Chapter

11

Macro Reference

This reference chapter consists of two sections: “ReportBasic conventions” and “Command reference.”

- “ReportBasic conventions” lists and describes the objects and events to which you can link a macro, such as Before Loading a Report or at Application Start Up. It then goes on to describe programming conventions of the ReportBasic macro language and data types for variables.
- “Command reference” lists and describes each ReportBasic and DataSet Control command and then lists and describes ReportBasic commands you can use to build ReportSmith macros.

ReportBasic conventions

Earlier chapters have shown you examples of linking macros to events to control the execution of the macro. This chapter details the types of events to which you can link macros.

Linking to events

Table 11.1 shows the ReportSmith events available for linking to macros, grouped by the object with which the events are associated.

Table 11.1: ReportSmith events for linking to macros

Macro type	Object type	Event	Comments
Global	**Application** (ReportSmith itself)	Keystroke	Links macro execution to a user-generated keystroke. You specify the key that must be pressed to execute the macro.
		Application startup	Links macro to opening of ReportSmith itself, independent of having any report open.

Table 11.1: ReportSmith events for linking to macros (continued)

Macro type	Object type	Event	Comments
Global	**Application** (ReportSmith itself)	Before New Report	Links macro execution to creation of a new report. The macro is executed immediately before the report is created.
		After New Report	Links macro execution to creation of a new report. The macro is executed immediately after the report is created so report data is available to the macro.
		New File Icon Click	Links macro execution to a user-generated (or simulated) click on the New File toolbar button.
		Before Report Load	Links macro execution to opening of an existing report. The macro is executed immediately before a report is opened.
		After Report Load	Links macro execution to opening of an existing report. The macro is executed immediately after a report is opened so report data is available to the macro.
		After Report Connects	Links macro execution to a specific point in report creation. Once a connection has been specified, the macro is executed.
		Before Executing SQL	Links macro execution to a specific point in report creation: the report type and style have been specified (for new reports), tables have been selected and linked, but SQL has not yet been executed.
		On SQL Execution Error	Links macro execution to an error in executing an SQL query. If no error occurs, then the macro does not run.
		SQL Icon Click	Links macro execution to a user-generated, or simulated, click on the SQL toolbar button.
		Before Report Print	Links macro execution to the printing of a report. Once the print operation has been chosen, the macro is executed immediately before the report is printed.
		Before Report Save	Links macro execution to the saving of a report. Once the Save (or Save As) operation has been chosen, the macro is executed immediately before the report is saved.
		After Report Save	Links macro execution to the saving of a report. Once the Save (or Save As) operation has been chosen, the macro is executed immediately after the report is saved.
		Before Report Close	Links macro execution to the closing of a report. Once the Close operation has been chosen, the macro is executed immediately before a report (*any* report) is closed.
		After Report Close	Links macro execution to the closing of a report. Once the Close operation has been chosen, the macro is executed immediately after a report (*any* report) is closed.

Table 11.1: ReportSmith events for linking to macros (continued)

Macro type	Object type	Event	Comments
Global	**Application** (ReportSmith itself)	Before Application Close	Links macro execution to the closing of ReportSmith itself. Once the Close operation is chosen, the macro is executed immediately before ReportSmith closes.
Report	**Report**	Keystroke	Links macro execution to a user-generated keystroke. You specify the key that must be pressed to execute the macro.
		Before Report Load	Links macro execution to opening of this report. The macro is executed immediately before the report is opened.
		After Report Load	Links macro execution to opening of this report. The macro is executed immediately after this report is opened so report data is available to the macro.
		Before SQL Execution	Links macro execution to a specific point in report update. The macro is executed immediately before the SQL query is executed.
		Before Print	Links macro execution to the printing of this report. Once the print operation has been chosen, the macro is executed immediately before the report is printed.
		Before Report Save	Links macro execution to the saving of this report. Once the Save (or Save As) operation has been chosen. The macro is executed immediately before the report is saved.
		Before Report Close	Links macro execution to the closing of this report. Once the Close operation has been chosen, the macro is executed immediately before the report is closed.
		Menu Selection	Links macro execution to the selection of any menu item in this report.
		On SQL Error	Links macro execution to an error in executing an SQL query. If no error occurs, then the macro does not run.
	Data Field	Display	Links macro execution to the display of a particular data field, derived field, or summary field in this report. (The same data field, for example, used in another report, will not trigger execution of this macro unless the macro is explicitly linked to *that* data field in *that* report.)
	Group Header	Creation	Links macro execution to the creation of a header for either the entire report or a particular group.
	Group Footer	Creation	Links macro execution to the creation of a footer for either the entire report or a particular group.

Note: You link macros to events and objects using the Macro Links dialog box. For a conceptual understanding of macros and how they link to objects and events, see “Using the DataSet Control” on page 215.

Application events

Global macros are linked to the ReportSmith application so you can link global macros only to *application* events, as shown in Table 11.1 on page 247. Application events occur when you run ReportSmith and when you run any report.

For information on how to link events to specific reports, refer to “Report events” on page 252.

The following sections describe in detail each application event listed in Table 11.1 and provide specific scenarios to show you how to practically apply them.

Keystroke

The *KeyStroke* event is unique in that it has additional options tied to it. When you link a macro to the KeyStroke event, it can be to tied any key or key combination on your keyboard. A macro linked to the KeyStroke event runs when the user presses the key or key combination you specify, or when this keystroke is simulated through the macro language.

For example, suppose you want to link a macro to the keystroke, *Ctrl+R*, so that whenever you press *Ctrl+R*, ReportSmith loads a report that you work on often.

Note: Because local macro scope takes precedence over expanded scope, the active report has a report macro linked to the same keystroke, and then that report macro is executed first (See “Keystroke” on page 250.).

Before creating a new report

A macro linked to this event runs *after* you choose File|New (or click the New button to create a new report), but *before* you choose the tables that you want to include in the report. This event lets you perform tasks, such as providing or denying certain users the ability to create new reports.

For example, you can have a global macro display a dialog box that prompts for a password. If the user-entered password is correct, you can enable the user to continue with the New operation. If it’s incorrect, you can cancel the New operation and display an informative message describing why the operation is canceled.

After creating a new report

A macro linked to this event runs *after* you choose File|New to create a new report, and *after* ReportSmith executes a query based on the tables you choose in this report. You might want to link to this event to change the default configuration for new reports.

For example, you can change the margins, specify the display mode (draft or presentation), turn the report boundaries on or off, turn the grid on or off, and so on.

Before starting the application

A macro linked to this event runs immediately after you click the ReportSmith icon to open the application and *after* ReportSmith displays its About box ("splash screen"). You can link a macro to this event to change the default ReportSmith environment or perform other tasks before you open ReportSmith.

For example, you can add new menu items, disable or remove existing menu items, load the reports you use on a daily basis, or execute a menu command, such as New or Open on the File menu.

You can also launch other Windows applications simultaneously, such as Visual Basic, Excel, and PowerBuilder.

Before printing a report

A macro linked to this event runs *after* you select Print from the File menu, but *before* the report is actually sent to the printer. You can link a macro to this event when you want to perform print-related tasks.

For example, you can have a macro display a dialog box that specifies the printer being used. Or you might want to display an interactive dialog box that lets the user enter printer parameters, such as margin specifications, number of copies, page orientation, paper size, and so on.

You can also link a macro to this event to warn users when a report is large, and then have the macro display a dialog box that gives them the opportunity to cancel the print.

Before loading a report

A macro linked to this event runs *after* you select File|Open and *after* you select an .RPT file from the Open Report dialog box. However, it runs *before* ReportSmith actually opens and displays the corresponding report. You can use this event to determine whether the report can actually be opened.

For example, suppose you want to ensure that a user has only one .RPT file open at a given time. You can create a global macro and link it to this event so that it saves and closes all active reports before allowing any new report to be opened.

After loading a report

A macro linked to this event runs *after* you select a report using File|Open, and *after* ReportSmith actually opens and displays the selected report. You might want to use this event to determine how reports appear when they are first opened.

For example, you can display all reports in draft mode by creating a macro that turns draft mode on when a report is opened. In a similar way, you might want to create a macro that resets the margins of all opened reports according to your corporate standards.

Here's another example of when you might want to link a macro to this event: Suppose you want to keep track of all the report files that a given user opens. A macro linked to this event can identify the title of each report and write its title and the time it was opened in an ASCII file.

Before saving a report

A macro linked to this event runs *after* you select File|Save (or File|Save As), but *before* ReportSmith actually saves the active report in an .RPT file. If this is your first time saving the report, or if you're saving an existing report under a different name, the macro runs *before* the dialog box prompts you for a filename.

For example, by linking a macro to this event, you can prevent a user from overriding certain .RPT files that you don't want modified. Suppose you want to prevent users from modifying all reports created in the month of May. Simply have the macro display a message indicating that the report can't be modified whenever a user attempts to modify and save a May report, and then cancel the Save (or Save As) operation. In a similar way, you can use this event to verify sufficient disk space, and then display a warning message if sufficient disk space is unavailable.

After saving a report

A macro linked to this event runs *after* you select File|Save (or File|Save As), and *after* ReportSmith saves the active report in an .RPT file. A macro linked to this event can, for example, close reports immediately after ReportSmith saves them. In a similar way, you can use this event to automatically create backup copies of saved reports in a backup directory.

Before closing the application

A macro linked to this event runs *after* you select File|Exit to close ReportSmith, but *before* ReportSmith actually closes. You can link a macro to this event for tasks such as preventing ReportSmith from closing under certain circumstances.

For example, you might not want ReportSmith to close if a driving application, such as PowerBuilder, is still open. In this case, you can have a macro cancel the close, and then display a message informing the user to close the driving application first.

You could also link a macro to this event to automatically save all open reports before ReportSmith closes, rather than have ReportSmith prompt the user to save each open report individually.

Report events

A report macro is linked to a *specific* report and, therefore, it becomes a part of that report. You link report macros to report events. Macro commands in this macro are unavailable to any other report, unless the macro is saved to an .MAC file and explicitly loaded for use in other reports.

Note: To link events to the ReportSmith application or to reports in general, see "Application events" on page 250.

The following subsections describe each report event in detail and provide specific scenarios to show you how to use them in real-world applications.

Keystroke

The *KeyStroke* event is unique in that it has additional options tied to it. When you link a macro to the KeyStroke event, it can mean any key or key combination on your keyboard. A macro linked to the KeyStroke event runs when you press the key or key combination you specify, and the report to which the macro is also linked is the active report.

Note: A KeyStroke event linked to a report macro overrides one that is linked to a global macro.

For more detailed information and for examples of how you can use the KeyStroke event, refer to “Keystroke” on page 250.

Before opening the report

A macro linked to this event runs *after* you open this particular report, but *before* ReportSmith executes the query to run the report. You can use this event to automatically set report variables for the report’s selection criteria.

Report variables enable you to create dialog boxes that prompt users for query values before a report is run. If you create a macro that sets report variables automatically, ReportSmith knows not to display the dialog box.

If the macro linked to this event calls the Set Report Variable command (refer to “GetSQL$ (dataset object)” on page 340), it sets the report variables for the report being loaded, rather than the active report.

After opening the report

A macro linked to this event runs *after* you open the specific report that the macro is linked to, and *after* ReportSmith runs the query and displays the report. You can use this event to trigger an action immediately after the report opens.

For example, you can automatically send a specific report to the printer after it first opens. Or, you can have the opening of a specific report trigger the loading of additional, related reports.

You can also use this event if you want a specific report to appear in a different display mode than all other reports in your company. For example, suppose all reports are set up to appear in presentation mode when you first open them, and you want a specific report to appear in draft mode instead. You can create a report macro and link it to this event so that only the specific report appears in draft mode every time you open it.

Before printing the report

A macro linked to this event runs *after* you select File|Print, but *before* ReportSmith actually prints the specific report to which the macro is also linked. Suppose you have a specific report that is particularly large, and you don't want to tie up the department's printer. You can link a macro to this event to automatically send that report to another printer which is not used as often.

You can also use this event to warn users that the specific report is particularly large and that printing it will be a time-consuming process. Then you can provide them the opportunity to cancel the print operation.

Before saving the report

A macro linked to this event runs *after* you attempt to save a specific report (to which the macro is also linked), but *before* ReportSmith actually saves it in an .RPT file. If this is your first time saving the report, the macro runs *before* the dialog box prompts you for a filename.

By linking a macro to this event, you can prevent a user from overriding a specific report that you don't want modified. Simply have the macro display a message indicating that the report can't be modified, and then cancel the save operation.

Before closing the report

A macro linked to this event runs *after* you select File|Close to close a specific report, but *before* ReportSmith actually closes it. You can link a macro to this event to restore options set by a macro that was linked to the After Opening the Report event.

For example, suppose the macro that was run after you opened the report changed the display mode from Presentation to Draft. You can link another macro to this event to restore the display mode back to Presentation.

Selecting a menu item

Like the Keystroke event, which lets you link to specific keystrokes, this event lets you link to *specific menu items*.

A macro linked to this event runs when you select the menu item to which the macro is also linked, but before the corresponding action of that menu item takes place. You can use the *ResumeEvent* command to determine whether or not the corresponding action is executed.

Suppose you have your own help file built for a specific report, and you want this file to appear (rather than ReportSmith's standard help file) whenever you select Index from the ReportSmith Help menu. You can link a macro to this event to replace the ReportSmith help file with the new help file for the specific report only.

Data field events

Currently, ReportSmith uses only one data field event called the *Display* event. A macro linked to this event runs whenever ReportSmith generates a value for the data field object to which the macro is also linked.

The primary purpose of the Display event is to enable you to do conditional formatting. When you want a macro to do conditional formatting, you create it based on criteria to which certain values in the report columns can correspond.

When you create a conditional formatting macro, you can use the *FieldFont* and *FieldText* macro commands. These commands let you tell the macro how to format the values that fulfill the specified criteria.

Group Header/Footer events

Group Header and Group Footer objects can take only one event: the Creation event. When you link a macro to the Header/Footer Creation event, you must choose the grouping level of the header or footer to which you want to link.

A macro linked to these events can call the *ResumeEvent* command with a parameter of 0 to suppress the creation of an individual group header or footer based on the data in the report.

Evaluation of expressions

When evaluating expressions, ReportBasic gives precedence to operators. To override the default precedence you can use parentheses to control relative priority of each expression or formula.

The following table describes each operator. The operators are listed in order of precedence; the first operator is the first to be evaluated.

Table 11.2: Default precedence of ReportBasic operators

Operator	Description
( )	Array element.
. [Period]	Record member—the left operand must be a record variable, and the right operand must be the name of a field.
Imp	Implication—operands can be Integer or Long. The operation is performed bitwise. (A Imp B) is the same as ((Not A) OR B ()).
Eqv	Equivalence—operands can be Integer or Long. The operation is performed bitwise. (A Eqv B) is the same as (Not (A X or B)).
Xor	Exclusive Or—operands can be Integer or Long. The operation is performed bitwise.
Or	Inclusive Or—operands can be Integer or Long. The operation is performed bitwise.
And	And—operands can be Integer or Long. The operation is performed bitwise.
Not	Unary Not—operand can be Integer or Long. The operation is performed bitwise (one's complement).

Table 11.2: Default precedence of ReportBasic operators (continued)

Operator	Description
>, <, =, <=, >=, <>	Sequence used by the language specified by the user using the Windows Control Panel. The result is 0 for FALSE and –1 for TRUE.
–, +	Numeric addition and subtraction. The + operator is also used for string concatenation.
Mod	Modulus or Remainder. The operands can be Integer or Long.
\	Integer division. The operands can be Integer or Long.
*, /	Numeric multiplication or division. For division, the result is a Double value.
–,+	Unary minus and plus.
^	Exponentiation.

Data types of variables

A variable declared *inside* of a procedure has scope local to that procedure. A variable declared *outside* of a procedure has scope local to the module.

It is permissible for a procedure to declare a variable with the same name as a module variable. When this happens, priority is given to the more local variable, and the module variable is not accessible by the procedure.

The *Shared* keyword is included for backward compatibility with older versions of BASIC. It is not allowed in *Dim* statements inside of a procedure; it has no effect.

BASIC allows a variable to be automatically declared without the use of a Dim statement. If a variable is first used with a type character (such as $, for example) as a suffix to its name, the variable is automatically declared to be a *local* variable of the specified type. If no type character is specified, the variable is automatically declared to be a local variable of type Double. It is considered good programming practice to declare all variables and not make use of this feature. It is also recommended that you place all procedure-level Dim statements at the beginning of the procedure.

Numeric types

ReportBasic supports use of four numeric types for use in variables. The four numeric types are:

Type	Range
Integer	From –32,768 to 32,767
Long	From –2,147,483,648 to 2,147,483,647
Single	From –3.402823e+38 to –1.401298e–45, 0.0, 1.401298e–45 to 3.402823466e+38
Double	From –1.797693134862315d+308 to –4.94065645841247d–308, 0.0, 2.2250738585072014d–308 to 1.797693134862315d+308

Numeric values are always signed.

Boolean types

BASIC has no true Boolean variables. BASIC considers 0 to be FALSE and any other numeric value to be TRUE. Only numeric values can be used as Booleans. Comparison operator expressions always return 0 for FALSE and –1 for TRUE.

Integer constants

Integer constants can be expressed in decimal, octal, or hexadecimal notation:

- Decimal constants are expressed by simply using the decimal representation.
- To represent an octal value, precede the constant with “&O” or “&o” (e.g., &o177).
- To represent a hexadecimal value, precede the constant with “&H” or “&h” (e.g.,&H8001).

Example

```
'The following constant is a decimal integer.
NumberVar = 5

'The following constant is an octal integer. (Note the difference between
the O-letter 'character and the 0 (zero) character in the integer.)
NumberVar = &O0177

'The following constant is a hexadecimal integer.
NumberVar = &HF017
```

Strings

BASIC strings can be either fixed or dynamic:

- Fixed strings have a length specified when they are defined. The length cannot be changed.
- Fixed strings cannot be 0 length.
- Dynamic strings have no specified length.
- Any string (fixed or dynamic) can vary in length from 0 to 32,767 characters.
- There are no restrictions on the characters which can be included in a string. For example, the character whose ANSI value is 0 can be embedded in strings.
- Local string variable names require a $ at the end of the name of the variable.

Example

```
MyStringVar$ = "A text string or field?!"
```

Records

Record variables, including dialog boxes, are declared by using a *Dim...As* clause and a type name which has previously been defined using the Type statement.

The syntax for Records looks like this:

```
Dim VariableName As TypeName
```

Records are made up of a collection of data elements called fields. These fields can be of any numeric, string, or previously-defined record type.

For details on accessing fields within a record, see “Type” on page 413.

Arrays

Arrays are created by specifying one or more subscripts at declaration or Redim time. Subscripts specify the beginning and ending index for each dimension. If only an ending index is specified, the properties of the beginning index are based on the Option Base setting. Array elements are referenced by enclosing the proper number of index values in parentheses after the array name, for example, arrayname(i,j,k).

The syntax for Arrays can be either of the following:

```
Dim variable( [ subscriptRange, ... ] ) As typeName
Dim variable_with_suffix([ subscriptRange, ... ])
```

where subscriptRange is of the format:

```
[ startSubscript To ] endSubscript
```

For more information, see “Dim” on page 299.

Example

```
Dim DS as DataSet 'Declare a DataSet
Dim A(5,2) 'Declare an array
Dim A as Integer 'Declare a variable
```

Application Data Types (ADTs)

Application Data Types are specific to each application that embeds the macro language. ADT variables have the appearance of standard BASIC records. The main difference is that they can be dynamic. Creating, modifying or querying the ADT or its elements causes application-specific actions to occur. ADT variables and arrays are declared using the *Dim* or *Global* statements just like any other variable.

Dialog-box records

Dialog-box records look like any other user-defined data type. Elements are referenced using the same recordname.elementname syntax. The difference is that each element is tied to an element of a dialog box. Some dialog boxes are defined by the application, others by the user.

For more information, see “Begin Dialog...End Dialog” on page 270.

Conversions

BASIC automatically converts data between any two numeric types. When converting from a larger type to a smaller type (for example, Long to Integer), a numeric overflow can occur. This indicates that the value of the larger type is too large for the target data type.

Important Loss of precision is not a run-time error (for example, when converting from Double to Single, or from either float type to either integer type.)

BASIC also automatically converts between fixed strings and dynamic strings:

- When converting a fixed string to dynamic, a dynamic string which has the same length and contents as the fixed string is created.
- When converting from a dynamic string to a fixed string, some adjustment can be required. If the dynamic string is shorter than the fixed string, the resulting fixed string is extended with spaces. If the dynamic string is longer than the fixed string, the resulting fixed string is a truncated version of the dynamic string. No run-time errors are caused by string conversions.

No other implicit conversions are supported. In particular, BASIC does not automatically convert between numeric and string data. Use the functions *Val* and *Str$* for such conversions.

Trappable errors

Table 11.3 lists the run-time errors which the macro language returns.

These errors can be trapped by the “On Error” on page 370. The “Err (function)” on page 309 can be used to query the error code, and the “Error$” on page 311 can be used to query the error text.

Table 11.3: ReportBasic trappable errors

Error code	Error text	Error code	Error text
5	Illegal function call	**62**	Input past end of file
6	Overflow	**63**	Bad record number
7	Out of memory	**64**	Bad file name
9	Subscript out of range	**68**	Device unavailable
10	Duplicate definition	**71**	Disk not ready
11	Division by zero	**74**	Can’t rename with different drive
14	Out of string space	**75**	Path/File access error
19	No resume	**76**	Path not found
20	Resume without error	**102**	Command failed
28	Out of stack space	**901**	Input buffer would be larger than 64K
35	Sub or function not defined	**902**	Operating system error
48	Error in loading DLL	**903**	External procedure not found

Table 11.3: ReportBasic trappable errors (continued)

Error code	Error text	Error code	Error text
52	Bad file name or number	**904**	Global variable type mismatch
53	File not found	**905**	User-defined type mismatch
54	Bad file mode	**906**	External procedure interface mismatch
55	File already open	**907**	Push-button required
58	File already exists	**908**	Module has no MAIN
61	Disk full	**910**	Dialog box not declared

Command reference

The commands available to you in ReportBasic are listed here in alphabetical order. Note that some commands are actually methods or properties of the dataset object or the report object, rather than the report itself or the displayed report “surface.” This is noted in the command descriptions, wherever applicable.

Commands by alphabetical listing

'$CStrings

The *'$CStrings* metacommand tells the compiler to treat a backslash character inside a string (\) as an escape character. This treatment is based on the ‘C’ language (and its variants).

Syntax '$CSTRINGS

Comments The supported special characters are:

Newline (Linefeed): \n
Horizontal Tab: \t
Vertical Tab: \v
Backspace: \b
Carriage Return: \r
Formfeed: \f
Backslash: \\
Single Quote: \'
Double Quote: \"
Null Character: \0 (zero)

The instruction Hello\r World is the equivalent of Hello + Chr$(13)+ World. In addition, any character can be represented as a 3-digit octal code or a 3-digit hexadecimal code: Octal Code\ddd or Hexadecimal Code\xddd. For both hexadecimal and octal, fewer than 3 characters can be used to specify the code as long as the subsequent character is not a valid (hex or octal) character. To tell the compiler to return to the default string processing mode, where the backslash character has no special meaning, use the *'$NoCStrings* metacommand.

Example '$CStrings

’$Include metacommand

Tells the compiler to include statements from another file.

Syntax ’$Include: "filename"

Parameters filename—A file with BASIC source code to compile along with the current source.

Comments Comments which include metacommands are only recognized at the beginning of a line. For compatibility with other versions of BASIC, you can use single quotes (‘) to enclose the filename.

A file extension of .SBH is suggested for the macro language include files. This is only a recommendation, and any other valid file extension can be used. If no directory or drive is specified, the compiler searches for filename on the source file search path.

Example '$Include "MYLIB.BAS"

‘$NoCStrings metacommand

Tells the compiler to treat a backslash inside a string as a normal character. This is the default.

Syntax '$NocStrings

Comments You can use the *‘$CStrings* metacommand to tell the compiler to treat a backslash character inside of a string as an escape character.

Example '$NocStrings

Abs

Returns the absolute value of a specified numeric expression.

Syntax Abs(numeric_expression)

Parameters numeric_expression—A field or variable representing a number.

Returns Matches the type of numeric expression. This includes variant expressions that return a result of the same vartype as input, except vartype 8 (string) is returned as vartype 5 (double) and vartype 0 (empty) is returned as vartype 3 (long).

Example Value1 and Value2 below are integer variables.

```
Difference=ABS(Value1-Value2)
```

ActiveTitle$

Gets the window title of the currently active report. Used only as a function.

Syntax ActiveTitle$

Comments For saved reports, the title contains the catalog, folder(s), and file name of the report's .RPT file.

Example ActiveReport$=ActiveTitle$

Msgbox "Your report is called" + ActiveReport$

AddGroup

Adds grouping criteria at the specified level. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see "Using the DataSet Control" on page 215.

Syntax [Object].AddGroup Table$, DataBase$, Field$, Level, Type, [NumRecs]

Parameters Table$, DataBase$, Field$—*Table$, DataBase$*, and *Field$* serve to identify the field to be grouped upon.

The level parameter specifies the grouping level that you want information about where 0 is the entire report group, 1 is the primary grouping criteria, 2 is the secondary grouping criteria, and so forth.

Valid levels are 1 to 1+ the current number of groups defined

0 Same value
1 Every *n* records (*n* is the NumRecs parameter)
2 Daily
3 Monthly
4 Weekly
5 Annually
6 Quarterly
7 Hourly
8 Every minute
9 Every second
10 Every $^{1}/_{10th}$ of a second
11 Every $^{1}/_{100th}$ of a second
12 Every $^{1}/_{1000th}$ of a second

Types 2–12 are valid only for date and/or time fields.

NumRecs—When grouping by every *n* records, this parameter specifies how many records per group. This parameter is used only with Type 1.

Returns 0 on success, a non-zero value on error.

Comments If a group exists at the given level, all groups at that level and higher are adjusted up one level to accommodate the new group.

Example
```
'Group by DEPT_ID, break on same value
MyData.AddGroup "dbo.emp","hr", "DEPT_ID",1,0,0
```

AddMenu

Allows you to add your own commands to ReportSmith's Main Menu to execute a macro when the command is chosen.

Syntax AddMenu MenuText$, Macro$, AfterMenu$, HelpText$

Parameters MenuText$—A string that specifies the text of the new menu item.

Macro$—A string that specifies which macro should be run when the new menu item is chosen.

AfterMenu$—A string that specifies which existing menu item the new menu item should follow. Specify this existing menu item by listing the menu name, followed by a vertical bar, and the menu item (omitting accelerator characters and so on), as in File|New or Tools|Macro.

HelpText$—A string, enclosed in double-quotes, specifying the text of the hint that appears in the ReportSmith status bar when the new menu item is selected.

Comments You can have ReportSmith execute an active report macro or global macro by specifying a macro name. (The name that appears in the active macro list in the macro dialog box.) You can also have ReportSmith execute a .MAC file by passing a string with a path and file name of the .MAC file you wish to run.

If this command is entered in a *global* macro linked to the Application Startup event, the menu will be customized each time ReportSmith is opened.

This command can be used to check the state of the menu used by default for new reports, by placing an exclamation point (!) before the menu name. This can be done whether the menu item is specified by command or relative location.

Examples AddMenu "Open Sales Reports","LoadSales","!File|Open",‘Opens sales reports’

—or—

AddMenu "Open Sales Reports", "C:\RPTSMITH\MACROS\LSALES.MAC","File|Open",""

AddSort

Adds a sorting criteria to the current dataset at the specified level. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AddSort Table$, Database$, Column$, Ascending, Level

Parameters Table$—Defines a string of the form: Owner.TableName, or for local databases such as dBASE, the file name of the local database file.

Database$—For local databases or servers that don’t require that a database be specified, this parameter should be set to a null string.

Column$—The field that is being sorted.

Ascending—Set to zero to sort from smallest to largest,set to non-zero to sort from largest to smallest.

Level—Indicates its priority among other sorting criteria for this dataset. Valid values for this parameter are 1 to ([the number of current sorting criteria] +1).

