



RepGrid • RepSocio • RepNet  
RepServe • RepScript

# ***Rep 5***

**Conceptual Representation Software**

## RepScript Manual for Version 1.0

July 2009

Copyright © 2009, Brian R Gaines and Mildred L G Shaw

Centre for Person-Computer Studies

3635 Ocean View, Cobble Hill, BC V0R 1L1, Canada

<mailto:cpcs@shaw.ca>

<http://repgrid.com>

# Contents

1 Introduction	1-1
1.1 Editing, executing and debugging <i>RepScript</i> scripts	1-1
1.2 Garbage collection	1-1
2 Basic features of the <i>RepScript</i> programming language	2-1
2.1 Data Types	2-1
2.2 Literals	2-1
2.3 Variables	2-1
2.4 Arrays	2-1
2.5 Constants	2-2
2.6 Operators	2-2
2.7 Expressions and assignments	2-2
2.8 Comments	2-2
3 Flow of control	3-1
3.1 If...Then...Else	3-1
3.2 For...Next	3-1
3.3 Do...Loop	3-1
3.4 While...Wend	3-2
3.5 Select Case	3-2
4 User-defined functions, subroutines, modules, classes and objects	4-1
4.1 Functions	4-1
4.2 Subroutines	4-1
4.3 Classes and objects	4-1
4.4 Modules, namespaces	4-2
5 Pre-defined functions	5-1
5.1 Constants	5-1
5.1 Bitwise functions	5-1
5.2 Arithmetic functions	5-1
5.3 Arithmetic rounding functions	5-1
5.4 Trigonometric functions	5-2
5.5 Strings formatted from numbers	5-2
5.6 String-numeric conversion functions	5-2
5.7 String modify functions	5-3
5.8 String field and array function	5-3
5.9 Other String functions	5-4
5.10 Array functions	5-5
5.11 Color functions	5-5
5.12 Time functions	5-5
5.13 Random functions	5-5
5.14 Popup menu functions	5-6
5.15 Data type function	5-6
6 Script specifications, calls, and global hash and vector stores	6-1
6.1 Script specifications	6-1
6.2 Script calls	6-1
6.3 Access to script state and name	6-2

6.4 Script execution	6-2
6.5 Hash table functions	6-2
6.6 Vector functions	6-3
7 Stream and text window functions	7-1
7.1 Stream write functions	7-1
7.2 Text window write functions	7-1
7.3 Text window create and access functions	7-1
7.4 Text window title functions	7-1
7.5 Text window state functions	7-1
8 System functions	8-1
8.1 Access to system parameters	8-1
8.2 File functions	8-1
8.3 Time functions	8-2
8.4 Sound functions	8-2
8.5 Alert and confirm dialog windows	8-2
9 XML functions	9-1
9.1 XML document open and create functions	9-1
9.2 XML document navigation functions	9-1
9.3 XML document data access functions	9-1
9.4 XML document data creation functions	9-2
10 Grid functions	10-1
10.1 Opening a grid	10-1
10.2 Grid Data Items	10-1
10.3 Settable Grid Data Items	10-1
10.4 Grid Elements	10-2
10.5 Grid Constructs	10-3
10.6 Grid Items	10-3
10.7 Grid Values	10-3
10.8 Grid Selection	10-3
10.9 Grid Matches	10-4
10.10 Grid Analysis	10-4
11 Data structures used in grid functions	11-1
11.1 Reserved names for items in a grid	11-1
11.2 Element specification in a hash store	11-1
11.3 Construct Specification in a hash store	11-1
11.4 Variables used in grids	11-1
11.5 Display Parameters	11-2
11.6 Focus Parameters	11-3
11.7 Compare Parameters	11-3
11.8 PrinGrid Parameters	11-4
11.9 Match Parameters	11-5
11.10 Crossplot Parameters	11-5
11.11 Style Parameters	11-5
11.12 Selection and weight specification	11-6
12 RepGrid library	12-1
12.1 Script window management	12-1

12.2 Script window functions	12-1
12.3 Script calls supporting interaction	12-1
13 RepServe library	13-1
13.1 Getting client request	13-1
13.2 Sending server response	13-1
13.3 WebGrid functions	13-1
13.4 WebNet functions	13-1
13.5 Server log window functions	13-1
13.6 System function extensions	13-2
14 Net functions	14-1
14.1 Net Management	14-1
14.2 Net Data Items	14-1
14.3 Net Types	14-1
14.4 Net Nodes	14-1
14.5 Net Links	14-2
14.6 Script interaction in <i>RepNet</i>	14-2
14.7 Node and line specifications in a hash store	14-2

## 1 Introduction

*Rep 5* was designed to have an open architecture in which major parts of its operation are programmed in user-accessible scripts which can be extended to customize its major components such as *RepGrid*, *RepNet* and *RepServe* and enhance their functionality. Scripts are written in the *RepScript* which is an object-oriented modern programming language similar in many respects to Microsoft's widely used Visual Basic for Applications (VBA). It is a heavily extended version of REAL Software's *RBScript* which is itself based on the *REALbasic* language. Those developing *RepScript* programs may find it useful to download the *REALbasic* User's Guide and Language Reference (see *RBScript* section) from REAL Software's web site:

<http://www.realsoftware.com/download/>

*RepScript* has a common functionality across all its applications enhanced by specialist functionality for each of the environments in which it operates, *RepGrid*, *RepNet* or *RepServe*. This manual describes both the common and specialist functionality. It assumes that readers are already experienced in programming in basic-like object-oriented languages.

### 1.1 Editing, executing and debugging *RepScript* scripts

*RepScript* scripts are text files with the extension .txt in ASCII or UTF-8 and may be edited in any text editor that supports these encodings.

*Rep 5* provides a suitable text editor when a new text file is created from within *Rep 5* or a text file is opened in *Rep 5*. The *Rep 5* text editor allows a *RepScript* opened in it to be executed using the "Execute" command at the bottom of the "Edit" menu, or CMD-E.

*RepScript* traps syntax errors when a script is first executed, and runtime errors while it is executing. It reports these in terms of the nature of the error, the name of the script, the line number of the code producing the error, and the code itself.

Errors are normally output to the "Log" window, but may be redirected to another location, e.g. the *RepServe* script environment shows them in an HTML error page sent to the web client that invokes the script, and the *RepGrid* script environment shows them in the "Script" panel.

Also useful in debugging scripts is the command:

```
Print stringExpression
```

which outputs the value of the string expression to the log window.

### 1.2 Garbage collection

Garbage collection in *RepScript* is automatic based on reference counting. *RepScript* keeps track of the number of variables that reference a particular data item and when the count reaches zero, that is the item goes out of scope, the data item is marked as garbage and collected when space is needed. This makes it important to manage cyclic references between objects that may prevent garbage collection and create a memory leak by setting such references to nil when the structure created is to be deleted.

## 2 Basic features of the *RepScript* programming language

### 2.1 Data Types

*RepScript* supports the following data types:

- String—of UTF8 characters;
- Boolean, integer—32bits;
- Single—32bits;
- Double—64 bits;
- Color—32 bits interpreted as three 8 bit fields for RGB;
- Object—datatype defined through a class;
- Variant—a generic datatype which may hold any of the previous data types.

### 2.2 Literals

Numeric literals have the form: 123, -99, 0.573, 1.45E6

Numeric literals may also be expressed to a different base:

- `&b10000101` (binary);
- `&o3427` (octal);
- `&h5fa8b` (hex);

Boolean literals have the form: false, true

String literals have the form: “abcdef”

Color literals have the form: `&c88aabc` (RGB in hex)

Array literals have the form: `Array(1,2,7)` or `Array(“cat”,“dog”)`

The literal for a non-existent object is: nil

### 2.3 Variables

Variables name are case-insensitive and can be any string of letters, numbers or underlines not commencing with a number.

All variable must be declared through a Dim statement specifying their name and type, e.g.

```
Dim i, j, k As Integer, x, y As Double, b As Boolean
```

Variables are normally initialized to zero, null string, false or nil as appropriate but may be initialized to a specified value when declared, e.g.

```
Dim i As Integer = 10
```

```
Dim x As Single = i + 2.7
```

initializes x to 12.7.

### 2.4 Arrays

Array variables are zero-based, may be multidimensional and are specified by parentheses specifying the upper bound of each dimension followed by their type, e.g.

```
Dim a(3) As Integer, w(12,12) As Double, bb(10,8,8) As Boolean
```

Empty arrays may be defined with -1 as the upper bound, or by empty parentheses, e.g.

