for the function formal parameters and can be treated as local variables that have been assigned with the argument values upon invocation. Every variable or symbol defined within the procedure is considered local and will be not available or visible after return.

When a procedure is invoked, the execution control passes to the statement that immediately follows the PROCEDURE definition header and continues until a RETURN statement or the end of the procedure block (ENDPROC statement) is found. Then the execution control returns to the expression evaluation statement where the procedure has been invoked. Every function procedure has a return value, being it the value specified in the RETURN statement or, if the end of procedure is reached, the last expression that has been evaluated.

See Also:

User defined Functions
RETURN
Functions

PROTECT Statement

Usage:

PROTECT *sProtectTag*
   ...
END[ ]**PROTECT**

Description:

Defines a user code protection block. Any code between the PROTECT and ENDPROTECT block is preserved among script interpretations (generations). A user code protection block is identified by an unique tag specified by the character string *sProtectTag*.

See Also:

User Code Protection

RETURN Statement

Usage:

RETURN [*expr*]

Description:

Forces the execution control to return to the expression evaluation statement where the current executing procedure has been invoked. Specifies the return value of an user defined function procedure. The return value, given by the *expr* expression, can be of any type. If omitted, an empty string is assumed.

See Also:

User defined Functions
PROCEDURE

SELECT Statement

Usage:

SELECT
  CASE *iCondition*
    ...
 [**CASE** *iCondition*]
    ...
 [**OTHER**[**WISE**]]
    ...
END[ ]**SELECT**

Description:

Chooses one of several alternatives. If one of the *iCondition* expressions evaluates to a non-zero value then the execution control is passed to the statement just after the corresponding CASE and continues up to the next CASE or end of SELECT statement. Only one CASE condition within a SELECT construct should evaluate to a non-zero value (true). If none evaluates to a non-zero value the statements following the OTHERWISE statement, if exist, are executed.

See Also:

CASE
IF

WHILE Statement

Usage:

WHILE *iCondition*
...
END[ ]WHILE

Description:

The statements within this loop construct are executed repeatedly, as long as the *iCondition* expression evaluates to a non-zero value. If the condition evaluates to 0 (zero) the loop terminates and execution control passes to the statement following ENDWHILE.

See Also:

BREAK
CONTINUE
FOR
FOREACH

Functions   dCG Scripting Reference

See also: Statements

String functions

FIND	Find string occurence
LEFT	Get leftmost substring
LEN	Get string length
LOWER	Convert string to lowercase
PAD	Make string fixed length with padding
PREFIX	Test for two equally prefixed strings
PROPER	Convert string initials to uppercase
REPLACE	Replace string occurences
RIGHT	Get rightmost substring
SPACE	String of blank character(s)
STR	Convert integer to string
STRING	String of same character
STRIP	Strip all non-alphanumeric characters from a string
SUBSTR	Get substring
SUFFIX	Test for two equally terminated strings
TAB	String of tab character(s)
TRIM	Remove leading and trailing blanks
UPPER	Convert string to uppercase
VAL	Convert string to integer

Character functions

ASC	Get ANSI code from character
CHR	Get character from ANSI code

List functions

ADD	Add item(s) to list
AVG	Average item value of a list
COUNT	Get list item count
FIND	Find list item occurrence
LIST	Create list
MAX	Maximum item value of a list
MIN	Minimum item value of a list
REMOVE	Remove item from list
RESET	Reset list contents
SUM	Sum of item values of a list

Special functions

DATE	Get current date
LISTTOSTR	Convert list to delimited string
MEMBERLIST	Edit string using mask
PICTURE	Retrieve values from a list of objects
STRTOLIST	Convert delimited string to list
TIME	Get current time

File functions

EXIST	Test if file exists
FDIR	Get directory from pathname
FEXT	Get extension from pathname
FNAME	Get filename from pathname
REMOVEFILE	Deletes a file
RENAMEFILE	Renames a file

Directory functions

CHDIR	Change directory
CURDIR	Get current directory
MKDIR	Create directory
RMDIR	Remove directory

Dialog functions

DIRDIALOG	Directory selection dialog
FILEDIALOG	File Open/Save dialog
MSGBOX	Message dialog box
PROMPT	Simple input dialog box

Data Type functions

ISINTEGER	Test for integer data type
ISLIST	Test for string data type
ISSTRING	Test for list data type
ISOBJECT	Test for object data type
OBJECT	Create registered type object

I/O functions

CLOSE	Close an open file stream
EOF	Load data item from a disk file
LOADITEM	Open a file stream for further processing
OPEN	Test for end-of-file of an input file stream
READ	Read a string from an input file stream
READLN	Read a line from an input file stream
SAVEITEM	Store data item into a disk file
WRITE	Write a string to an output file stream
WRITELN	Write a line to an output file stream

Environment functions

DCGDIR	Get component installation directory
DCGNAME	Get component installation filename
DCGVERSION	Get component installation version string

ADD Function

Usage:

list **= ADD(** *list***,** *expr* [**,** ...] **)**
*list***.ADD(** *expr* [**,** ...] **)**

Description:

Appends one or more items to a list object. Each function argument will be became a new item of the returned list object, can be of any data type and type mixing is allowed. Returns the new list object.

See Also:

COUNT
FIND
LIST
REMOVE
RESET

ASC Function

Usage:

iCharCode **= ASC(** *sCharacter* **)**
iCharCode **=** *sCharacter***.ASC()**

Description:

Returns the ANSI character code of the character specified in *sCharacter* argument.