Returns If an invalid index is specified, the command fails and returns an error. This command returns 0 on success, non-zero on error.

Example MyData.AddSort "dbo.emp","HR","DEPT_ID",1,1

AddSummary

Adds a summary field to the specified grouping level and index. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AddSummary Table$, DataBase$, Field$, Level, Type

Parameters Table$,DataBase$,Field$—Identify the table, database, and particular field to be summed.

Level—Specifies the grouping level for creating a summary field. The *Level* parameter specifies the grouping level that you want information about, where 0 is the entire report group, 1 is the primary grouping criteria, 2 is the secondary grouping criteria, and so forth. Valid values for levels are zero to the number of defined groups.

Type—Specifies the type of summary:

1 Sum
2 Daily
3 Count
4 Minimum
5 Maximum
6 Average
7 First
8 Last
9 Standard Deviation
10 Variance

Returns Non-zero value for an error and the *Error$* property, set to indicate the error.

Example MyData.AddSummary "dbo.emo","HR","SALARY",1,1

AddTable

Adds a table to a dataset object. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AddTable Table$,DBase$

Parameters Table$—Defines a string of the form: Owner.TableName or, for local databases such as dBASE, the file name of the local database file.

dBase$—This parameter is for data servers that require databases. (For servers that don’t require a database, it should be set to a NULL string.)

Returns 0 on success and a non-zero on error.

Comments Before you can add a table to a data set, you must establish a connection.

Example 'Add the EMP table from the PUBS database with the outer dbo
MyDataSet.AddTable "dbo.emp","PUBS"

AllDataBases$

Returns a comma-delimited list of all databases available under the current connection. This command represents a property—an object variable—of the dataset object, which, in, turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AllDataBases$

Comments A connection must be made before the list of all databases can be retrieved from a dataset object.

Example ListofDataBases$=AllDataBases$

AllOwners$

Returns a comma-delimited list of all owners available under the current connection. This command represents a property—an object variable—of the dataset object, which, in turn, represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AllOwners$

Comments Before the list of all owners can be retrieved from a dataset object, a connection must be made.

Example OwnerList$ = MyData.AllOwners$

AllTables$

Returns a list of all tables, separated by commas, for a connection. This command represents a property—an object variable—of the dataset object, which, in turn, represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].AllTables$ [=stringexpression]

Comments The table in the string is separated by commas and access to individual tables can be achieved by using the “GetField$” on page 335.

Example ‘This example creates a message box displaying the message “All Tables:” followed by a list of all available tables.

Msgbox " All Tables: " + MyData.AllTables$

AppActivate statement

Syntax AppActivate string-expression

Comments AppActivate statement is used to activate an application window. String-expression is the name in the title-bar of the application window you want to activate. String-expression must match the name of the window character for character, but comparison is not case-sensitive. If there is more than one window with name matching string-expression, a window is chosen by random.

AppActivate changes the focus to the specified window but does not change whether the window is minimized or maximized. AppActivate can be used together with SendKeys statement to send keys to another application.

Asc

Converts the first character in a string from character code to an ASCII code number.

Syntax Asc(string_expression$)

Parameters string_expression$—The string from which to get the first character.

Returns An integer corresponding to the ASCII code of the first character in the specified numeric expression. The return value is single-precision for an integer, currency or single-precision numeric expression. The return value is double-precision for a long, variant, or double-precision numeric expression.

Comments Generates an error if the parameter is null.

Example MsgBox "The character "%" has the ASCII code:" +Str$(ASC("%"))

Assert

Declares a condition that must be true for continued macro execution.

Syntax Assert condition

Parameters condition—An expression or condition which you want to assert as true.

Comments Triggers an error if the condition is FALSE. An assertion error cannot be trapped by the ON ERROR statement.

The Assert statement is intended to help ensure that a procedure is performing in the expected manner.

Example Assert x>0

Atn

Calculates the arctangent of a numeric expression that indicates a ratio. This can be used only as a function.

Syntax Atn(numeric_expression)

Parameters numeric_expression—A field or variable representing a number.

Returns Returns the angle (in radians) corresponding to the arctangent of the specified numeric expression.

Comments The return value is single-precision for an integer, currency or single-precision numeric expression. The return value is double-precision for a long, variant or double-precision numeric expression.

Example x=Atn(0.33)

Beep

The Beep statement produces a short beeping tone that can be used to alert a user.

Syntax Beep

Comments The Beep statement produces a single short beeping tone through the computer speaker. Users might have this tone mapped to another sound through the use of a sound card and/or sound software.

Example
```
If Value>Limit then
Beep
MsgBox "The value was greater than allowed"
End If
```

Begin Dialog...End Dialog

Starts/ends the dialog box declaration for a user-defined dialog box.

Syntax
```
Begin Dialog dialogName [x, y,] dx, dy
[dialog box definition statements]
End Dialog
```

Parameters [x, y,]—The x and y parameters give the coordinates that position the dialog box. These coordinates designate the position of the upper left corner of the dialog box, relative to the upper left corner of the client area of the parent window. The x parameter is measured in units that are ¼ the average width of the system font. The y parameter is measured in units 1/8 the height of the system font. (For example, to position a dialog box 20 characters in and 15 characters down from the upper left hand corner, enter 80 and 120 as the x, y coordinates.) If these parameters are omitted, the dialog box is centered in the client area of the parent window.

dx, dy—The *dx* and *dy* (*Dx* [Delta-x] and *Dy* [Delta-y] are also sometimes seen and represent the same concept: Parameters ("change in x" and "change in y") specify the width and height of the dialog box (relative to the beginning x and y coordinates). The dx parameter is measured in ¼ system-font character-width units. The dy parameter is measured in 1/8 system-font character-width units (i.e., to create a dialog box 320 characters wide, and 120 characters in height, enter 320 and 120 as the *dx*, *dy* coordinates).

Comments The *Begin Dialog* statement assumes that if only two parameters are given, they are the *dx* (width) and *dy* (height) parameters. Unless the *Begin Dialog* statement is followed by at least one other dialog box definition statement and the *End Dialog* statement, an error results.

The other definition statement must include an *OKButton*, a *CancelButton,* or a *Button* statement. If this statement is left out, there is no way to close the dialog box, and the procedure is unable to continue execution.

This command defines the dialog box, but does not display it. To display the dialog box, you create a dialog record variable with the *Dim* statement, and then display the dialog box using the *Dialog* statement. In the *Dim* statement, *dialogName* is used to identify the dialog definition.

Example Begin Dialog MyDialog 120,130
Caption "This is My Dialog"
End Dialog

Button

Defines a custom push button. (This allows the use of push buttons other than OK and Cancel.) It is used in conjunction with the *ButtonGroup* statement.

Syntax Button x, y, dx, dy, text$

Parameters x, y—The *x* and *y* parameters set the position of the button relative to the upper left corner of the dialog box.

dx, dy—*dx* and *dy* set the width and height of the button. A *dy* value of 14 typically accommodates text in the system font of the dialog box.

text$—Contains a message that is displayed in the push button. If the width of this string is greater than *dx*, trailing characters are truncated.

Comments The *Button* statement can be used only between a *Begin Dialog* and an *End Dialog* statement. A *dy* value of 14 typically accommodates text in the system font.

Example Button 10,20,14,14, "Hello User!"

ButtonGroup

Begins definition of the buttons, when custom buttons are to be used.

Syntax ButtonGroup.field

Parameters .field—A variable that contains the number of the button in the group that was selected. Buttons are numbered by the order they are entered in the dialog definition.

Comments *ButtonGroup* establishes the dialog-record field that contains the user's selection. If *ButtonGroup* is used, it must appear before any *Button* statement which creates a push button. Only one *ButtonGroup* statement is allowed within a dialog box definition.

The *ButtonGroup* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

Example

```
Begin Dialog MyDialog 360,260
ButtonGroup.Pressed
Button 20, 130, 150, 30, "Load Report"
Button 190, 130, 150, 30, "Print Report"
End Dialog

'Execute dialog in this code
Dim ViewDialog as MyDialog
ViewDialog.Pressed
'This will have the value of 1 if Load Report is pressed and 2 if Print
'Report is pressed.
```

Call

Transfers control to a subprogram procedure or application-defined dialog box.

Syntax A Call subprogram_name [(parameterlist)]

Syntax B Subprogram_name parameterlist

Syntax C Call app_dialog (recordName)

Syntax D App_dialog {recordName | dotList}

Parameters The *Call* statement parameter consists of a subprogram and its parameters.

Comments Used to call a subprogram written in BASIC or to call C procedures in a DLL. These C procedures must be described in a *Declare* statement or be implicit in the application.

The parameters to the subprogram must match the parameters as specified in the definition of the subprogram. The parameters can be either variables or expressions. Parameters are passed by reference to procedures written in Basic. If you pass a variable to a procedure which modifies its corresponding formal parameter, and you do not wish to have your variable modified, enclose the variable in parentheses in the *Call* statement. This tells BASIC to pass a copy of the variable. This is less efficient and should not be done unless necessary.

When a variable is passed to a procedure which expects its parameter by reference, the variable must match the exact type of the formal parameter of the function. (This restriction does not apply to expressions.)

Similarly to subprogram invocation, functions associated with application-defined dialog boxes can be invoked using *Call* syntaxes listed as C and D above. In Syntax C, the name inside the parentheses must be a variable previously *Dimmed* as an application-defined dialog record. In Syntax D, the dialog box name can be followed by either a dialog record variable or a comma-separated list of dialog box fields settings, for example:

```
SearchFind .SearchFor="abc", .Forward=1
```

When calling an external DLL procedure, parameters can be passed by value rather than by reference. This is specified either in the *Declare* statement, the *Call* itself, or both, using the *ByVal* keyword. If *ByVal* is specified in the declaration, then the *ByVal* keyword is optional in the call; if present, it must precede the value. If *ByVal* was not specified in the declaration, it is illegal in the call unless the datatype specified in the declaration was *Any*. Specifying *ByVal* causes the parameter's value to be placed on the stack, rather than a far reference to it.

Example

```
Sub Callable (parameter$)
'
End Sub
Sub Macro
Call Callable "My parameter"
```

CancelButton

The *CancelButton* statement determines the position and size of a cancel button.

Syntax CancelButton x, y, dx, dy

Parameters x, y—The *x* and *y* parameters set the position of the Cancel button relative to the upper left corner of the dialog box.

dx, dy—*dx* and *dy* set the width and height of the button. A dy value of 14 can usually accommodate text in the system font.

Comments The *CancelButton* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

If the Cancel button is pushed at run time, the dialog box is removed from the screen and an Error 102 is triggered.

Example CancelButton 10,20,14,14

Caption

Defines the text to be used as the title of a dialog box.

Syntax Caption text$

Parameters text$—Text to appear as your dialog-box caption.

Comments The *Caption* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

If no *Caption* statement is specified for the dialog box, a default caption is used.

Example Caption "Dialog Title"

CCur

Syntax CCur(expression)

Returns The CCur function converts the value of expression to a currency.

Comments CCur accepts any type of expression. Numbers that do not fit in a currency will result in an "Overflow" error. Strings that cannot be converted to a currency will result in a "Type Mismatch" error. Variants containing nulls will result in an "Illegal Use of Null" error.

To convert a value to a different data type, see CDbl, CInt, CLng, CSng, CStr, CVDate and CVar.

CDbl

The *CDbl* function converts an expression to double-precision floating point. This can be used only as a function.

Syntax CDbl (numeric_expression)

Parameters numeric_expression—A field or variable representing a number.

Returns Converts an expression to a double-precision floating point.

Comments *CDbl* accepts any type of expression. Strings that cannot be converted to a double-precision value result in a “Type Mismatch” error. Variants containing nulls result in an “Illegal Use of Null” error.

To convert an expression to a different data type, see CCur, CInt, CLng, CSng, CStr, CVDate and CVar.

Example Cdbl("Table.Numberfield1"+"Table.Numberfield2")

ChDir

Changes the default directory for the specified drive.

Syntax ChDir pathname$

Parameters pathname$—A string expression identifying the new default directory.

Comments The *ChDir* statement changes the default directory for the specified drive. It does not change the default drive. (To change the default drive, use *ChDrive*).

The syntax for *pathname*$ is: [drive:] [\] directory [\directory]. The drive parameter is optional, since this command cannot change the default drive. If omitted, *ChDir* changes the default directory on the current drive.

Example ChDir "c:\rptsmith\macros"

ChDrive

Changes the default drive.

Syntax ChDrive drivename$

Parameters drivename$—Drivename$ is a string expression designating the new default drive. This drive must exist, and must be within the range specified in the CONFIG.SYS file.

Comments If a null parameter (“”) is supplied, the default drive remains the same. If the *drivename$* parameter is a string, *ChDrive* uses the first letter only. If the parameter is omitted, an error message is produced. (To change the current directory on a drive, use *ChDir*.)

Example ChDrive "g:"

CheckBox

The CheckBox statement places a check box within a dialog box.

Syntax CheckBox x, y, dx, dy, text$, .field

Parameters x, y—The *x* and *y* parameters give the coordinates that position the check box. These coordinates designate the position of the upper left corner of the check box, relative to the upper left corner of the dialog box. The x parameter is measured in ¼ system-font character-width units. The y parameter is measured in 1/8 system-font character-height units. (See “Begin Dialog...End Dialog” on page 270.)

dx—The *dx* parameter is the combined width of the check box and the text$ field containing the check box label. Because proportional spacing is used, the width needed will vary with the characters used. To estimate the needed width, multiply the number of characters in the text$ field (including blanks and punctuation) by 4, then add 12 for the checkbox itself.

dy—The *dy* parameter is the height of the text$ field. A *dy* value of 12 is standard and will accommodate typical default fonts. If larger fonts are used, the value should be increased. As the *dy* number grows, the checkbox and the accompanying text moves downward within the dialog box.

text$—The *text$* field contains the label shown to the right of the check box. If the width of this string is greater than *dx,* trailing characters are truncated. If you want to include accelerator characters so that the check box selection can be made from the keyboard, the character must be preceded with an ampersand (&).

.field—The *.field* parameter is the name of the dialog-record field that holds the current check box setting. If its value is 0, the box is unchecked; if its value is –1 the box appears grayed; if its value is 1, the box is checked. The macro language treats any other value of *.field* the same as a 1.

Comments The *CheckBox* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

Example CheckBox 10,10,14,14,"My statement", .check_value-holder

Chr$

Syntax Chr[$](numeric expression)

Returns The Chr$ function returns the one-character string corresponding to an ANSI code.

The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function will return a variant of vartype 8 (string).

Comments Numeric expression must evaluate to an integer between 0 and 255. See Asc.

CInt

Converts the value of expression to an integer by rounding. Used as a function.

Syntax CInt (numeric_expression)

Parameters (numeric_expression)—The parameter given is any numeric expression.

Comments *CInt* accepts any type of expression. After rounding, the resulting number must be within the range of –32767 to 32767, or an error occurs.

Strings that cannot be converted to an integer result in a “Type Mismatch” error. Variants containing nulls result in an “Illegal Use of Null” error. To convert a numeric expression to a different data type, see the “CDbl” on page 274, “CLng” on page 277 and “CSng” on page 292.

Example CInt("table.Decimalfield")

CLng

The *CLng* command converts the value of expression to a long by rounding. Used as a function.

Syntax CLng (numeric_expression)

Parameters (numeric_expression)—The parameter given is any numeric expression.

Returns A round long number.

Comments After rounding, the resulting number must be within the range of: –2,147,483,648 to 2,147,483,647, or an error occurs.

Strings that cannot be converted to a long result in a “Type Mismatch” error. Variants containing nulls result in an “Illegal Use of Null” error.

CLng generates the same result as you would get by assigning the numeric expression to a Long variable. To convert a value to a different data type, see the “CDbl” on page 274, “CInt” on page 276 and “CSng” on page 292.

Example Clng("Table.Numfield"*10000)

Close

Closes a file, concluding input/output to that file.

Syntax Close [[#] filenumber% [, [#] filenumber%...]]

Parameters filenumber%—*filenumber*% is an integer expression identifying the file to close. It is the number used in the *Open* statement for the file. If this parameter is omitted, all open files are closed.

Comments Once a *Close* statement is executed, the association of a file with *filenumber%* is ended, and the file can be reopened with the same or different file number.

When the *Close* statement is used, the final output buffer is written to the operating system buffer for that file. *Close* frees all buffer space associated with the closed file. Use the *Reset* statement so that the operating system flushes its buffers to disk.

Example Close file number 1
close number 1

CloseReport

Closes the active report, but not ReportSmith.

Syntax CloseReport Conditional%

Parameters Conditional%—If the value of this integer parameter is zero, the report closes unconditionally. If *CloseReport* is called using a non-zero value for this parameter, reports that were modified since last being opened prompt the user to save the report before closing it, and allow the user to cancel the Close operation.

Returns Returns 0 if the active report was closed successfully. It returns 1 if there is no active report, or –1 if the user canceled a conditional close.

Comments When you use this command as a *function* (rather than a *statement*), you must enclose its parameter within parentheses.

Example CloseReport 1

CloseRS

Closes ReportSmith.

Syntax CloseRS Conditional%

Parameters Conditional%—If the value of this integer parameter is zero, ReportSmith closes unconditionally. If *CloseRS* is called using a non-zero value for this parameter, ReportSmith prompt the user to save reports before closing them (and various other “house-keeping” tasks), and allows the user to cancel the Close operation.

Returns Returns non-zero if a conditional close was canceled.

Comments When you use this command as a function (rather than a statement), you must enclose its parameters within parentheses.

Example
```
If User_Response$ = 'No'
CloseRS 0
End if
```

ComboBox

The *ComboBox* statement is used to create a combination text box and list box.

Syntax ComboBox x, y, dx, dy, text$, .field

Parameters x,y—The *x* and *y* parameters give the horizontal and vertical (respectively) coordinates that position the upper left corner of the list box, relative to the upper left corner of the dialog box. The *x* parameter is measured in ¼ system-font character-width units. The *y* parameter is

measured in 1/8 system-font character-width units. (See “Begin Dialog...End Dialog” on page 270.)

dx,dy—The *dx* and *dy* parameters specify the width and height of the combo box in which the user enters or selects text.

text$— The *text*$ field specifies the name of the string containing the list variables.

.field—The *.field* parameter is the name of the dialog-record field that holds the text string entered in the text box or chosen from the list box. The string in the text box is recorded in the field designated by the *.field* parameter when the OK button (or any button other than CANCEL) is clicked.

Comments The *ComboBox* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

Example ComboBox 10,10,14,14,"My Statement",.text_entered_holder

Command$

Syntax Command[$]

Returns The Command$ function returns a string containing the command line specified when the MAIN subprogram is invoked.

The dollar sign ($) in the function name is optional. If specified, the return type is ‘string’. If omitted. the function will return a variant of vartype 8 (string).

Comments After the MAIN subprogram returns, further calls to the Command$ function will yield an empty string. This function may not be supported in some implementations of SBL.

Commit

Creates a default report based on the query specified in the dataset object. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Commit

Returns This function returns a 0 on success and a non-zero on error.

Comments When you associate a dataset object with a report, using the *SetFromActive* method, changing one dataset object changes another. The changes to the report appear on the report surface when the report is reloaded or recalculated (with the report level *Recalc* command and not the dataset *Recalc* command).

However, if you create a default report from a dataset control object using the *Commit* method, the changes are not associated with the new report. If you want the changes to be associated with the new report, simply use the *SetFromActive* method after the *Commit* method.

Example MyData.Commit

Connect

Opens a connection to a database server.

Syntax Connect Type, Server$, UserId$, Password$, Database$

Parameters Type—A number that identifies the type of database to which you are connecting:

Native connections

0 - Reserved for named connections
1 - Reserved
2 - dBASE
3 - Excel
4 - Paradox
5 - Ascii
6 - SQL Server
7 - Oracle
8 - DB2
9 - NetSQL
10 - Sybase
11 - Btrieve
12 - Gupta
13 - Ingres
14 - Watcom
15 - Ocelot
16 - Teradata
17 - DB2Gupta
18 - AS400
19 - Unify
20 - dBASE for Windows Query
21 - Delphi
22 - Sybase 10

ODBC connections

40 - dBASE ODBC
41 - Excel ODBC
42 - Paradox ODBC
43 - SQL Server ODBC
44 - Oracle ODBC
45 - DB2 ODBC

46 - NetSQL ODBC
47 - Sybase ODBC
48 - Btrieve ODBC
49 - Gupta ODBC
50 - Ingres ODBC
51 - DB2Gupta ODBC
52 - Teradata ODBC
53 - AS400 ODBC
54 - Watcom ODBC
55 - Generic ODBC - all other ODBC connections not specifically listed (MS Access,etc).
56 - Unify ODBC

BDE Connections

61 - BDE Paradox
62 - BDE dBASE
63 - BDE Ascii
64 - BDE Oracle
65 - BDE Sybase
66 - BDE NovSQL
67 - BDE Interbase
68 - BDE IBMEE
69 - BDE DB2
70 - BDE Informix

Server$—A string identifying the server that will be used to make the connection.

UserId$—A string identifying the user making the connection.

Password$—A string containing the password of the user making the connection.

Database$—A string naming the database you want to connect to, or naming the file name of a local database.

Comments For local databases (such as dBASE), *Server$, UserId, Password$*, and *Database$* should be set to an empty string. If any of these parameters are not valid for your connection type, use a null string.

Example ‘This example connects to an SQL Server database named "mydb."
Connect 6, "sqlsvr","myuser","mypassword","mydb"

Connect (dataset object)

Replaces any previous connection information in a dataset object with the supplied connection information. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object.] Connect Type, Server$, UserId$, Pswrd$, DBase$

Parameters Type—The Type parameter can take the following values

Native connections

0 - Reserved for named connections
1 - Reserved
2 - dBASE
3 - Excel
4 - Paradox
5 - Ascii
6 - SQL Server
7 - Oracle
8 - DB2
9 - NetSQL
10 - Sybase
11 - Btrieve
12 - Gupta
13 - Ingres
14 - Watcom
15 - Ocelot
16 - Teradata
17 - DB2Gupta
18 - AS400
19 - Unify
20 - dBASE for Windows Query
21 - Delphi
22 - Sybase 10

ODBC connections

40 - dBASE ODBC
41 - Excel ODBC
42 - Paradox ODBC
43 - SQL Server ODBC
44 - Oracle ODBC
45 - DB2 ODBC
46 - NetSQL ODBC
47 - Sybase ODBC
48 - Btrieve ODBC
49 - Gupta ODBC
50 - Ingres ODBC
51 - DB2Gupta ODBC
52 - Teradata ODBC
53 - AS400 ODBC
54 - Watcom ODBC
55 - Generic ODBC - all other ODBC connections not specifically listed (MS Access,etc).
56 - Unify ODBC

BDE Connections
61 - BDE Paradox
62 - BDE dBASE
63 - BDE Ascii
64 - BDE Oracle
65 - BDE Sybase
66 - BDE NovSQL
67 - BDE Interbase
68 - BDE IBMEE
69 - BDE DB2
70 - BDE Informix

Server$—Name of the data server or local data file name.

UserId$—Name of the user to make the connection for connections that require a user ID.

Pswrd$—The user password.

dBASE$—The database name for connections that require a database. (Null for Oracle.)

Returns 0 on success. On error a non-zero value and the *Error$* property, containing text that describes the error.

Comments Note that for Oracle and other connections without databases, the *dBase$* parameter should be set to a null string.

Using the *Connect* method clears any previously defined tables or table columns, sorting and grouping information, and so forth.

Example

```
ErrorCode=MyDataSet.Connect(6,"SQLSRVR","John_Doe","PW","")
```

Const

Declares symbolic constants for use in a BASIC program.

Syntax [Global] Const constantName = expression [,constantName=expression]...

Parameters constantName—The name of the constant being defined.

expression—The value to assign to the constant.

Comments BASIC is a strongly typed language. The available data types for constants are numbers and strings.

The type of the constant can be specified by using a type character as a suffix to the *constantName*. If no type character is specified, the type of the *constantName* is derived from the type of the expression.

If Global is specified, the constant is validated at module load time; if the constant has already been added to the run-time global area, the constant's type and value are compared to the previous definition, and the load fails if a mismatch is found. This is useful as a mechanism for detecting version mismatches between modules.

Example Const True=1, False=0, User$="John Doe", Pi=3.1415

ConvertClearCache

Clears the conversion caches defined by the macros ConvertNamedConnInfoCache, ConvertConnInfoCache, ConvertTableInfoCache, and ConvertFieldInfoCache.

Syntax ConvertClearCache

Parameters None

Comments It is extremely important to call this macro after conversion of reports has been completed. If not called, an attempt will be made to apply the conversion information in the conversion caches to each report as it is opened - a very undesirable system action.

Example
```
Sub conversion()
ConvertConnInfoCache
7,"test","myid","mydatabase","",6,"prod","mynewid","mynewdatabase",""
ConvertClearCache
End Sub
```

ConvertConnInfoCache

You use this statement to change the database type used to create a report (for example, a database change from SQL Server to Oracle).

Syntax ConvertConnInfoCache [OldType],[OldServer$],[UserID$],[OldDBName$],[OldOption$],[NewType],[NewServer$],[NewUserID$],[NewDBName$],[NewOptions$]

Parameters OldType - The connection number, which is the internal ReportSmith number. Possible numbers are listed below. Note that there are several ways to connect to some databases. For example, Sybase can be connected through number 10, 22, and 47. Please inspect the list carefully before choosing a connection number.

Native connections

0 - Reserved for named connections
1 - Reserved
2 - dBASE
3 - Excel
4 - Paradox
5 - Ascii

6 - SQL Server
7 - Oracle
8 - DB2
9 - NetSQL
10 - Sybase
11 - Btrieve
12 - Gupta
13 - Ingres
14 - Watcom
15 - Ocelot
16 - Teradata
17 - DB2Gupta
18 - AS400
19 - Unify
20 - dBASE for Windows Query
21 - Delphi
22 - Sybase 10

ODBC connections

40 - dBASE ODBC
41 - Excel ODBC
42 - Paradox ODBC
43 - SQL Server ODBC
44 - Oracle ODBC
45 - DB2 ODBC
46 - NetSQL ODBC
47 - Sybase ODBC
48 - Btrieve ODBC
49 - Gupta ODBC
50 - Ingres ODBC
51 - DB2Gupta ODBC
52 - Teradata ODBC
53 - AS400 ODBC
54 - Watcom ODBC
55 - Generic ODBC - all other ODBC connections not specifically listed (MS Access,etc).
56 - Unify ODBC

BDE Connections

61 - BDE Paradox
62 - BDE dBASE
63 - BDE Ascii
64 - BDE Oracle
65 - BDE Sybase
66 - BDE NovSQL
67 - BDE Interbase
68 - BDE IBMEE
69 - BDE DB2
70 - BDE Informix

OldServer$ - For ODBC connections, this is the old ODBC data source name. For remote connections, where the connection dialog box asks for a server name, use the old server name.

UserId$ - For remote connections, where the connection dialog box asks for a User ID, use the old User ID used to log on.

OldDBName$ - For local ODBC connections, this is the directory name where the tables reside. For remote connections, where the connection dialog box asks for a database name, use the old database name.

OldOptions$ - If the connection dialog has an 'Options' edit field, enter the old information in that field.

NewType - The new connection type. See possible numbers in 'OldType' listed above.

NewServer$ - For ODBC connections, this is the new ODBC data source name. For remote connections, where the connection dialogbox asks for a server name, this is the new server name.

NewUserId$ - For remote connections, where the connection dialog box asks for a User ID, this is the new User ID used to log on.

NewDBName$ - For local ODBC connections, this is the new directory where the tables reside. For remote connections, where the connection dialog box asks for a database name, this is the new database name.

NewOptions$ - If the connection dialog box has an 'Options' edit field, this is the new information for that field.

Comments If any of the above parameters have no value, use a set of double quotes (“”) as a placeholder. This example shows a change from an Oracle database that exists on a server called demo to an SQL server database on a server called prod.