Dim b(-1) As Integer, x() As Single

Arrays may be initialized using the Array function, e.g.

Dim a(3) As Integer=Array(4,7,9,2)

Arrays may re-dimensioned dynamically by the redim statement, e.g.

Redim a(10)

Redim w(8,12)

The function UBound may be used to find the dimension of an array, e.g.

UBound(a) or a.UBound will return 10

For multidimensional arrays UBound return the first dimension, and may be used with an argument to specify a specific dimension, e.g.

UBound(w) or UBound(w,1) will return 8

UBound(w,2) will return 10

For any array UBound with a second argument of -1 will return the number of dimensions, e.g.

UBound(a,-1) will return 1

UBound(w,-1) will return 2

## **2.5 Constants**

Constants can be declared as: Const name = literal, e.g.:

Const pi = 3.141352673723846

Const MachineName = "JoesComputer"

Const red = &cff0000

## **2.6 Operators**

Arithmetic operators, numeric type to numeric type: +, -, \*, /

Arithmetic operators, integer to integer: \, Mod (integer divide, remainder)

Comparison operators, all types to boolean: <, =, >, <=, >=, <>

Logical operators between boolean expressions: And, Or, Not (shortcut operators)

String operators: + (concatenation)

## **2.7 Expressions and assignments**

Expressions are well-formed combinations of literals, variables, functions and operators, such as a + 7.234 / sin(0.89).

Assignments have the form: variable = expression

Type conversion between numeric types is automatic.

Expressions of any type may be assigned to a variable of type variant.

Type conversion from a variant to any other type is automatic.

## **2.8 Comments**

Program lines commencing are treated as comments if they commence with: ' or //

### 3 Flow of control

*RepScript* supports the normal flow of control commands expected in a programming language.

#### 3.1 *If...Then...Else*

The general form of the “If...Then...Else” construct is:

```
If boolean expression Then
    program statements
ElseIf boolean expression Then
    program statements
Else
    program statements
End
```

where the ElseIf and Else clauses may be omitted and there can be as many ElseIf clauses as required.

If one-line program statements are adequate then the “If...Then...Else” construct can be one-line:

```
If boolean expression Then statement
If boolean expression Then statement Else statement
```

#### 3.2 *For...Next*

The general form of the “For...Next” construct is:

```
For counter variable = start value To | DownTo end expression [Step step value]
    program statements
    [Continue]
    [Exit]
    program statements
Next
```

where the counter variable can be of type Integer, Single or Double. If the Step clause is absent the value is taken to be 1.

The counter variable is initialized to the start value and if it is not greater than the end expression the program statements are executed until the Next statement. The counter variable is then incremented (To) or decremented (DownTo) by the step expression and if it is not greater than the end expression the program statements are executed, and so on.

The optional Continue statement skips directly to the Next statement. The optional Exit statement skips to the statement following the Next statement.

#### 3.3 *Do...Loop*

The general form of the “Do...Loop” construct is:

```
Do [Until boolean expression]
    program statements
    [Continue]
```

```
[Exit]
program statements
```

```
Loop [Until boolean expression]
```

The Do...Loop statement continues to execute statements repeatedly until one of the optional boolean expressions is True or an optional Exit statement within the loop is executed.

The optional Continue statement skips directly to the Loop statement. The optional Exit statement skips to the statement following the Loop statement.

### **3.4 While...Wend**

The general form of the “While...Wend” construct is:

```
While boolean expression
    program statements
    [Continue]
    [Exit]
    program statements
Wend
```

If the boolean expression evaluates to True, all statements are executed until the Wend statement is reached. If the boolean expression still evaluates to True the process is repeated. If False, then execution continues with the statement following the Wend statement.

The optional Continue statement skips directly to the Wend statement. The optional Exit statement skips to the statement following the Wend statement.

### **3.5 Select Case**

The general form of the “Select Case” construct is:

```
Select Case testExpression
    [Case expression-n
        program statements-n]
    [Else
        program statements-else]
End
```

Where testExpression is any expression that evaluates to a value. The expression can be of any data type or an object.

Where expression-n is any expression or list of expressions. You can use a function that evaluates to the data type of testExpression. The expression can be a single value, a comma-delimited list of values, a function that returns a value, a range of values specified with the 'To' keyword, an expression that uses the Is keyword to do an equality or inequality test, or an expression that uses IsA to determine the data type of an object.

Where program statements-n are executed if expression-n is true.

Where program statements-else are executed if none of the expression-n's are true.

## 4 User-defined functions, subroutines, modules, classes and objects

### 4.1 Functions

*RepScript* supports subroutines and functions with arguments specified by type and passed by value or by reference. Variable numbers of arguments of the same type can be passed and are accessed as a parameter array. Subroutines and functions may be called before they are defined. The Call statement allows a function to be called as if it were a subroutine.

The general form of a function definition is:

```
Function(parameter list) As datatype
    program statements
    [Return expression]
    [Exit]
    program statements
End Function
```

Where the parameter list has the form:

```
Name Type, Name Type, .....
```

specifying each parameter by name and by type. Array parameters are indicated by a pair of opening and closing parentheses. Multi-dimensional arrays have commas within the parentheses to indicate the number of dimensions.

The ByRef keyword in front of a parameter name indicates that the parameter is passed by reference and hence the value of the calling variable may be changed.

The ParamArray keyword in front of the last or only parameter name indicates that the parameter will accept any number of values of the type specified and make them available as an array.

A Return expression statement exits the function, returning the value of the expression.

If there is no Return statement the default value of the datatype of the functions is returned. An Exit statement returns immediately with the default value.

Variables defined in Dim statements within the function are local to the function.

### 4.2 Subroutines

The general form of a subroutine definition is:

```
Sub(argument list)
    program statements
    [Exit]
    program statements
End Sub
```

The same considerations apply as for functions except that there is no value returned.

### 4.3 Classes and objects

A set of constants, variables, functions and subroutines may be encapsulated in a class which acts as a template for objects instantiating the class. In essence, each class defines a new data type.

Classes may inherit from one another. The functions and subroutines within a class are usually termed its methods.

The general form of a class definition is:

```
Class classname
    [Inherits classname]
    constant, variable, function and subroutine definitions
End Class
```

A variable can be declared as having the type of a class in a Dim statement:

```
Dim variable As classname
```

The New statement can be used to create an object instantiating a class:

```
New classname
```

A class can have Constructor subroutines/methods that are called by a New statement when the classname specification has the form of a subroutine call with an argument signature that matches that of the Constructor subroutine.

A class can have a Destructor subroutine/method that is called automatically when an object of the type of the class goes out of scope.

#### ***4.4 Modules, namespaces***

Modules may be used to encapsulate sections of a program within a namespace such that all constants, variables, functions, subroutines and classes defined within the module have to be referenced with the name of the module followed by a period and the name of the construct referenced.

The general form of a module definition is:

```
Module name
    constant, variable, functions, subroutine and class definitions
End Module
```

## 5 Pre-defined functions

### 5.1 Constants

BOM As String—UTF8 BOM

CR As String—CR

CRLF As String—CRLF

EOL As String—end-of-line character for operating system under which *Rep 5* is operating

REVFLAG As Integer—flag used in construct number to indicate it is reversed

REVMSK As Integer—bit-pattern to mask out flag from construct number

TAB As String—TAB

### 5.1 Bitwise functions

The bitwise functions operate on integers as if they were strings of bits.

bAnd(a As Integer, b As Integer) As Integer—logical conjunction of bits in a and b

bComp(a As Integer) As Integer—logical complement of bits in a

bCount(a As Integer) As Integer—number of bits in a that are set to 1

bOr(a As Integer, b As Integer) As Integer—logical disjunction of bits in a and b

bSet(a As Integer, mask As Integer, b As Integer) As Integer—set bits in a exposed by mask from bits that are 1 in b,

bTest(a As Integer, mask As Integer) As Boolean—return true if any bit in a exposed by mask is 1

bXor(a As Integer, b As Integer) As Integer—logical exclusive-or of bits in a and b

### 5.2 Arithmetic functions

Exp(x As Double) As Double—e to power of x

Log(x As Double) As Double—natural logarithm of x

Min(x As Double, y As Double) As Double—minimum of x and y

Max(x As Double, y As Double) As Double—maximum of x and y

MinMax(x As double, mi As double, ma As double) As double—x if in range [mi,ma], mi if  $x < mi$ , ma if  $x > ma$

Pow(x As Double, y As Double) As Double—x to power of y

Sqrt(x As Double) As Double—square root of x

### 5.3 Arithmetic rounding functions

Abs(x As Double) As Double—absolute value of x

Ceil(x As Double) As Double—x rounded up to nearest integer

Floor(x As Double) As Double—x rounded down to nearest integer

Round(x As Double) As Double—x rounded to nearest integer

## 5.4 Trigonometric functions

Trigonometric functions represent angles in radians.