See Also:

CHR
SPACE
STRING
TAB

AVG Function

Usage:

iAvg **= AVG(** *list* **)**
iAvg **=** *list***.AVG()**

Description:

Returns the average value of the items in the *list* argument. The *list* should be made of items of the same data type, and only integer, string and list item data types are meaningful. For integer items the average item value is computed. For character string items the average string length is returned. If the argument is a list of lists the average list item count is returned.

See Also:

COUNT
FIND
LIST
MAX
MIN
SUM

CHDIR Function

Usage:

iSuccess **= CHDIR(** *sDirectory* **)**
iSuccess **=** *sDirectory***.CHDIR()**

Description:

Changes the current directory to a new one, specified by *sDirectory* argument. If the character string *sDirectory* does not specify a path, the new directory is assumed relative to the current process directory. Returns a non-zero value if the current directory has been changed successfully. Returns 0 (zero) if an error has occurred or the specified directory does not exist.

See Also:

CURDIR
DCGDIR
DIRDIALOG
FDIR
FILEDIALOG
MKDIR
RMDIR

CHR Function

Usage:

sCharacter **= CHR(** *iCharCode* **)**
sCharacter **=** *iCharCode***.CHR()**

Description:

Returns a single character whose ANSI character code is given by the *iCharCode* integer argument.

See Also:

ASC
SPACE
STRING
TAB

CLOSE Function

Usage:

CLOSE(*iFile* **)**
*iFile***.CLOSE()**

Description:

Closes a file stream currently open. The argument *iFile* must be a valid file handle value that was returned by a previous call to function OPEN, otherwise the function call will fail.

See Also:

EOF
OPEN
READ
READLN
WRITE
WRITELN

COUNT Function

Usage:

iCount **= COUNT(** *list* **)**
iCount **=** *list***.COUNT()**

Description:

Returns the number of items of a list object.

See Also:

ADD
AVG
FIND
LIST
MAX
MIN
REMOVE
RESET
SUM

CURDIR Function

Usage:

sDirectory **= CURDIR()**

Description:

Returns the current process directory witch includes the current drive specification. No trailing directory slash is appended to the returned string, unless it is the root directory.

See Also:

CHDIR
DCGDIR
DIRDIALOG
FDIR
FILEDIALOG
MKDIR
RMDIR

DATE Function

Usage:

sDate **= DATE()**

Description:

Retrieves the current date. Returns the current date as a character string in the format *yyyymmdd*.

See Also:

PICTURE
TIME

DCGDIR Function

Usage:

sDirectory **= DCGDIR()**

Description:

Returns the dCG installation directory which includes the installation drive specification. The installation directory is where the dCG executable file is currently located. No trailing directory slash is appended to the returned string, unless it is the root directory.

See Also:

CHDIR
DIRDIALOG
DCGNAME
DCGVERSION
FDIR
FILEDIALOG
MKDIR
RMDIR

DCGNAME Function

Usage:

sFilename **= DCGNAME()**

Description:

Returns the dCG program module filename which includes the file extension.

See Also:

DCGDIR
DCGVERSION
FEXT
FILEDIALOG
FNAME

DCGVERSION Function

Usage:

sVersion **= DCGVERSION()**

Description:

Returns the current dCG program module version string.

See Also:

DCGDIR
DCGNAME
DCGVERSION

DIRDIALOG Function

Usage:

sDirectory **= DIRDIALOG(** *sTitle* **)**

Description:

Displays a directory selection dialog, with caption title specified by *sTitle*. The user will have the option to create, remove or select a directory from the file system. Returns the complete path of the selected directory. If the Cancel button is chosen an empty string is returned. The initial directory is assumed to be the current process directory.

See Also:

CHDIR
CURDIR
DCGDIR
FDIR
FILEDIALOG
MKDIR

EOF Function

Usage:

iEof **= EOF(** *iFile* **)**

Description:

Test if a file stream has reached the end. Returns a non-zero value if the file stream with handle *iFile* has the end-of-file status, otherwise it will return a zero value. The argument *iFile* must be a valid file handle value that was returned by a previous call to function OPEN, otherwise the function call will fail.

See Also:

CLOSE
OPEN
READ
READLN
WRITE
WRITELN

EXIST Function

Usage:

iExist **= EXIST(** *sFilename* **)**
iExist **=** *sFilename***.EXIST()**

Description:

Tests if the specified file exists. Returns a non-zero value if the filename is valid, 0 (zero) otherwise.

See Also:

DIRDIALOG
FDIR
FEXT
FILEDIALOG
FNAME
REMOVEFILE
RENAMEFILE

FDIR Function

Usage:

sDirectory **= FDIR(** *sFilename* **)**
*sFilename***.FDIR()**

Description:

Returns the directory portion of the filename specification given by *sFilename* string argument. The returned string includes the leading drive specification. If the character string *sDirectory* does not specify a path, the current directory will be returned.

See Also:

DIRDIALOG
EXIST
FEXT
FILEDIALOG
FNAME

FEXT Function

Usage:

sExtension **= FEXT(** *sFilename* **)**
*sFilename***.FEXT()**

Description:

Returns the file extension portion of the filename specification given by *sFilename* string argument. The returned string includes the leading period (.).

See Also:

DIRDIALOG
EXIST
FEXT
FILEDIALOG
FNAME

FILEDIALOG Function

Usage:

sFilename **= FILEDIALOG(** *sTitle***,** *sMask* [**,** *iSave*] **)**

Description:

Displays a file open dialog (or file save dialog), with caption title specified by *sTitle* and a file specification mask (wildcard) given by *sMask*. If a file save dialog is to be displayed the *iSave* argument should be supplied with a non-zero value. Returns the complete path of the selected file. If the Cancel button is chosen an empty string is returned. The initial directory is assumed to be the current process directory.