Example

```
Sub conversion()
ConvertConnInfoCache
7,”test”,”myid”,”mydatabase”,””,6,”prod”,”mynewid”,”mynewdatabase”,””
ConvertClearCache
End Sub
```

ConvertDeleteFieldInfoCache

Use this command when you delete a field from the underlying database.

Syntax ConvertDeleteFieldInfoCache [NewType],[NewServer$],[NewUserID$],[NewDatabase$],[NewOwner$],[NewTable$],[OldField$]

Parameters NewType - This is the internal ReportSmith connection number. If converting connection information at the same time as changing field names, this is the new connection type, not the old type. See the discussion in ConvertConnInfoCache above for more detailed information on the valid numbers for this entry.

NewServer$ - For ODBC connections, this is the ODBC data source name. For remote connections, where the connection dialog box asks for a server name, this is the server name. If converting connection information at the same time as changing field names, use the new server name, not the old server name.

NewUserId$ - For remote connections, where the connection dialog asks for a User ID, this is that user ID. If converting connection information at the same time as changing table names, use the new UserID, not the old one.

NewDatabase$ - This is the database name for the table that owned the field as shown in the table selection dialog box, not the connection dialog. If converting table names at the same time as deleting field names, this is the new database name, not the old one.

NewOwner$ - This is the owner name for the table that owned the field as shown in the table selection dialog box, not the connection dialog box. If converting table names at the same time as deleting field names, use the new table name, not the old one.

NewTable$ - This is the table name for the table that owned the field being deleted. If converting table names at the same time as deleting field names, use the new table name, not the old table name.

OldField$ - This is the field name of the field that's been deleted from the database.

Comments If any of the above parameters have no value, use a set of double quotes (“”) as a placeholder.

Example This example shows the deletion of a field called middle_name.

```
sub fieldeletion()
ConvertDeleteFieldInfoCache
6,”demo”,”myid”,”mydatabase”,”sysadm”,”employee”,”middle_name”
ConvertClearCache
End Sub
```

ConvertFieldInfoCache

Use this command if a field name changes in your underlying database.

Syntax ConvertFieldInfoCache [NewType],[NewServer$],[NewUserID$],[NewDatabase$],[NewOwner$],[NewTable$],[OldField$],[NewField$]

Parameters NewType - This is the internal ReportSmith connection number. If converting connection information at the same time as changing field names, this is the new connection type, not the old type. See the discussion in ConvertConnInfoCache above for more detailed information on the valid numbers for this entry.

NewServer$ - For ODBC connections this is the ODBC data source name. For remote connections, where the connection dialog asks for a server name, this is the server name. If converting connection information at the same time as changing field names, this is the new server name, not the old server name.

NewUserId$ - For remote connections, where the connection dialog asks for a User ID, this is that user ID. If converting connection information at the same time as changing table names, this is the new UserID, not the old one.

NewDatabase$ - This is the database name for the table that owns the field as shown in the table selection dialog, not the connection dialog. If converting table names at the same time as converting field names, this is the new database name, not the old one.

NewOwner$ - This is the owner name for the table that owns the field as shown in the table selection dialog, not the connection dialog. If converting table names at the same time as converting field names, this is the new table name, not the old one.

NewTable$ - This is the table name for the table that owns the field being renamed. If converting table names at the same time as converting field names, this is the new table name, not the old table name.

OldField$ - This is the old field name.

NewField$ - This is the new field name.

Comments If any of the above parameters have no value, use a set of double quotes (“”) as a placeholder.

Example This example shows a field name changing from birth_date to bday.

```
sub fieldnamechange()
ConvertFieldInfoCache 6,”demo”,”myid”,”mydatabase”,”sysadm”,
”employee”,”birth_date”,”bday”
ConvertClearCache
End Sub
```

ConvertNamedConnInfoCache

Use this command if the named connection used in a report changes.

Syntax ConvertNamedConnInfoCache [OldNamedConn$],[NewNamed Conn$]

Note: The above statement should be on one line.

Parameters OldNamedConn$ - the name of the old named connection used in the report.

NewNamedConn$ - the name of the new named connection.

Comments When adding a named connection to the cache, specify the old named connection and the new named connection. As reports are opened that use the old named connection, they are converted to use the new named connection. Both named connections must exist in the named connection file (which is RPTSMITH.CON and, by default, resides in the Windows install directory).

Example This example shows a named connection changing from test_connection to production_connection.

```
sub namedconnectionchange()
ConvertNamedConnInfoCache “test_connection”,”production_
connection”
ConvertClearCache
End Sub
```

ConvertTableInfoCache

Use this command if a table name changes in your underlying database.

Syntax ConvertTableInfoCache [NewType],[NewServer$],[NewUserID$],[OldDatabase$],[OldOwner $],[OldTable$],[NewDatabase$],[NewOwner$],[NewTable$]

Parameters NewType - This is the internal ReportSmith connection number. If converting connection information at the same time as changing table names, this is the new connection type, not the old type. See the discussion in ConvertConnInfoCache above for more detailed information on the valid numbers for this entry.

NewServer$ - For ODBC connections, this is the ODBC data source name. For remote connections, where the connection dialog asks for a server name, this is the server name. If converting connection information at the same time as changing table names, use the new server name, not the old server name.

NewUserId$ - For remote connections, where the connection dialog asks for a User ID, this is that user ID. If converting connection information at the same time as changing table names, use the new User ID, not the old one.

OldDatabase$ - Use the old database name for the table as shown in the table selection dialog box, not as shown in the connection dialog.

OldOwner$ - This is the old owner name for the table as shown in the table selection dialog box.

OldTable$ - This is the old table name.

NewDatabase$ - This is the new database name for the table as shown in the table selection dialog box, not as shown in the connection dialog.

NewOwner$ - This is the new owner name for the table as shown in the table selection dialog box.

NewTable$ - This is the new table name for the table.

Comments If any of the above parameters have no value, use a set of double quotes (“”) as a placeholder.

Example This example shows a table name changing from employee to emp.

```
sub tablenamechange()
ConvertTableInfoCache
6,”production_server”,”myid”,”pubs”,”admin”,”employee”,”pubs”,”adm
in”,”emp”
ConvertClearCache
End Sub
```

Cos

Returns the cosine of a value expressed in radians.

Syntax Cos(angle)

Parameters angle—A variable, field, or number representing an angle.

Returns The *Cos* function returns the cosine of an angle expressed in radians. The return value is between –1 and 1. The return value is single-precision if the angle is an integer or single-precision value. The return value is double precision for a long or double-precision value.

Comments The angle is specified in radians, and can be either positive or negative.

Example

```
'Calculate the Cos 450
Pi radians = 1800
The_cos = cos(45x3.1415/180)
'The mathematical equation in parentheses represents the
conversion 'factor.
```

CreateObject

Syntax CreateObject(string expression)

Returns The CreateObject function will create a new Ole2 automation object.

Comments String expression should be the name of the application, a period, and the name of the object to be used. Refer to the documentation provided with your Ole2 server applications for correct application and object names.

```
Dim Ole2 As Object
Set Ole2 = CreateObject("spoly.cpoly")
Ole2.reset
```

CreateReport Method

Syntax CreateReport [Type%], [Style], [Crosstabstyle], [DraftModeRecLimit]

Definition This command allows a DataSet control with a defined query to create one of four different report types: columnar, crosstab, form, or label. It also takes style information and draft date.

Parameters The Type% parameter is optional. If this argument is omitted, then a columnar report will be created. If the type parameter is specified, one of the following reports will be created.

The Style parameter specifies the report style to be used on a columnar report. You can use either system (supplied by ReportSmith) or custom report styles, but this parameter entry must exactly match the style name, including upper- and lower-case characters.

The Crosstabstyle parameter is used only for crosstab reports, and functions in a manner similar to that of the Style parameter.

DraftModeRecLimit is an integer value that represents the number of records you want ReportSmith to display when you are using draft mode.

Report	Creates
0	A columnar report as the old commit function did.
1	A label report (brings up the insert field dialog box).
2	A crosstab report (brings up the crosstab dialog box).
3	A form report (uses the default form layout).

Returns This function returns 0 if a macro is found and successfully executed.

CrosstabStyle$ Property

Syntax [object].CrosstabStyle$

Definition A string that holds the last crosstab style name selected. This property is read only.

Returns Returns the last selected crosstab style name selected in the New Report Style dialog box.

Example
```
Sub GetTabStyle()
dim MyDialog as newReportDialog
MyDialog.rundialog
MsgBox Str(MyDialog.ReportType)+ Chr$(13) +
MyDialog.CrosstabStyle$
End Sub
```

CSng

The *CSng* function converts the value of expression to a single-precision floating point.

Syntax CSng (numeric_expression)

Parameters numeric_expression—The parameter given is any numeric expression.

Returns A numeric expression.

Comments Accepts any type of expression. The numeric expression must have a value within the range allowed for the Single data type or an error occurs.

Strings that cannot be converted to an integer result in a “Type Mismatch” error. Variants containing nulls result in an “Illegal Use of Null” error.

To convert a numeric expression to a different data type, see the “CDbl” on page 274, “CInt” on page 276, and “CLng” on page 277.

Example S=Csng(1/3)

CurDir$

This function determines the current directory of the specified drive.

Syntax CurDir$[(drivename$)]

Parameters [(drivename$)]—*drivename*$ is a string expression identifying which drive is to return the default directory. This drive must exist, and must be within the range specified in the CONFIG.SYS file. If a null parameter (“”) is supplied, or if no drivename is indicated, the path for the default drive is returned.

Returns The path (including the drive letter) that is the current default directory for the specified drive. The dollar sign ($) in the function name is optional. If specified, the return type is ‘string’. If omitted, the function returns a variant of vartype 8 (string).

Comments To change the current drive, use *ChDrive*; to change the current directory, use *ChDir* command from the list box.

Example
```
'Get the current directory of Drive C
CurrentDir$=CurDir$("C")
```

Current

Returns the record number to which the data set (belonging to the active report) is pointing. That is, this function tells you what record number the *Field$* function returns when executed.

Syntax Current

Returns This function returns the record number to which the data set of the currently active report is pointing.

Example
```
If Current = 1 then
MsgBox "At the beginning"
EndIf
```

CurrentPage

Returns the number of the page currently being displayed in the active report. Used only as a function.

Syntax CurrentPage

Returns Returns the displayed page number.

Comments This function is useful in changing the active report for functions that work on the currently active report.

Example
```
'This example creates a message box that tells the message what page the 'active report is displaying.

MsgBox "The active report is on page: " + str$(CurrentPage)
```

CVar

Syntax CVar(expression)

Returns The CVar function converts the value of expression to a variant.

Comments CVar accepts any type of expression.

CVar generates the same result as you would get by assigning the expression to a Variant variable. To convert a value to a different data type, see CCur, CDbl, CInt, CLng and CSng.

CVDate function

Syntax CVDate(expression)

Returns The CVar function converts the value of expression to a variant date.

Comments The argument given is any expression. It accepts both string and numeric values.

The CVDate function returns a variant of vartype 7 (date) that represents a date from January 1, 100 through December 31, 9999. A value of 2 represents January 1, 1900. Times are represented as fractional days.

To convert a value to a different data type, see CCur, CDbl, CInt, CLng, CSng, or CStr. To convert a value to a different variant type, see CVar.

DataBase$

Returns the current database for the current connection. This command represents a property—an object variable—of the dataset object, which, in turn, represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].DataBase$

Comments Before a current database can be retrieved from a dataset object, a connection that has databases must be made.

Example CurrentDataBase$=MyData.DataBase$

Date$

Retrieves the system date as a string.

Syntax Date$

Returns A string representing the current date. The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function returns a variant of vartype 8 (string).

Comments The *Date$* function returns a ten character string.

Example Today$=Date$

DateSerial

Syntax DateSerial(year%, month%, day%)

Returns The DateSerial function returns a date value for year, month, and day specified.

Comments The DateSerial function returns a variant of vartype 7 (date) that represents a date from January 1, 1900 through December 31, 9999, where January 1, 1900 is 2.

The range of numbers for each DateSerial argument should conform to the accepted range of values for that unit. You also can specify relative dates for each argument by using a numeric expression representing the number of days, months, or years before or after a certain date.

DateValue

Syntax DateValue(string expression$)

Returns The DateValue function returns a date value for the string specified.

Comments The DateValue function returns a variant of vartype 7 (date) that represents a date from January 1, 100 through December 31, 9999, where January 1, 1900 is 2.

DateValue accepts several different string representations for a date. It makes use of the operating system's international settings for resolving purely numeric dates.

Day

Syntax Day(expression)

Returns The Day function returns the day of the month component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments The Day function returns an integer between 1 and 31, inclusive.

It accepts any type of *expression*, including strings, and attempts to convert the input value to a date value.

Declare

The *Declare* statement has two uses—forward declaration of a procedure whose definition is to be found later in this module, and declaration of a procedure which is to be found in an external Windows DLL or external BASIC module. The *Declare* statement should always precede the macro itself.

Syntax A Declare Sub name [libSpecification] [(parameter [As type])]

Syntax B Declare Function name [libSpecification] [(parameter [As type])]

Parameters The parameters are specified as a comma-separated list of parameter names. The data type of a parameter can be specified by using a type character or by using the *As* clause. Record parameters are declared by using an *As* clause and a type which has previously been defined using the *Type* statement.

A forward declaration is needed only when a procedure in the current module is referenced before it is used. In this case, the BasicLib, Lib and Alias clauses are not used.

Array parameters are indicated by using empty parentheses after the parameter. Array dimensions are not specified in the *Declare* statement.

External DLL procedures are called with the PASCAL calling convention (the actual parameters are pushed on the stack from left to right). By default, the actual parameters are passed by far reference. For external DLL procedures, there are two additional keywords, *ByVal* and *Any*, that can be used in the parameter list.

When *ByVal* is used, it must be specified before the parameter it modifies. When applied to numeric data types, *ByVal* indicates that the parameter is passed by value, not by reference. When applied to string parameters, *ByVal* indicates that the string is passed by far pointer to the string data. By default, strings are passed by far pointer to a string descriptor.

Any can be used as a type specification, and permits a call to the procedure to pass a value of any datatype. When *Any* is used, type checking on the actual parameter used in calls to the procedure is disabled (although other parameters not declared as type *Any* are fully type-safe). The actual parameter is passed by far reference, unless *ByVal* is specified, in which case the actual value is placed on the stack (or a pointer to the string in the case of string data). *ByVal* can also be used in the call. It is the external DLL procedure's responsibility to determine the type and size of the passed-in value.

Returns A Sub procedure does not return a value. Function returns a value, and can be used in an expression. Function names must end with a type character. This specifies the return value of the function. The name parameter names the Sub or Function being declared.

Comments If the libSpecification uses the format of BasicLib libName, the procedure is to be found in another BASIC module named libName. In this case, the other module is loaded on demand whenever the procedure is called. The macro language will not automatically unload modules that are loaded in this fashion. The macro language detects errors of mis-declaration with very high (but not perfect) reliability.

If the libSpecification uses the format of Lib libName [Alias ordinal], the procedure is to be found in a Dynamic Link Library (DLL) named libName. The ordinal parameter specifies the ordinal number of the procedure within the external DLL. If the ordinal is not specified, the DLL function is accessed by name, which can cause the module to load more slowly. It is recommended that the ordinal be used whenever possible.

ReportBasic supports two different behaviors when an empty string ("") is passed by value to an external procedure. The implementor of the macro language can specify which behavior by using the macro language API function *SetInstanceFlags*. In any specific implementation which uses the macro language, one of these two behaviors should be used consistently. We recommend the second behavior, which is compatible with Microsoft's VB Language. The following two paragraphs describe the two possible behaviors.

When an empty string ("") is passed b y value to an external procedure, the external procedure receives a NULL pointer. If you wish to send a valid pointer to an empty string, use Chr$(0).

When an empty string ("") is passed by value to an external procedure, the external procedure receives a valid (non-NULL) pointer to a character of 0. To send a NULL pointer, *Declare* the procedure parameter as ByVal As Any, and call the procedure with a parameter of 0&.

Example

```
Declare Function ShowWindow Lib
"User" (ByVal hwnd as Integer,
ByVal nCmdShow as Integer)as Integer
```

Deftype

Specifies the default data type of a variable specified in varTypeLetters.

Syntax DefInt varTypeLetters
DefLng varTypeLetters
DefSng varTypeLetters
DefDbl varTypeLetters
DefStr varTypeLetters

Parameters varTypeLetters—The name of the variable you want to use to define the data type.

Comments The *varTypeLetters* are specified as a comma-separated list of letters. A range of letters can also be specified. For example, a–d indicates the letters a, b, c, and d.

The case of the letters is not important, even in a letter range. The letter range a–z is treated as a special case. It denotes all alpha characters, including the international characters.

The *Deftype* statement only affects the module in which it is specified. It must precede any variable definition within the module.

Variables defined using *Global* or *Dim* can override the *Deftype* statement by using an *As* clause or a type character.

Example
```
DefInt MyInt1, Another Int, LastInt
DefStr Name, Caption
```

DerivedField

Sets the value of a Derived Field.

Syntax DerivedField Value$

Parameters Value$—A quoted value or string variable.

Comments The *DerivedField* command is only valid in macros you use to derive fields. This command takes a string (Value$) that is used to name the derived field. If the value for the derived field is a number, convert it to a string using the *STR$* function.

Example
```
'This command creates a derived field named "Bill Smith". It does not
'specify the source of the data contained in the field.
DerivedField "Bill Smith"
```

Dialog

Displays a dialog box. Use the *Begin Dialog...End Dialog* command pair, with intervening dialog definition statements, to define a dialog box and populate it with controls. Seond, use the Dim statement to create and name the dialog-box variable, and finally, use the Dialog statement to display the named dialog-box variable. The *Dialog* command does not define or create the dialog box; it merely displays it.

Syntax Dialog DialogName$

Parameters DialogName$—The name of a user-defined dialog box created with the Dim statement.

Comments The data for the controls of the dialog box comes from the dialog box record *recordName*.

The dialog box *recordName* must have been declared using the *Dim* statement. If the user exits the dialog box by choosing the Cancel button, a run-time error is triggered, which can be trapped using *On Error.*

The *Dialog* statement does not return until the dialog box is closed.

Example
```
Dim The_Diag as UserDiag
Dialog The_Diag
```

Dim

Declares variables for use in a BASIC program.

Syntax Dim [Shared] variableName [As type] [,variableName [As type]]...

Comments BASIC is a strongly typed language. The available data types are: numbers, strings, records, arrays, dialog boxes and Application Data Types (ADTs).

If the *As* clause is not used, the type of the variable can be specified by using a type character as a suffix to the *variableName* parameter. The two different type-specification methods can be intermixed in a single *Dim* statement (although not on the same variable).

Names

Variable names must begin with a letter and contain only letters, numbers and underscores. Variable names can also be delimited by brackets, and any character can be used inside the brackets except other brackets.

```
Dim my_1st_variable As String
Dim [one long and strange! variable name] As String
```

Numbers

Numeric variables can be declared using the *As* clause and one of the following numeric types: Currency, Integer, Long, Single, Double. Numeric variables can also be declared by including a type character as a suffix to the name.

Strings

BASIC supports two types of strings, fixed-length and dynamic. Fixed-length strings are declared with a specific length (between 1 and 32767) and cannot be changed later. Use the following syntax to declare a fixed-length string:

```
Dim variableName As String* length
```

Dynamic strings have no declared length, and can vary in length from 0 to 32767. The initial length for a dynamic string is 0. Use the following syntax to declare a dynamic string:

```
Dim variableName$
```

—or—

```
Dim variableName As String
```

Records

Record variables are declared by using an *As* clause and a *typeName* which has previously been defined using the *Type* statement. The syntax to use is:

```
Dim variableName As typeName
```

Records are made up of a collection of data elements called fields. These fields can be of any numeric, string, variant, or previously-defined record type. See *Type* for details on accessing fields within a record.

You can also use the *Dim* statement to declare a dialog record. In this case type is specified as:

```
[Dialog] dialogName
```

where *dialogName* matches a dialog box name previously defined using *Begin Dialog...End Dialog.* The dialog record variable can then be used in a *Dialog* statement.

Dialog records exhibit the same behavior as other records—they differ only in the way they are defined. Some applications may provide a number of pre-defined dialog boxes.

Objects

Object variables are declared by using an *As* clause and a *typeName* of a class. Object variables can be set to refer to an object, and then used to access members and methods of the object using “dot notation.”

```
Dim Ole2 As Object
Set Ole2 = CreateObject("spoly.cpoly")
Ole2.reset
```

An object may be declared as New for some classes. In such instances, the object variable does not need to be Set; a new object is allocated when the variable is used. Note that *the class object does not support the New operator.*

```
Dim variableName As New className
variableName.methodName
```

Arrays

The available data types for arrays are: numbers, strings, variants, objects and records. Arrays of arrays, dialog box records, and ADTs are not supported.

Array variables are declared by including a subscript list as part of the *variableName*. The syntax to use for *variableName* is:

```
Dim variable([subscriptRange,...]) As typeName
```

—or—

```
Dim variable_with_suffix( [ subscriptRange, ... ] )
```

where *subscriptRange* is of the format:

```
[startSubscript To] endSubscript
```

If *startSubscript* is not specified, 0 is used as the default. The *Option Base* statement can be used to change the default.

Both the *startSubscript* and the *endSubscript* are valid subscripts for the array. The maximum number of subscripts which can be specified in an array definition is 60. The maximum total size for an array is limited only by the amount of memory available.

If no *subscriptRange* is specified for an array, the array is declared as a dynamic array. In this case, the *ReDim* statement must be used to specify the dimensions of the array before the array can be used. A variable declared inside of a procedure has scope local to that procedure. A variable declared outside of a procedure has scope local to the module. It is permissible for a procedure to declare a variable with a name that matches a module variable. When this happens, the module variable is not accessible by the procedure.

Variables can be shared across modules. See the Global statement for details.

The *Shared* keyword is included for backward compatibility with older versions of BASIC. It is not allowed in *Dim* statements inside of a procedure. It has no effect. BASIC allows a variable to be automatically declared, without the use of a *Dim* statement. If a variable is first used with a type character as a suffix to its name, the variable is automatically declared to be a local variable of the specified type. If no type character is specified, the variable is automatically declared to be a local variable of type *Variant*. It is considered good programming practice to declare all variables, and not make use of this feature. To force all variables to be explicitly declared use the *Option Explicit* statement. It is also recommended that you place all procedure-level *Dim* statements at the beginning of the procedure. Regardless of what mechanism you use to declare a variable, you can choose to use or omit the type character when referring to the variable in the rest of your program. The type suffix is not considered part of the variable name.

Example Dim My_var as Array

Dir$

Returns a file that matches the given directory and/or wild card. Can be used only as a function.

Syntax Dir$ [(filespec$)]

Parameters filespec$—A string expression identifying a path or filename. This parameter can also include a drive specification. It may also include “wildcard” characters (?) and (*).

Attrib—An integer expression specifying the file names that need to be added to the list. The default value for attrib% is 0.

Returns The (‘) function returns a filename that matches the specified pattern. The dollar sign ($) in the function’s name is optional. If specified, the return type is string. If omitted, the function returns a variant of vartype 8 (string).

Comments This parameter can include a drive specification. It can also include the “wildcard” characters ’?’ and ’*’. *Dir$* returns the first filename that matches the *filespec$* parameter. To retrieve additional filenames that match the *filespec$*, call the *Dir$* function again, omitting the *filespec$* parameter. If no file is found, an empty string (“”) is returned. For *Attrib%*, *Dir$* returns only files without directory, hidden, system, or volume label attributes set.

The possible values for attrib% are:

0 — Return normal files
2 — Add hidden files
4 — Add system files
8 — Return volume label
16 — Add directories

Example

```
'Count .rpt files in "c:s"
Count=1
A$=Dir$("C:\*.rpt")
While Dir$<>""
Count=Count+1
Wend
```

Disconnect

Allows you to remove a connection that was previously set with the Connect method. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Disconnect

Returns Error if an active report is using the connection. The return code for this command is 0 (zero) on success. If it is not zero, then the *Error$*property contains text that describes the error.

Comments This function is executed successfully only if there are no other dataset objects or reports using the same connection.

This command returns an error if any other report or active dataset object is using the connection that this object is trying to disconnect.

Example The following example shows you how to:

Create a DataSet control
Add a table
Create a report
Print a report
Close a report
Disconnect the connection

In the following example, assuming use of a server called X:ORASRV, and a user called SCOTT with a password of TIGER.

```
Sub MakeAReport()
Dim NewData as DataSet
NewData.Connect 7, "X:ORASRV","SCOTT","TIGER"," "
```

```
'Add a Table (DataBase is NULL for Oracle)
NewData.AddTable "SCOTT.DEPARTMENT"," "
'Create the Report Object
NewData.Commit
PrintReport 0,0,"","",""
CloseReport 0
NewData.Disconnect
End Sub
```

DoEvents

Allows other Windows applications to process messages.

Syntax DoEvents

Comments Use this command when you want your BASIC code to yield processor time, to allow other applications to process messages.

Do...While

Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax A

```
Do [{While | Until} condition]
[statementblock]
[Exit Do]
[statementblock]
Loop
```

Syntax B

```
Do
[statementblock]
[Exit Do]
[statementblock]
Loop [{While | Until} condition]
```

Parameters condition—*Condition* is any expression that BASIC can determine to be true (non zero) or false (0).

Comments BASIC repeats the program lines contained in the statement block(s) as long as a *While* condition is true or until an *Until* condition is false.

When an *Exit Do* statement is executed, control is transferred to the statement that follows the loop. When used within a nested loop, an *Exit Do* statement moves control out of the immediately enclosing loop.

Example

```
'search to the end of the R&D Group
do GetNext
Loop While Field$("Dept") = "R&D"
```

DraftMode Property

Syntax [object].DraftMode

Definition A flag. If it is non-zero, it indicates that the user checked the Draft Mode check-box.

Returns Returns a non-zero value if the Draft Mode button was checked.

Example

```
Sub IsDraftMode()
dim MyDialog as newReportDialog
xx.rundialog
MsgBox Str(MyDialog.ReportType)+ Chr$(13) +
MyDialog.DraftMode
End Sub
```

EnableIcon

Enables and disables icons and combo boxes on the Toolbar and Ribbon.

Syntax EnableIcon GroupNo, ItemNo, EnableFlag

Parameters GroupNo—Index of the icon group to which the icon or combo box belongs. (See illustration in example.)

ItemNo—Index of the item within the group. (See Table 11.4.)

EnableFlag—0 is to disable, 1 is to enable.

Comments The following table shows toolbar and ribbon groups.

Table 11.4: Toolbar/Ribbon groupings

Toolbar/Ribbon Item	Group Number	Index Number
New File	1	1
Open File		2
Save File		3
Print	2	1
View entire page	3	1
View at 100%		2
View page width		3
Column mode	4	1
Form mode		2
Group header	5	1
Group footer		2
Ascending sort	6	1
Descending sort		2

Table 11.4: Toolbar/Ribbon groupings (continued)

Toolbar/Ribbon Item	Group Number	Index Number
Sum column	7	1
Absolute value		2
Minimum value		3
Maximum value		4
Item count		5
Merge reports	8	1
Edit SQL		2
Best fit for columns		3
Font	12	1
Point size	13	1
Bold	14	1
Italic		2
Underline		3
Left align	15	1
Center align		2
Right align		3
Currency format	16	1
Numeric format		2
Percentage format		3
Insert text	17	1
Insert picture		2
Insert graph		3
Insert crosstab		4
Extract style	18	1
Apply report style		2

Example The command below disables the Italic button so that it is not available:

```
EnableIcon 14,2,0
```

EnableMenu

Enables or disables a menu command.

Syntax EnableMenu Menu$, EnableCode%

Parameters Menu$—The menu name and subname (separated by a vertical bar) you want to enable or disable.

EnableCode%—Specify 1 to enable or 0 (zero) to disable.

Returns When used as a function, returns zero if a menu was removed successfully and –1 if a menu of the given name was not found.

Comments This command uses a string that specifies a menu item or a submenu item. The string uses this format:

```
"MenuName|SubMenuName"
```

The names must match ReportSmith menu commands, not including keyboard accelerators and ellipsis (...) characters. If you omit the pipe and submenu name, the routine assumes you're working with a top-level menu. If a top-level menu is disabled, all of its submenu items are also disabled.