Acos(x As Double) As Double—inverse cosine of x

Asin(x As Double) As Double—inverse sine of x

Atan(x As Double) As Double—inverse tan of x

Atan2(x As Double, y As Double) As Double—arctangent of the point specified in x, y coordinates, i.e. angle from the x-axis to a line drawn through the origin and the point specified

Cos(x As Double) As Double—cosine of x

Sin(x As Double) As Double—sine of x

Tan(x As Double) As Double—tangent of x

## 5.5 Strings formatted from numbers

Format(x As Double, fmt As String) As String—convert x to string using fmt as format

NStr(x As Double, fmt As String) As String—as for Format but padding with spaces if necessary

sNumeric(s As String) As Boolean—true if s in a valid numeric form as might be created by Format

The format specification string comprises one or more special characters that control how the number will be formatted:

- #—placeholder that displays the digit from the value if it is present. If fewer placeholder characters are used than in the passed number, then the result is rounded;
- 0—placeholder that displays the digit from the value if it is present. If no digit is present, 0 (zero) is displayed in its place;
- .—placeholder for the position of the decimal point;
- ,—placeholder that indicates that the number should be formatted with thousands separators;
- %—displays the number multiplied by 100;
- (—displays an open parenthesis;
- )—displays a closing parenthesis;
- +—displays the plus sign to the left of the number if the number is positive or a minus sign if the number is negative;
- —displays a minus sign to the left of the number if the number is negative. There is no effect for positive numbers;
- E or e—displays the number in scientific notation.;
- \character—displays the character that follows the backslash.

The difference between Format and NStr is that NStr pads out a number with spaces on the left to comply with the width specification, whereas Format does not.

## 5.6 String-numeric conversion functions

Asc(s As String) As Integer—ASCII value of first character of s

AscB(s As String) As Integer—ASCII value of first byte of s  
BooStr(b As Boolean) As String—“false” if b is false, otherwise “true”  
Chr(x As Double) As String—string consisting of character whose ASCII value is x  
ChrB(x As Double) As String—string consisting of one-byte character whose ASCII value is x  
CStr(x As Double) as String—x as string using system-defined character for decimal separator  
Hex(i As Integer) As String—hexadecimal form of i  
Oct(i As Integer) As String—octal form of i  
Str(x As Double) As String—x as string using “.” as decimal separator  
Val(s As String) As Double—numeric form of numeric string in s

### **5.7 String modify functions**

Lowercase(s As String) As String—convert all characters in s to lowercase  
Titlecase(s As String) As String—start each word in s with an uppercase character  
Uppercase(s As String) As String—convert all characters in s to uppercase  
sCapitalize(s As String) As String—convert first character of s to uppercase

LTrim(s As String) As String—trim whitespace on left of s  
RTrim(s As String) As String—trim whitespace on right of s  
Trim(s As String) As String—trim whitespace on left and right of s  
sReplaceEOL(s As String, linend As String) As String—replace line endings with linend  
sClean(s As String) As String—replace TAB with space and line endings with CRLF  
sFill(s As String, len As Integer[, fill As String][, right As Boolean]) As String—extend the string to be at least as long as len by adding the optional fill parameter to the left if the optional right parameter is false or to the left if it is true—if the fill parameter is absent it is a space—if the right parameter is absent it is false

Base64Encode(s As String) As String—encode s to Base64  
Base64Decode(s As String) As String—decode s from Base64  
sURLEncode(String) As String—encode s to be an encoded URL  
sURLDecode(String) As String—decode s as encoded URL to be a normal string

### **5.8 String field and array function**

The field functions treat the first string argument as a 1-based array of fields separated by the (case-insensitive) second string argument.

CountFields(s As String, sep As String) As Integer—number of fields in first string  
NthField(s As String, sep As String, idx As Integer) As String—field indexed by idx—empty if idx is greater than the number of fields

sFind(s As String, find As String[, sep As String]) As Integer—returns the index of field find in the fields in s, zero if not present—if sep is absent it is TAB

sGet(s As String[, sep As String][, first As Integer][, last As Integer]) As String—gets the fields from first through last—if sep is absent it is TAB—if last is absent the field specified by first is returned—if last is -1 the field from first to the end of s are returned—if first and last are absent the number of fields in s is returned

sGetI(s As String[, sep As String][, first As Integer][, last As Integer]) As Integer—converts the result to be an Integer

sGetD(s As String[, sep As String][, first As Integer][, last As Integer]) As Double—converts the result to be a Double

sMakeDel(del As String, ParamArray par As String) As String—returns a variable length list of strings separated by del strings

sMake(ParamArray par As String) As String—returns a variable length list of strings separated by TAB characters

sReplaceNth(s As String, rep As String[, sep As String], n As Integer) As String—replaces field n in s with rep—field is treated as separated by sep or by TAB if sep is absent

There are also functions for transforming field-based delimited arrays to string arrays and *vice versa*.

Join(s() As String[, sep As string]) As String—concatenates the elements of the one-dimensional array s as fields separated by an optional separator sep, or the space character if one is not specified

Split(s As String[, sep As string]) As String()—creates an array of the fields in s separated by an optional separator sep, or the space character if one is not specified

More generally strings may be decomposed using patterns in a regular expression.

sSplit(src As String, regex As String) As String()—splits the string into an array of substrings using the patterns in the regular expression regex

### 5.9 Other String functions

InStr([startpos As Integer,] s As String, sep As String) As Integer—position in s of the first occurrence of the (case-insensitive) string sep in the first—starting at the location specified by startpos, 1 if that argument is not present—returns 0 if sp is not in s

InStrB([startpos As Integer,] s As String, sep As String) As Integer—as for Instr except treats string s as bytes rather than characters

Left(s As String, num As Integer) As String—returns num characters from beginning of s

LeftB(s As String, num As Integer) As String—treats string as bytes rather than characters

Len(s As String) As Integer—number of characters in s

LenB(s As String) As Integer—number of bytes in s

Mid(s As String, startpos As Integer[, num As Integer]) As String—returns number of characters in s specified by num commencing with that specified by startpos—if num is absent, returns from startpos to the end of s

MidB(s As String, startpos As Integer[, num As Integer]) As String—treats string as bytes rather than characters

Replace(s As String, sep As String, rep As String) As String—returns s with the first occurrence of the case-insensitive string sep replaced by rep

ReplaceB(s As String, sep As String, rep As String) As String—treats string as bytes rather than characters

ReplaceAll(s As String, sep As String, rep As String) As String—returns s with all occurrences of the case-insensitive string sep replaced by rep

ReplaceAllB(s As String, sep As String, rep As String) As String—treats string as bytes rather than characters

Right(s As String, num As Integer) As String—returns num characters from the end of s

RightB(s As String, num As Integer) As String—treats string as bytes rather than characters

StrComp(s1 As String, s2 As String, mode As Integer) As Integer—compares s1 with s2 according to a mode and returns -1 if s1<s2, 0 if s1=s2, 1 if s1>s2—modes are 0 for case-sensitive comparison, 1 for lexicographic

sUID() As String—a unique identifier based on date and time in microseconds

sWeb(s As String) As String—replaces HTML syntax characters (&<>”) by &..; form—the W prefix to gGet goes this also

### **5.10 Array functions**

array.Append value—appends a value of the type of the array to the end of the array

array.Insert integer, value—inserts a value of the type of the array at the specified index

array.Remove integer—removes the item at the specified index

array.Pop As value of type of array—returns and removes last item in the array

array.Sort—sorts a one-dimensional array of strings, integers, singles or doubles in ascending order

array.SortWith(array1[,...arrayN])—same as sort but also one or more additional arrays in the same order as the base array

array.Shuffle—sorts the array in random order

array.IndexOf(value[,integer]) As Integer—returns position of first occurrence of value in the array, starting at the optional integer index if present—returns -1 if value is not present

### **5.11 Color functions**

CMY(cyan As Double, magenta As Double, yellow As Double) As Color—color from cyan, magenta, yellow

HSV(hue As Double, saturation As Double, value As Double) As Color—color from hue, saturation, value

RGB(red As Double, green As Double, blue As Double) As Color—color from red, green, blue

### **5.12 Time functions**

Microseconds As Double—number of microseconds since computer was started

Ticks As Integer—number of ticks (60<sup>th</sup> of second) since computer was started

### ***5.13 Random functions***

Rnd As Double—random number in range 0.0 to 1.0

### ***5.14 Popup menu functions***

Scripts may present a popup menu specified as a list of items separated by “\”. An item is either a simple item or itself a list of items separated by “#” specifying a main menu item and submenu items. An item may have an “@” character within it, in which case the text before the “@” is put in the menu and the text after it is returned when that menu item is selected. If the item is selected is in a submenu then the main menu item followed by a space is added to the front of the submenu item selected in the text returned.