See Also:

DIRDIALOG
EXIST
FEXT
FDIR
FNAME
REMOVEFILE
RENAMEFILE

FIND List Function

Usage:

iIndex **= FIND(** *list***,** *item* **)**
iIndex **=** *list***.FIND(** *item* **)**

Description:

Scans the *list* argument for the first occurrence of *item*. Returns the item position index of the first occurrence of *item*. Only integer and string items are meaningful for the scan.   If *item* does not occur in *list*, a negative (-1) index will be returned. The scanning direction is first-to-last. The returned index is zero based which implies that 0 (zero) stands for the first list item, 1 (one) for the second, and so on.

See Also:

ADD
AVG
COUNT
LIST
MAX
MIN
REMOVE
RESET
SUM

FIND String Function

Usage:

iIndex **= FIND(** *sString*, *sTarget* **)**
iIndex **=** *sString***.FIND(** *sTarget* **)**

Description:

Scans the character string *sString* argument for the first occurrence of *sTarget*. Returns the character position index of the starting position of *sTarget*. If *sTarget* does not occur in *sString*, a negative (-1) index will be returned. The scanning direction is left-to-right. The returned position index is zero based which implies that 0 (zero) stands for the first character position, 1 (one) for the second, and so on.

See Also:

LEN
LISTTOSTR
PREFIX
REPLACE
STRTOLIST
SUBSTR
SUFFIX

FNAME Function

Usage:

sName **= FNAME(** *sFilename* **)**
*sFilename***.FNAME()**

Description:

Returns the name and extension portion of the filename specification given by *sFilename* string argument.

See Also:

DIRDIALOG
EXIST
FEXT
FDIR
FILEDIALOG

ISINTEGER Function

Usage:

iIsInteger **= ISINTEGER(** *expr* **)**

Description:

Returns a non-zero value if *expr* given as argument is of integer type.

See Also:

ISLIST
ISSTRING
ISOBJECT

ISLIST Function

Usage:

iIsList **= ISLIST(** *expr* **)**

Description:

Returns a non-zero value if *expr* given as argument is a list object.

See Also:

ISINTEGER
ISSTRING
ISOBJECT

ISSTRING Function

Usage:

iIsString **= ISSTRING(** *expr* **)**

Description:

Returns a non-zero value if *expr* given as argument is of character string type.

See Also:

ISINTEGER
ISLIST
ISOBJECT

ISOBJECT Function

Usage:

iIsObject **= ISOBJECT(** *expr* **)**

Description:

Returns a non-zero value if *expr* given as argument is an object.

See Also:

ISINTEGER
ISLIST
ISSTRING

LEFT Function

Usage:

sString **= LEFT(** *sString*, *iLength* **)**
*sString***.LEFT(** *iLength* **)**

Description:

Returns the leftmost *iLength* characters of the *sString* argument.

See Also:

LEN
PREFIX
RIGHT
SUBSTR
SUFFIX

LEN Function

Usage:

iLength **= LEN(** *sString* **)**
iLength **=** *sString***.LEN()**

Description:

Returns the character string *sString* argument length in characters.

See Also:

LEFT
PAD
PREFIX
RIGHT
STRIP
SUBSTR
SUFFIX
TRIM

LIST Function

Usage:

list **= LIST(** [*expr* [**,** ...]] **)**

Description:

Creates a list object with items given by the variable number of arguments specified. Each function argument will became an item of the returned list object, can be of any data type and type mixing is allowed. If no arguments are given an empty list is returned.

See Also:

ADD
AVG
COUNT
FIND
MAX
MIN
REMOVE
RESET

LISTTOSTR Function

Usage:

sString **= LISTTOSTR(** *list*, *sSeparator* **)**
sString **=** *list***.LISTTOSTR(** *sSeparator* **)**

Description:

Converts a list object into a delimited character string. Each item of the argument *list* will be concatenated, separated by each other by the character string *sSeparator*. Returns the complete concatenated string.

See Also:

FIND (in list)
FIND (in string)
REPLACE
STRTOLIST

LOADITEM Function

Usage:

item **= LOADITEM(** *sFilename* **)**

Description:

Reads a single data item from the external file with name given by *sFilename*. The data item can be of any data type and should be previously stored with SAVEITEM function. Only one data item can be stored in a file at a time, however this can be a compound object or list. The return value is the data item that is read and should be checked for the correct data type. In case of a read error, or the file doesnt exist or cannot be found, the script will be aborted immediately.

See Also:

SAVEITEM
ISINTEGER
ISLIST
ISSTRING
ISOBJECT

LOWER Function

Usage:

sString **= LOWER(** *sString* **)**
*sString***.LOWER()**

Description:

Converts all characters of *sString* to lowercase. Returns the lowercase version of argument *sString*.

See Also:

PAD
PROPER
STRIP
TRIM
UPPER

MAX Function

Usage:

iMax **= MAX(** *list* **)**
iMax **=** *list***.MAX()**

Description:

Returns the maximum value of the items in the *list*. The *list* should be made of items of the same data type, and only integer, string and list item data types are meaningful. For integer items the maximum item value is computed. For character string items the maximum string length is returned. If the argument is a list of lists the maximum list item count is returned.

See Also:

AVG
COUNT
FIND
LIST
MIN
SUM

MEMBERLIST Function

Usage:

lMembers **= MEMBERLIST(** *lObjects*, *sMemberName* **)**

Description:

Returns the list of object member values of the object list given by *lObjects*. The member name to be extracted is given by *sMemberName*. The list of *lObjects* must be all made of items with the same object structure, or the function will fail, aborting the script.