When you use this command as a function (rather than a statement) you must enclose its parameters within parentheses. For more information on the differences between functions and statements, refer to "Using the DataSet Control" on page 215.

Example The following line of code disables the ReportSmith File|New menu item, and assigns the resulting value to a variable called "Success."

```
Success = EnableMenu("File|New", 0)
```

EnableRMenu

Enables or disables pop-up menus (activated by right-clicking the mouse) by object type.

Syntax EnableRMenu ObjectType, EnableCode%

Parameters ObjectType—Specifies object type for which to enable/disable pop-up menus.

1 Text Fields
2 Sections
3 Draw Windows
4 Crosstabs\
5 Crosstab Cells
6 General
7 Reserved

EnableCode%

0 Zero disables the menu; non-zero enables the menu

Comments By placing an exclamation point (!) before the menu name, this command can be used to check the default menu state for new reports. This can be done whether the menu item is specified by command or relative location.

Example The following example will disable pop-up menus for text fields.

```
EnableRMenu 1,0
```

Environ$

Retrieves strings from the operating system's environment table.

Syntax A Environ$(environment-string$)

Syntax B Environ$(n%)

Parameters environment-string$—The name of a keyword in the operating system environment. If this parameter is given, it must be entered in uppercase, or it returns a null string. The value associated with the keyword is returned.

n%—One of the strings from the operating system environment. This can be any numeric expression, but it is rounded to a whole number by *Environ$*. If this parameter is used, *Environ$* returns the nth string from the environment table. This string uses the form "keyword = value."

Returns A string from the operating system's environment table.

Comments The parameter of the *Environ$* function can be either a string (environment-string$) or an integer (n%). A null string is returned if the specified parameter cannot be found.

Example

```
'Get the Users Path
MyPath = Environ$("Path")
```

Eof

Indicates whether the end of a file has been reached.

Syntax Eof (filenumber%)

Parameters filenumber%—The number used in the *Open* statement of the file.

Returns A value indicating whether the end of a file has been reached.

Comments The *Eof* Function returns a (–1) if the end-of-file condition is true for the specified file.

Example The following fills a string-array from a file. (!= means not equal.)

```
While Eof(2)!=–1
Input#2, A$(x)
x=x+1
Wend
```

Erase

Syntax Erase Array [, Array]

Comments The Erase statement reinitializes the contents of a fixed array and frees the storage associated with a dynamic array. The effect of using Erase on the elements of a fixed array varies with the type of element:

Element Type	Erase Effect
Numeric	Each element set to zero.
Variable length string	Each element set to zero length string.
Fixed length string	Each element's string is filled with zeros.
Variant	Each element set to Empty.
User defined type	Members of each element are cleared as if the members were array elements, i.e. numeric members have their value set to zero, etc.
Object	Each element is set to the special value Nothing.

Erl

Gets the line number of the last trapped error. Can be used only as a function.

Syntax Erl

Returns The line number where an error was trapped.

Comments Using the *Resume* or *On Error* statements resets the *Erl* value to 0. If you want to maintain the value of the line number returned by *Erl*, assign it to a variable.

The value of the *Erl* function can be set indirectly through the *Error* statement.

Example MsgBox "Error on Line:" + str$(Erl)

Err (function)

Determines the last run time error code. It can be used only as a function.

Syntax Err

Returns The run-time error code for the last error that was trapped.

Comments Using the *Resume* or *On Error* statements resets the *Err* value to 0. If you want to maintain the value of the error code returned by *Err*, you should assign it to a variable.

The value of the *Err* function can be set directly through the *Err* statement and indirectly through the *Error* statement. See “Trappable errors” on page 259.

Example MsgBox "Error #" + Str$(Err)

Err (statement)

Sends error information between procedures.

Syntax Err = n%

Parameters n% —An integer that contains the code of the error.

Comments The parameter *n%* must be a 0 (indicating that no run-time error has been trapped) or an integer expression indicating a run-time error code (having a value between 1 and 32,767).

This statement can be used with *Error$* to generate application specific errors.

Example
```
'Set to user error number 276
Err=276
```

Error

Error errorcode% simulates the occurrence of a macro language-defined or user-defined error.

Syntax Error errorcode%

Parameters errorcode%—Represents the error code, must be an integer between 1 and 32,767.

Comments If an *errorcode%* is one which the macro language already uses, the *Error* statement simulates an occurrence of that error.

User-defined error codes should employ values greater than those used for standard macro language error codes. To help ensure that non-macro language error codes are chosen, user-defined codes should work down from 32,767.

If an *Error* statement is executed and there is no error-handling routine enabled, the macro language produces an error message and halts program execution. If an *Error* statement specified an error code not used by the macro language, the message “User-defined error” is displayed.

Example Generate an error using a code of 7.
```
Error 7
```

Error$

Returns an error message for the given error code. Can be used only as a function.

Syntax Error$ [(errorcode%)]

Parameters errorcode%—A number from 1–32,767

Returns The error message that corresponds to the specified error code.

Comments If the *errorcode%* is omitted, BASIC returns the error message for the run-time error which has occurred most recently.

If no error message is found to match the *errorcode*, null string(“”) is returned. See “Trappable errors” on page 259.

Example MsgBox "Error #75 is" + Error$(75)

Error$ (dataset object and report object)

Contains a string that describes the error that occurred in the last dataset control method executed. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object.]Error$ [=stringexpression]

Example This example first sets the value of a variable called X, by assigning it the return value of an *AddTable* command. If X is a non-zero value, meaning that an error has occurred, a message box appears displaying the current value of the *Error$* property of the DS dataset object.

```
X=DS.AddTable("Invalid.Table","Bogus")
If x<>0 then msgbox DS.ERROR$
End If
```

ExecSQL

Executes stored procedures and other SQL strings that do not return a result set.

Syntax ExecSQL SQL$, Type, Server$, UserId $, Password$, Database$

Parameters SQL$—The SQL statement to be executed.

Type—A number identifying the type of database to which you are connecting:

Native connections

0 - Reserved for named connections
1 - Reserved
2 - dBASE
3 - Excel
4 - Paradox
5 - Ascii
6 - SQL Server
7 - Oracle
8 - DB2
9 - NetSQL
10 - Sybase
11 - Btrieve
12 - Gupta
13 - Ingres
14 - Watcom
15 - Ocelot
16 - Teradata
17 - DB2Gupta
18 - AS400
19 - Unify
20 - dBASE for Windows Query
21 - Delphi
22 - Sybase 10

ODBC connections

40 - dBASE ODBC
41 - Excel ODBC
42 - Paradox ODBC
43 - SQL Server ODBC
44 - Oracle ODBC
45 - DB2 ODBC
46 - NetSQL ODBC
47 - Sybase ODBC
48 - Btrieve ODBC
49 - Gupta ODBC
50 - Ingres ODBC
51 - DB2Gupta ODBC
52 - Teradata ODBC
53 - AS400 ODBC
54 - Watcom ODBC
55 - Generic ODBC - all other ODBC connections not specifically listed (MS Access,etc).
56 - Unify ODBC

BDE Connections

61 - BDE Paradox
62 - BDE dBASE
63 - BDE Ascii

64 - BDE Oracle
65 - BDE Sybase
66 - BDE NovSQL
67 - BDE Interbase
68 - BDE IBMEE
69 - BDE DB2
70 - BDE Informix

Server$—Identifies the server used in making the connection.

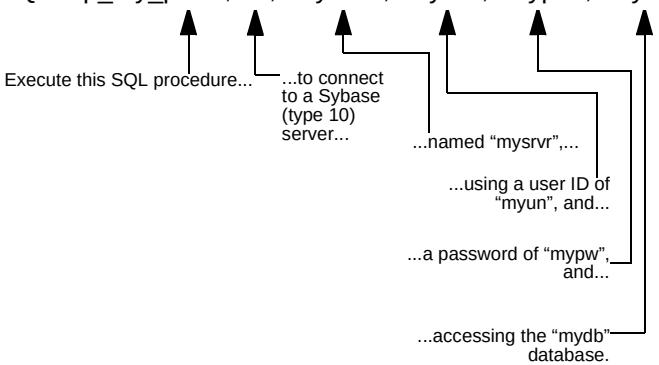
UserID$—Identifies the user making the connection.

Password$—Identifies the user's password.

Database$—The name of the database to connect to, or the file name of a local database.

Comments For local databases (such as dBASE), Server$, UserID$, and Password$ should each be set to an empty string.

Example ExecSQL "sp_my_proc", 10, "mysrvr", "myun", "mypw", "mydb"



ExecuteMenu

Simulates a user clicking one of the ReportSmith menu items.

Syntax ExecuteMenu Menu$

Parameters Menu$—The menu name, or menu/submenu combination, that you want to execute.

Returns 0 (zero) if a menu was executed successfully, and –1 if a menu of the given name wasn't found.

Comments This command takes a string that specifies a menu item or a submenu item. The string uses this format:

"MenuName|SubMenuName"

The names must match ReportSmith menu commands, not including keyboard accelerators and ellipsis (...) characters. If you omit the pipe

(vertical bar character) and submenu name, the routine assumes you're working with a top level menu.

By placing an exclamation point (!) before the menu name, this command can be used to check the default menu state for new reports. This can be done whether the menu item is specified by command or relative location.

Example 'The following code executes the File|Exit command in ReportSmith, 'thereby closing the application and prompting for unsaved changes.
Execute Menu "File|Exit"

Exit

Allows the program flow to escape from a loop function or subroutine.

Syntax Exit {Do | For | Function | Sub}

Parameters Do | For—When used with either of these parameters, Exit terminates loop statements.

Function | Sub—When used with either of these parameters, Exit transfers control from the current procedure back to the original calling procedure.

Comments *Exit Do* can be used only within a *Do...Loop* statement. *Exit For* can be used only within a *For...Next* statement. In both cases, control is transferred to the statement which follows the loop statement. When used within a nested loop, an *Exit* statement moves control out of the immediately enclosing loop.

The *Exit Function* and *Exit Sub* statements transfer control from the current procedure back to the original calling procedure. *Exit Function* must be used in a function procedure. *Exit Sub* can be used only to exit from a Sub procedure.

Example
```
'Wait until 11am
Do
If time$="11:00" then Exit Do
Loop
```

Exp

Returns the value *e* raised to the numeric-expression power. Used only as a function.

Syntax Exp(numeric expression)

Parameters numeric-expression—The numeric expression is the value to which you want to raise the exponent.

Returns The value of (*e* raised to the numeric-expression power).

Comments The return value is single-precision for an integer or single-precision numeric expression. It is double precision for a long or double-precision numeric expression.

Example The following example finds the value of $5^{2.333}$.

```
value = 5 Exp(2.333)
```

ExportTable

This command is a method of the dataset object that represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

The *ExportTable* command enables you to use 32-bit ODBC 2.0 drivers (dBASE, Paradox, Excel, Oracle, or any other supported ODBC driver type) to create tables

Syntax [Object].ExportTable
[TblPath],[Type],[DataSource],[UserId],[Password], [Database]

Parameters TblPath—Path to the new table, for PC-based tables, or fully qualified table name for server-based tables.

Type—Integer value representing the type of table to be exported:

Native connections

0 - Reserved for named connections
1 - Reserved
2 - dBASE
3 - Excel
4 - Paradox
5 - Ascii
6 - SQL Server
7 - Oracle
8 - DB2
9 - NetSQL
10 - Sybase
11 - Btrieve
12 - Gupta
13 - Ingres
14 - Watcom
15 - Ocelot
16 - Teradata
17 - DB2Gupta
18 - AS400
19 - Unify
20 - dBASE for Windows Query
21 - Delphi
22 - Sybase 10

ODBC connections

40 - dBASE ODBC
41 - Excel ODBC
42 - Paradox ODBC
43 - SQL Server ODBC
44 - Oracle ODBC
45 - DB2 ODBC
46 - NetSQL ODBC
47 - Sybase ODBC
48 - Btrieve ODBC
49 - Gupta ODBC
50 - Ingres ODBC
51 - DB2Gupta ODBC
52 - Teradata ODBC
53 - AS400 ODBC
54 - Watcom ODBC
55 - Generic ODBC - all other ODBC connections not specifically listed (MS Access,etc).
56 - Unify ODBC

BDE Connections

61 - BDE Paradox
62 - BDE dBASE
63 - BDE Ascii
64 - BDE Oracle
65 - BDE Sybase
66 - BDE NovSQL
67 - BDE Interbase
68 - BDE IBMEE
69 - BDE DB2
70 - BDE Informix

DataSource—This parameter comes into play only when “55” is used as the value of the *Type* parameter. For example, if you specify “2” as the value of the *Type* parameter, ReportSmith will use the first dBASE ODBC driver it encounters, regardless of the number of such drivers you may have installed. By specifying “55” as the value of the *Type* parameter, then specifying the exact (including matching upper- and lowercase) name of the driver you want to use, you can force ReportSmith to use only the intended ODBC driver. (If you receive error message #9025 while exporting a table to Type 55, it usually means that you have misspelled or mismatched case on the *DataSource* parameter.)

UserId, Password—Used only for server-based databases (use null strings for PC-based tables). *UserId* represents your user identification, while *Password* represents your user password.

DataBase—Specifies the database for server-based databases. Because this is specified in the TblName parameter for those connections requiring it, you can usually specify a null string for this parameter's value.

Returns 0 (zero) on success, or –1 on failure.

Comments The *DataSource* parameter is case-sensitive, so you must exactly match both the spelling and the case of the ODBC driver you intend to use.

Examples In the examples that follow, each *ExportTable* command line should be written on a single line.

```
Sub TheExporter()
Dim ds As DataSet
ds.SetFromActive
ds.ExportTable "X:\MyTable", 55, "Btrieve 6","","","",""
ds.ExportTable "X:\MYTABLE", 55, "RS_dBase","","","",""
ds.ExportTable "SCOTT.VIDEO_EMPLOYEE", 55, "Oracle7
ODBC",
"SCOTT","TIGER",""
ds.ExportTable "indigo.dbo.video_Employee", 55,
"SQLServer_ODBC","sa", "secretpw", ""
ds.ExportTable "SYSADM.DEDUCTIONS", 55, "SQLBase",
"SYSADM", "", ""
End Sub
```

This macro uses a named connection to determine the directory in which to create the new table:

```
Windows API function declaration
Declare Function GetPrivateProfileString Lib "Kernel" (ByVal
lpApplicationName As String, ByVal lpKeyName As String, ByVal
lpDefault As String, ByVal lpReturnedString As String, ByVal nSize
As Integer, ByVal lpFileName As String) As Integer

Sub Export()
Dim ThePath As String
ThePath = Space(256)
'Replace "MyNamedConnection" with yours.
Length = GetPrivateProfileString ("MyNamedConnection",
"DataFilePath", "", ThePath, Len(ThePath), "RPTSMITH.CON")
ThePath = Left(ThePath, Length) 'Remove last garbage character
'Add backslash if not there.
If Right(ThePath, 1) <> "\" Then ThePath = ThePath + "\"
'Replace the "MyTable" table name with yours.
ThePath = ThePath + "MyTable"
dim ds As DataSet
ds.SetFromActive
ds.ExportTable ThePath, 55, "RS_Paradox", "", "", ""
End Sub
```

Field$

Retrieves the value of the specified field for the record number to which the data set of the currently active report is pointing. This statement is always used as a function.

Syntax Field$(FieldName$)

Parameters FieldName$—The name of the report field for which you want the current value.

Returns The value of the specified field *as a string,* regardless of the retrieved field's data type. If the specified field is not found, returns "N\A" or "<ERROR>".

Comments The name of the field should exactly match the database column name.

You can link two tables together that have one or more column names in common. In this case, it's necessary to use the fully qualified field name to insure that you're getting the correct field. The fully qualified field name includes the table name followed by a period (.) followed by the field name. You can also use a field or table alias. The surest way to get the correct fully qualified field name is to drag the field you want from the list box that appears on the left of the Edit Macro dialog box into the Formula box.

Example
```
'This code line creates a string variable called "NextId$" and assigns
'it the current value of the "Employee_Id" field.
NextId$ = Field$("Employee_Id")
```

Field$ (dataset object)

Returns the value of the specified data field for the current data set record. You can set the current record with the data set record property. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see "Using the DataSet Control" on page 215.

Syntax [object].Field$(FieldName$)

Parameters FieldName$—The name of the field from which to return data.

Returns The value of the specified data field for the current dataset record.

Comments Before data can be retrieved from a dataset object, a connection must be made, links must be set and a *Commit* or *Recalc* must be successfully performed.

Example
```
MyData.record = 5
Salary = val(MyData.Field$("SALARY"))
```

FieldFont

Changes the font type, style attribute, point size, or color of a field in your report. This command is usually used in a macro that is linked to the display event of a field.

Syntax FieldFont Facename$, PointSize, Style, ForColor, BackColor

Parameters Facename$—The font name.

PointSize—Size, in points, of the field font.

Style—The style code indicating:

–1	No change
0	Normal text
1	Bold text
2	Italic
4	Strikeout
8	Underline

ForColor—The foreground text color.

BackColor—The background text color.

Comments Add the style codes to combine attributes. For example, a 3 designates bold italic. (Using the codes on the previous page, 1 + 2 = 3). However, do not add 0 or –1 to other numbers, as this can produce unexpected results. For example, you might set ~~strikeout~~ text and add –1, thinking it would produce no *additional* change. Instead, it will produce *bold* italic text (4 – 1 = 3), as summed values here assume the use of positive integers only.

For the fourth and fifth parameters (foreground and background colors), use a color value. You can specify one of 16 million colors using the Rgb command. Windows substitutes the closest color to the one you select. Use –1 as the color value for any color you want unchanged.

Example The following example changes the font to red, italic, 14-point Arial:

```
Field Font "Arial", 14, 2, RGB(255,0,0), –1
```

FieldText

Applies to macro fields that are linked to the display event of a data field object. It changes the display of the field.

Syntax FieldText Text$

Parameters Text$—The literal text string (in quotes) you want displayed in the field.

Example This example searches the “FirstName” field of the correct record, for a value of “James.” If this value is found, then “Jim” is substituted as the text displayed in the “FirstName” field.

```
If field$("FirstName")="James" then
FieldText "Jim"
End If
```

FileAttr

Returns information about an open file. Used only as a function.

Syntax FileAttr (filenumber%, attribute%)

Parameters filenumber%—The number used in the *Open* statement to open the file.

attribute%—Either a 1 or 2. If *attribute%* is 2, FileAttr returns the operating system handle for the file.

Returns The *FileAttr* function returns information about an open file. Depending on the attribute chosen, this information is either the file mode or the operating system handle. The following list shows the return values and corresponding file modes if *attribute%* is 1.

1—Input

2—Output

3—Append

Example The following example gets the DOS file handle of the second open tile report.

```
handle=FileAttr(2,2)
```

FileCopy

Syntax FileCopy SourceFile$, DestFile$

Comments FileCopy makes a copy of SourceFile in DestFile. Both SourceFile and DestFile are String expressions that contain the file names with no wild cards. SourceFile cannot be copied if it is opened by BASIC for anything other than Read access.

FileDateTime

Syntax FileDateTime(filename$)

Returns The FileDateTime function returns a string that indicates when filename was last modified.

Comments The argument filename is a String expression that contains the name of the file to query. Wildcards are not allowed. Filename can contain optional path and disk information.

FileLen function

Syntax FileLen(filename$)

Returns The FileLen function returns a Long that indicates the length of the specified file.

Comments The argument filename is a String expression that contains the name of the file to query. Wildcards are not allowed. Filename can contain optional path and disk information.

If the specified file is open, FileLen returns the length of the file before it was opened.

Fix

Returns the integer part of a numeric expression. Used only as a function.

Syntax Fix (numeric-expression)

Parameters numeric-expression—The parameter given is any numeric-expression.

Returns The integer part of a numeric-expression. The return type matches the type of numeric expression (including variant expressions which return a result of the same vartype as input), except vartype 8 (string) is returned as vartype 5 (double) and vartype 0 (empty) is returned as vartype 3 (long).

Comments The parameter given is any numeric expression. *Fix* removes the fractional part of the expression and returns only the integer part for both positive and negative numeric expressions. See the “CInt” on page 276 and “Int” on page 349.

Examples The following returns 6:

```
Fix (6.2)
```

The following returns –6:

```
Fix (–6.2)
```

For...Next

Repeats the statement block a fixed number of times, determined by the values of *start, end*, and *step*.

Syntax

For counter = start TO end [STEP increment]
[statementblock]
[Exit For]
[statementblock]
Next [counter]

Parameters

start—The initial value.

end—The maximum value (inclusive) used by the loop.

increment—The amount to add to the counter each time through the loop.

counter—A variable to hold the count.

statementblock—BASIC functions, statements, or methods to be executed in the loop.

Comments

In order for a *For...Next* loop to be properly executed, the *start* and *end* values must be consistent with *increment*. If *end* is greater than *start*, *increment* must be positive. If *end* is less than *start*, *increment* must be negative (effectively creating a decrement, rather than an increment). BASIC compares the sign of (*end-start*) with the sign of *Step*. If the signs are the same, and *end* does not equal *start*, execution of the *For...Next* loop begins. If not, the loop is omitted in its entirety.

With a *For...Next* loop, the program lines following the *For* statement are executed until the *Next* statement is encountered. At this point, the *Step* amount is added to the counter and compared with the final value, *end*. If the beginning and ending values are the same, the loop is executed once, regardless of the *Step* value. Otherwise, the *Step* value controls the loop as follows:

Step Value	Loop Execution
Positive	If *counter* is less than or equal to *end*, the *Step* value is added to *counter*. Control returns to the statement after the *For* statement and the process repeats. If *counter* is greater than *end*, the loop is exited; execution resumes with the statement following the *Next* statement.
Negative	The loop repeats until *counter* is less than *end*.
Zero	The loop repeats indefinitely.

Within the loop, the value of the counter should not be changed, as changing the counter makes programs more difficult to edit and debug.

For...Next loops can be nested within one another. Each nested loop should be given a unique variable name as its counter. The *Next* statement for the inside loop must appear before the *Next* statement for the outside loop. The *Exit For* statement can be used as an alternative exit from *For...Next* loops.

If the variable is left out of a *Next* statement, the *Next* statement matches the most recent *For* statement. If a *Next* statement occurs prior to its corresponding *For* statement, BASIC returns an error message.

Multiple consecutive *Next* statements can be merged together. If this is done, the counters must appear with the innermost counter first and the outermost counter last.

Example

```
For i = 1 To 10
[statementblock]
For j = 1 To 5
   [statementblock]
Next j, i
```

Format$ function

Syntax Format[$](expression [, fmt])

Returns The Format$ function converts the value of expression to a string based upon the fmt specified.

The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function returns a variantvarianttype of vartypeglosvartype 8 (string).

Comments Format$ will format expression as a number, date, time, or string depending upon the fmt argument.

Expression specifies the value to be formatted. It may be a number, variant, or string.

Fmt is any string expression. It specifies how the output string is to be constructed. See below for a detailed description of format strings.

Formatting Numbers

Numeric values may be formatted as either numbers or date/times. If a numeric expression is supplied and the fmt argument is omitted or null, the number will be converted to a string without any special formatting.

The following are predefined numeric formats with their meanings:

General Number	Display the number without thousand separator.
Fixed	Display the number with at least one digit to the left and at least two digits to the right of the decimal separator.
Standard	Display the number with thousand separator and two digits to the right of decimal separator.
Scientific	Display the number using standard scientific notation.
Currency	Display the number using a currency symbol as defined in the International section of the Control Panel. Use thousand separator and display two digits to the right of decimal separator. Enclose negative value in parentheses
Percent	Multiply the number by 100 and display with a percent sign appended to the right; display two digits to the right of decimal separator
True/False	Display False for 0, True for any other number
Yes/No	Display No for 0, Yes for any other number
On/Off	Display Off for 0, On for any other number

Here are the rules for creating user-defined numeric formats:

A simple numeric format consists of digit characters and optionally, a decimal separator. Two format digit characters are provided: zero (0) and number sign (#). A zero forces a corresponding digit to appear in the output; while a number sign causes a digit to appear in the output if it is significant (in the middle of the number or non-zero).

Number	Fmt	Result
1234.56	#	1235
1234.56	#.##	1234.56
1234.56	#.#	1234.6
1234.56	######.##	1234.56
1234.56	00000.000	01234.560
0.12345	#.##	.12
0.12345	0.##	0.12

A comma placed between digit characters in a format causes a comma to be placed between every three digits to the left of the decimal separator.

Number	Fmt	Result
1234567.8901	#,#.##	1,234,567.89
1234567.8901	#,#.####	1,234,567.8901

Note that while period (.) is always used in the fmt to denote the decimal separator, the output string contains the appropriate character based upon the current international settings for your machine. Likewise, while comma is always used in the fmt specification, the output contains the appropriate separator from the current international settings.

Numbers may be scaled either by inserting one or more commas before the decimal separator or by including a percent sign in the fmt specification. Each comma preceding the decimal separator (or after all digits if no decimal separator is supplied) scales (divide) the number by 1000. The commas do not appear in the output string. The percent sign causes the number to be multiplied by 100. The percent sign appears in the output string in the same position as it appears in fmt.

Number	Fmt	Result
1234567.8901	#,.##	1234.57
1234567.8901	#,,.####	1.2346
1234567.8901	#,#,.##	1,234.57
0.1234	#0.00%	12.34%

Characters may be inserted into the output string by being included in the fmt specification. The following characters are automatically inserted in the output string in a location matching their position in the fmt specification:

- + $ ( ) space : /

Any set of characters may be inserted by enclosing them in double quotes. Any single character may be inserted by preceding it with a backslash, "\".

NumberFmtResult

1234567.89$#,0.00 $1,234,567.89

1234567.89"TOTAL:" $#,#.00TOTAL: $1,234,567.89

1234\=\>#,#\<\==>1,234<=

You may wish to use the SBL $CSTRINGScstrings metacommand or the Chrchr function if you need to embed double quotation marks in a format specification. The character code for double quote is 34.

Numbers may be formatted in scientific notation by including one of the following exponent strings in the fmt specification:

E- E+ e- e+

The exponent string should be preceded by one or more digit characters. The number of digit characters following the exponent string determines the number of exponent digits in the output. Fmt specifications containing an upper case E results in an upper case E in the output. Those containing a lower case e results in a lower case e in the output. A minus sign following the E causes negative exponents in the output to be preceded by a minus sign. A plus sign in the fmt causes a sign to always precede the exponent in the output.

Number	Fmt	Result
1234567.89	###.##E-00	123.46E04
1234567.89	###.##e+#	123.46e+4
0.12345	0.00E-00	1.23E-01

A numeric fmt can have up to four sections, separated by semicolons. If you use only one section, it applies to all values. If you use two sections, the first section applies to positive values and zeros, the second to negative values. If you use three sections, the first applies to positive values, the second to negative values, and the third to zeros. If you include semicolons with nothing between them, the undefined section is printed using the format of the first section. The fourth section applies to Null values. If it is omitted and the input expression results in a NULL value, Format$ will return an empty string.

Number	Fmt	Result
1234567.89	#,0.00;(#,0.00);"Zero";"NA	" 1,234,567.89
-1234567.89	#,0.00;(#,0.00);"Zero";"NA	" (1,234,567.89)
0.0	#,0.00;(#,0.00);"Zero";"NA#	"Zero
0.0	#,0.00;(#,0.00);;"NA	"0.00
Null	#,0.00;(#,0.00);"Zero";"NA	"NA
Null	"The value is: " 0.00	

Formatting Date Times

Both numeric values and variants may be formatted as dates. When formatting numeric values as dates, the value is interpreted according the standard Basic date encoding scheme. The base date, December 30, 1899, is represented as zero, and other dates are represented as the number of days from the base date.