Menu(items As String) As String—show a popup menu as specified by items and return the item the user selected or the empty string if none

### ***5.15 Data type function***

object IsA object class As Boolean—returns true if object is of type object class, false otherwise or if object is nil

## 6 Script specifications, calls, and global hash and vector stores

*RepScript* supports script calls with argument passing between scripts. It also supports calls specifying a return script.

Scripts operate as independently compiled programs with access only to data structures defined within them. To allow data structures to be maintained across script calls *RepScript* supports two types of general-purpose storage structure that are maintained in the script environment and accessible to all scripts operating within it. Both structures allow the storage of arbitrary mixed data types. One provides hash tables where a data item is stored indexed by a string that names it. The other provides one-dimensional vectors where a data item is stored indexed by an integer. In each case *RepScript* provides a set of named stores so that a number of each type of store may be defined for differing purposes.

### 6.1 Script specifications

A script specification has the form “scriptpath/scriptstate” where the scriptpath specifies the file path of the script in the scripts directory associated with the scripted application:

NetScripts for *RepNet* scripts;

GridScripts for *RepGrid* scripts;

ServerScripts for *RepServe* Scripts.

*RepScript* looks for the GridScripts and ServerScripts directories in the “Documents/Rep 5” or “My Documents/Rep 5” directory first and in the application directory second. It looks for the NetScripts directory in the directory holding the net and its subdirectories first, and then in the “Rep 5” and application directories.

If the script is not found at the end of the path *RepScript* searches for it backwards along the path through successively enclosing directories.

If the scriptpath begins with a “/” it is an absolute reference to the scripts directory, otherwise it is relative to the location of the calling script.

The extension “.txt” is automatically appended to the script name.

A script may include a “#INCLUDE scriptpath” statement in which case the sub-script specified by the scriptpath is substituted for the line starting with the #INCLUDE statement. Where sub-scripts are included in this way any error messages generated specify the name of the sub-script and the line number within it.

Scripts are automatically compiled and cached when first used so that they do not need to be recompiled if used again. Each environment has facilities for clearing its script cache so that when scripts are being developed the latest version is used.

### 6.2 Script calls

ScriptCall(callscript As String, ParamArray arg As String)—executes the script specified in callscript and puts the parameters passed in the vector store named by the empty string.

If callscript begins with “\$” then “Library/” is substituted for “\$ simplifying calls to a script named “Library” intended to include generally useful subroutines and functions. Note that a relative script specification is generated allowing there to be a local Library

script in the same directory as the calling script or a more global one in a containing directory.

`ScriptFlow([callscript][,returnscript])`—if the optional argument `callscript` is present the script specified is pushed on the call stack to become the next one to be executed when the current one terminates—if, in addition, the optional argument `returnscript` is present the script specified is pushed on the return stack—if no arguments are specified then a script is popped from the returnstack and pushed on the call stack.

Because `ScriptCall` executes another script and returns after execution it can be used to access libraries of subroutines and functions. Note that the script specified by `ScriptFlow` is stacked and not executed until the script containing the `ScriptFlow` call itself terminates making `ScriptFlow` more suitable for transfer of control between scripts. The `returnscript` parameter enables control to be transferred elsewhere when the called script terminates.

The `ScriptWait` call in the *RepGrid* scripting environment extends the `ScriptFlow` system by specifying that *RepScript* wait for keyboard or mouse input from the user and then execute the script that has been stacked on the `ScriptFlow` return stack.

### 6.3 Access to script state and name

`ScriptState()` As String—script state set up by the call as the last field of the script specification—this is the simplest way to pass parameters to a script

`ScriptName()` As String—script name—mainly used in debugging using print statements in different scripts

### 6.4 Script execution

The direct execution of scripts is supported

`ScriptExecute(src As String)`—compiles and executes `src`

`ScriptExecute(names() As String, src() As String)`—compiles and executes the joined array of strings in `src()` giving each subscript the name specified in `names()` for purposes of error messages

Scripts operate as independently compiled programs with access only to data structures defined within them. To allow data structures to be maintained across script calls *RepScript* supports two types of general-purpose storage structure that are maintained in the script environment and accessible to all scripts operating within it. Both structures allow the storage of arbitrary mixed data types. One provides hash tables where a data item is indexed by a string that names it. The other provides one-dimensional vectors where a data item is indexed by an integer. In each case *RepScript* provides a set of named stores so that a number of each type of store may be defined for differing purposes.

### 6.5 Hash table functions

Hash tables are named stores containing named items having values of any type. The default store is named by the empty string. The stores are created automatically when values are set.

`hSet(v As datatype, hitem As String[, hstore As string])`—sets the value of `hitem` in the store `hstore` to be `v` where `v` can be of type `String`, `Integer`, `Double` or `Boolean`—if `hstore` is absent it is taken as the empty string

**hGet**(hitem As String[, hstore As String]) As String—gets the value of hitem in the store hstore as a String—if hstore is absent it is taken as the empty string  
**hGetI**(hitem As String[, hstore As String]) As Double—gets the value as an Integer  
**hGetD**(hitem As String[, hstore As String]) As Double—gets the value as a Double  
**hGetB**(hitem As String[, hstore As String]) As Double—gets the value as a Double  
**hCheck**(hitem As String[, hstore As String]) As Boolean—returns true if there is an item hitem in the store hstore—if hstore is absent it is taken as the empty string  
**hCheckGet**(ByRef s As String, hitem As String[, hstore As String]) As String—as hCheck but returns the value in the String s  
**hCount**([hstore As String]) As Integer—number of items in store hstore—if hstore is absent it is taken as the empty string  
**hKeysA**([hstore As String]) As String()—array of keys indexing items in store hstore—if hstore is absent it is taken as the empty string  
**hEmpty**([hstore As String])—empties hstore of items—if hstore is absent it is taken as the empty string  
**hRemove**(hitem As String[, hstore As string])—removes hitem from the store—if hstore is absent it is taken as the empty string  
**hKill**([hstore As String])—removes hstore—if hstore is absent it is taken as the empty string  
**hDump**([hstore As string]) As String—content of hstore as a list of item names and values—if hstore is absent then all hash stores are dumped

## 6.6 Vector functions

Vector stores are named stores containing a vector of items having values of any type. The default store is named by the empty string. The vectors are created automatically when values are set and expand automatically to accommodate any index.

**vSet**(v As datatype, i As Integer[, vstore As string])—sets the value of item i in the vector vstore to be v where v can be of type String, Integer, Double or Boolean—if vstore is absent it is taken as the empty string  
**vInsert**(v As datatype, i As Integer[, vstore As string])—inserts a new item at index i in the vector vstore and sets its value to be v—if vstore is absent it is taken as the empty string  
**vPush**(v As datatype[, vstore As string])—pushes a new item at the end of the vector vstore and sets its value to be v—if vstore is absent it is taken as the empty string  
**vGet**(i As Integer[, vstore As string]) As String—gets the value of item i in the vector vstore as a String—if vstore is absent it is taken as the empty string  
**vGetI**(i As Integer[, vstore As string]) As Integer—gets the value of item as an Integer  
**vGetD**(i As Integer[, vstore As string]) As Double—gets the value of item a Double  
**vGetB**(i As Integer[, vstore As string]) As Boolean—gets the value of item as a Boolean  
**vExtract**(i As Integer[, vstore As string]) As String—gets the value of item i in the vector vstore as a String and removes the item from the vector—if vstore is absent it is taken as the empty string

vExtractI(i As Integer[, vstore As string]) As Integer—extracts the value of item as an Integer  
vExtractD(i As Integer[, vstore As string]) As Double—extracts the value of item as a Double  
vExtractB(i As Integer[, vstore As string]) As Boolean—extracts the value of item as a Boolean  
vPop([, vstore As string]) As String—gets the value of the last item in the vector vstore as a String and removes the item from the vector—if vstore is absent it is taken as the empty string  
vPopI([, vstore As string]) As Integer—pops the value of item as an Integer  
vPopD([, vstore As string]) As Double—pops the value of item as a Double  
vPopB([, vstore As string]) As Boolean—pops the value of item as a Boolean  
vCount([, vstore As string]) As Integer—dimension of the vector vstore—if vstore is absent it is taken as the empty string  
vCountSet(n As Integer [, vstore As string]) As Integer—set the dimension of the vector vstore to n—if vstore is absent it is taken as the empty string  
vOK([, vstore As string]) As Integer—return true if the vector vstore has any items in it—if vstore is absent it is taken as the empty string—used with vPop to get items from stack  
vDump([, vstore As string]) As String—content of vstore as a numbered list of values—if vstore is absent it is taken as the empty string

## 7 Stream and text window functions

*RepScript* supports writing text to named streams which are text buffers that automatically expand as necessary. It enables text windows to be created with associated streams and the text in those streams to be flushed to appear in the window.