See Also:

LIST
LISTTOSTR
STRTOLIST

MIN Function

Usage:

iMin **= MIN(** *list* **)**
iMin **=** *list***.MIN()**

Description:

Returns the minimum value of the items in the *list*. The *list* should be made of items of the same data type, and only integer, string and list item data types are meaningful. For integer items the minimum item value is computed. For character string items the minimum string length is returned. If the argument is a list of lists the minimum list item count is returned.

See Also:

AVG
COUNT
FIND
LIST
MAX
SUM

MKDIR Function

Usage:

iSuccess **= MKDIR(** *sDirectory* **)**
iSuccess **=** *sDirectory***.MKDIR()**

Description:

Creates a new directory, specified by *sDirectory* argument. If the character string *sDirectory* does not specify a path, the new directory is assumed relative to the current process directory. Returns a non-zero value if the new directory has been created successfully. Returns 0 (zero) if an error has occurred or the specified directory already exists.

See Also:

CHDIR
CURDIR
DCGDIR
DIRDIALOG
FDIR
FILEDIALOG
RMDIR

MSGBOX Function

Usage:

iResult **= MSGBOX(** *sTitle*, *sText* [, *iType*] **)**

Description:

Displays a message in a message box. The message box will have a caption title of *sTitle*, a message text given by *sText* and *iType* can represent the icon and buttons displayed in the box. Returns a value according to the button the user chooses in the message box.

iType is the sum of three values, one from each of the following groups.

Group	Value	Meaning
Button	0	OK button (default).
	1	OK and Cancel buttons.
	2	Abort, Retry, and Ignore buttons.
	3	Yes, No, and Cancel buttons.
	4	Yes and No buttons.
	5	Retry and Cancel buttons.
Icon	0	No icon (default).
	16	Stop icon.
	32	Question icon.
	48	Exclamation icon.
	64	Information icon.
Default	0	First button is the default (default).
	256	Second button is the default.
	512	Third button is the default.

The following values can be returned:

-1	First button chosen (leftmost).
0	Second button chosen.
1	Third button chosen.

See Also:

DIALOG
FILEDIALOG
DIRDIALOG
PRINT
PROMPT

OBJECT Function

Usage:

oObject **= OBJECT(** *sObjType* **)**
oObject **=** *sObjType***.OBJECT()**

Description:

Returns an object structure of type given by *sObjType* argument. The object type is case sensitive and must be a registered one, otherwise the script will abort. All object members come initialized to their default values.

See Also:

LIST
MEMBERLIST

OPEN Function

Usage:

iFile **= OPEN(** *sFilename***,** *iOpenMode* **)**
iFile **=** *sFilename***.OPEN(** *iOpenMode* **)**

Description:

Opens a file stream for further processing. The filename is given by *sFilename* argument and the processing mode by the value of *iOpenMode*. The return value must be kept to feed any file stream processing function, and is known to be the open file handle (dont confuse it with file system DOS/WINDOWS file handle, the *iFile* value is supposed to be recognized only by the dCG engine).

The processing mode *iOpenMode* can be any of the following predefined values:

iOpenMode	**Value**	**Description**
O_READ	0	Opens the file *sFilename* for reading only (input). If it doesnt exist the function will return the null (zero) value.
O_WRITE	1	Opens the file *sFilename* for writing only (output). If the file doesnt exist it will be created.
O_APPEND	2	Opens the file *sFilename* for writing and positions the file pointer at the end of file. Any output will be appended to the end of the file. If the file doesnt exist it will be created (same as **O_WRITE**).

If in any case, the file cannot be opened the null value (zero) will be returned or the function will fail, aborting the script.

See Also:

CLOSE
EOF
READ
READLN
WRITE
WRITELN

PAD Function

Usage:

sString **= PAD(** *sString***,** *iLength* [**,** *cCharacter*] **)**
*sString***.PAD(** *iLength* [**,** *cCharacter*] **)**

Description:

Returns the character string *sString* argument, right-padded to fill *iLength* characters in length, with the character *sCharacter*. If *sCharacter* is not given, the space (blank) character is assumed.

See Also:

CHR
LEN
LEFT
PREFIX
RIGHT
SPACE
STRING
STRIP
SUBSTR
SUFFIX
TAB
TRIM

PICTURE Function

Usage:

sString **= PICTURE(** *sString*, *sPicture* **)**
*sString***.PICTURE(** *sPicture* **)**

Description:

String formatting by picture template. Formats a character string, *sString* argument, using a template picture *sPicture*, similar to the COBOL editing picture clauses, but slightly different and limited; it doesn't support count repetition of template characters (example: "9(5)" will be not the same as "99999"; here you must supply the later). Returns the formatted string.

Picture type prefixes:

@C	Character string type (alphanumeric).
@N	Numeric value type (numeric).
@D	Date string value type (date, see below).
@L	Logical value type (logical).
@M	Memo field type (memo).
@F	Floating point value type (numeric).
@A	Array of alphanumeric strings (delimited with ;).
@K	Encrypted character string type (alphanumeric).

Picture template characters:

X	Any alpha-numeric character (alphanumeric).
9	Decimal digits only (numeric).
Z	Blank zeros on left (numeric).
V	Assumed decimal point position (numeric).
+	Force signal output (+/-) (numeric).
-	Force signal output on negative numbers (numeric).