As with numeric formats, there is a number of predefined formats for formatting dates and times:

General Date	If the number has both integer and real parts, display both date and time. (e.g., 11/8/93 1:23:45 PM); if the number has only integer part, display it as a date; if the number has only fractional part, display it as time
Long Date	Display a Long Date. Long Date is defined in the International section of the Control Panel
Medium Date	Display the date using the month abbreviation and without the day of the week. (e.g, 08-Nov-93)
Short Date	Display a Short Date. Short Date is defined in the International section of the Control Panel
Long Time	Display Long Time. Long Time is defined in the International section of the Control Panel and includes hours, minutes, and seconds.
Medium Time	Do not display seconds; display hours in 12-hour format and use the AM/PM designator
Short Time	Do not display seconds; use 24-hour format and no AM/PM designator.

When using a user-defined format for a date, the fmt specification contains a series of tokens. Each token is replaced in the output string by its appropriate value.

A complete date may be output using the following tokens:

Token	Output
c	The date time as if the fmt was: "ddddd ttttt". See the definitions below.
ddddd	The date including the day, month, and year according to the machine's current Short Date setting. The default Short Date setting for the United States is m/d/yy.

dddddd	The date including the day, month, and year according to the machine's current Long Date setting. The default Long Date setting for the United States is mmmm dd, yyyy.
ttttt	The time including the hour, minute, and second using the machine's current time settings. The default time format is h:mm:ss AM/PM.

Finer control over the output is available by including fmt tokens that deal with the individual components of the date time. These tokens are:

Token	Output
d	The day of the month as a one or two digit number (1-31).
dd	The day of the month as a two digit number (01-31).
ddd	The day of the week as a three letter abbreviation (Sun-Sat).
dddd	The day of the week without abbreviation (Sunday-Saturday).
w	The day of the week as a number (Sunday as 1, Saturday as 7).
ww	The week of the year as a number (1-53).
m	The month of the year or the minute of the hour as a one or two digit number. The minute is output if the preceding token is an hour; otherwise, the month is output.
mm	The month or the year or the minute of the hour as a two digit number. The minute is output if the preceding token is an hour; otherwise, the month is output.
mmm	The month of the year as a three letter abbreviation (Jan-Dec).
mmmm	The month of the year without abbreviation (January-December).
q	The quarter of the year as a number (1-4).
y	The day of the year as a number (1-366).
yy	The year as a two-digit number (00-99).
yyyy	The year as a four-digit number (100-9999).
h	The hour as a one or two digit number (0-23).
hh	The hour as a two digit number (00-23).
n	The minute as a one or two digit number (0-59).
nn	The minute as a two digit number (00-59).

s	The second as a one or two digit number (0-59).
ss	The second as a two digit number (00-59).

By default, times will be displayed using a military (24-hour) clock. Several tokens are provided in date time fmt specifications to change this default. They all cause a 12 hour clock to be used. These are:

Token	Output
AM/PM	An uppercase AM with any hour before noon; an uppercase PM with any hour between noon and 11:59 PM.
am/pm	A lowercase am with any hour before noon; a lowercase pm with any hour between noon and 11:59 PM
A/P	An uppercase A with any hour before noon; an uppercase P with any hour between noon and 11:59 PM.
a/p	A lowercase a with any hour before noon; a lowercase p with any hour between noon and 11:59 PM.
AMPM	The contents of the 1159 string (s1159) in the WIN.INI file with any hour before noon; the contents of the 2359 string (s2359) with any hour between noon and 11:59 PM. Note, ampm is equivalent to AMPM.

Any set of characters may be inserted into the output by enclosing them in double quotes. Any single character may be inserted by preceding it with a backslash (\). See number formatting above for more details.

Formatting Strings

Strings are formatted by examining the fmt specification and transferring one character at a time from the input expression to the output string.

By default, formatting will transfer characters working from left to right. The exclamation point (!) format character may be used to change this default. Its presence in the fmt specification will cause characters to be transferred from right to left.

By default, characters being transferred will not be modified. The less than (<) and the greater than (>) characters may be used to force case conversion on the transferred characters.The less than sign forces output characters to be in lowercase. The greater than sign forces output characters to be in uppercase.

Character transfer is controlled by the ‘at’ sign (@) and ‘ampersand’ (&) characters in the fmt specification. These operate as follows:

Character	Interpretation
@	Output a character or a space. If there is a character in the string being formatted in the position where the @ appears in the format string, display it; otherwise, display a space in that position.
&	Output a character or nothing. If there is a character in the string being formatted in the position where the & appears, display it; otherwise, display nothing.

A fmt specification for strings can have one or two sections separated by a semicolon. If you use one section, the format applies to all string data. If you use two sections, the first section applies to string data, the second to Null values and zero-length strings.

FreeFile

Used when you need to supply a file number, and want to make sure that you are not choosing a file number which is already being used. Used as a function.

Syntax FreeFile

Returns The lowest unused file number.

Comments The value returned can be used in a subsequent *Open* statement.

Example
```
FileNumber=FreeFile
Open for output as filenumber "Temp.txt"
```

Function ... End Function

Defines a function procedure. The statement enclosed within this command pair can be used only as a function. The purpose of a function is to produce and return a single value of a specified type.

Syntax
```
Function name [(parameter [As type]...)]
name = expression
End Function
```

Parameters The parameters are specified as a comma-separated list of parameter names. The data type of a parameter can be specified by using a type character or by using the *As* clause. Record parameters are declared by using an *As* clause and a type which has previously been defined using the *Type* statement.

Array parameters are indicated by using empty parentheses after the parameter. The array dimensions are not be specified in the *Function* statement. All references to an array parameter within the body of the function must have a consistent number of dimensions.

Returns Specify the return value by assigning it to the function name as if it were a variable or parameter. If no such assignment occurs, the value returned is 0 for numeric functions and the empty string ("") for string functions. The function returns to the caller when the *End Function* statement is reached or when an *Exit Function* statement is executed.

Comments Recursion is supported.

In the *Function* statement, the name of the function can end with a type character, which specifies the type that the function returns. When calling the function, you need not specify the type character.

BASIC procedures use the call-by-reference convention. This means that if a procedure assigns a value to a parameter, it modifies the variable passed by the caller. This feature should be used with great care.

Use Sub to define a procedure which has no return value.

Example

```
Function = triangle(leg1 as double, leg2 as double)
Triangle = (leg1^2+leg2^2)^.5
End Function
```

Get

Syntax Get [#] filenumber%, [recordnumber&], variable

Comments Get is used to read a variable from a file opened in Random or Binary mode.

Filenumber% is an integer expression identifying an open file from which to read. See the Open statement for more details.

Recordnumber& is a Long expression containing the number of the record (for Random mode) or the offset of the byte (for Binary mode) at which to start reading. Recordnumber is in the range 1 to 2,147,483,647. If recordnumber is omitted, the next record or byte is read. Note that the commas are required, even if no recordnumber is specified.

Variable is the name of the variable into which Get reads file data. Variable can be any variable except Object, Application Data Type or Array variables (single array elements may be used).

For Random mode, the following apply:

Blocks of data are read from the file in chunks whose size is equal to the size specified in the Len clause of the Open statement. If the size of the variable is smaller than the record length, the additional data is discarded. If the size of the variable is larger than the record length, an error occurs.

For variable-length String variables, Get reads two bytes of data that indicate the length of the string, then reads the data into the variable.

For Variant variables, Get reads two bytes of data that indicate the type of the variant, then it reads the body of the variant into the variable. Note that Variants containing strings contain two bytes of type information followed by two bytes of length followed by the body of the string.

User defined types are read as if each member were read separately, except no padding occurs between elements.

Files opened in Binary mode behave similarly to those opened in Random mode except:

Get reads variables from the disk without record padding.

Variable length Strings that are not part of user defined types are not preceded by the two byte string length. Instead, the number of bytes read into a string variable is equal to the length of the existing string variable.

GetAllFields$

Gets a comma-delineated list of the fields available for a given table. A connection must first be made, and the table added to the data set, before its field list can be retrieved.

Syntax [object.]GetAllFields (Table$, Database$)

Parameters Table$—The name of the table for which you're getting the list of fields.

Database$—The name of the table's database. This applies only to tables with databases.

Returns A list of all fields that are available in the given table.

Comments The Table$ parameter is the path and file name for local databases. For database servers, it takes the form Owner.TableName. For local databases or servers that don't require that a database be specified, the Database$ parameter should be left blank.

Example MsgBox GetAllField$ ("dbo.emp","Indigo")

GetAllFields$ (dataset object)

Lists all fields that are available in the given table. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetAllFields$(Table$,DBase$)

Parameters Table$—The Table$ parameter is the path and file name for local databases.

dBASE$—The database that contains the table for connections that have databases.

Comments A connection must first be made and the table added to the dataset before its field list can be retrieved.

For database servers *Table$* takes the form: Owner.TableName.For local databases or servers that don’t require that a database be specified, the *dBase$* parameter should be set to a null string.

See the “GetField$” on page 335 to get an individual field out of the list of fields.

Example AvailableField$=MyData.GetAllFields$("dbo.emp", "hr")

GetAttr

Syntax GetAttr(filename$)

Returns The GetAttr function returns the attributes of the file, directory or volume label indicated by filename.

Here is a description of file attributes returned by GetAttr:

Value	Meaning
0	Normal file
1	Read-only file
2	Hidden file
4	System file
8	Volume label
16	Directory
32	Archive - file has changed since last backup

Comments Filename is a String expression that indicates the name of the file whose attributes are returned. Filename may not contain wild cards.

GetColumnAlias$

Gets the alias for the specified field in the specified table. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetColumnAlias$(Table$, Database$, Column$)

Parameters Table$—Defines a string of the form: Owner. TableName or, for local databases such as dBASE, the file name of the local database file.

Database$—The name of the database that contains the table, for connections that use databases.

Column$—The column for which to set an alias.

Returns The alias for the specified field in the specified table.

Comments For database servers Table$ takes the form: Owner.TableName. For local databases or servers that don’t require that a database be specified, the *Database$* parameter should be set to a null string.

Example ColumnAlias$=MyData.GetColumnAlias$("dbo.emp", "hr","DEPT_ID")

GetCurValues

Syntax GetCurValues recordName

Returns The GetCurValues statement stores the current values for the application dialog box associated with the specified record.

Comments RecordName must have been previously dimensioned as an application dialog box.

GetDataSources$

Gets the available data sources, as a comma-delineated string list.

Syntax [object].GetDataSources$

Returns A comma-delineated list of all of the data sources available to ReportSmith, including ODBC sources.

Example ‘This example creates a message box displaying a message of, “The available data sources:” followed by the comma-delineated list of data sources.

```
MsgBox "The available data sources: "+ GetDataSources$
```

GetDataSources$ (dataset object)

Gets the available data sources (for the dataset object), as a comma-delineated string list. *T*his command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object.]GetDataSources$

Returns Returns a list of all of the data sources available to ReportSmith.

Example ‘This example creates a variable called *DataSourcesAvailable$* and assigns it the value returned by *GetDataSources$*, on the *MyData* dataset.

```
DataSourcesAvailable$=MyData.GetDataSources$
```

GetField$

Returns a substring from a delimited string. Can be used only as a function.

Syntax GetField$(string$, field_number%, separator_chars$)

Parameters string$—The source string is considered to be divided into fields by separator characters.

field_number%—The number of the substring to fetch.

separator_chars$—The character used to separate field; e.g., in a comma-delimited string this character would be a comma.

Returns A substring from a source string.

Comments Multiple separator characters can be specified. The fields are numbered starting with one. If *field_number* is greater than the number of fields in the string, the empty string is returned.

Example GetField$("value1, value2, value3",2,",") would return “value2.”

GetFieldList$

Lists all fields that have been included in the given table. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object.]GetFieldList$(Table$,DBase$)

Parameters Table$—Defines a string of the form: Owner. TableName or, for local databases such as dBASE or Excel, the file name of the local database file.

dBase$—Specifies the table, for connections using databases.

Returns A list of all fields that have been included in the given table.

Comments A connection must first be made and the table added to the data set before its field list can be retrieved.

For database servers, *Table$* takes the form: Owner.TableName. For local databases or servers that don’t require that a database be specified, the dBASE$ parameter should be set to a null string.

Example IncludedField$=MyData.GetFieldList$("dbo.emp", "hr")

GetFieldName$

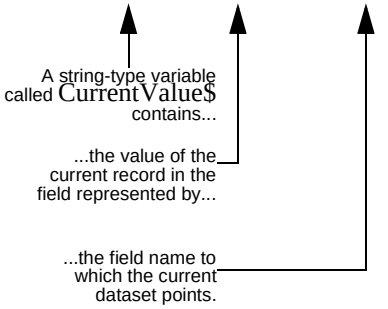
This function is a global filter, returning the column name of the data field for which the filter macro is being called. This command is used only as a function.

Syntax GetFieldName$()

Returns Returns the column name of the data field.

Comments See also “SetDataFilter” on page 397.

Example CurrentValue$=Field$(GetFieldName$())



GetGroup$

Gets a string that provides information about grouping at the specified level. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetGroup$(Level)

Parameters Level—Specifies the grouping level that you want information about, where 0 is the entire report group, 1 is the primary grouping criterion, 2 is the secondary grouping criterion, and so forth.

Returns A string that provides information about grouping at the specified level.

Comments If an invalid index is specified, a null string is returned and the *Error$* property is set to indicate the error. Valid values for levels are zero to the number of defined groups.

Example PrimaryGroup$=MyData.GetGroup(1)

GetIncludePath$

Gets the default directory for macro include files.

Syntax GetIncludePath$()

Returns A string that is the default path for macro include files.

Example This example creates a message box displaying the text “Looking for include files in” followed by the default path for macro include files.

```
MsgBox"Looking for include files in"+GetIncludePath$()
```

GetNext

Causes the data set in an active report to point to the record that comes immediately after the record to which it is currently pointing.

Syntax GetNext

Example This example steps through the entire report counting the employees with the first name ‘John.’

```
GetRandom (1)
For x = 1 to RecordCount()
If Field$("First_Name")="John" then
Count=Count+1
End If
GetNext
Next X
```

GetPrevious

Causes the data set in an active report to point to the record that comes immediately before the record to which it is currently pointing.

Syntax GetPrevious

Comments The *GetPrevious* command can be used to change the current record of a data set. The current record determines what data the *Field$*, *SumField$* and *DateField$* functions retrieve. This function could be used in a macro-defined summary field. When a macro defined summary field is dropped in a group footer the current record is the last record in that group. For this reason, a macro derived field can step backwards through the group performing custom summary operations. See also “GetNext” on page 337, “GetRandom” on page 338, “Field$” on page 318, “hWin_Active()” on page 344.

Example
```
'Position to the 3rd record
GetRandom 3
'Now go to 2
GetPrevious
```

GetRandom

Causes the data set in an active report to point to the record specified by the RecordNumber% parameter (if that record number exists).

Syntax GetRandom RecordNumber%

Parameters RecordNumber%—The index number of the data record to which you want to navigate.

Comments See also “GetPrevious” on page 338, “GetNext” on page 337, “Field$” on page 318, “String$” on page 407, “hWin_Active()” on page 344, and “TestSelection$” on page 410.

Example
```
'Point to the 23rd record in a set of data
  GetRandom 23
```

GetRecordLimit

Gets the total number of records that ReportSmith will download for any loaded or created report. *Used only as a function.*

Syntax GetRecordLimit

Comments This limit is set with the function SetRecordLimit.

Example TheLimit=GetRecordLimit

GetRepVar

Retrieves the value of a report variable in the active report. This command is only used as a function.

Syntax GetRepVar(ReportVariable$)

Parameters ReportVariable$—The name of a report variable in your report.

Returns The value of the specified report variable as a string. This function returns "<ERROR>" if a report variable of the specified name cannot be found in the active report.

Comments Report variables *are* case sensitive. *GetRepVar* takes a string parameter that specifies the name of the report variable being retrieved. See also "GetSQL$ (dataset object)" on page 340, "Val" on page 415, and "SetTableAlias" on page 401.

Example Var_name$=GetRepVar("Rep_var_name")

GetSort$

Gets a string indicating sorting criteria at the given level. *T*his command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see "Using the DataSet Control" on page 215.

Syntax [object].GetSort$(Level)

Parameters Level—Specifies the sorting level that you want information about where 1 is the primary sorting criteria, 2 is the secondary sorting criteria, and so forth.

Returns A string indicating sorting criteria at the given level.

Comments Valid values for the level parameter are 1 to the number of current sorting criteria. If an invalid index is specified, a null string is returned and the *Error$* property is set to "Invalid Index."

Example PrimarySort$=MyData.GetSort$(1)

GetSQL

Returns a string that is the text of the last SQL statement that ReportSmith executed.

Syntax GetSQL

Comments This function can be used along with the SetSQL statement in a macro that is linked to the “Before SQL is Executed” to change the SQL string “On the Fly.”

Example The following stores the last generated SQL statement in a variable called The_SQL$.

```
The_SQL$ = GetSQL$
```

GetSQL$ (dataset object)

Gets the last SQL statement that was executed for this data set control object. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetSQL$

Returns The last SQL statement that was executed for this data set control object.

Example MySQL$=MyData.GetSQL$

GetSummary$

Gets a string that provides information about a summary field at the specified grouping level and index. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, *a*s shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetSummary$(Level, Index)

Parameters Level—Specifies the grouping level that you want information about where 0 is the entire report group, 1 is the primary grouping criteria, 2 is the secondary grouping criteria, and so forth.

Index—Matches the order in which the tables are originally added.

Returns If an invalid index or level is specified, a null string is returned and the *Error$* property is set to indicate the error.

Example SecondSummary$=MyData.GetSummary(1,2)

GetTable$

Gets a string that describes the table at the specified index. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetTable$(Index)

Parameters Index—Matches the order in which the tables are originally added.

Returns A string that describes the table at the specified index.

Comments If an invalid index is given, this function returns a null string and the *Error$* property is set to an appropriate error message.

Example SecondTable$=MyData.GetTable$(2)

GetTableAlias$

Gets the alias for the specified table. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetTableAlias$(Table$, DBase$)

Parameters Table$—Defines a string of the form: Owner. TableName or, for local data bases such as dBASE, the file name of the local database file.

dBase$—Specifies the table (for connections that have databases).

Returns The alias for the specified table.

Comments For local databases or servers that don’t require that a databases be specified, the *dBase$* parameter should be set to a null string.

Example TableAlias$=MyData.GetTableAlias$("dbo.emp","hr")

GetTableLink$

Gets a string that provides information about the table link at the given index. *T*his command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].GetTableLink$(Index)

Parameters Index—The link for which to retrieve information.

Returns A string that provides information about the table link at the given index (if one exists).

Comments If an invalid index is specified, a null string is returned and the *Error$* property is set to indicate the error.

Example SecondTableLink$ = MyData.GetTableLink(2)

Global

Declares global variables for use in a Basic program.

Syntax Global variableName [As type] [,variableName [As type]]

Parameters variableName—The name of your variable.

type—A valid data type to assign your variable. (See “Data types of variables” on page 256 for a list of valid data types.)

Comments BASIC is a strongly typed language. The available data types are: numbers, strings, records, arrays, dialog boxes, and Application Data Types (ADTs).

Global data is shared across all loaded modules. Attempting to load a module with a declared global variable of a different data type than an existing global variable of the same name will cause failure of the module load.

If the *As* clause is not used, the type of the global variable can be specified by using a type character as a suffix to *variableName*. The two different type-specification methods can be intermixed in a single *Global* statement (although not on the same variable).

Regardless of which mechanism you use to declare a global variable, you can choose to use or omit the type character when referring to the variable in the rest of your program. The type suffix is not considered part of the variable name.

Example Global A as string

GoTo

Changes a program flow by branching to a label.

Syntax GoTo Label

Parameters Label—The name of a labeled procedure to which code should branch.

Comments *GoTo* sends control to a label. ReportBasic does not support the use of line numbers.

A label has the same format as any other BASIC name. To be recognized as a label, a name must begin in the first column and be followed immediately by a colon (:). Reserved words are not valid labels.

GoTo cannot be used to transfer control out of the current function or sub.

Use of *GoTo* is not recommended. Instead, use *While_Loops*, *Do_Loops*, *Select_Case* statements for loops and subroutines.

Example
```
Start:
If Value > Limit then GoTo Done
value=value+somemore
GoTo Start
Done:
End Sub
```

GroupBox

Sets up a box that encloses sets of items, such as option boxes and check boxes that you wish to group together in a dialog box.

Syntax GroupBox x, y, dx, dy, text$

Parameters x, y—The *x* and *y* parameters set the position of the group box relative to the upper left corner of the dialog box. See “Begin Dialog...End Dialog” on page 270.

dx, dy—*dx* and *dy* set the width and height of the box.

text$—The *text$* field contains a title that is embedded in the top border of the group box.

Comments The *GroupBox* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

In the *text$* field, trailing characters are truncated if *text$* is wider than *dx*. If the *text$* parameter is an empty string (“”), the top border of the group box will be a solid line.

Example GroupBox 10,10,80,50, "My Group"

Hex$

Converts a value from decimal or integer to a hexadecimal string.

Syntax Hex$(numeric-expression)

Parameters numeric-expression—A decimal or integer value to convert.

Returns A hexadecimal representation (as a string) of a numeric-expression.

Comments If the numeric expression is an integer, the string contains up to four hexadecimal digits; otherwise, the expression is converted to a long integer, and the string can contain up to 8 hexadecimal digits.

Example MsgBox "The hexadecimal representation of 175 is:" + Hex$(175)

Hour

Returns the hour of day component (0-23) of a date-time value.

Syntax Hour(expression)

Returns The Hour function returns the hour of day component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments The Hour function returns an integer between 0 and 23, inclusive.

It accepts any type of *expression* including strings and will attempt to convert the input value to a date value.

hWin_Active()

Gets the window handle of the currently active report.

Syntax hWin_Active()

Comments This function can be used along with Windows API functions. You make the API functions available to ReportBasic through the use of the *Declare* function.

Example

```
Declare Function ShowWindow Lib "User"(ByVal hWnd
As Integer, ByVal nCmdShow As Integer) As Integer
Sub MyMacro()
'Force the active report to an Icon
Result=ShowWindow(hWin_Active(),2)
```

hWin_RS()

Gets the ReportSmith main window handle.

Syntax hWin_RS()

Comments You can use this function along with Windows API functions. You make the API functions available to ReportBasic through the use of the *Declare* function.

Example RS_Handle=hWin_RS()

Id (dataset object and report object)

Stores an integer value. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Id = [integerexpression]

Comments The *Id* property function can be used to track information related to the dataset control. As with the *Name$* property this value has no meaning to ReportBasic and it is completely up to the programmer how it is to be used. This property is read/write at run time.

Example MyData.Id=127

If...Then...Else

Organizes alternative actions into separate, conditional blocks of code.

Syntax A IF condition THEN action [ELSE Alternative_Action]

Syntax B
```
IF condition_a THEN
action_statement_block
[ELSEIF condition_b THEN
Alternative_Action_statement_block]...
[Else
No_Conditions_Met_statement_block]
End If
```

Parameters In Syntax A, condition represents an evaluated condition which must be true for the action to be performed. The *Else* clause in this syntax is optional; if omitted, execution flows to the statement following the *If* statement, whenever the stated condition is not met. If the *Else* clause is used, Alternative_Action is executed whenever the *If* condition is false.

In Syntax B, additional alternatives are provided. This code block first evaluates condition_a; if it is true, then action_statement_block is executed. If condition_a is not true, execution proceeds to the next *ElseIf* statement, and condition_b is evaluated. If condition_b is true, then Alternative_Action_statement_block is executed. Any number of *ElseIf* statements are permitted, although large numbers of such statements can be difficult to track. Finally, if none of the *If* or *ElseIf* conditions are true, No_Conditions_Met_statement_block is executed.

Comments The syntax in both formats above, including the placement and organization of items on each line, must be followed exactly because the resulting action depends on the logical value of one or more conditions expressed in the structure.

The condition can be any expression which is evaluated as TRUE (non-zero) or FALSE (zero).

In the single-line version of the *If* statement (Syntax A), action and alternative_action can be any valid single statement. Multiple statements separated by colons (:) are not allowed. When multiple statements are required in either the *Then* or *Else* clauses, use the block version of the *If* statement (Syntax B).

In the block version of the *If* statement (Syntax B), the statement blocks can be made up of zero or more statements, separated by colons (:) or on different lines.

Example 1

```
If TOTAL > LIMIT Then
MsgBox "Over Limit"
Else
MsgBox "Under Limit"
End If
```

Example 2

```
If TOTAL > LIMIT Then
MsgBox "Over Limit"
ElseIf TOTAL < LIMIT Then
MsgBox "Under Limit"
ELSE
MsgBox "Total = Limit"
End If
```

IncludeFields$

Adds columns from a table to your data set. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object.] IncludeFields$,Table$, DBase$, IncludeList$

Parameters Table$—Defines a string of the form: Owner. TableName or, for local databases such as dBASE, the file name of the local database file.

dBase$—Contains the table for connections that have databases.

IncludeList$—A list of fields in a table to include as part of a table.

Comments The field list should be one string with the names of the included fields separated by commas. The names should be provided exactly as they appear in ReportSmith dialog boxes; this usage is case-sensitive.

Example
```
'Type this code on one line.
MyData.IncludeFields
"dbo.emp","Pubs","First_Name,Last_Name,Dept,Emp_Id"
```

Input$

Reads a specified number of characters from a file. Can be used only as a function.

Syntax Input$ (numchars%, [#]filenumber%)

Parameters numchars%—Contains the number of characters (bytes) to read from the file.

filenumber%—An integer expression identifying the open file to read from.

Returns A string containing the characters read.

Comments The file pointer is advanced by the number of characters read. Unlike the *Input #* statement, *Input$* returns all characters it reads, including carriage returns, line feeds, and leading spaces.

This function is useful when you want to read in characters such carriage returns, commas, or other characters that are used as delimiters for the *Input$* statement.

Example The following example reads 20 characters from File2.

```
Input$ (20,#2)
```

Input

Reads data from a sequential file and assigns the data to variables.

Syntax Input [#] filenumber%, variable, [variable]
Input [prompt$,] variable, [variable]

Parameters filenumber%—An integer expression identifying the open file to read from. This is the number used in the *Open* statement to open the file.

prompt$—An optional string that can be used to prompt for keyboard input.

variable—Lists the variables that are assigned the values read from the file. The list of variables is separated by commas.

Comments If *filenumber* is not specified, the user is prompted for keyboard input with a question mark "?", unless *prompt$* is specified.

Examples Open "c:\MyFile.Txt" for input as 1

—or—

input #2, Name$, age, address$

InputBox$

Displays a dialog box containing a prompt and a text box for user input. Used only as a function.

Syntax InputBox$(prompt$[,title$ [,default$ [,xpos%,ypos%]]])

Parameters prompt$—A string expression containing the text to be shown in the dialog box. The length of *prompt$* is restricted to 255 characters. (This figure is approximate and depends on the width of the characters used.) Note that a carriage return and a line-feed character must be included in *prompt$* if a multiple-line prompt is used.

title$,default$—*Title$* represents the caption that appears in the dialog box's title bar. *Default$* is the string expression that is shown in the edit box as the default response. If either of these parameters is omitted, nothing is displayed.

xpos%, ypos%—Numeric expressions, specified in dialog box units, that determine the position of the dialog box. *Xpos%* determines the horizontal distance between the left edge of the screen and the left border of the dialog box. *Ypos%* determines the horizontal distance from the top of the screen to the dialog box's upper edge. If these parameters are not entered, the dialog box is centered roughly one third of the way down the screen. A horizontal dialog box unit is ¼ of the average character width in the system font; a vertical dialog box unit is 1/8 of the height of a character in the system font. If you want to specify the dialog box's position, you must enter both of these parameters. If you enter one without the other, the default positioning is set.

Returns After the user presses *Enter*, or chooses the OK button, *InputBox$* returns the text contained in the input box. If the user chooses Cancel, the *InputBox$* function returns a null string.

Comments Once the user has entered text, or made the button choice being prompted for, the contents of the box are returned.

Example Street$=InputBox$("Enter Street Name","MyTitle","Default Exp",15,15)

InStr

Finds the occurrence of a substring in a string. The *InStr* command can be used only as a function.

Syntax InStr ([position%,] string$, substring$)

Parameters position%—Indicates the index of the character within *string$* where the search should start. If not specified, the search starts at the beginning of the string (equivalent to a position% of 1).

string$—The string being searched.

substring$—The substring to search for within *string$*.