### 7.1 Stream write functions

`write(s As String[, stream As String])`—write *s* to the text stream specified—if stream is absent it is taken as the empty string

`writeln([s As String][, stream As String])`—write *s* plus EOL to the text stream specified—if stream is absent it is taken as the empty string—if *s* is also absent it is taken as the empty string

`writeStream(stream1 As String[, stream2 As String])`—appends the text in stream1 to that in streams and empties stream1—if stream2 is absent it is taken as the empty string

### 7.2 Text window write functions

`wFlush(s As String[, stream As String])`—write *s* to the text stream specified and flush the stream to the associated window, if any—if stream is absent it is taken as the empty string

`wFlushln([s As String][, nam As String])`—write *s* plus EOL to the text stream specified and flush the stream to the associated window, if any—if stream is absent it is taken as the empty string—if *s* is also absent it is taken as the empty string

`wSetText([s As String][, nam As String])`—sets the text in window *nam* to be *s*, replacing the existing text and not affecting the stream—if *nam* is absent it is taken as the empty string

### 7.3 Text window create and access functions

`wOpen(Window(title As String[, stream As String])`—creates a new window having the title and text stream specified—if stream is absent it is taken as the empty string

`wEnsureWindow (title As String[,stream As String])`—if there is an existing text window associated with the text stream specified then set its title as specified—otherwise, if there is an existing text window with the title specified then associate it with the text stream specified—otherwise create a stream and associated text window as in `wOpen`—if stream is absent it is taken as the empty string

### 7.4 Text window title functions

`WindowTitle([stream As String]) As String`—title of window associated with stream specified—if stream is absent it is taken as the empty string

`WindowSetTitle(title As String[,stream As String])`—set the title of window associated with the stream specified—if stream is absent it is taken as the empty string

### 7.5 Text window state functions

`wSetChanged(changed As Boolean[,stream As String])`—set the changed flag of window associated with the stream specified—if stream is absent it is taken as the empty string

## 8 System functions

### 8.1 Access to system parameters

`xGet("System", hash)`—puts system information in the hash store specified—additional items may be added in specific environments such as *RepServe*—the standard items are:

<b>Application:</b>	application and version, e.g. "Rep 5 1.00"
<b>Date:</b>	date in international format, e.g. "2009-01-12"
<b>Time:</b>	time in international format, e.g. "12:35:20"
<b>GMT:</b>	date in unix format, e.g. "Mon, 12 Jan 2009 19:35:20 GMT"
<b>LocalIP:</b>	network address of machine, e.g. "64.124.9.20"
<b>MAC:</b>	MAC number of machine, e.g. "00:24:53:68:AE:EA"
<b>User:</b>	name of current user account, e.g. "smethurst"
<b>Machine:</b>	name set up for machine on local network, e.g. "Colossus"
<b>Objects:</b>	number of active objects, e.g. "1086"
<b>Memory:</b>	memory used in bytes, e.g. "21591680"

`xSet(ParamArray arg As String)`

### 8.2 File functions

*RepScript* can access files, generally in one of two directories: the directory containing the *Rep 5* application, known by the rootcode, "App"; and the "Rep 5" directory in the user's "Documents" or "My Documents" directory, known by the rootcode, "Rep". File specifications are in terms of one of these directories and a file path within it.

`FileAppend(s As String, path As String[, rootcode As String])`—append *s* to file—if rootcode is absent it is "Rep"

`FileDelete(path As String[, rootcode As String])`—delete file—if rootcode is absent it is "Rep"

`FileExists(path As String[, rootcode As String][, dir As Boolean]) As Boolean`—*dir* is true if file being checked should be a directory—return true if specified file exists—if *dir* is absent it is false—if rootcode is also absent it is "Rep"

`FileRename(newname As String, path As String[, rootcode As String])`—rename the file—if rootcode is absent it is "Rep"

`FileSave(dat As String, path As String[, rootcode As String][, bom As Boolean])`—save *dat* in the file—if *bom* is true add a unicode BOM to the start of the file to indicate UTF-8 text—if *bom* is absent it is true—if rootcode is also absent it is "Rep"

`FileSave(dat As String, path As String, rootcode As String)`—if rootcode is absent it is "Rep"

`FileStr(path As String[, rootcode As String]) As String`—if rootcode is absent it is "Rep"

`FileURLDialog(prompt As String, name As String, filter As String) As String`—calls the operating system's file open dialog and returns the file specified in the form of the file identifier that *RepNet* generates when a file is dragged to a net—returns an empty string if user cancels

wOpenURL(ByRef fid As String) As Boolean—true if file exists that is specified by the identifier that *RepNet* generates when a file is dragged to a net then return true and open the file in its associated application

### **8.3 Time functions**

Time() As Double—time in microseconds since machine started

Time(t As Double) As Double—if t is positive sets a variable to the time in microseconds since machine started and returns that time—if t is negative returns the time in microseconds since machine started minus the time previously set in the variable.

### **8.4 Sound functions**

Bleep()—make a beep sound

### **8.5 Alert and confirm dialog windows**

Alert(Heading As String, Message As String)—show an alert modal dialog window

Confirm(Heading As String, Message As String) As Boolean—show a modal confirm dialog with “OK” and “Cancel” buttons—return true is user clicks “OK.”

## 9 XML functions

*RepScript* can open, create and navigate xml document trees, and access, change and create items within them. this is done through the function `xml` which takes a variable number of argument strings.

`xml(argument strings)`—may return a value

If there is a return value it is also of type `String`. Other types can be specified by a suffix letter, `xmlI` for Integer, `xmlD` for Double, `xmlB` for Boolean and `xmlX` for a call without a return value.

### 9.1 XML document open and create functions

`xml("OpenDocument")`—open a file as an xml document

`xml("NewDocument")`—create a new xml document

`xml("String")`—document as string suitable for filing

### 9.2 XML document navigation functions

`xml("Down")`—if there are child nodes of top of stack then push the first one and return number—if stack is empty return -1, if no children return 0 AND do not change stack

`xml("Up")`—pop the stack

`xml("Top")`—go to the root of the document tree

`xml("Next")`—replace top of stack with next node and return true—if no more then pop stack and return false

`xml("Previous")`—step back to previous node

### 9.3 XML document data access functions

`xml("Name")`—name of current node

`xml("Prefix")`—namespace prefix of current node

`xml("LocalName")`—name of current node without namespace prefix

`xml("NameSpace")`—namespace URI of current node

`xml("Value")`—value of current node

`xml("Text")`—first text item

`xml("AttributeCount")`—number of attributes

`xml("Attribute",attribute)`—value of attribute specified

`xml("AttributeName",attribute)`—value of attribute specified

`xml("Type")`—type of current node—an integer indicating the following types:

- 1 Element
- 2 Attribute
- 3 Text
- 4 CDATA Section
- 5 Entity Reference
- 6 Entity

- 7 Processing Instruction
- 8 Comment Node
- 9 Document Node
- 10 Document Type
- 11 Document Fragment
- 12 Notation
- 13 Other

#### ***9.4 XML document data creation functions***

`xml("NewElement",name,text)`— create a new element with text specified

`xml("NewAttribute",attribute,value)`— create a new attribute with value specified

## 10 Grid functions

*RepScript* provides functions to access a grid, getting and setting its data. The grid accessed is initialized by *RepGrid* and *WebGrid* to be the grid being elicited or edited, and can be set up by a *RepNet* script to refer to a stored grid associated with a net.

`gGet(argument strings)` gets data from a grid.

`gSet(argument strings)` changes data in a grid.

The arguments to these functions are always of type `String`. The default return type for `gGet` is `String` but can also be another type as specified by a suffix letter, `gGetI` for Integer, `gGetD` for Double, `gGetB` for Boolean, `gGetW` for web encoded (i.e. the quote character become “&quot;” and so on for angle brackets and ampersand), `gGetX` for a call without a return value.