Particular to date picture templates (**@D** prefix):

YY	Year in 2-digit format (no-century).
YYYY	Year in 4-digit format.
MM	Month in 2-digit format (numeric, left padded with zero).
MMM...	First *n* characters of month name (alphanumeric format, right padded with spaces).
DD	Day of month in 2-digit format (numeric, left padded with zero).
DDD...	First *n* characters of day of week name (alpha-numeric format, right padded with spaces).

Templates with at least one "**X**" are assumed alphanumeric and will be left justified without any blank character padding; otherwise they are assumed purely numeric and will be right justified by template width and decimal point position. The decimal point position in the source string is determined by the period character, "**.**"; on the picture template is assumed at the position where the "**V**" character appears; the source decimal point character, period "**.**", will never appear at output; this must be explicit coded in the template string, usually right beside the assumed decimal position template character ("**V**"); any non-template characters are inserted on output string. If there are more characters that can be filled by the template truncation will occur.

The formatting using date picture templates (**@D**) are special in that they only accept source character strings in the format *yyyymmdd*, beeing *yyyy* the 4-digit year, *mm* the 2-digit month and *dd* the 2-digit day of month.

See Also:

DATE
STR
TIME
VAL

PREFIX Function

Usage:

iLength **= PREFIX(** *sString1***,** *sString2* [**,** *iCaseSensitive*] **)**
iLength **=** *sString1***.PREFIX(** *sString2* [**,** *iCaseSensitive*] **)**

Description:

Returns the length of the identical string prefix of the two argument character strings *sString1* and *sString2*. The optional argument *iCaseSensitive* must be supplied with a non-zero value if the comparison is to be case sensitive. The default is a non case sensitive comparison.

See Also:

CHR
LEN
LEFT
PROPER
RIGHT
SPACE
STRING
STRIP
SUBSTR
SUFFIX
TAB
TRIM

PROMPT Function

Usage:

sResult **= PROMPT(** *sTitle***,** *sText* [**,** *sDefault* [**,** *iMaxLength*]] **)**

Description:

Displays a simple dialog box to prompt for input of a single character string. The prompt dialog has a caption title, given by *sTitle*, a prompt text *sText*, an edit box input field with initial value given by *sDefault*, a maximum length of *iMaxLength* characters, and an Ok and Cancel buttons. If *sDefault* is not given, the initial input value defaults to an empty string. If *iMaxLength* is omitted or has 0 (zero) value the length of the input field is unlimited. Returns the character string the user has typed in the input field (edit box) if the Ok button is pressed. If the Cancel button is pressed an empty string is returned.

See Also:

DIALOG
FILEDIALOG
DIRDIALOG
MSGBOX
PRINT

PROPER Function

Usage:

sString **= PROPER(** *sString* **)**
*sString***.PROPER()**

Description:

Converts all characters of *sString* to lowercase except the first character of any non alphanumeric-delimited word which is converted to uppercase if it is alphabetic (A-Z). Returns the proper case version of argument *sString*.

See Also:

LOWER
PAD
STRIP
TRIM
UPPER

READ Function

Usage:

sString **= READ(** *iFile*, *iLength* **)**
sString **=** *iFle***.READLN(***iLength* **)**

Description:

Reads a string from an input file stream. Returns a character string with length given by *iLength* argument, or less if the end-of-file has been reached. The first argument *iFile* must be a valid file handle value that was returned by a previous call to the function OPEN in input mode (O_READ), otherwise the function call will fail.

See Also:

CLOSE
EOF
OPEN
READLN
WRITE
WRITELN

READLN Function

Usage:

sString **= READLN(** *iFile* [**,** *iMaxLength*] **)**
sString **=** *iFle***.READLN()**

Description:

Reads a line from an input file stream. Returns a character string with no terminating end-of-line character (i.e. it doesnt include any character of the CRLF character pair). If the file with file handle *iFile* has the end-of-file status, an empty string will be returned. The first argument *iFile* must be a valid file handle value that was returned by a previous call to the function OPEN in input mode (O_READ), otherwise the function call will fail. The optional second argument, *iMaxLength*, specifies the maximum length of the expected input line. If omitted, it is assumed that a text line will not exceed 4K (4096) bytes in length.

See Also:

CLOSE
EOF
OPEN
READ
WRITE
WRITELN

REMOVE Function

Usage:

list **= REMOVE(** *list*, *iIndex* **)**
*list***.REMOVE(** *iIndex* **)**

Description:

Removes the list item with index *iIndex* from the *list* argument. The index *iIndex* value is zero based, i.e. the first element has index 0 (zero), the second has index 1, and so on.

See Also:

ADD
AVG
COUNT
FIND
LIST
MAX
MIN
RESET
SUM

REMOVEFILE Function

Usage:

iRemoved **= REMOVEFILE(** *sFilename* **)**
iRemoved **=** *sFilename***.REMOVEFILE()**

Description:

Removes the file with name given by *sFilename* from the file system. Returns a non-zero value if the file has been successfully deleted.

See Also:

EXIST
FILEDIALOG
RENAMEFILE

RENAMEFILE Function

Usage:

iRenamed **= RENAMEFILE(** *sOldFilename*, *sNewFilename* **)**
iRenamed **=** *sOldFilename***.RENAMEFILE(** *sNewFilename* **)**

Description:

Renames the file with name given by *sOldFilename* with a new name as *sNewFilename*. Returns a non-zero value if the file has been successfully renamed.

See Also:

EXIST
FILEDIALOG
REMOVEFILE

REPLACE Function

Usage:

sString **= REPLACE(** *sString*, *sTarget*, *sReplace* **)**
*sString***.REPLACE(** *sTarget*, *sReplace* **)**

Description:

Scans the character string *sString* argument for all occurrences of *sTarget*, replacing each one by *sReplace*. Returns the modified character string.