Returns An integer representing the position of the first occurrence of a substring within another string.

Comments These parameters can be string variables, string expressions, or string literals.

If the *position%* parameter is greater than the length of the substring, if the *string$* parameter is a null string, or if the *substring$* cannot be located, *InStr* returns a zero. If the *substring$* parameter is a null string, then the *position%* parameter is returned.

The index of the first character in a string is 1, not zero.

Example The following starts at the first character of a string, searching for “JR.” in the field Last_Name from the Employee table.

```
InStr(1,"Employee.Last_Name","JR.")
```

Int

Returns the integer portion of a number. Can be used only as a function.

Syntax Int (numeric-expression)

Parameters numeric-expression—The parameter given is any numeric expression.

Returns The integer part of a numeric-expression.

Comments For positive numeric expressions, *Int* removes the fractional part of the expression and returns the integer part only. For negative numeric-expressions, *Int* returns the largest integer less than or equal to the expression.

Example For example, Int (6.2) returns 6; Int(–6.2) returns –7. See “CInt” on page 276 and “Fix” on page 321.

Is Operator

Syntax objectExpression Is objectExpression

Returns -1 (True) if the two object expressions refer to the same object, zero (False) if they do not.

Comments *Is* checks if two object expressions refer to the same object. *Is* may also be used to test if an object variable has been Set to Nothing

IsDate

Syntax IsDate (expression)

Returns IsDate determines whether or not a value is a legal date.

Comments IsDate returns -1 (True) if the expression is of vartype 7 (date) or a string that may be interpreted as a date; otherwise it returns 0 (False).

IsEmpty

Syntax IsEmpty(variant)

Returns The IsEmpty function returns a value that signifies whether or not a variant has been initialized.

Comments IsEmpty returns -1 (True) if the variant is of Vartype 0 (empty); otherwise it returns 0 (False). Any newly-defined Variant defaults to being of Empty type, to signify that it contains no initialized data. An Empty Variant converts to zero when used in a numeric expression, or an empty string in a string expression.

IsMenuChecked

Determines whether a given menu item has a check mark next to it. This command is only used as a function.

Syntax IsMenuChecked (Menu$)

Parameters Menu$—The menu and submenu name that you are interested in.

Returns Returns 1 if the menu is checked, 0 if it isn't checked, and –1 if a menu of the given name was not found.

Comments This function takes a string that specifies a menu item or a submenu item. The string is of the form "MenuName|SubMenuName." The names must match ReportSmith menu commands, not including keyboard accelerators and ellipsis (...) characters. If you omit the pipe (vertical bar character) and submenu names, the routine assumes you're working with a top level menu.

This function cannot correctly return the state of a menu item if it is called before the menu is visible, as in the case of a macro linked to the 'Application Startup' event.

This command can be used to check the state of the menu that is used for new reports by default by placing an exclamation point (!) before the menu name. This can be done whether the menu item is specified by command or relative location. Refer to the second example. Also, refer to "EnableMenu" on page 306, "KillMenu" on page 353, "CloseReport" on page 278, "ExecuteMenu" on page 313 and "IsMenuEnabled" on page 351.

Example Success = IsMenuChecked ("View|Boundaries")

—or—

```
If IsMenuChecked ("!View|Boundaries") = 1 Then
ExecuteMenu "View|Boundaries"
End if
```

IsMenuEnabled

Determines if a given menu item is enabled or dimmed. This command is used only as a function.

Syntax IsMenuEnabled (Menu$)

Parameters Menu$—The menu and submenu name whose status you want to determine.

Returns 1 if the menu is enabled, 0 if it is disabled, and –1 if a menu of the given name was not found.

Comments This command takes a string that specifies a menu item or a submenu item. The string uses this format:

"MenuName|SubMenuName"

The names must match ReportSmith menu commands, not including keyboard accelerators and ellipsis (...) characters. If you omit the pipe and submenu name, then the routine assumes you're working with a top level menu.

This command can be used to check the state of the menu that is to be used as the default for new reports by placing an exclamation point (!) before the menu name. This can be done whether the menu item is specified by command or relative location. Refer to the second example. Also, refer to “EnableMenu” on page 306, “KillMenu” on page 353, “CloseReport” on page 278 and “ExecuteMenu” on page 313.

Example Success = IsMenuEnabled ("Edit|Cut")

—or—

```
'Check if the tables Menu is enabled
If IsMenuEnabled ("Tools|Tables") = 1 Then
MsgBox "Tables Menu Enabled"
End if
```

IsNull

Syntax IsNull(variant)

Returns The IsNull function returns a value that signifies whether or not an expression has resulted in a null value.

Comments IsNull returns -1 (True) if the variant contains the Null value; otherwise it returns 0 (False). Null variants have no associated data and serve only to represent invalid or ambiguous results. Null is not the same as Empty, which indicates that a variant has not yet been initialized.

IsNumeric

Syntax IsNumeric(variant)

Returns The IsNumeric function returns a value that signifies whether or not a variant is of a numeric type.

Comments IsNumeric returns -1 (True) if the variant is of Vartypes 2-6 (numeric) or a string that may be interpreted as a number; otherwise it returns 0 (False).

Kill

Deletes files from disk.

Syntax Kill filespec$

Parameters filespec$—A string expression that specifies a valid DOS file specification. This specification can contain paths and wildcards.

Comments *Kill* deletes only files, not directories. Use the *RmDir* function to delete directories.

Example Kill "c:\Temp.Txt"

KillMenu

Removes one of the ReportSmith menu items.

Syntax KillMenu Menu$

Parameters Menu$—The menu and submenu names that you want to remove.

Returns 0 if a menu was removed successfully, and –1 if a menu of the given name was not found.

Comments It takes a string that specifies a menu item or a submenu item. The string uses this format:

```
"MenuName|SubMenuName"
```

The names must match ReportSmith menu commands, not including keyboard accelerators and "..." characters. If you omit the pipe and submenu name, the routine assumes you're working with a top-level menu.

This command can be used to check the state of the menu that will be used for new reports by default by placing an exclamation mark (!) before the menu name. This can be done whether the menu item is specified by command or relative location.

When you use this command as a function (rather than a statement), you must enclose its parameters within parentheses. For more information on the differences between functions and statements, refer to "Using the DataSet Control" on page 215.

Example The following code fragment will remove the File|New menu Item from ReportSmith.

```
Success = KillMenu("File|New")
```

LBound

Declares the number that represents the first element of an array. Can be used only as a function.

Syntax LBound (arrayVariable [, dimension])

Parameters arrayVariable—The name of the array used to declare a lower bound.

dimension—The number that represents the first element of an array.

Returns The lower bound of the subscript range for the specified dimension of the *arrayVariable*.

Comments The dimensions of an array are numbered starting with 1, not zero. If the dimension is not specified, 1 is used as a default.

LBound can be used with *UBound* to determine the length of an array.

Example
```
'Start employee array at element 5
LBound (Employee, 5)
```

LCase$

Converts the characters in a string to lower case. Can be used only as a function.

Syntax LCase$ (string$)

Parameters string$—The string to convert.

Returns A copy of the source string, with all upper-case letters converted to lower case.

Comments The translation is based on the country specified in the Windows Control Panel.

Example The following example changes “First Street” to “first street.”

```
Lower_Case = LCase$("First Street")
```

Left$

Returns the left n characters of a string. Used only as a function.

Syntax Left$ (string$, length%)

Parameters string$—The string to characters from.

length%—The number of characters to return.

Returns A string of a specified length, copied from the beginning of the source string.

The dollar sign ($) in the function name is optional. If it is specified, the return type is string. If it is omitted, the function typically returns a variant of vartype 8 (string). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments If the length of *string$* is less than *length%*, *Left$* returns the whole string.

Left$ accepts expressions of type string and any type of expression including numeric values and converts the input value to a string.

Example The following will return “C:\”

```
left$ ("c:\RPTSMITH",3)
```

Len

Returns the length of a string or the number of bytes used to store a non-string value. Used only as a function.

Syntax Len (string$)
Len (non-string)

Parameters string$,non-string—If the parameter is a string, the number of characters in the string is returned; otherwise, the length of the built-in datatype or user-defined type is returned.

Returns The length of the parameter.

Comments The parameter can be of any type. If the parameter is a string, the number of characters in the string is returned. If the parameter is a variant variable the number of bytes required to represent its value as a string is returned. If not, the length of the built-in datatype or user-defined type is returned.

Example The following example returns a value of 8.

```
Length_Var = Len ("Thursday")
```

Let

Assigns a value to a BASIC variable. The keyword *Let* is optional.

Syntax [Let] variable = expression

Parameters Let—Used to assign to a numeric, string or record variable. You can also use the *Let* statement to assign to a record field or to an element of an array.

Comments You can also use this statement to assign to a record field or to an element of an array. When assigning a value to a numeric or string variable, standard conversion rules apply.

Example

```
A =1
Let Name$ = "John"
```

Like Operator

Syntax string LIKE pattern

Comments The Like operator returns true (-1) if the string matches pattern, and false (0) if it does not. “string” may be any string expression. “pattern” may also be any string expression where the following characters have special meaning:

Character	Meaning
?	match any single character
*	match any set of zero or more characters
#	match any single digit character (0-9)
[chars]	match any single character in chars
[!chars]	match any single character not in chars
[schar-echar]	match any single character in range schar to echar
[!schar-echar]	match any single character not in range schar to echar

Both ranges and lists may appear within a single set of square brackets. Ranges are matched according to their ANSI values. In a range, schar must be less than echar.

If either string or pattern is NULL then the result value is NULL.

The Like operator respects the current setting of Option Compare.

Line Input

Reads a line from a sequential file into a string variable or brings up a dialog box with an edit control that allows you to enter a string.

Syntax Line Input [#] filenumber%, variable$
Line Input [prompt$,] variables

Parameters Line Input [#]—The *Line Input* # statement reads a line from a sequential file into a string variable.

filenumber%—An integer expression identifying the open file from which to read. This is the number used in the *Open* statement to open the file.

prompt$—An optional string t used to prompt for keyboard input.

variable$—A string variables into which the line from the input file is read.

Comments If *filenumber* is not specified, the user is prompted for keyboard input with a question mark (?) unless *prompt$* is specified.

Example Line Input #1, A$

LinkMacro Method

Syntax [dataset].LinkMacro (MacroName$, Object% ,Event%, [Item$], [IgnoreDialog%])

Definition The LinkMacro Method links a macro in an active list to the specified Object, Event, and Item. If you are linking to an Application event, the macro must be in the list of active global macros. Otherwise, it must appear in the list of macros for the report to which the Dataset object is associated .

Parameters MacroName$ — defines the name of a macro in the list of active macros for which you want to define a link.

Object% — a number that specifies the object to link the macro to.

Event% — a number that specifies the event to link the macro to.

IgnoreDialog% — If IgnoreDialog% is specified and is non-zero, then the macro dialog will not be updated by this method.

Item$ — If you are linking to the keystroke event, Item$ must be a string that specifies a link to a keystroke.

Valid strings are "F1"- "F12" for function keys, "[CTRL] A"- "[CTRL] Z" or "[SHIFT] [CTRL] A" "[SHIFT] [CTRL] Z" as support for other events are added this string might represent a datafield, group header or footer name, or a menu item. At this time all events other than the keystroke events expect the Item$ argument to be omitted or set to a NULL string.

Returns Non-zero on error.

Argument	Description
Object%	Event%
0 - APPLICATION	
	0 - Keystroke
	1 - Before New Report
	2 - After New Report
	3 - Application Startup
	4 - Before Executing SQL
	5 - Before Report Print
	6 - Before Report Load
	7 - After Report Load
	8 - Before Report Save
	9 - After Report Save

10 - Before Application Close

11 - On SQL Execution Error

12 - New File Icon Click

13 - SQL Icon Click

14 - After Report Connects

15 - Before Report Close

16 - After Report Close

1 - REPORT

0 - Keystroke

1 - Before Report Open

2 - After Report Open

3 - Before SqL Execution

4 - Before Print

5 - Before Report Save

6 - Before Report Close

7 - Not Used

8 - On SQL Error

Note: Some link Objects and events are not available using this command. This command currently does not support the DataField, Header, or Footer Objects or the MenuItem Event. Under certain circumstances, some function keys are trapped before the macro links can be executed. The F1 and F12 keys will fail to execute macros linked to them.

Example ds.LinkMacro "ReportLoader", 0,0,"[CTRL] L" ' links a macro called "ReportLoader" the CTRL+L keystroke

ListBox

Used to create a list of choices.

Syntax A ListBox x, y, dx, dy, text$, .field

Syntax B ListBox x, y, dx, dy, stringarray$(),.field

Parameters x, y—Coordinates of the upper left corner of the list box, relative to the upper left corner of the dialog box. The x parameter is measured in ¼ system-font character-width units. The y parameter is measured in 1/8-system-font character-width units. (See “Begin Dialog...End Dialog” on page 270.)

dx, dy—Specify the width and height of the list box.

text$—A string containing the selections for the list box. This string must be defined, using a *Dim* statement, before the *Begin Dialog* statement is executed.

.field—The name of the dialog-record field that holds the selection made from the list box. When the user selects OK (or selects the customized button created using the Button statement), a number representing the selection's position in the *text$* string is recorded in the field designated by the .field parameter. The numbers begin at 0. If no item is selected, it is –1.

Comments The *ListBox* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

When syntax A is used, the *text$* parameter is a string containing the selections for the list box. This string must be defined, using a *Dim* statement, before the *Begin Dialog* statement is executed.

Where *dimname* is the name of a *String* variable defined in a *Dim* statement, *listchoice* is the text that appears as a selection in the list box, and *Chr$(9)* is the function call that produces a tab character. Note that multiple selections can be specified in *text$* by separating the list choices with tab characters, as shown above.

Example The parameters in the *text$* string are entered as shown in the following example.

```
dimname = "listchoice" + Chr$(9)+ "listchoice" + Chr$(9) +
"listchoice" +
Chr$(9)...,.Selection_Holder
```

Load

Replaces any previous connection information in a dataset control with information from the file that is specified by the *Filename$* parameter. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see "Using the DataSet Control" on page 215.

Syntax [object].Load Filename$

Parameters Filename$—The name of a file from which the data set control object will read.

Returns 0 on success. If it is not zero, then the *Error$* property contains text that describes the error.

Comments The file must be created with the *Save* method and can have any extension.

Example MyData.Load("c:\RPTSMITH\MyData.DSC")

LoadMacro

Syntax [dataset].LoadMacro (FileName$,[MacroType%], [IgnoreDialog%])

Definition The LoadMacro command allows you to load a macro into the active macro list from a .MAC file. If only the filename parameter is provided then the macro will be loaded into the dataset controls macro collection. This means that if you have associated your dataset control with a report using the SetFromActive command, then this method will load the macro into that report's macro collection regardless of whether it is the active report or not. If the MacroType argument is specified, and if it is 1, then the macro will be loaded as a global macro. If the Macro Commands dialog box is up then the macro will be loaded into the dialog box as if the load button were used. If the IgnoreDialog parameter is specified, and if it is non-zero, then the modification will be made to the control object and the dialog will not be updated.

Note: If the dialog box is editing the same list of macros that the dataset control is associated with, then the macro loaded may be unloaded when the Macro Commands dialog box is closed if the IgnoreDialog parameter is non-zero.

Parameters FileName$ — Name of the macro file to load. If the extension is omitted then the default extension of .MAC will be used.

MacroType% — Specifies the what collection the macro will be loaded into. If the parameter is 0 or not specified the macro will be loaded into the currently active report. If no report is loaded then the macro will be loaded as a global. If the parameter is 1 then the macro will be loaded as a global only.

IgnoreDialog% — If this parameter is specified and is non-zero, then the macro dialog will not be updated by this method. Error codes are:

1 Invalid File Name

2 A macro with the same name is already in the active list and must be removed before this macro may be loaded

LoadReport

Loads a specified report.

Syntax LoadReport Filespec$, InitString$

Parameters Filespec$—Path and name of the report (.RPT file) you want to run.

InitString$—You can use the *InitString* parameter to set report variables before SQL is executed. Report variables set in this manner will *not* prompt the user to enter values. This is the format for the *InitString$* parameter:

@Report_Variable1=<Value1>,@ReportVariable2=<Value2>,...

Specify the report variables you would like to set in the report. The @ symbols in the example shown below are optional.

Returns Non-zero on error.

Comments Enter the full path name to the .RPT file you want the macro to load, then make it the active report. Report variables *are* case-sensitive, so use care when entering the *InitString$* parameter.

Example LoadReport "c:\summary" , "@RV1=<40>,@RV2=<Smith>"

Note: There must be a space before and after the comma that follows \summary".

Loc

Syntax Loc(filenumber%)

Returns The Loc function returns the current offset within the open file specified by filenumber%. For files opened in Random mode, Loc returns the number of the last record read or written. For files opened in Append, Input, or Output mode, Loc returns the current byte offset divided by 128. For files opened in Binary mode, Loc returns the offset of the last byte read or written.

Comments Filenumber% is an integer expression identifying the open file to query. The filenumber% is the number used in the Open statement of the file.

Lock, Unlock statements

Syntax Lock [#]filenumber% [, [start&] [To end&]]

Unlock [#]filenumber% [, { record& | [start&] To end& }]

Comments The Lock and Unlock statements are used to control access by other process to some or all of an open file.

Filenumber% is an integer expression identifying the open file to Lock or Unlock. The filenumber% is the number used in the Open statement of the file.

Start is a Long integer that specifies the offset of the first record or byte to Lock or Unlock.

End is a Long integer that specifies the offset of the last record or byte to Lock or Unlock.

For Binary mode, start and end are byte offsets. For Random mode, start and end are record numbers. If start is specified without end, then only the record or byte at start is locked. If To end is specified

without start, then all records or bytes from record number or offset 1 to end are locked.

For Input, Output, and Append modes, start and end are ignored and the whole file is locked.

Lock and Unlock always occur in pairs with identical parameters. All locks on open files must be removed before closing the file or unpredictable results will occur.

Lof

Determines the length of a file in bytes. Used only as a function.

Syntax Lof (filenumber%)

Parameters filenumber%—An integer expression identifying the open file from which the file length is read.

Returns The length in bytes of the file specified by *filenumber%*.

Comments The *filenumber%* is the number used in the *Open* statement of the file.

Example MsgBox = "c:\Sample.txt is"; Lof (7); "Bytes Long"

Log

Determines the natural logarithm of an expression. Used only as a function.

Syntax Log(numeric expression)

Parameters numeric-expression—Any number or numerical expression for which you want to find the natural logarithm.

Returns The natural logarithm of *numeric expression*.

Comments The return value is single-precision for an integer or single-precision numeric expression, or double precision for a long or double-precision numeric expression.

Example MsgBox "The natural log of 12,000 is:" + Str$ (log(12,000))

Lset statement

Syntax A Lset string$ = string-expression

Syntax B Lset variable1 = variable2

Comment If the first form of Lset statement is used and string$ is shorter than string-expression, Lset copies leftmost character of string-expression into string$. The number of characters copied is equal to the length of string$.

If string is longer than string-expression, all characters of string-expression are copied into string$ filling it from left to right. All leftover characters of string$ are replaced with spaces.

The second form of Lset is used to assign one user-defined type variable to another. The number of characters copied is equal to the length of the shorter of variable1 and variable2.

Lset cannot be used to assign variables of different user-defined types if either contains a variant or a variable-length string.

LTrim$

Returns a copy of the source string with all leading spaces removed. Used only as a function.

Syntax LTrim$ (string$)

Parameters string$—The string from which to strip leading spaces.

Returns A copy of the source string, with all leading space characters removed. The dollar sign ($) in the function name is optional. If it is specified, the return type is string. If it is omitted, the function typically returns a variant of vartype 8 (string). If the value of expression is null a variant of vartype 1 (null) is returned.

Comments *LTrim$* accepts expressions of type string and any type of expression including numeric values and converts the input value to a string.

Example ‘The following example results in “October 7.”

```
LTrim$("    October 7")
```

Mid

Replaces part of a string with another string.

Syntax Mid (string$, position%[, length%]) = subst-string$

Parameters string$—The string expression into which the substring is placed.

position%—The character position in a string where substring is inserted.

length%—The number of characters for the substring to insert at the string.

subst-string$—A string to insert in another string.

Returns Replaces the specified substring in *string$* with *subst-string$*.

Comments If the l*ength%* parameter is left out, or if there are fewer characters in a string than specified in *length%*, then *Mid$* replaces all the characters from the *position%* to the end of the string. If *position%* is larger than the number of characters in the indicated *string$*, then *Mid$* appends *subst-string%* to *string$*.

The index of the first character in a string is 1.

Example
```
A$ = "One rotten day"
Mid$ (A$, 5, 6) = "Happy"
```

Will leave *A$* equal to "One happy day".

Mid$

Returns a substring of a specified length% from a source expression, starting with the character at the specified position%. Can be used only as a function.

Syntax Mid$ (string$, position%[, length%])

Parameters string$—The string expression from which a sub-string is taken.

position%—The start of the sub-string.

length%—The length of the sub-string.

Returns A substring of *length%* from a source *string$*, starting with the character at the specified *position%*.

The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function typically returns a variant of vartype 8 (string). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments If the *length%* parameter is left out, or if there are fewer characters in a string than specified in *length%*, then *Mid$* returns all the characters from the *position%* to the end of the string. If *position%* is larger than the number of characters in the indicated *string$*, then *Mid$* returns a null string.

The index of the first character in a string is 1.

To modify a portion of a string value, see “Mid” on page 363.

Example The following example returns “press.”

```
Return_value = Mid$ ("expression", 3,4)
```

Minute

Syntax Minute(expression)

Returns The Minute function returns the minute component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments The Minute function returns an integer between 0 and 59, inclusive.

It accepts any type of *expression*, including strings, and will attempt to convert the input value to a date value.

MkDir

Makes a new directory.

Syntax MkDir pathname$

Parameters pathname$—A string expression identifying the new default directory.

Comments The syntax for pathname$ is:

```
[drive:] [\] directory [\directory]
```

The drive parameter is optional. If omitted, *MkDir* makes a new directory on the current drive. The directory parameter is a directory name.

Example The following example creates a directory called “2ndQrt” under the “RptSmith” directory.

```
MkDir "c:\RptSmith\2ndQrtr"
```

Month

Syntax Month(expression)

Returns The Month function returns the month component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments The Month function returns an integer between 1 and 12, inclusive.

It accepts any type of *expression*, including strings, and attempts to convert the input value to a date value.

Msgbox

Displays a message in a dialog box.

Syntax A (function)Msgbox(message$[,type%[, caption$]])

Syntax B (statement)MsgBox message$[,type%[,caption$]]

Parameters message$—A string to display to the user.

type%—Governs the icons and buttons that are displayed in the dialog box. This parameter is the sum of values describing the number and type of buttons that appear, the icon style, and the default button. One selection can be made from each group. If *type%* is omitted, a single OK button appears.

Group 1: Buttons

0OK only
1OK, Cancel
2Abort, Retry, Ignore
3Yes, No, Cancel
4Yes, No
Retry, Cancel

Group 2: Icons

16Critical Message (STOP)
32Warning Query (?)
64Information Message (i)

Group 3: Defaults

0First button
256Second button
512Third button

caption$—Appears as the message-box title.

Returns When used as a function: an integer value indicating the button chosen.

The return values for the *Msgbox* function are:

1 OK
2 Cancel
3 Abort
4 Retry
5 Ignore
6 Yes
7 No

Comments The message displayed must be no more than 1024 characters long. A message string greater than 255 characters without intervening spaces is truncated after the 255th character. Once the user has chosen a button, *MsgBox* returns a value indicating the user's choice. When using MsgBox as a statement, you will usually want to include only an OK button (and possibly a Cancel button) together with an icon, as a return value (which button was chosen) will not be used.

Example Button_Pressed=Msgbox("Please pick Yes or No",4,"Title")

Name

Renames a file. It can also be used to move a file from one directory to another.

Syntax Name oldfilename$ As newfilename$

Parameters oldfilename$,newfilename$—String expressions that designate, respectively, the file to rename and the new name for that file.

Comments A path can be part of the *filename$*. If the paths are different, the file is moved to the new directory.

If the file *oldfilename$* is open, BASIC generates an error message. A file must be closed in order to be renamed. If the file *newfilename$* already exists, BASIC generates an error message.

Example Name "win.ini" as "win.bak"

Name$ (dataset object and report object)

Returns the current name for the current situation. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Name$ = [stringexpression]

Comments This property is read/write at run time.

Example MyData.Name$="Susan's Data"

New Operator

Syntax Set objectVar = New className

Dim objectVar As New className

Comments In the Set statement, New allocates and initializes a new object of the named class.

In the Dim statement, New marks the object variable so that a new object will be allocated and initialized when the object variable is first used. If the object variable is not referenced, then no new object will be allocated.

Note: An object variable that was declared with New will allocate a second object if the variable is Set to Nothing and referenced again.

Nothing

Syntax Set variableName = Nothing

Returns An object value that doesn't refer to an object

Comments Nothing is the value object variables have when they do not refer to an object, either because the have not been initialized yet or because they were explicitly Set to Nothing.

```
If Not objectVar Is Nothing then
   objectVar.Close
   Set objectVar = Nothing
End If
```

Now

Syntax Now( )

Returns The Now function returns the current date and time.

Comments The Now function returns a variant of vartype 7 (date) that represents the current date and time according to the setting of the computer's system date and time.

Null

Syntax Null

Comments Null returns a variant value set to the null value. Null is used to explicitly set a variant to the null value

variableName = Null

Note that variants are initialized by Basic to the empty value, which is different from the null value.

See also IsNull and IsEmpty.

Object Class

Syntax dim variableName As Object

Comments Object is a class that provides access to Ole2 automation objects. To create a new Object, first dimension a variable, and then Set the variable to the return value of CreateObject.

Dim Ole2 As Object

Set Ole2 = CreateObject("spoly.cpoly")

Ole2.reset

Oct$

Returns an octal representation of a number. Used only as a function.

Syntax Oct$(numeric-expression)

Parameters numeric-expression—A decimal number to convert to octal.

Returns An octal representation of a numeric-expression, as a string. If the numeric expression is an integer, the string contains up to six octal digits; otherwise, the expression is converted to a long integer, and the string can contain up to 11 octal digits.

The dollar sign ($) in the function name is optional. If specified,the return type is string. If omitted, the function typically returns a variant of vartype 8 (string).

Comments The octal value is returned as a string of characters.

Example MsgBox "The octal representation of 175 is " + oct$(175)

OKButton

Specifies the position and size of an OK button.

Syntax OKButton x, y, dx, dy [,.id]

Parameters x, y—Set the position of the OK button relative to the upper left corner of the dialog box. See “Begin Dialog...End Dialog” on page 270.

dx, dy—Set the width and height of the button.

id—An optional identifier used by the dialog statements that act on this. A *dy* value of 14 typically accommodates text in the system font.

Comments The *OKButton* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

Example OKButton 10,10,90,20

On Error

Enables an error-handling routine, specifying the location of that routine within procedure.

Syntax On [Local] Error {GoTo label [Resume Next] GoTo 0}

Parameters Local—Keyword allowed in error-handling routines at the procedure level. Used to ensure compatibility with other variants of BASIC.

GoTo label—Enables the error-handling routine that starts at *label*. If the designated label is not in the same procedure as the *On Error* statement, BASIC generates an error message.

Resume Next—Establishes that when a run-time error occurs, control is passed to the statement which immediately follows the statement in which the error occurred. At this point, the *Err* function can be used to retrieve the error code of the run-time error.

GoTo 0—Disables any error handler that has been enabled.

Comments *On Error* can also be used to disable an error-handling routine. Unless an *On Error* statement is used, any run-time error will be fatal. ReportBasic will terminate the execution of the program.

When it is referenced by an *On Error GoTo label* statement, an error-handler is enabled. After this enabling occurs, a run-time error results in program control switching to the error-handling routine and “activating” the error handler. The error handler remains active from the time the run-time error has been trapped until a *Resume* statement is executed in the error handler.