There are also some grid functions that return arrays: `gGetA`, `gGetAI`, `gGetA2S`.

### 10.1 Opening and saving a grid

In the *RepGrid* and *RepServe* environments the grid being edited is automatically made available for access through the `gGet` and `gSet` function. In other environments such as *RepNet* one may wish to open a grid and the following functions are provided.

`OpenGrid(path As String[, rootcode As String]) As Boolean`—opens a grid within the directory specified by `rootcode` (“App” or “Rep”) along the path specified within that directory and return true if successful—`rootcode` is “Rep” if absent

`OpenGridURL(ByRef fid As String) As Boolean`—opens a grid using the file identifier that *RepNet* generates when a file is dragged to a net and return true if successful—search may be involved to find the file and the function updates `fid` to the actual value it uses

`gSet(“Save”, path[, rootcode])`—saves the grid within the directory specified by `rootcode` (“App” or “Rep”) along the path specified within that directory—`rootcode` is “Rep” if absent

### 10.2 Grid Data Items

`gGet()`—grid data structure encoded as if for filing

`gGet(“Grid”, hstore)`—grid data structure encoded in the hash store specified

`gGet(“nE”)`—number of elements

`gGet(“nC”)`—number of constructs

`gGet(“Identifier”)`—an identifier for the grid

`gGet(“File”)`—grid file name, if any

`gGet(“TypesSupported”)` As Integer—a bit-pattern corresponding to the rating types supported

### 10.3 Settable Grid Data Items

These are accessed through `gGet(“Item Name”)` and `gSet(“Item Name”, value)`.

`gGet(“E”)`—singular term for an element

`gGet(“Es”)`—plural term for elements

`gGet(“C”)`—singular term for a construct

`gGet(“Cs”)`—plural term for constructs

gGet("UID")—grid UID  
gGet("Date")—date when the grid was initiated  
gGet("Time")—time when the grid was initiated  
gGet("Place")—location where the grid was initiated  
gGet("Name")—name of grid  
gGet("Note")—note attached to name  
gGet("Context")—purpose of grid  
gGet("Annotation")—annotation of grid  
gGet("Origin")—path to the original grid if it derives from one  
gGet("Meta")—bit pattern specifying the meta values allowed  
gGet("MinR")—default minimum rating value  
gGet("MaxR")—default maximum rating value  
gGet("Command")—command string controlling elicitation  
gGet("Status")—status number used to indicate the derivation of a grid  
gGet("Display")—parameter values specified for Display analysis  
gGet("Focus")—parameter values specified for Focus analysis  
gGet("PrinGrid")—parameter values specified for PrinGrid analysis  
gGet("Compare")—parameter values specified for Compare analysis  
gGet("Crossplot")—parameter values specified for Crossplot analysis  
gGet("Statistics")—parameter values specified for Statistics analysis  
gGet("Matches")—parameter values specified for Matches analysis  
gGet("Style")—parameter values specified for the styles used in analysis  
gGet("Types")—types number used in WebGrid  
gGet("Control")—control number used in WebGrid  
gGet("Body")—body specified in WebGrid  
gGet("Header")—header specified in WebGrid  
gGet("Body")—body specified in WebGrid  
gGet("Trailer")—trailer specified in WebGrid  
gGet("LimitE")—limit on number of elements  
gGet("LimitC")—limit on number of constructs  
gGet("XXX")—additional parameter XXX specified in a script or web page

#### ***10.4 Grid Elements***

gGet("E", En)—name of the element number En  
gGet("E", En, hstore) also puts the full specification of the element in the hash store specified  
gSet("E", En, hstore) sets the element values specified in the hash store

gSet("NewE", hstore) adds an element with the values specified in the hash store  
gSet("NewE", En, hstore) inserts an element with the values specified in the hash store  
gSet("RemoveE", En) removes the element specified  
gSet("Move", vstore, En) moves the vector of elements specified by number in vstore to position  
En

### **10.5 Grid Constructs**

gGet("C", Cn)—name or identifier of the construct number Cn  
gGet("C", Cn, hstore) also puts the full specification of the construct in the hash store specified  
gSet("C", Cn, hstore) sets the construct values specified in the hash store  
gSet("NewC", hstore) adds a construct with the values specified in the hash store  
gSet("RemoveC", Cn) removes the construct specified  
gSet("Move", vstore, Cn) moves the vector of constructs specified by number in vstore to  
position Cn

### **10.6 Grid Items**

gGet("I", item)—value of item specified  
gSet("I", item, value)—set value of item as specified—if item is a reserved name an underline  
character is appended before its name (it is a good convention for user items to have  
names commencing with underline to distinguish them from system variables)

### **10.7 Grid Values**

gGet("V", Cn, En)—value in the grid for construct number Cn, element number En  
gSet("V", Cn, En, value) sets the value for construct number Cn, element number En  
gSet("EndV", Cn, En, end) sets the value for construct number Cn, element number En to the  
maximum if end is true, minimum if false  
gGet("RawV", Cn, En)—raw value  
gGet("NumV", Cn, En)—numeric value scaled in the range [-1.0,+1.0]  
gGet("SortV", Cn, vstore) sets up a vector of values on the construct sorted by value  
gGet("PossV", Cn, vstore) sets up a possible values for the construct sorted by value  
  
gGet("Open")—number of open values in the grid  
gGet("OpenC")—number of open constructs in the grid  
gGet("OpenE", En)—number of open values in the element specified  
gGet("OpenC", Cn)—number of open values in the construct specified

### **10.8 Grid Selection**

gGet("SelectE", boolean)—number of elements with selection as specified  
gGet("SelectE", boolean, vstore)—also puts those elements in the vector store  
gGet("SelectE", En)—gets the selection of element number En

gSet("SelectE", En, value)—sets the selection of element number En to value  
gSet("SelectE", value)—sets the selection of all the elements to value  
gGetAI("SelectE, boolean) As Integer()—array of element numbers with selection as specified

gGet("SelectC", boolean)—number of constructs with selection as specified  
gGet("SelectC", vstore)—also puts those constructs in the vector store  
gGet("SelectC", Cn)—gets the selection of construct number Cn  
gSet("SelectC", Cn, value)—sets the selection of construct number Cn to value  
gSet("SelectC", value)—sets the selection of all the constructs to value  
gGetAI("SelectC, boolean) As Integer()—array of construct numbers with selection as specified

### **10.9 Grid Matches**

gGet("MatchE", threshold, selwt)—a randomly selected element match above threshold, as a string of three numbers: E1 TAB E2 TAB match level, empty string if none

gGet("MatchE", threshold, vstore[,vstore2])—number of element matches above threshold—puts them in the specified vector store—puts match values in vstore2 if present

gGet("MatchE", En, threshold, vstore[,vstore2])—number of matches to element En above threshold—puts them in the specified vector store—puts matchvalues in vstore2 if present

gGetA2S(("MatchE", Cselected As Boolean, Cweight As Boolean, power As Double) As Single(.))—element match matrix—if Cselected then based on selected constructs—if Cweight then using construct weights—power specifies Minkowski metric

gGet("MatchC", threshold, selwt)—a randomly selected construct match above threshold, empty string if none

gGet("MatchC", threshold, vstore[,vstore2])—number of construct matches above threshold—puts them in the specified vector store—puts match values in vstore2 if present

gGet("MatchC", Cn, threshold, vstore[,vstore2])—number of matches to construct Cn above threshold—puts them in the specified vector store—puts match values in vstore2 if present

gGetA2S(("MatchC", Eselected As Boolean, Eweight As Boolean, power As Double) As Single(.))—Construct match matrix—if Cselected then based on selected elements—if Cweight then using element weights—power specifies Minkowski metric

## 10.10 Grid Analysis

`gGet("Display", param, selwt)`—returns a TAB-separated string of fields for a grid Display analysis based on the parameters and selection/weight specification passed—first field is image width, second height, third data regarding the locations of elements and constructs in the image, fourth the PNG image of the Display analysis

`gGet("Focus", param, selwt, hash, anEOL)`—returns a TAB-separated string of fields for a grid Focus analysis based on the parameters and selection/weight specification passed—first field is image width, second height, third data regarding the locations of elements and constructs in the image, fourth the PNG image of the Focus analysis—any text output requested is returned as items in the hstore with the specified EOL string