See Also:

FIND
LISTTOSTR
LOWER
PAD
PREFIX
PROPER
STRIP
STRTOLIST
SUFFIX
TRIM
UPPER

RESET Function

Usage:

list **= RESET(** *list* **)**
*list***.RESET()**

Description:

Returns an empty list object, resetting all contents of the *list* argument.

See Also:

ADD
AVG
COUNT
FIND
LIST
MAX
MIN
REMOVE

RIGHT Function

Usage:

sString **= RIGHT(** *sString*, *iLength* **)**
*sString***.RIGHT(** *iLength* **)**

Description:

Returns the rightmost *iLength* characters of the *sString* argument.

See Also:

LEFT
LEN
PREFIX
SUBSTR
SUFFIX

RMDIR Function

Usage:

iSuccess **= RMDIR(** *sDirectory* **)**
iSuccess **=** *sDirectory***.RMDIR()**

Description:

Removes the directory specified by *sDirectory* argument. If the character string *sDirectory* does not specify a path, the directory to be removed is assumed relative to the current process directory. Returns a non-zero value if the current directory has been changed successfully. Returns 0 (zero) if an error has occurred or the specified directory does not exist.

See Also:

CURDIR
DCGDIR
DIRDIALOG
FDIR
FILEDIALOG
MKDIR
RMDIR

SAVEITEM Function

Usage:

lSaved **= SAVEITEM(** *sFilename***,** *expr* **)**
lSaved **=** *sFilename***.SAVEITEM(** *expr* **)**

Description:

Writes a single data item into the external file with name given by *sFilename*. The data item, given by *expr*, can be of any data type.   The value can be recovered later with LOADITEM function. Only one data item can be stored in a file at a time, however this can be a compound object or list. The return value indicates the success of the operation. It is non-zero if the data item has been stored successfully, zero otherwise.

See Also:

LOADITEM

SPACE Function

Usage:

sString **= SPACE(** [*iLength*] **)**
sString **=** *iLength***.SPACE()**

Description:

Returns a character string of length *iLength*, filled with space characters. If *iLength* is not given, a space string of only one character is returned.

See Also:

CHR
PAD
STRING
STRIP
TAB
TRIM

STR Function

Usage:

sString **= STR(** *iInteger* **)**
sString **=** *iInteger***.STR()**

Description:

Converts a integer numeric value *iInteger* into a character string *sString*.

See Also:

PICTURE
VAL

STRING Function

Usage:

sString **= STRING(** *sCharacter* [**,** *iLength*] **)**
*sCharacter***.STRING(** [*iLength*] **)**

Description:

Returns a character string of length *iLength*, filled with the character given in *sCharacter*. If *iLength* is not given, a string of only one character is returned.

See Also:

CHR
PAD
SPACE
STRIP
TAB
TRIM

STRIP Function

Usage:

sString **= STRIP(** *sString* **)**
*sString***.STRIP()**

Description:

Returns a character string that is a version of the argument sString stripped from all characters that are not alphanumeric (A-Z,0-9) or underscore (_). If the first character is not alphabetic (A-Z) it is stripped too. In other words, characters that are not valid for a simple identifier are removed from the string, including leading, embedded or trailing blanks.

See Also:

LEFT
LEN
PREFIX
PAD
PROPER
RIGHT
SPACE
SUBSTR
SUFFIX
TAB
TRIM

STRTOLIST Function

Usage:

list **= STRTOLIST(** *sString*, *sSeparator* **)**
list **=** *sString***.STRTOLIST(** *sSeparator* **)**

Description:

Converts a delimited character string into a list object. The *sString* argument is scanned for all occurrences of the character string *sSeparator*. Each sub-string of *sString* delimited by the *sSeparator* occurrences will become an individual item of the returned list object.

See Also:

FIND (in list)
FIND (in string)
LISTTOSTR
REPLACE

SUBSTR Function

Usage:

sString **= SUBSTR(** *sString*, *iIndex*, *iLength* **)**
*sString***.SUBSTR(** *iIndex*, *iLength* **)**

Description:

Retrieve a sub-string of a character string. Returns the portion of *sString* argument, starting on *iIndex* position with length of *iLength* characters. The position index *iIndex* is zero based which implies that 0 (zero) stands for the first character position, 1 (one) for the second, and so on.

See Also:

LEN
LEFT
RIGHT

SUFFIX Function

Usage:

iLength **= SUFFIX(** *sString1***,** *sString2* [**,** *iCaseSensitive*] **)**
iLength **=** *sString1***.SUFFIX(** *sString2* [**,** *iCaseSensitive*] **)**

Description:

Returns the length of the identical string suffix of the two argument character strings *sString1* and *sString2*. The optional argument *iCaseSensitive* must be supplied with a non-zero value if the comparison is to be case sensitive. The default is a non case sensitive comparison.

See Also:

CHR
LEN
LEFT
PREFIX
PROPER
RIGHT
SPACE
STRING
STRIP
SUBSTR
TAB
TRIM

SUM Function

Usage:

iSum **= SUM(** *list* **)**
iSum **=** *list***.SUM()**

Description:

Returns the total sum value of the items in the *list* argument. The *list* should be made of items of the same data type, and only integer, string and list item data types are meaningful. For integer items the sum of item values is computed. For character string items the total string length is returned. If the argument is a list of lists the total sum of item count is returned.