If another error occurs while the error handler is active, ReportBasic searches for an error handler in the procedure which called the current procedure (if this fails, the macro language looks for a handler

belonging to the caller's caller, and so on). If a handler is found, the current procedure terminates, and the error handler in the calling procedure is activated.

It is an error (No Resume) to execute an End Sub or End Function statement while an error handler is active. The *Exit Sub* or *Exit Function* statement can be used to end the error condition and exit the current procedure.

Example On Error Goto ErrorProc

Open

Enables I/O to a file or a device.

Syntax Open filename$ For mode As [#] filenumber% [Len=reclen]

Parameters filename$—A string expression specifying the file to open.

mode—Specifies one of the following: Input (sequential input mode), Output (sequential output mode), or Append (sequential output mode).

filenumber—An integer expression with a value between 1 and 255.

reclen—Specifies the record length for files opened in random mode. It is ignored for all other modes.

Comments A file must be opened before any input/output operation can be performed on it.

The *FreeFile* function can be used to return the next available filenumber.

Example Open "Data.txt" for input as 1

Option Base

Specifies the default lower bound to be used for array subscripts.

Syntax Option Base lowerBound%

Comments The *lowerBound* must be either 0 or 1. If no *Option Base* statement is specified, the default lower bound for array subscripts is 0.

The *Option Base* statement is not allowed inside a procedure, and must precede any use of arrays in the module. Only one Option Base statement is allowed per module.

Example Option Base 1

OptionButton

Used to define the position and text associated with an option button. Because option buttons are used to specify alternative and mutually exclusive options, there must be at least two *Option Button* statements. They are used in conjunction with the *OptionGroup* statement.

Syntax OptionButton x, y, dx, dy, text$

Parameters x, y—Set the position of the button relative to the upper left corner of the dialog box.

dx, dy—Set the width and height of the button. A *dy* value of 12 typically accommodates text in the system font.

text$—Contains the caption that appears to the right of the option button icon. If the width of this string is greater than *dx*, trailing characters are truncated.

Comments The *OptionButton* statements can be used only between a *Begin Dialog* and an *End Dialog* statement.

If you want to include accelerator characters so that the option selection can be made from the keyboard, the character must be preceded with an ampersand (&).

Example OptionButton 10,10,110,20,"Sales Reports"

Option Compare

Syntax Option Compare { Binary | Text }

Comments The Option Compare statement specifies the default method of string comparison. Binary comparisons are case sensitive. Text comparisons are case insensitive. Binary comparisons compare strings based upon the ANSI character set. Text comparison are based upon the relative order of characters as determined by the country code setting for your system.

Option Explicit

Syntax Option Explicit

Comments The Option Explicit statement specifies that all variables in a module must be explicitly declared. By default, BASIC will automatically declare any variables that do not appear in a Dim, Global, Redim, or Static statement. Option Explicit causes such variables to produce a "Variable Not Declared" error.

OptionGroup

Used in conjunction with *OptionButton* statements to set up a series of related options.

Syntax OptionGroup.field

Parameters .field—The name of the dialog box field that contains the index of the selected options button.

Comments The *OptionGroup* statement begins definition of the option buttons and establishes the dialog-record field that contains the current option selection. *Field* contains a value 0 when the choice associated with the first *OptionButton* statement is selected, a value of 1 when the choice associated with the second *OptionButton* statement is chosen, and so on.

The *OptionGroup* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

Example OptionGroup.selected_option

Owner$

Returns the current owner for the current situation. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Owner$

Comments Before the current owner can be retrieved from a DataSet object, a connection that has owners must be made.

Example CurrentOwner$=MyData.Owner$

Page Property

Syntax [object].Page

Definition Gets or sets the current report page.

Returns Not applicable.

Example

```
Sub GetPageInfo()
dim MyReport as Report
MyReport.SetFromActive
ActivePage$ = Str$( MyReport.Page)
NumberOfPages$ = str$(MyReport.TotalPages)
End Sub
```

PasswordBox$

Syntax PasswordBox[$](prompt$ [,[title$] [,[default$] [,xpos%, ypos%]]])

Returns The PasswordBox$ function returns a string entered by the user. The user's type-in will not be echoed.

The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function will return a variant of vartype 8 (string).

Comments The PasswordBox$ function displays a dialog box containing a prompt. Once the user has entered text, or made the button choice being prompted for, the contents of the box are returned.

The prompt$ argument is a string expression containing the text to be shown in the dialog box. The length of prompt$ is restricted to 255 characters. This figure is approximate and depends on the width of the characters used. Note that a carriage return and a line-feed character must be included in prompt$ if a multiple-line prompt is used.

The title$ argument is the caption that appears in the dialog box's title bar. Default$ is the string expression shown in the edit box as the default response. If either of these arguments is omitted, nothing is displayed.

The xpos% and ypos% arguments are numeric expressions, specified in dialog box units, that determine the position of the dialog box. Xpos% determines the horizontal distance between the left edge of the screen and the left border of the dialog box. Ypos% determines the horizontal distance from the top of the screen to the dialog box's upper edge. If these arguments are not entered, the dialog box is centered roughly one third of the way down the screen. A horizontal dialog box unit is 1/4 of the average character width in the system font; a vertical dialog box unit is 1/8 of the height of a character in the system font.

Note: If you wish to specify the dialog box's position, you must enter both of these arguments. If you enter one without the other, the default positioning is set.

Once the user presses Enter, or selects the OK button, PasswordBox$ returns the text contained in the input box. If the user selects Cancel, the PasswordBox$ function returns a null string.

Print

Outputs data to the specified *filenumber%*.

Syntax Print [# filenumber%,] expressionlist [{ ; | , }]

Parameters filenumber% —An integer expression identifying the print destination.

expressionlist—The values that are printed. The parameter can contain numeric or string expressions.

Comments If the *expressionlist* is omitted, a blank line is written to the file.

Filenumber% is optional. If this parameter is omitted, the *Print* statement outputs data to the screen.

Expressions are separated by either a semicolon (;) or a comma (,). A semicolon indicates that the next value should appear immediately after the preceding one without intervening white space. A comma indicates that the next value should be positioned at the next print zone. Print zones begin every 14 spaces.

The optional {;|,} parameter at the end of the *Print* statement determines where output for the next *Print* statement to the same output file should begin. A semicolon places output immediately after the output from this *Print* statement on the current line; a comma starts output at the next print zone on the current line. If neither separator is specified, a CR-LF (carriage return-line feed) pair is generated and the next *Print* statement prints to the next line.

The *Print* statement supports only elementary BASIC data types.

Example Print #1, MyData$, Index, Amount

PrintReport

Prints the specified pages of the active report to the specified printer.

Syntax PrintReport StartingPage%, EndingPage%, Printer$, Port$, Driver$

Parameters StartingPage%—The number (inclusive) of the report page on which you would like to start printing.

EndingPage%—The number (inclusive) of the last page you would like to print.

Printer$—The name of the printer to which you would like to print the report.

Port$—The correct port specification for the designated printer.

Driver$—The correct driver specification.

Copies%—The number of copies to print.

Returns Non-zero on error.

Comments To print all report pages, use 0 for the start and end page parameters. To use the default printer, use null strings for the *Printer$, Port$,* and *Driver$* parameters. To specify a printer, see the Devices section of your WIN.INI file. You'll see printers listed in this format:

```
Printer=Driver,Port1,Port2, ...
```

When you specify the printer, driver and port, use the same text you see in the WIN.INI file.

When you use this command as a function (rather than a statement), you must enclose its parameters within parentheses. For more information on the differences between functions and statements, refer to "Using the DataSet Control" on page 215.

Example PrintReport 1,5, "HPLaserJet 4/4M", "LPT3", "HPPCL5E"

Put statement

Syntax Put [#] filenumber%, [recordnumber&], variable

Comments Put is used to write a variable to a file opened in Random or Binary mode.

Filenumber% is an integer expression identifying an open file to which to write. See the Open statement for more details.

Recordnumber& is a Long expression containing the number of the record (for Random mode) or the offset of the byte (for Binary mode) at which to start writing. Recordnumber is in the range 1 to 2,147,483,647. If recordnumber is omitted, the next record or byte is written. Note that the commas are required, even if no recordnumber is specified.

Variable is the name of the variable from which Get writes file data. Variable can be any variable except Object, Application Data Type or Array variables (single array elements may be used).

For Random mode, the following apply:

Blocks of data are written to the file in chunks whose size is equal to the size specified in the Len clause of the Open statement. If the size of variable is smaller than the record length, the record is padded to the correct record size. If the size of variable is larger than the record length, an error occurs.

For variable length Strings variables, Put writes two bytes of data that indicate the length of the string, then writes the string data.

For Variant variables, Put writes two bytes of data that indicate the type of the Variant, then it writes the body of the variant into the variable. Note that Variants containing strings contain two bytes of type information, followed by two bytes of length, followed by the body of the string.

User defined types are written as if each member were written separately, except no padding occurs between elements.

Files opened in Binary mode behave similarly to those opened in Random mode except:

Put writes variables to the disk without record padding.

Variable length Strings that are not part of user defined types are not preceded by the two byte string length.

Randomize

Seeds the random number generator.

Syntax Randomize [numeric-expression%]

Parameters numeric-expression%—An integer value between –32768 and 32767.

Comments If no *numeric-expression%* parameter is given, BASIC uses the *Timer* function to initialize the random number generator.

Example Randomize Timer

Recalc

Recalculates the currently active report so as to reflect changes made to report variables, or changes made with an associated dataset control object.

Syntax Recalc

Example The following example sets the starting date report variable to today and recalculates.

```
SetRepVar("StartDate",Date$)
Recalc
```

Recalc (dataset object and report object)

Recalculates the data for this dataset object. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Recalc

Comments Because this command is associated with a dataset control, any changes to the result are not reflected in associated reports until a report-level recalculation is performed. This command is generally used with dataset control functions that do not have associated reports. When used as a method of the report object, this command affects only the associated report, regardless of which is the currently active report.

Example MyData.Recalc

Record

Returns the current record in the dataset. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Record

Comments Before the number of records can be retrieved from a dataset object, a connection must first be made, links must be set, and a *Commit* or *Recalc* must be performed successfully.

Example If MyData.Record = 1 then MsgBox "You are at the beginning"

RecordCount (dataset object)

Returns the total number of records in the dataset. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RecordCount$

Comments Before the number of records can be retrieved from a dataset object, a connection must be made, links must be set, and a commit or recalc must be successfully performed.

Example TotalRecords=MyData.RecordCount$

RecordCount

Gets the total number of records in the data set that belongs to the currently active report.

Syntax RecordCount

Returns The number of records in the currently active report or 0 if no active report exists.

Comments This command is useful when writing macros that step through the data in a report.

Example The following code fragment counts all of the customers from the city of Spokane in a customer database and displays the result.

```
GetRandom 1
For X = 1 to RecordCount
If Field$("CITY") = "Spokane" then Count = Count + 1
GetNext
Next X
MsgBox "The total number of customers located in Spokane are: "+
Str$(Count)
```

ReDim

Changes the upper and lower bounds of a dynamic array’s dimensions.

Syntax ReDim [Preserve] variableName (subscriptRange,...) [As[New] type],...

Parameters variableName—An array to re-dimension.

subscriptRange—New dimensions.

Comments Memory for the dynamic array is reallocated to support the specified dimensions, and the array elements are reinitialized. *ReDim* cannot be used at the module level—it must be used inside of a procedure.

The *Preserve* option is used to change the last dimension in the array while maintaining its contents. If *Preserve* is not specified the contents of the array will be reinitialized. Numbers are set to zero. String variants are set to empty.

A dynamic array is normally created by using *Dim* to declare an array without a specified *subscriptRange*. The maximum number of dimensions for a dynamic array created in this fashion is 8. If you need more than 8 dimensions, you can use the *ReDim* statement inside of a procedure to declare an array which has not previously been declared using *Dim* or *Global*. In this case, the maximum number of dimensions allowed is 60.

The available data types for arrays are: numbers, strings, and records. Arrays of arrays, dialog box records, and ADTs are not supported.

If the *As* clause is not used, the type of the variable can be specified by using a type character as a suffix to the name. The two different type-specification methods can be intermixed in a single *ReDim* statement (although not on the same variable).

The *ReDim* statement cannot be used to change the number of dimensions of a dynamic array once the array has been given dimensions. It can change only the upper and lower bounds of the dimensions of the array. The *LBound* and *UBound* functions can be used to query the current bounds of an array variable's dimensions.

Do not use the *ReDim* statement on an array in a procedure that has received a reference to an element in the array in an parameter. The result is unpredictable.

The subscript*Range* is of the format:

[startSubscript To] endSubscript

If *startSubscript* is not specified, 0 is used as the default. The *Option Base* statement can be used to change the default.

Example

```
'dimension to a 2x13 array
Redim DynaList (3 to 5, 7 to 20)
```

Rem

Inserts a comment (or “remark”) in a BASIC program. Everything from *Rem* to the end of the line is ignored, but you must preface each new line of code with the *Rem* statement or it will be treated as executable code.

Syntax Rem arbitrary text

Parameters arbitrary text—Because the remainder of the line following the *Rem* statement is ignored for purposes of code execution, you are free to enter any text here, including all alphanumeric and extended characters.

Comments The single quote (’) can also be used to initiate a comment. Metacommands (e.g., CSTRINGS) **must** be preceded by the single quote comment form.

Example
```
Rem this comment line is not compiled.
'This statement is also not compiled. It is a comment.
```

RemoveGroup

Removes a grouping criterion from a report. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RemoveGroup Level

Parameters Level—Specifies the grouping level you want to remove, where 0 is the entire report group, 1 is the primary grouping criterion, 2 is the secondary grouping criterion, and so forth.

Returns 0 on success, a non-zero value on error.

Comments If an invalid index is specified, a null string is returned and the *Error$* property is set to indicate the error.

Example MyData.RemoveGroup 1

RemoveSort

Removes the sorting criteria at the given level. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RemoveSort Level

Parameters Level—Specifies the grouping level for which you want to remove sorting criteria, where 0 is the entire report group, 1 is the primary grouping criterion, 2 is the secondary grouping criterion, and so forth.

Returns 0 on success, a non-zero value on error.

Comments Valid values for the *Level* parameter are 1 to the number of current sorting criteria.

Example MyData.RemoveSort 1

RemoveSummary

Returns a summary field from a report. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RemoveSummary Level, Index

Parameters Level—Specifies the grouping level from which you want you want to remove summary information, where 0 is the entire report group, 1 is the primary grouping criterion, 2 is the secondary grouping criterion, and so forth.

Index—Matches the order in which the tables were *originally* added.

Returns If an invalid index is specified, a null string is returned and the *Error$* property is set to indicate the error.

Example MyData.RemoveSummary 1,2

RemoveTable

Removes the table at the specified index. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RemoveTable Index

Parameters Index—Matches the order in which the tables were originally added.

Returns 0 on success, a non-zero on error.

Comments The table at any given index can be determined using the *GetTable* function.

Example MyData.RemoveTable 2

RemoveTableLink

Removes the table link for the specified index from the dataset control object. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].RemoveTableLink, Index

Parameters Index—The index of the link for which to retrieve information.

Comments If an invalid index is specified, a null string is returned and the Error$ property is set to indicate the error.

Example MyData.RemoveSummary 1,2

ReplaceTable

Replaces one table in a dataset with another. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].ReplaceTable Table$, Database$, NewTable$, NewDataBase$

Parameters Table$—Path and file name of the table to be replaced.

Database$—Database (if applicable) containing the table to be replaced.

NewTable$—The name of the replacement table.

NewDataBase$—Database (if applicable) containing the replacement table.

Returns 0 on success, a non-zero value on error.

Comments Database and table names *must* be entered entirely in upper-case characters. Any fields that do not have a direct match in the old table are excluded from the dataset, and those fields on the report surface are removed or show *#ref*.

For database servers this method takes the form: Owner.TableName. For local databases or servers that don't require that a database be specified, the *Database$* parameter should be set to a null string.

Example MyData.ReplaceTable "DBO.EMP","HR","DBO.EMP2", "NEW_HR"

ReportType Property

Syntax [object].ReportType

Definition Indicates the type of report selected. This property is read only.

Returns ReportType

Number	Type
0	columnar
1	label
2	crosstab
3	form

Example
```
Sub GetRepDatType()
dim MyDialog as NewReportdialog
MyDialog.rundialog
MsgBox Str(MyDialog.ReportType)
End Sub
```

Reset

Closes all disk files that are open and writes any data still remaining in the operating system buffers to disk.

Syntax Reset

Example Reset

Resume

Halts an error-handling routine.

Syntax Resume Next
Resume label
Resume [0]

Parameters Resume Next—When used, control is passed to the statement which immediately follows the statement in which the error occurred.

Resume label—When used, control is passed to the statement which immediately follows the specified label.

Resume [0]—When used, control is passed to the statement in which the error occurred.

Comments The location of the error handler that caught the error determines where execution resumes. If an error is trapped in the same procedure as the error handler, program execution resumes with the statement that caused the error. If an error is located in a different procedure from the error handler, program control reverts to the statement that last called out the procedure containing the error handler.

Example The following example goes to the *Reset_Proc* procedure after an error.

```
Resume Reset_Proc
```

ResumeEvent

Enables a macro linked to an event to determine whether the event should be executed or aborted.

Syntax ResumeEvent ResumeCode%

Parameters ResumeCode%—The code which indicates the event:

0 Abort the event to which this macro is linked.

1 Perform the event as usual. (Default)

Comments This only applies to certain events. For most events, 0 means abort and 1 means proceed. Some events recognize more codes. See the individual event for more information.

Example The following example stops an event.

```
ResumeEvent 0
```

ReturnCode Property

Syntax [object].ReturnCode

Definition Indicates if the user selected OK or Cancel to exit the Dialog. This property is read only.

Returns ReturnCode

Number	Meaning
1	User Exited with OK
2	User Exited with Cancel

Example

```
Sub GetReturnCode()
dim MyDialog as newReportDialog
MyDialog.rundialog
MsgBox Str(MyDialog.ReportType)+ Chr$(13) +
MyDialog.ReturnCode$
End Sub
```

RGB

As a statement, sets the color used by the *FieldFont* command to set or change field text color. As a function, returns a long-integer value for the color used by the *FieldFont* command for its color assignment.

Syntax RGB (Red%, Green%, Blue%)

Parameters Red%, Green%, Blue%—The intensity of red, green, and blue in color.

Returns A long integer representing the color you've specified.

Comments This command uses three integers from 0 to 255. The first integer designates the intensity of red, the second number the intensity of green, and the third the intensity of blue. This provides 16.5 million color combinations, and Windows 95 then uses the closest match to the specified color.

Example This example tells the *FieldFont* command to use the color red.

```
Dim Red as Integer
Red = RGB (255, 0, 0)
```

Tip Use the Custom Color option in the Windows 95 Control Panel to select a color. Take note of the RGB settings for the color you select. Then use these values in the RGB command to get the color you need.

Right$

Returns a string of a specified length. Used only as a function.

Syntax Right$(string$, length%)

Parameters string$—The string from which to copy characters.

length%—The number of characters to copy.

Returns A string of a specified length copied from the right-most length characters from the *string$*.

Comments If the length of *string$* is less than *length%*, *Right$* returns the whole string.

Example The following example returns “RptSmith.”

```
Return_Var$ = Right$("c:\RptSmith",8)
```

RmDir

Removes a directory.

Syntax RmDir pathname$

Parameters pathname$—A string expression identifying the directory to remove.

Comments The syntax for *pathname$* is: [drive:] [\] directory [\directory]

The drive parameter is optional. The directory parameter is a directory name. The directory to be removed must be empty, except for the working (.) and parent (..) directories.

Example

```
'Remove a temporary directory under ReportSmith
RmDir "c:\RptSmith\Temp"
```

Rnd

Generates a random number between 0 and 1. Used only as a function.

Syntax Rnd [(number!)]

Parameters The following summarizes the valid values for *number*:

< 0	Same random number for a given number
> 0	Next random number
= 0	Last random number
Omitted	Next random number

Returns A single-precision random number between 0 and 1.

Comments The same sequence of random numbers is generated whenever the program is run, unless the random number generated is re-initialized by the *Randomize* statement.

Example The following example generates a random number between 1 and 100

```
x = (RND*100) +1
```

Rset

Syntax Rset string$ = string-expression

Comment Rset is used to right-align string-expression within string$. If string$ is longer than string-expression, the left-most characters of string$ are replaced with spaces.

If string$ is shorter than string-expression, only the leftmost characters of string-expression are copied.

Rset cannot be used to assign variables of different user-defined types.

RTrim$

Removes trailing spaces.

Syntax RTrim$(string$)

Parameters string$—A string from which to remove trailing spaces.

Returns A copy of the source string, with all trailing space characters removed.

Example This example trims trailing spaces from the string “John “and the string “ Doe “, then concatenates the two to produce “John Doe”.
RTrim$ ("John ") + RTrim$(“Doe ") = "John Doe"

Run Dialog Method

Syntax [object].RunDialog

Definition This command executes the Create A New Report dialog box. (Not necessary if this appears in a different color.)

Returns This method returns 0 on success. Non-zero means that there was a problem displaying the dialog box. This value should not be confused with the return code property which indicates how the end user closed the dialog box.

Example

```
dim MyDialog as NewReportdialog
' Run The dialog
MyDialog.rundialog
```

RunMacro

Executes a ReportBasic macro from within another macro.

Syntax RunMacro Macro$, Parameters$

Parameters Macro$—Specifies the macro to run.

Parameters$—The parameters defined in your macro, if they exist. If your macro contains no parameters, just use a null string for this parameter (“”).

Returns 0 (zero) if the specified macro is found and successfully executed, or non-zero if the macro cannot be found or successfully executed.

Comments The macro language first looks for an active *global* macro that matches the name and then searches for active *report* macros matching the specified macro name. You can also specify the file name of a .MAC (stored macro) file. To make sure the correct .MAC file is executed, specify the full path of the .MAC file.

Example
```
RunMacro "c:\rptsmith\macros\greeting.mac",""
RunMacro "my_macro","a=5"
```

Save

Specifies a file name under which the contents of the dataset object should be saved. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Save Filename$

Parameters Filename$—The file to which to save dataset contents.

Returns 0 on success. If the return value is not zero, then the *Error$* property contains text that describes the error.

Example MyData.Save "c:\MyData.Dat"

SaveReport

Syntax SaveReport ([FileName$],[ExportType%])

Definition This command allows you to save the active report under its last saved name, save it as a new name and path, or export it to one of several export field types.

Parameters FileName$ — The name of the file to which you want tosave the report. If this Argument is omitted, then the function will attempt to save the file under the same name under which it was last saved. If the file has never been saved and no filename is provided the function will return an error. You may not save a report with one of ReportSmith's default names (REPORT1.RPT, REPORT2.RPT).

ExportType% — Specifies the type of file that ReportSmith saves. If it is omitted, ReportSmith saves it in its default file format. The valid Codes are:

Code	File Type	Default Extension
0	Standard ReportSmith Report	.RPT
1	Report Query File	.RQF
2	Excel Spread Sheet	.XLS
3	Text File	.TXT
4	Lotus Spread Sheet	.WKS
5	Comma Delimited Text	.CSV
7	Data Interchange Format	.DIF
8	Quattro Pro	.WKQ

Error codes:

1 No Active report to save

2 Cannot save unnamed file

3 General Error Saving file

4 Invalid File name

5 Cannot overwrite exported file

6 Invalid Export Code

4002 File not found

4003 Path not found

4004 Too many open files

4005 Access denied

4008 Not enough memory

4010 Bad environment

4011 Bad format

4012 Invalid access

4013 Invalid data

4014 Invalid drive

4018 No more files

4019 Write protect error

4026 Not MS-DOS disk

4031 General failure

4032 Sharing violation

Second

Syntax Second(expression)

Returns The Second function returns the second component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null, a variant of vartype 1 (null) is returned.

Comments The Second function returns an integer between 0 and 59, inclusive.

It accepts any type of *expression*, including strings, and attempts to convert the input value to a date value.

Seek

As a function, determines the current position in a file. As a statement, sets the position within a file for the next read or write.

Syntax A (function)Seek (filenumber%)

Syntax B (statement)Seek [#] filenumber%, position&

Parameters filenumber%—An integer expression identifying the open file from which to read the file position.

position&—A numeric expression that indicates (in a *Seek* statement) where the next write or read occurs. Value must be between 1 and 2,146,483,647.

Returns As a function, returns the current file position for the file specified by *filenumber%*. For files opened in Random mode, *Seek* returns the number of the next record to be read or written. For all other modes, Seek returns the file offset for the next operation. The first byte in the file is at offset 1, the second byte is at offset 2, and so on. The return value is a Long.

Comments See “Open” on page 371 for more details. If you write to a file after seeking beyond the end of the file, the file length is extended. BASIC returns an error message if a *Seek* operation is attempted which specifies a negative or zero position.

Example Position1 = Seek(1)

Select Case

Executes one of a series of statement blocks, depending on the value of an expression.

Syntax Select Case testexpression
[Case expressionlist
[statement_block]]
[Case expressionlist
[statement_block]]

f

[Case Else
[statement_block]]
End Select

Parameters test expression—Any numeric or string expression for which you test. Each statement_block can contain any number of statements on any number of lines.

Comments The expressionlist(s) can be a comma-separated list of expressions of the following forms:

•expression
•expression-to-expression
•*Is comparison_operator* expression

The type of each expression must be compatible with the type of testexpression.

When there is a match between testexpression and one of the *Case* expressions, the statement block following the *Case* clause is executed. When the next *Case* clause is reached, execution control passes to the statement which follows the *End Select* statement.

Note that when the *To* keyword is used to specify a range of values, the smaller value must appear first. The *comparison_operator* used with the *Is* keyword is one of: <, >, =, <=, >=, <>.

Example
```
Select Case price
Case is > 100
MsgBox "Too expensive"
Case 50 to 99
MsgBox "Good price"
Case is < 50
MsgBox "Sale!"
End Select
```

Selection$

Gets or sets the selection criteria for a dataset Control Object. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Selection$

Comments A table must be added to the data set before the selection criteria can be written or read. A change in *Selection$* does not change the data until a *Commit* method or a *Recalc* command is executed. This property is read/write at run time.

You can get the individual tables from the list by using the *GetField$*function.

Example MyData.Selection$="Salary>40000"

SelectReport

Sets the input focus to the report that has the indicated title.

Syntax SelectReport ReportTitle$

Parameters ReportTitle$—The title of the report on which to set focus.

Comments This statement is useful in changing the active report in order to use commands that work only on the currently active report.

Example SelectReport "c:\rptsmith\sales.rpt"

SendKeys statement

Syntax SendKeys string-expression [, wait]

Comments The SendKeys statement is used to send keystrokes to the active application.

The keystrokes are represented by characters of string-expression. If the wait parameter is True, SendKeys does not return until all keys are processed. Otherwise, SendKeys does not wait for an application to process the keys. The default value for wait is False.

To specify an ordinary character, use this character in string-expression. For example, to send character 'a' use "a" as string-expression. Several characters may be combined in one string: string-expression "abc" means send 'a', 'b', and 'c'.

To specify that Shift, Alt, or Control keys should be pressed simultaneously with a character, prefix the character with

+ to specify Shift,

% to specify Alt, and

^ to specify Control.

Parentheses may be used to specify that Shift, Alt, or Control key should be pressed with a group of characters. For example, "%(abc)" is equivalent to "%a%b%c".

Since '+', '%', '^' ,'(' and ')' characters have special meaning to SendKeys, they must be enclosed in braces if need to be sent with SendKeys. For example string-expression"{%}" specifies a percent character '%'

The other characters that need to be enclosed in braces are '~' which stands for a newline or "Enter" if used by itself and braces themselves: use {{} to send '{' and {}} to send '}'. Brackets '[' and ']' do not have special meaning to SendKeys but may have special meaning in other applications, therefore, they need to be enclosed inside braces as well.

To specify that a key needs to be sent several times, enclose the character in braces and specify the number of keys sent after a space: for example, use {X 20} to send 20 characters 'X'.