`gGet("PrinGrid", param, selwt, hash, anEOL)`—returns a TAB-separated string of fields for a grid PrinGrid analysis based on the parameters and selection/weight specification passed—first field is image width, second height, third eigenvector percentages, fourth the PNG image of the grid PrinGrid analysis—any text output requested is returned as items in the hstore with the specified EOL string

`gGet("PrinGridClick", param, selwt, x, y)`—returns a string containing the type and number of an item clicked in a PrinGrid analysis image based on the parameters and selection/weight specification passed

`gGet("Crossplot", param, selwt)`—returns a TAB-separated string of fields for a grid Crossplot analysis based on the parameters and selection/weight specification passed—first field is image width, second height, fourth the PNG image of the Crossplot analysis

`gGet("CrossplotClick", param, selwt, x, y) )`—returns a string containing the type and number of an item clicked in a Crossplot analysis image based on the parameters and selection/weight specification passed

`gGet("Compare", param, selwt, g2path, g2rootcode) )`—returns a TAB-separated string of fields for a grid Compare analysis based on the parameters and selection/weight specification passed—first field is image width, second height, third data regarding type of Compare analysis, fourth number of elements in common out of total elements, fifth number of constructs in common out of total constructs, sixth data regarding type of Compare analysis, seventh an identifier for the second grid, eight data regarding the locations of elements and constructs in the image, ninth the PNG image of the Compare analysis

## 11 Data structures used in grid functions

The grid functions defined in the previous section use a number of data structures that are defined in this section.

### 11.1 *Reserved names for items in a grid*

Name, Note, Context, Annotation, UID

E, Es, C, Cs

MinR, MaxR

Meta, Types

Date, Time, Place

Display Focus, PrinGrid, Crossplot, Compare, Statistics, Matches, Style

Header, AddHead, AddBody

Control, Status, Analysis

Script

### 11.2 *Element specification in a hash store*

When a hash store is used to get or set elements, the items in it are:

**Name:** name As String

**Note:** note As String

**Weight:** weight As Integer

**Select:** selected As Boolean

### 11.3 *Construct Specification in a hash store*

When a hash store is used to get or set constructs, the items in it are:

**Type:** string, first character:- R for ratings; C categories, I integers, F floats

**Name:** name As String

**Note:** note As String

**LHP:** left hand pole name As String

**RHP:** right hand pole name As String

**Range:** minimum value, TAB, maximum value

**Weight:** weight As Integer

**Labels:** labels and ranges As String and numbers

**Level:** level As Integer

**Ordered:** ordered As Boolean

**Output:** output As Boolean

**Select:** selected As Boolean

**Identifier:** identifier As String (only set up by gGet, not used by gSet)

### 11.4 *Variables used in grids*

**Types**—integer containing 1-bit flags determining available data types

1—ratings

2—categories

- 4—integers
- 8—numbers

If types is zero then simple rating scales without names, weights, etc are assumed

**Meta**—integer containing 1-bit flags determining available meta values

- 1—open (?)
- 2—unknown (!)
- 4—any (\*)
- 8—none (^)
- 16—inapplicable (~)

**Status**—integer indicating grid type

- 0—new grid
- 1—open existing grid
- 2—exchange grid
- 3—elements grid
- 4—constructs grid
- 5—copy grid

**Analysis**—integer containing 1-bit flags used in analysis

- 1—use selected elements in analysis
- 2—use selected constructs in analysis
- 4—use element weights in analysis
- 8—use construct weights in analysis
- 16—deselect rather than edit

**Control**—integer containing 1-bit flags used in WebGrid

- 1—turn help off
- 2—turn off annotation display
- 4—use selected elements in pair and triad
- 8—hide analysis parameters (Display, Focus, PrinGrid, Compare, etc)
- 16—hide elements (Main)
- 32—hide constructs (Main)
- 64—hide grid parameters (Options)
- 128—hide items (Options)
- 256—hide WebGrid parameters (Options)

### ***11.5 Display Parameters***

**Field 1**—integer containing 1-bit flags—default value is 7

- 1—show plot
- 2—show title
- 4—constructs as rows
- 8—show element and construct numbers
- 16—shade values
- 32—show element notes

- 64—show construct notes
- 128—black and white plot

### ***11.6 Focus Parameters***

**Field 1**—integer containing 1-bit flags—default value is 23

- 1—show plot
- 2—show title
- 4—constructs as rows
- 8—show element and construct numbers
- 16—shade values
- 32—tree on right
- 64—show element notes
- 128—show construct notes
- 256—black and white plot
- 512—interior clustering

**Field 2**—integer, element tree cut-off—default value is 25

**Field 3**—integer, construct tree cut-off—default value is 25

**Field 4**—integer, tree scale—default value is 100

**Field 5**—integer containing 1-bit flags—default value is 62

- 1—output text
- 2—output for elements
- 4—output for constructs
- 8—matches
- 16—links
- 32—sorts

**Field 6**—float, power for Minkowski metric—default value is 1.0

### ***11.7 Compare Parameters***

**Field 1**—integer containing 1-bit flags—default value is 27

- 1—show plot
- 2—show title
- 4—show element and construct numbers
- 8—shade values
- 16—show graph
- 32—show match values
- 64—show percent
- 128—black and white plot
- 256—show element notes and use in element ID matching
- 512—show construct notes and use in construct ID matching
- 1024—reverse order of grids in comparison

**Field 2**—3 2-bit fields for Minus and for different values of opt, plus 64 if split—default value is 11

**Field 3**—integer, cut-off—default value is 50

**Field 4**—integer, threshold—default value is 75

**Field 5**—integer, scale—default value is 100

**Field 6**—float, power for Minkowski metric—default value is 1.0

### ***11.8 PrinGrid Parameters***

**Field 1**—integer containing 1-bit flags—default value is 63

1—show plot

2—show title

4—show axes

8—show dimensions

16—show elements

32—show constructs

64—reverse horizontal

128—reverse vertical

256—reverse depth

512—3D plot

1024—show element and construct numbers

2048—show element notes

4096—show construct notes

8192—black and white plot

16384—do not spread

32768—do not show variance

65536—do not fit element and construct spreads

131072—do not centre on means

**Field 2**—integer containing 1-bit flags—default value is 14

1—output text

2—output variance

4—output element loadings

8—output construct loadings

**Field 3**—integer, plot scale—default value is 125

**Field 4**—integer, horizontal component—default value is 0

**Field 5**—integer, vertical component—default value is 1

**Field 6**—integer, depth component—default value is 2

**Field 7**—integer, horizontal rotation degrees—default value is 20

**Field 8**—integer, vertical rotation degrees—default value is 10

**Field 9**—integer, depth rotation degrees—default value is 0

default

### ***11.9 Match Parameters***

**Field 1**—integer containing type of match and 1-bit flags—default value is 12

- 1,2—integer, type of match
  - 0— all
  - 1— selected with all
  - 2— selected with selected
  - 3— selected with non-selected
- 4— elements
- 8— constructs
- 16— numbers
- 32— notes
- 64— separated

**Field 2**—integer, threshold—default value is 80

**Field 3**—float, power for Minkowski metric—default value is 1.0

### ***11.10 Crossplot Parameters***

**Field 1**—integer containing 1-bit flags—default value is 3

- 1— show plot
- 2— show title
- 4— reverse horizontal
- 8— reverse vertical
- 16— reverse depth
- 32— 3D
- 64— show element and construct numbers
- 128— show element notes
- 256— show construct notes
- 512— black and white plot
- 1024— do not spread

**Field 2**—integer scale—default value is 100

**Field 3**—integer, construct on horizontal axis—default value is 0

**Field 4**—integer, construct on vertical axis—default value is 1

**Field 5**—integer, construct on depth axis—default value is 2

**Field 6**—integer, horizontal rotation degrees—default value is 20

**Field 7**—integer, vertical rotation degrees—default value is 10

**Field 8**—integer, depth rotation degrees—default value is 0

### ***11.11 Style Parameters***

**Field 1**—integer. font size—default value is 12

**Field 2**—string. font—default value is “Arial”

**Field 3**—color. background—default value is FFFFE1

**Field 4**—color. title—default value is 40000

**Field 5**—color. elements—default value is C80000

**Field 6**—color. constructs—default value is 000080

**Field 7**—color. ratings—default value is 004000

*Display/Focus/Compare fields*

**Field 8**—color, lines—default value is DCDCDC

**Field 9**—color. low value shading—default value is FFFFFFFF

**Field 10**—color. middle value shading—default value is DCDCDC

**Field 11**—color. high value shading—default value is B4B4B4

*PrinGrid/Crossplot fields*

**Field 12**—color, links—default value is DCDCDC

**Field 13**—color, dimensions—default value is 9FDCCF

**Field 14**—color. axes—default value is 787878

**Field 15**—color, plane—default value is FFCC00

### ***11.12 Selection and weight specification***

The selwt parameter specifies whether selected and weighted elements and constructs should be used in analysis. It is an integer consisting of four 1-bit flags

- 1—use element weights
- 2—use construct weights
- 4—use selected elements
- 8—use selected constructs

## 12 RepGrid library

The *RepGrid* script environment provides some additional functions largely concerned with managing the script window which is a text window into which scripts may enter styled text and the user may type text or click on the mouse.