See Also:

ADD
AVG
COUNT
FIND
LIST
MAX
MIN
REMOVE
RESET

TAB Function

Usage:

sString **= TAB(** [*iLength*] **)**
sString **=** *iLength***.TAB()**

Description:

Returns a character string of length *iLength*, filled with tab characters. If *iLength* is not given, a tab string of only one character is returned.

See Also:

CHR
PAD
SPACE
STRING
STRIP
TRIM

TIME Function

Usage:

sTime **= TIME()**

Description:

Retrieve the current time. Returns the current time as a character string in the format *hh*:*mm*:*ss*.

See Also:

DATE
PICTURE

TRIM Function

Usage:

sString **= TRIM(** *sString* **)**
*sString***.TRIM()**

Description:

Discards all leading and trailing non printable characters (white-spaces) from a character string *sString*. Returns the trimmed version of argument *sString*.

See Also:

LEFT
LEN
PREFIX
PAD
PROPER
RIGHT
SPACE
SUBSTR
SUFFIX
TAB

UPPER Function

Usage:

sString **= UPPER(** *sString* **)**
*sString***.UPPER()**

Description:

Converts all characters of *sString* to uppercase. Returns the upper case version of argument *sString*.

See Also:

LOWER
PAD
PICTURE
PROPER
REPLACE
STRIP
TRIM

VAL Function

Usage:

iInteger **= VAL(** *sString* **)**
iInteger **=** *sString***.VAL()**

Description:

Converts a character string *sString* into an integer numeric value. Return the numeric value of *sString*. If *sString* does not begin with a numeric digit, VAL returns 0 (zero).

See Also:

PICTURE
STR

WRITE Function

Usage:

WRITE(*iFile*, *sString* **)**
iFile.**WRITE(** *sString* **)**

Description:

Writes a character string into an output file stream. The text to be written is given by the second argument *sString*. The first argument *iFile* must be a valid file handle value that was returned by a previous call to function OPEN in output mode (O_WRITE or O_APPEND open modes), otherwise the function call will fail.

See Also:

CLOSE
EOF
OPEN
READ
READLN
WRITELN

WRITELN Function

Usage:

WRITELN(*iFile*, *sString* **)**
iFile.**WRITELN(** *sString* **)**

Description:

Writes a line into an output file stream. The line text to be written is given by the second argument *sString* with no terminating end-of-line character (i.e. it doesnt include the any character of the CRLF pair, which are actually appended to the file stream). The argument *iFile* must be a valid file handle value that was returned by a previous call to function OPEN in output mode (O_WRITE or O_APPEND open modes), otherwise the function call will fail.

See Also:

CLOSE
EOF
OPEN
READ
READLN
WRITE

Options command (View menu)

Modifies dCG settings that control script interpretation generation, editing, extension drivers, and other options.

Select the tab for the options you want.

General tab

Modifies the general settings of the application and editor windows, such
as the script include search path list, the internal editor options and the INSPECT dialog appearance.

Scripting tab

Modifies specific script interpreter settings, such as the default comment and script markers, user protection tags and the PRINT statement behaviour.

Editor tab

Modifies editing settings, where you specify whether you use the internal editor or an external editor program or DDE server for script file editing.

Filters tab

Specifies the script file extension filters that are used by the file open and save dialogs, and the number of the most recently used files that appear in the File menu.

Drivers tab

Modifies the permanent system list of registered type drivers that are loaded whenever you start or use the dCG system.

General tab
Options command (View menu)

Modifies the general settings of the application and editor windows, such
as the script include search path list, the internal editor options and the INSPECT dialog appearance.

Include Path
Specifies the script include search path list. Several directories can be specified with the semicolon separator, and they will be searched for include files if one cannot be found in the current working directory.

Messages
Message window options.

Maximum Lines
Specifies the maximum number of message lines that the message window can show at one time. If this number is exceeded during the interpretation/generation process, the oldest message lines will be discarded to conform to this count.

Reset Messages on Generate
Indicates if the messages window is reset whenever the interpretation/generation process is started.

Internal Editor Options
Options for the local editing windows.

Tab Stop width
Specifies the width in number of characters, based on current editor font, of the tab (ASCII code 9) control character or key.

Soft-Tabs
Indicates that the tab character (ASCII code 9) is to be replaced by as many spaces (ASCII code 32) as the number given in **Tab Stop width**. Otherwise it is inserted as is, often called *hard-tabs*.

Keep Edit Selection
Indicates that some formatting commands, like Indenting and Script Mark Toggle, are to keep the affected text selected. If this option is not set, the selection is reset , that is unselected.

INSPECT Dialog Options
Options for the INSPECT dialog appearance and behaviour.

Use Registered Type Editing
Indicates that the editor supllied by the registered type driver, if applicable, is to be used instead of the normal editing whithin the INSPECT dialog.

Extended Type Editing
Indicates that object member editing is allowed, that is, the user can add new members, modify member definitions or remove existing members of the inspected object. This option, if set, instructs the INSPECT dialog to show the member editing buttons (**Add**, **Edit**, **Remove**)

Member Buttons on the Right
Indicates that the INSPECT dialog is to show the member editing buttons (**Add**, **Edit**, **Remove**) on the right of the window. If this option is not set, the buttons are shown on the bottom of the dialog window.

Scripting tab

Options command (View menu)

Modifies specific script interpreter settings, such as the default comment and script markers, user protection tags and the PRINT statement behaviour.

Enable Comments

Indicates that script comments are allowed and are to be detected and virtually removed during the interpretation process. If this option is not set, no comments are detected and every script portion is assumed to be either a statement or expression.