To send one of the non-printable keys use a special keyword inside braces:

Key	Keyword
Backspace	{BACKSPACE} or {BKSP} or {BS}
Break	{BREAK}
Caps Lock	{CAPSLOCK}
Clear	{CLEAR}
Delete	{DELETE} or {DEL}
Down Arrow	{DOWN}
End	{END}

Enter	{ENTER}
Esc	{ESCAPE} or {ESC}
Help	{HELP}
Home	{HOME}
Insert	{INSERT}
Left Arrow	{LEFT}
Num Lock	{NUMLOCK}
Page Down	{PGDN}
Page Up	{PGUP}
Right Arrow	{RIGHT}
Scroll Lock	{SCROLLLOCK}
Tab	{TAB}
Up Arrow	{UP}

To send one of function keys F1-F15, simply enclose the name of the key inside braces. For example, to send F5 use "{F5}"

Note that special keywords can be used in combination with +, %, and ^. For example: %{TAB} means Alt-Tab. Also, you can send several special keys in the same way as you would send several normal keys: {UP 25} sends 25 Up arrows

SendKeys can send keystrokes only to the currently active application. Therefore, you have to use the AppActivate statement to activate the application before sending keys unless it is already active.

SendKeys cannot be used to send keys to an application which was not designed to run under Windows.

Set

Syntax Set variableName = expression

Comments variableName must be an object variable or a variant variable. Expression must be an expression that evaluates to an object, typically a function, an object member or Nothing

Dim Ole2 As Object

Set Ole2 = CreateObject("spoly.cpoly")

Ole2.reset

Note: If you omit the keyword Set when assigning an object variable, Basic will try to copy the default member of one object to the default member of another. This usually results in a runtime error.

```
' Incorrect code - tries to copy default member!
Ole2 = CreateObject("spoly.cpoly")
```

SetAttr

Syntax SetAttr filename$, attributes%

Comments The SetAttr statement sets the attributes for a file.

Filename is a String expression containing the name of the file whose attributes are to be modified. Wildcards are not allowed. It is an error to attempt to modify the attributes of a file opened for other than Read access.

Attributes is an Integer containing the new attributes for the file. Here is a description of attributes that can be modified:

Value	Meaning
0	Normal file
1	Read-only file
2	Hidden file
4	System file
32	Archive - file has changed since last backup

SetColumnAlias

Sets or changes the Alias for a column in a report's table. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see "Using the DataSet Control" on page 215.

Syntax [object].SetColumnAlias Table$, Databases$, Column$, Alias$

Parameters Table$—The path and file name for local databases.

Database$—The database that contains the table.

Column$—The name of the field for which to set an alias.

Alias$—The new alias for the field.

Returns 0 on success, a non-zero on error.

Comments For database servers the *Table$* parameter takes the form: Owner.TableName. For local databases or servers that don't require that a database be specified, the *Database$* parameter should be set to a null string.

Example MyData.SetColumnAlias "dbo.emp","hr","DEPT_ID","Departments"

SetDataFilter

Calls a specified macro to modify displayed data. This command is executed before any data field column value is calculated. The specified macro can then use the *FieldText* and *FieldFont* functions to change either the text or the appearance of the data.

Syntax SetDataFilter MacroName$

Parameters MacroName$—The path and file name of a .MAC file, or the name of a global macro, to be used as the macro data filter.

Comments This command can be costly in performance (under some circumstances), as the specified macro is executed once for each field visible on the report surface. The filter function can be disabled by calling this function with a null macro name.

Example SetDataFilter "FilterMacro"

SetDirtyFlag

This command is used to mark a report as modified, or "dirty."

Syntax SetDirtyFlag Conditional%

Parameters Conditional%—When set to zero (FALSE), the report is marked as "clean" (unmodified), and the Save Report dialog box will not appear when the report is closed. If *Conditional%* is non-zero (TRUE), the Save Report dialog box appears when the report is closed, prompting the report user to save changes.

Example The following code prompts the user to save changes to the report before closing it, whether or not the report has been modified.

```
SetDirtyFlag 1
```

SetField$

Sets a value in a list of values whose fields are all separated by the same character.

Syntax SetField$(string$, field_number%, field$, separator_chars$)

Parameters string$—String list of items to update.

field_number%—Number of the item in the list to update.

field$—New item to be placed in the list.

separator_chars$—Character used to separate individual items in the list.

Returns A string created from a copy of the source string with a substring replaced.

Comments The source string is considered to be divided into fields by separator characters. Multiple separator characters can be specified. The fields are numbered starting with one.

If *field_number* is greater than the number of fields in the string, the returned string is extended with separator characters to produce a string with the proper number of fields. If more than one separator character was specified, the first one is used as the separator character.

It is legal for the new field value to be a different size than the old field value.

Example The following returns "one\2\three."

```
SetField ("one\two\three",2,"2","\")
```

SetFieldLabel

Changes the label that appears over the given field.

Syntax SetFieldLabel FIELD$, NEWLABEL$

Parameters FIELD$ — A text string specifying a field from a table used in the report.
NEWLABEL$ — The label to display in place of the default field label.

Example The following code refers to a table field called EMPYEE_ID, and sets its label (within the report) to “The Employee ID” (without the quotes).

```
SetFieldLabel “EMPLYEE_ID”, “The Employee ID”
```

SetFromActive (dataset object and report object)

Replaces any previous connection information in a dataset object with a reference to the data description for the currently active report. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].SetFromActive

Comments Use this method to change a currently active report. It can also be used to save dataset information for a report that can then be reloaded, changed and used to create other reports.

Example The following example uses the SetFromActive method along with the Selection$ property to change the selection criteria for the active report.

```
Sub ChangeActiveSelection()
'Create a DataSet named DS
dim DS as DataSet
DS.SetFromActive
DS.Selection$ = " Department = 'Accounting' "
Cause the report to update to reflect the change
Recalc
End Sub
```

SetFromLoading

Associates the dataset control object with a report that is being loaded (before the SQL is executed for this report). This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].SetFromLoading

Returns 0 on success, a non-zero value on error.

Comments This function is only valid when used in the “Before report is opened” event.

Example MyData.SetFromLoading

SetIncludePath

Sets to default directory for macro include files.

Syntax SetIncludePath Path$

Parameters Path$—The path where ReportBasic will search by default to find include files.

Returns Non-zero on error.

Comments The macro compiler first looks in this directory for include files, then it looks in the standard search path. The beginning value is the default macro path in the RPTSMITH.INI file.

Example SetIncludePath "c:\macros\include"

SetRecordLimit

Sets the total number of records that ReportSmith downloads for any loaded or created report. A value of 0 allows an unlimited number of records to be downloaded.

Syntax SetRecordLimit Limit

Parameters Limit—The maximum number of records that ReportSmith downloads for a single report.

Comments This statement is helpful for implementing a draft mode where you can work with a subset of a large report until you are ready to work with the entire report. Some operations continue to work against the entire result set, such as selections, sorting, and summary fields, so that performance does not change for these operations.

Example SetRecordLimit 100

SetRepVar

Stores a value in a case-sensitive report variable in the active report.

Syntax SetRepVar ReportVariable$, Value$

Parameters ReportVariable$—A string specifying the name of the report variable being set.

Value$—A string parameter that specifies to what value to set the report value.

Returns Non-zero on error.

Example This example sets “Smith” as the value of a report variable called “Repvar1”.

```
SetRepVar "Repvar1", "Smith"
```

SetSQL

Replaces the SQL string that would normally be generated by ReportSmith.

Syntax SetSQL SQL$

Parameters SQL$—A quoted, valid SQL statement.

Comments The *SetSQL* statement is only valid in a macro that is linked to the “Before SQL is Executed” event. Care should be used when executing this command, as the string is not verified before it is executed. This command can be used along with the *GetSQL* command in a macro that is linked to the “Before SQL is Executed” event, to dynamically change the SQL string.

Example SetSQL "Select ENAME, EMP_ID from SCOTT.EMP"

SetTableAlias

Sets or changes the alias for a table in a report. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].SetTableAlias Table$, Database$, Alias$

Parameters Table$—The path and file name for local data sources.

Database$—The database (if any) that contains the table.

Alias$—The new alias for the table.

Returns 0 on success, a non-zero value on error.

Comments For database servers the Table$ parameter takes the form: Owner.TableName. For local databases or servers that don’t require that a database be specified the Database$ parameter should be set to a null string.

Example MyDate.SetTableAlias "dbo.emp","hr","Human_Resource"

SetTableLink

Defines a link between two tables. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].SetTableLink Table1$,DBase1$,Field1$,Table2$,DBase2$,Field2$,Operation, JoinType

Parameters Table1$—The first table to link.

DBase1$—The database (if applicable) that contains Table1.

Field1$—The field to link on from the first table.

Table2$—The second table to link.

DBase2$—The database (if applicable) that contains Table2.

Field2$—The field to link on from the second table.

Operation—The relation between the linked fields. It can have one of the following values:

0 Field 1 = Field 2

1 Field 1 < Field 2

2 Field 1 <= Field 2
3 Field 1 > Field 2
4 Field 1 >= Field 2

JoinType—The type of link. It can have one of the following values:

0 Inner join
1 Left outer join
2 Right outer join
3 Full outer join

Comments Before a table link can be defined, both tables must be added to the dataset object, using the *AddTable* function.

Example This example links the emp table to the dept table by the department id excluding all unmatched records. The following code fragment should be entered as one unbroken line of BASIC code.

```
SetTableLink
"dbo.emp","Indigo","Dept_Id","dbo.dept","Indigo","Dept_Id",0,0
```

SetUserSQL (dataset object)

Places the dataset object into user-entered SQL mode with the provided SQL. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax **[**object].SetUserSQL SQL$

Parameters SQL$—The complete SQL string to be used for this dataset query.

Returns 0 on success, a non-zero on error.

Comments You must have a connection to the appropriate server (or local database) that supports the SQL you generate, in order for the *SetUserSQL* command to work properly.

Example This code fragment should be entered as a single line of BASIC code.

```
MyData.SetUserSQL"SELECT
dbo.emp.First_Name,dbo.emp.Last_Name FROM dbo.emp"
```

Sgn

Determines the sign of a number. Can be used only as a function.

Syntax Sgn (numeric-expression)

Parameters numeric-expression—The number for which you want to get the sign.

Returns A value indicating the sign of the numeric expression. The value that the Sgn function returns depends on the sign of the expression:

> 0, Sgn (numeric-expression) returns 1.

= 0, Sgn (numeric-expression) returns 0.

< 0, Sgn (numeric-expression) returns –1.

Example ‘The following example makes a number positive.

```
number_sign = number * sgn(number)
```

Shell

Runs an executable program.

Syntax Shell (commandstring$, [windowstyle%])

Parameters commandstring$—The name of the program to execute. It can be the name of any valid .COM, .EXE., .BAT, or .PIF file. Parameters or command line switches can also be included.

windowstyle%—The style of the window in which the program is to be executed. It can be one of the following:

1Normal window with focus

2Minimized with focus

3Maximized with focus

4Normal window without focus

5Minimized without focus

If *windowstyle%* is not specified, the default of windowstyle% = 1 is assumed (normal window with focus).

Returns A unique number that identifies the running program.

Comments If *commandstring$* is not a valid executable file name or if *Shell* cannot start the program, an error message is generated.

Example The following command launches Excel.

```
Task_id = Shell("c:\Excel\Excel.Exe",1)
```

ShowRS

Hides, shows, minimizes, or maximizes ReportSmith.

Syntax ShowRS Code%

Parameters Code%—The following codes are valid:

0 Hides ReportSmith and passes activation to another window.

1 Activates and displays ReportSmith. If ReportSmith is minimized or maximized, Windows 95 restores it to its original size and position.

2 Activates ReportSmith and displays it as a Taskbar icon.
3 Activates ReportSmith and displays it maximized.
4 Displays ReportSmith in its most recent size and position. The window that is currently active remains active.
5 Activates ReportSmith and displays it in its current size and position.
6 Minimizes ReportSmith and activates the top-level window in the system's list.
7 Displays ReportSmith as an icon. The window that's currently active remains active.
8 Displays ReportSmith in its current state. The window that's currently active remains active.
9 Activates and displays ReportSmith. If ReportSmith is minimized or maximized, Windows 95 restores it to its original size and position.

Example

```
'Force ReportSmith to be a Taskbar icon
ShowRS 2
```

Sin

Calculates the sine of an angle specified in radians. Used only as a function.

Syntax Sin(angle)

Parameters angle—The angle (in radians) for which you are computing the sine.

Returns The sine of an angle. The return value is between –1 and 1. The return value is single-precision if the angle is an integer or single-precision value, double precision for a long or double-precision value.

Comments The angle is specified in radians, and can be either positive or negative.

Example The following calculates the sine of 60 degrees.

```
Value = sin ((60x3.1415)/180)
```

Space$

Generates a string with the given number of spaces. Used only as a function.

Syntax Space$(numeric expression)

Parameters numeric expression—Number of spaces which the returned string contains.

Returns A string of spaces.

Comments Any numeric data type can be used, but the number is rounded to an integer. The numeric expression must be between 0 and 32,767.

Example The following example evaluates to “One_ _ _ _ _Two_ _ _ _ _Three.”

```
"One" + Space$(5) + "Two" + Space$(5) + "Three"
```

Spc

Syntax Spc (numeric-expression)

Comments The Spc function can be used only inside the Print statement. Numeric-expression specifies the number of spaces that should be output.

When the Print # statement is used, the Spc function will use the following rules for determining the number of spaces to output:

If the width of the output line is not set (the width of the line can be set with the Width statement), SPC outputs the number of spaces equal to numeric-expression. Otherwise, it outputs numeric-expression Mod width spaces, unless the difference between the width of the line and the current print position is less than numeric-expression Mod width. In this case, the Spc function skips to the beginning of the next line and outputs (numeric-expression Mod width) - (width - current-position) spaces.

Sqr

Calculates the square root of a number. Used only as a function.

Syntax Sqr(numeric expression)

Parameters numeric-expression—Number for which you are getting the square root.

Returns The square root of numeric-expression.

Comments The return value is single-precision for an integer or single-precision numeric expression, or double precision for a long or double-precision numeric expression. A negative value causes a run-time illegal function call error.

Example Z = Sqr (x^2 + y^2)

Static

Syntax Static variableName [As type] [,variableName [As type]] ...

Comment Static is used inside procedures to declare variables and allocate storage space. Variables declared with the Static statement retain their value as long as the program is running. The syntax of Static is exactly the same as the syntax of the Dim statement

All variables of a procedure can be made static by using the Static keyword in definition of that procedure (see function or Sub for the details).

Stop

Halts program execution.

Syntax Stop

Comments *Stop* statements can be placed anywhere in a program to suspend its execution. While the *Stop* statement halts program execution, it does not close files or clear variables.

Example If ResumeCode = 0 Then Stop

Str$

Converts a number to a string. Used only as a function.

Syntax Str$(numeric-expression)

Parameters numeric-expression—A number to be converted to a string.

Returns The *Str$* function returns a string representation of a numeric-expression.

Comments The returned string is single-precision for an integer or single-precision numeric expression, double precision for a long or double-precision numeric expression.

Example MsgBox "The value is:" + Str$ (value)

StrComp

Syntax StrComp(string1$, string2$ [, comparetype%])

Returns The StrComp function compares two strings and returns -1 if the string1 is less than string2, 0 if the two strings are identical, 1 if string1 is greater than string2, and null if either string is NULL.

Comment The method of comparison is determined by comparetype%. If comparetype% is 0, a case sensitive comparison based on the ANSI character set sequence is performed. If comparetype% is 1, a case insensitive comparison is done based upon the relative order of characters as determined by the country code setting for your system. If omitted the module level default, as specified with Option Compare will be used.

The string1 and string2 arguments are both passed as variants. Therefore, any type of expression is supported. Numbers will be automatically converted to strings.

String$

Creates a string of the given character repeated the given number of times. Used only as a function.

Syntax String$(numeric expression, charcode%)
String$ (numeric expression, string expression$)

Parameters numeric expression—Specifies the length of the string to be returned. This number must be between 0 and 32,767.

charcode%—A decimal ANSI code of the character that is used to create the string. It is a numeric expression that BASIC will evaluate as an integer between 0 and 255.

string-expression$—A string parameter, the first character of which becomes the repeated character.

Returns The *String$* function returns a string consisting of a repeated character.

Example DerivedField String$(10,"=")

Sub...End Sub

Defines a subprogram procedure.

Syntax Sub name [(parameter [As type],...)] End Sub

Parameters The parameters are specified as a comma-separated list of parameter names. A parameter's data type can be specified either by using a type character or by using the *As* clause.

Record parameters are declared by using an *As* clause and a type which has previously been defined using the *Type* statement.

Array Parameters are indicated by using empty parentheses after the parameter. The array dimensions are not specified in the *Sub* statement. All references to an array parameter within the body of the subprogram must have a consistent number of dimensions.

Returns Returns to the caller when the *End Sub* statement is reached or when an *Exit Sub* statement is executed.

Comments A call to a subprogram stands alone as a separate statement. (See “Call” on page 272.)

Recursion is supported.

BASIC procedures use the call-by-reference convention. This means that if a procedure assigns a value to a parameter, it modifies the variable passed by the caller. This feature should be used with great care.

The MAIN subprogram has a special meaning. In many implementations of BASIC, MAIN is called when the module is “run.” The MAIN subprogram is not allowed to take parameters.

Example

```
Sub Hello()
MsgBox "Hello"
End Sub
```

SumField

Gives the value of a summary field.

Syntax SumField$(Field$, Table$, GroupLevel, Operation$)

Parameters Field$—The name of the field being summed.

Table$—The name of the table being summed.

GroupLevel—The group in the report at which the summary is reset.

Operation$—Summary operation performed on the Field.

Comments You can drag and drop this command from the list box. The best way to use it is to choose Summary Fields from the first list box and then double-click on your Summary Field. ReportSmith uses the *SumField$* command and fills in the Parameters for you.

Example This is how a Summary Field should be referenced in macros.

```
    my_var$=
SumField$("QTY_FIELD","OWNER.TABLE_NAME",0,"Count")
```

Style$ Property

Syntax [object].Style$

Definition A string that holds the last report style name selected.

Returns Returns the last selected style name chosen in the New Report dialog box.

Example
```
Sub GetTypeandStyle()
dim MyDialog as newReportDialog
xx.rundialog
MsgBox Str(MyDialog.ReportType)+ Chr$(13) + MyDialog.Style$
End Sub
```

Tab

Syntax Tab (numeric-expression)

Comments The Tab function can be used only inside the Print statement. It moves the current print position to the column specified by numeric-expression. The leftmost print position is position number 1.

When the Print # statement is used, the Tab function will use the following rules for determining the next print position:

If the width of the output line is not set (the width of the line can be set with the Width statement), the new print position is equal to numeric-expression. Otherwise, the new print position is equal to numeric-expression Mod width, unless the current print position is greater than numeric-expression Mod width. In this case, Tab skips to the next line and sets print position to numeric-expression Mod width.

Table$

Returns a list of tables included in a report, separated by commas. This command represents a property—an object variable—of the dataset object, which in turn represents the data contained in the currently active report. Access object properties the same way you access object methods: by using the object name followed by a period (.) and the property name. Some properties are read-only while others can be both read and written. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].Table$

Comments By using the *GetField$* function you can get the individual tables from the list.

Example
```
SecondTable$=GetField$(MyData.Table$,2,"","")
```

Tan

Returns the tangent of an angle. Used only as a function.

Syntax Tan(angle)

Parameters angle—The angle, in radians, for which you are calculating the tangent.

Returns The tangent of an angle. The return value is single-precision if the angle is an integer or single-precision value, or double precision for a long or double-precision value. The angle is specified in radians, and can be either positive or negative.

Example Find the tangent of pi/6.

```
value=tan (3.14159265359/6)
```

TestSelection$

Returns a string that tells how many records would be selected or an error message about the selection criteria. This command is a method of the dataset object, which represents the data contained in the currently active report. To use the command, preface it with the name of the dataset object and a period, followed by the command, as shown in the following syntax example. For detailed information on using the DataSet object, see “Using the DataSet Control” on page 215.

Syntax [object].TestSelection$

Comments In order to set and test a selection criterion, you must have a connection and at least one table.

Example

```
MyData.Selection$="Salary>40000"
Msgbox MyData.TestSelection$()
```

Text

Sets up line(s) of text in a dialog box.

Syntax Text x, y, dx, dy, text$

Parameters x, y—Set the position of the upper left hand corner of the text area relative to the upper left corner of the dialog box.

dx, dy—Set the width and height of the text area.

text$—Contains the text that appears to the right of the position designated by the x/y coordinates. If the width of this string is greater than dx, the spillover characters wrap to the next line. This continues as long as the height of the text area established by *dy* is not exceeded. Excess characters are truncated.

Comments The *Text* statement can be used only between a *Begin Dialog* and an *End Dialog* statement.

By preceding a character in *text$* with an ampersand (&), you enable a user to press that character on the keyboard and position the cursor in the combo or text box defined in the statement immediately following the Text statement.

Example Text 10,10,180,20 "This is my text"

TextBox

Creates a box, used within a dialog box, in which the user can enter and edit text.

Syntax TextBox x, y, dx, dy, .field

Parameters x, y—Set the position of the upper left hand corner of the text box relative to the upper left corner of the dialog box.

dx, dy—Set the width and height of the text area. A *dy* value of 12 will usually accommodate text in the system font.

.field—The name of the dialog record field that holds any text entered in the text box. When the user selects the OK button, or any push button other than cancel, the text string entered in the text box is recorded in the field.

Comments The *TextBox* statement can only be used between a *Begin Dialog* and an *End Dialog* statement.

Example Textbox 10,10,130,20 .MyData

Time$

Gets the current system time as a string. Used only as a function.

Syntax Time$

Returns A string representing the current time.

Comments The Time$ function returns an eight-character string. The format of the string is "hh:mm:ss" where hh is the hour, mm is the minutes, and ss is the seconds. The hour is specified in military style and ranges from 0 to 23.

Example CurrentTime$ = Time$

TimeSerial

Syntax TimeSerial(hour%, minute%, second%)

Returns The TimeSerial function returns a variant of vartype 7 (date) that represents a time specified by the hour%, minute%, and second% arguments.

Comments The range of numbers for each TimeSerial argument should conform to the accepted range of values for that unit. You also can specify relative times for each argument by using a numeric expression representing the number of hours, minutes, or seconds before or after a certain time.

TimeValue

Syntax TimeValue(string expression$)

Returns The TimeValue function returns a time value for the string specified.

Comments The TimeValue function returns a variant of vartype 7 (date/time) that represents a time between 0:00:00 and 23:59:59, or 12:00:00 A.M. and 11:59:59 P.M., inclusive.

Timer

Returns the number of seconds that have elapsed since midnight. Used only as a function.

Syntax Timer

Returns The number of seconds that have elapsed since midnight.

Comments The *Timer* function can be used in conjunction with the *Randomize* statement to seed the random number generator.

Example Use the number of seconds since midnight to seed the random number generator.

```
Randomize Timer
```

TotalPages (reports and report object)

Returns the number of pages in the currently active report. *Used only as a function.*

Syntax TotalPages

Example MsgBox "There are" + str$(Totalpages)+"in the current report"

TotalRecords

Returns the total number of records in the active report.

Syntax TotalRecords

Example r = TotalRecords()

—or—

r = TotalRecords

Trim$

Syntax Trim[$](expression)

Returns The Trim$ function returns a copy of the source expression with all leading and trailing space characters removed.

The dollar sign ($) in the function name is optional. If specified, the return type is string. If omitted, the function will typically return a variant of vartype 8 (string). If the value of expression is null, a variant of vartype 1 (null) is returned

Comments Trim$ accepts expressions of type string. Trim accepts any type of expression including numeric values and will convert the input value to a string.

Type

Declares a user-defined type which can then be used in the DIM statement to declare a record variable. Such a user-defined type is also sometimes referred to as a record type or a structure type.

Syntax
```
Type userType
field1 As type1
field2 As type2
End Type
```

Parameters Fieldn—Each field for which you declare a type.

Typen—A valid data type. (See “Data types of variables” on page 256 for a list of data types.)

Comments Between the *Type* and *Type End* you may define a number of elements known as fields. Each field can be a string (either dynamic or fixed), number (integer, long, single or double), or a previously-defined record type; it cannot be an array. However, arrays of records are allowed.

The Type statement is not valid inside a procedure definition.

To access the fields of a record, use notation of the form:

recordName.fieldName.

To access the fields of an array of records, use notation of the form:

arrayName (index).fieldName

Example

```
Type car
engine size as integer
number of cylinders as integer
color as string
make as string
model as string
End Type
Dim Honda as car
Honda.Color = "red"
```

Typeof

Syntax If Typeof objectVariable Is className then. . .

Returns -1 (True) if the objectVariable refers to an object of the given class, zero (False) otherwise

Comments Typeof may only be used in an *If* statement and may not be combined with other boolean operators. i.e. Typeof may only be used exactly as shown in the syntax above.

To test if an object does not belong to a class, use the following code structure:

```
If Typeof objectVariable Is className Then
Else
   Rem Perform some action.
End If
```

UBound

Determines the largest valid index for a particular dimension of an array. Used only as a function.

Syntax UBound (arrayVariable [, dimension])

Parameters array—The name of the array to check.

dimension—The number of the dimension to check.

Returns The upper bound of the subscript range for the specified dimension of the *arrayVariable*.

Comments The dimensions of an array are numbered starting with 1. If the dimension is not specified, 1 is used as a default.

LBound can be used with *UBound* to determine the length of an array.

Example The following example sets the last element of this array to an end marker.

```
Data$ (1, UBound (Data,2)) = "*End"
```

UCase$

Converts the characters of a string to upper case. Can be used only as a function.

Syntax UCase$ (string$)

Parameters string$—The string to convert.

Returns A copy of the source string, with all lower case letters converted to upper case. The translation is based on the country specified in the Windows 95 Control Panel.

Example The following example returns the value “STOP.”

```
Upper_case = UCase$("Stop")
```

Val

Converts a string to a number. Can be used only as a function.

Syntax Val(string$)

Parameters string$—The string or string field to convert.

Returns Returns a numeric value corresponding to the first number found in the specified string. Spaces in the source string are ignored. If no number is found, 0 is returned.

Example The following example turns a string into a number and sets the number variable to that result.

```
Num_var = Val("123.4")
```

Weekday

Syntax Weekday(expression)

Returns The Weekday function returns the day of the week for the specified date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null a variant of vartype 1 (null) is returned.

Comments The Weekday function returns an integer between 1 and 7, inclusive (1=Sunday, 7=Saturday).

It accepts any type of expression including strings and attempts to convert the input value to a date value.

While ... Wend

Controls a repetitive action.

Syntax
```
While condition
statementblock
Wend
```

Comments The condition is tested —If TRUE, the statement block is executed. This process is repeated until the condition becomes FALSE.

The *While* statement is included in ReportBasic for compatibility with older versions of BASIC. The *Do* statement is a more general and powerful flow control statement.

Example
```
While InputBox$ ("Type the secret word") <> "Secret"
MsgBox "That's not it"
Wend
```

Width statement

Syntax Width # filenumber%, width%

Comments The width statement sets the output line width for an open file.

Filenumber% is an integer expression identifying an open file to query for position. See the Open statement for more details.

Width is an integer expression in the range 0 to 255 specifying the number of characters on a line before a newline is started. A value of zero (0) for width indicates there is no line length limit. The default width for a file is zero (0).

Write

Writes data to a sequential file. The file must be opened in output or append mode.

Syntax Write [#] filenumber% [,expressionlist]

Parameters filenumber%—An integer expression identifying the open file to write to.

expressionlist—Specifies one or more values to be written to the file.

Comments An expression must be string and/or numeric expressions, separated by commas. If *expressionlist* is omitted, the *Write* statement writes a blank line to the file. (See “Input$” on page 347 or “Input #” on page 348.)

Example Write #1, A$, B

Year

Syntax Year(expression)

Returns The Year function returns the year component of a date-time value.

The return value is a variant of vartype 2 (integer). If the value of expression is null a variant of vartype 1 (null) is returned.

Comments The Year function returns an integer between 100 and 9999, inclusive.

Year accepts any type of expression including strings and attempts to convert the input value to a date value.