Text output to the window may be declared clickable by placing a code string followed by a backslash character at the beginning of the text string. When the text is clicked the code string is made available to the script.

### 12.1 Script window management

UndoSave(action As String)—saves the current grid on the undo stack before changing it, also storing the action that will be taken to change it

Halt(message As String)—stop current script and append the message to the text in the script window

SetMessage(message As String)—sets the message text above the script window

### 12.2 Script window functions

SetBackColor(c As Color)—set the background color of the script window

StyleAdd(name As String, aTextAlign As Integer, aTextStyle As Integer, aTextSize As Integer, aTextColor As color, aTextFont As String)—add a new style by name to a hash table of styles

StyleSet(style As String)—set the default style as specified

TextClear()—clear the script window

Output(s As String)—append s in the default style to that in the script window

OutputSelect(s As String )—append s in the default style to that in the script window and selects the new output

Output(s As String, style As String)—append s in the specified style to that in the script window—do not change the default style

### 12.3 Script calls supporting interaction

The *RepGrid* script environment manages user interaction with the script window by supporting script calls that wait for interaction as specified in an input mode. The modes are designated by an integer and constants are defined for some of them:

0:	script halted
1:	script running
kClick=2:	script waiting for mouse click
kText=3:	script waiting for text input terminated by return or mouse click if any defined
kCMenu=4:	script waiting for text input terminated by return or mouse click if any defined or construct menu defined in item "C" in the default hash store (named by the empty string)

kMenu=5: script waiting for text or click if any defined or menu defined in item "M"  
in the default hash store (named by the empty string)

ScriptWait(returnscript As String, imode As Integer)—the script specified is pushed on the  
return stack and then *RepScript* waits for user interaction as specified in imode—when  
the specified interaction is complete *RepScript* executes a Flow() function to return to the  
specified script

GetiMode() As Integer—current input mode

InCode() As String—the code specified in the clickable text that was clicked

Input() As String—the text entered by the user, either by typing or through a menu

## 13 RepServe library

The *RepServe* environment provides access to the client request including any POST data, and an output stream to buffer the text generated to be returned to the client.

### 13.1 Getting client request

GetRequest() As String—get the request up to and not including the POST data

GetRequestField(field As String) As String—value of request field specified

GetPost(name As String) As String—value of item named in form posted

GetPostI(name As String) As Integer—value as Integer of item named in form posted

CheckPost(name As String, ByRef value As String) As Boolean—true if item named was posted—value returned as String

CheckPostI(name As String, ByRef value As Integer) As Boolean—true if item named was posted—value returned as Integer

OpenGrid() As Boolean—sets up the grid from data embedded in hidden fields—returns true if grid is valid

### 13.2 Sending server response

Out(s As String)—appends s to the *RepServe* stream

Outln([s As String])—append s if present followed by CRLF to the *RepServe* stream

OutWebGrid()—outputs the grid data in hidden fields

GetOut() As String—return the output stream as a String and clear it

SendSocket(s As String)—sends s as the response

### 13.3 WebGrid functions

GetNextGridHeader(path As String, rootcode As String, ByRef n As Integer) As String—increment n and return the header information about the n'th grid in the directory specified—start with n zero—set n zero when there are no more grid files

### 13.4 WebNet functions

GetNetPNG(net As String) As String—returns a PNG representation of the net

GetNetClickNote(net As String, x As Integer, y As Integer) As String—note field of node at location specified in net—empty string if no node (or note field is empty)

### 13.5 Server log window functions

SendLog(s As String, time As Boolean)—append s to the log window including the time if time is true

### 13.6 System function extensions

xGet("System", hash)—adds the following items in the *RepServe* environment:

- RemoteIP:** numeric internet address of client
- RemoteDNS:** symbolic internet address of client
- IPAndPort:** internet internet address and port of server as specified by client
- Email:** the email address specified in the *RepServe* pane
- Flags:** the flags set by the checkboxes in the *RepServe* pane

The flgs are a bit-pattern set as follows:

- 1 Log in window
- 2 Log in log file
- 4 Show memory use in log
- 8 Show client browser in log
- 16 Show socket activity
- 32 Show request in log window
- 64 Show post data in log window
- 128 Show reply in log window
- 256 Capture grids
- 2048 Listening on local machine
- 4096 Also listening on Internet

## 14 Net functions

*RepScript* provides functions to access a net, getting and setting its data. The net accessed is initialized by *RepNet* to be the net that has been opened.

nGet(argument strings) gets data from a net.

nSet(argument strings) changes data in a net.

The arguments to these functions are always of type String. The default return type for nGet is String but can also be another type as specified by a suffix letter, nGetI for Integer, nGetD for Double, nGetB for Boolean, nGetX for a call without a return value.

There are also some net functions that return arrays: nGetA, nGetAI, gGetA2I.

### 14.1 Net Management

nSet("Update")—updates the visible appearance of the net for any changes made

nSet("Refresh")—refreshes the visible appearance of the entire net

nSet("UndoSave", action)—saves the current net on the undo stack before changing it, also storing the action that will be taken to change it

### 14.2 Net Data Items

nGetI("NN") As Integer—number of nodes

nGetI("NL") As Integer—number of links

nGet("Net", hash)—net encoded in hash store specified

nGetB("Lock") As Boolean—true if net locked

nSet("Lock", b)—set the lock to b, where b is "true" or "false"

nGet("File") As String—net file name if any

nGet("Field", field) As String—text in field specified, either "Label" or "Note"

nSet("Field", s, field)—put s in field specified, either "Label" or "Note"

### 14.3 Net Types

nGetA("NodeTypes") As String()—array of node type names

nGetA("LineTypes") As String()—array of line type names

nGetA("MarkeTypes") As String()—array of mark type names

### 14.4 Net Nodes

nGetA("Node") As String()—array of node labels

nGetA("Notes") As String()—array of node notes

nGetAI("Types") As Integer()—array of node types

nGetAI("Marks") As Integer ()—array of node marks

nSet("Mark", node, mark)—sets the specified node's mark

nSet("Note", node, note)—sets the specified node's note

### 14.5 Net Lines

nGetA2I("Lines") As Integer(.)—2D array of node lines as node number pairs

nSet(node1 As Integer, node2 As Integer, linetype As Integer)—add line from node1 to node2 of type line type, replacing any existing line

### 14.6 Script interaction in RepNet

When a net with a script is opened the script is called from the net with the scriptState set to "Start". When the net with a script is locked the script is called with the scriptState set to "Cursor" if the mouse moves; "Click" if the mouse is clicked or double-clicked; "Locked" if the user locks the net. The arguments passed are:

Start—sent when net is opened—no arguments

Locked—sent when net is locked—no arguments

Cursor—send when the mouse has moved and the cursor needs to be set—node number of the node under the cursor is sent in vGetI(0), -1 if not over a node—return a cursor number by vSet(cursornumber,0)

Click—send when the mouse has been clicked or double-clicked—node number of the node under the cursor is sent in vGetI(0), -1 if not over a node—vGetB(1) is true for a double-click.

### 14.7 Node and line specifications in a hash store

When a hash store is used encode a node, the items in it are:

**Label:** name As String

**Note:** note As String

**Type:** type As Integer

**TypeName:** type name As String

**Mark:** mark As Integer

**Selection:** selection As Integer

**Left:** left position As Integer

**Top:** top position As Integer

**Width:** width As Integer

**Height:** width As Integer

When a hash store is used to encode a line, the items in it are:

**Type:** type As Integer

**Start:** start node As Integer

**End:** end node As Integer

**Type:** type As Integer

When a hash store is used to encode a net, the items in it are:

**Net:** indicator that hash store contains net data

**Window:** window title As String if net is showing in a window

**Width:** net width As Integer if net is showing in a window

**Height:** net height As Integer if net is showing in a window

**N:** TAB-separated list of node ID's As String

**L:** TAB-separated list of line ID's As String

Plus a list of nodes and lines in the format above but with the name of each item suffixed with a period followed by its ID.