Begin Comment

Specifies the character sequence that marks the beginning of a comment area. If comments are enabled you must supply this sequence.

End Comment

Specifies the character sequence that marks the end of a comment area. The end of a comment must always appear in the same line or portion where a beginning sequence also is inscribed. All text between these sequences, and including them, are considered comments and will be discarded from interpretation. You must supply this character sequence if comments are enabled.

Line Comment

Specifies the character sequence that marks the beginning of a line comment area. All text after this sequence, and including it, to the end of the line or script portion is considered comments and will be discarded from interpretation. You must supply this character sequence if comments are enabled.

Script Markers

Indicates that text portions is the way the interpreter finds its source for parsing. These portions are delimited by the script markers specified below. Any text that doesnt fall between these markers are to be sent to current output, if any. If this option is not set, every text is to be fed to the interpreter and, accordingly, the script must be well formed as either statements or expressions.

Begin Script Marker

Specifies the character sequence that marks the beginning of a script source portion. If script markers are enabled, you must supply this character sequence.

End Script Marker

Specifies the character sequence that marks the end of a script source portion. The end marker of a script source text must always appear in the same line where a beginning sequence also is inscribed. All text between these sequences, but excluding them, are considered for interpretation and will be parsed either as a statement or an expression. You must supply this character sequence if script markers are enabled.

Ignore Blank Lines

Indicates that empty lines or lines with only white spaces are to be discarded from interpretation. If this option is not set, every blank or comment solely line will most probably generate a syntax error.

Code Protection

Specifies the literal text markers that defines an user protected code block in any generated output file. These tags are simple character sequences that must not appear in your target files in any way but for this purpose.

Begin Code Protection

Specifies the character sequence that marks the beginning of an user protected code block. In the generated output file, this character sequence will precede the protected tag that uniquely identifies the protected block.

End Code Protection

Specifies the character sequence that marks the end of an user protected code block. Any text of an existing file that falls between the code protection sequences are literally copied to the current output file, if it has the same name.

PRINT Statement

Options that determine the PRINT statement behaviour. If neither of the following options are set, the PRINT statement has no visible results, whatsoever.

Send to Messages

Indicates that each argument of the PRINT statement are to be shown in a line of the messages window. If this option is not set, no results are displayed as a message line.

Send to Output

Indicates that each argument of the PRINT statement are to be inserted in be current output file. It has no effect if no output file is being generated.

Editor tab

Options command (View menu)

Modifies editing settings, where you specify whether you use the internal editor, an external editor program or a DDE server for script file editing.

Internal Editor
Indicates that every open script file is to be locally edited within the program. This editor window is somewhat limited in a way that it cannot handle large files in size. The maximum size it can handle has been fixed to 32KB, just a reasonable size the standard edit control can hold in Windows.

Read-Only when using an External Editor
Indicates that the internal editor window, that is always open along with the file, is to be put in Read-Only mode when either the external editor options are chosen. When in this mode, the file is displayed as usual, but cannot be modified or saved to disk (you can still use the **Save As…** command, but this is not recommended when file load truncation occurs due to the limitations of the internal/Windows editor window).

External Editor - Command
Indicates that the preferred mode of editing script files is to spawn an editor program externally, in the same way you do in the console command line. There are two placeholders that you can specify in this command line, which will be substituted by the filename to be open (**%f**) and the line number to be located (**%l**), if applicable to your editor.

Command
Specifies the command line that will be used to invoke the external application program, as is submitted on the console command prompt. In this line, the **%f** specifier will be literally substituted by the filename to be opened and, optionally you can use **%l** for line number positioning.

External Editor - DDE
Indicates that the preferred mode for editing script files is to invoke an DDE server editor program. There are two placeholders you can use in the DDE command specification, that are substituted by the filename to be open (**%f**) and the line number to locate (**%l**), if applicable to the particular DDE server.

Server
Specifies the DDE server name, usually the program executable filename (without extension).

Topic
Specifies the DDE command topic.

Commands
Specifies the DDE command sequence to be invoked. Each command must be delimited by the **[** and **]** bracket characters to be recognised. Multiple commands can be chained in this way, and they are executed in a left-to-right sequence. The **%f** specifier will be literally substituted by the filename in question and, optionally, you can use **%l** for line number positioning, if applicable.

Filters tab

Options command (View menu)

Specifies the script filename pattern filters that are used by the file open and save dialogs, and the number of the most recently used files that appear in the File menu.

File Filters

Specifies the filename patterns to be used on the **File Open…** and **Save As…** command dialogs in the program. These filename patterns are also known as *wildcards* and can be more than one, if separated by semicolons.

Script Files

Specifies the filename patterns to be used for normal script filenames.

Include Files

Specifies the filename patterns to be used for included script filenames.

All Files

Specifies the filename patterns to be used for all other files.

MRU Menu Items

Specifies the maximum number of the most recently used (open or saved) filenames that are to be maintained in the File menu.

Drivers tab

Options command (View menu)

Modifies the permanent system list of registered type drivers that are loaded whenever you start or use the dCG system.

Registered Type Drivers
Displays the list of all the registered type drivers that are currently loaded in the system.

Registered Types
Displays the list of all the registered type names that are implemented by the currently selected driver.

Add… button
Shows a dialog where you choose a extension type driver filename to be loaded. The loaded driver will become a registered type driver in the DCG system list and, therefore, it will be loaded in any subsequent session.

Remove button
Unloads the selected registered type driver, removing it from the dCG system list.

Setup… button
Invokes the set-up or configuration proceedings for the selected registered type driver. If this button is disabled indicates that the driver module hasnt any set-up to be accomplished.