ReportSmith Glossary

Click on the individual terms to view their definitions.

A

alias
application event
argument
Avery codes

B

BDE

C

calculated field
call by reference
call by value
client disk option
client memory option
column
Column Editing mode
columnar report
comma delimited files
comment
connection
crosstab

D

data field
derived field
display event
Draft mode
Dynamic Data Access
Dynamic Data Exchange

E

event

F

field
field label
Field Editing mode
form report
function

G

global macro

alias

A substitute name for a field, table, or column. For tables, an alias is used when you use the same table in a report more than once. For fields, tables, or columns, use an alias to substitute a more easily recognizable, or shorter, name.

application event

An application event can occur globally, each time you run ReportSmith, and/or each time you run any report.

argument

Used to pass information from one macro to another, and then in turn, pass information back from the second macro to the first. This older term is interchangeable with the term “parameter”; the two terms mean exactly the same thing.

Avery codes

Avery label codes corresponding to the Avery label styles found in the Label Type and Dimensions list of the Page Setup dialog box. (Choose File | Page Setup.) Codes match those displayed on the Avery label box.

BDE (Borland Database Engine)

The Borland Database Engine provides a fast, safe, and easy-to-use means of connecting ReportSmith to Borland database products such as Visual dBASE and Paradox.

calculated field

A field created by modifying one or more existing fields that contain stored data values.

call by reference

Arguments passed by reference to a procedure may be modified by the procedure.   Procedures written in BASIC are defined to receive their arguments by reference.   If you call such a procedure and pass it a variable, and if the procedure modifies its corresponding formal parameter, it will modify the variable. Passing an expression by reference is legal in BASIC; if the called procedure modifies its corresponding parameter, a temporary value will be modified, with no apparent effect on the caller.

call by value

When an arguments is passed by value to a procedure, the called procedure receives a copy of the argument. If the called procedure modifies its corresponding formal parameter, it will have no effect on the caller. Procedures written in other languages such as C may receive their arguments by value.

client disk option

Tells ReportSmith to load data onto the client disk (as opposed to local memory, or on a server) each time you run ReportSmith.

client memory option

Tells ReportSmith to load data into client memory (as oposed to local memory, or on a server) each time you run ReportSmith.

column

Columns are vertical visual divisions of data, usually representing different categories or means of classifying that data. Columns in a report contain fields from tables or files.

Column Editing mode

Used to move columns and their labels horizontally across the report page. Compare Form Editing mode.

columnar report

Displays similarly classified or categorized fields in vertical columns across the report page.

comma-delimited files

Text files containing fields separated by commas, and using the file extension .CSV.

comment

A comment is text which documents the program. Comments have no effect on the program (except for metacommands). In BASIC, a comment begins with a single quote, and continues to the end of the line. If the first character in a comment is a dollar sign ($), the comment will be interpreted as a metacommand. Lines beginning with the keyword **Rem** are also interpreted as comments.

connection

Attaches ReportSmith to a server or particular folder/file structure on local computers, and points to the location(s) of files and tables used in a report.

crosstab

A summary report that displays data in a spreadsheet-like format. It can consist of rows, columns, row labels, column labels, and values. Fields can be calculated to show sums, averages, or counts.

data field

A single record of information contained in a table—for example, a place for a customer name in a column containing customer last names.

derived field

A field that is created and derived by concatenating, calculating, or otherwise modifying other existing fields. For example, a field called FULLNAME, derived by combining the F_NAME and L_NAME fields.

display event

An event linked to the display of a particular instance of a field in a report.

draft mode

Use draft mode to page through a report quickly, displaying graphics as light gray outlines, and "DRAFT" in the report background. Draft mode is intended to speed up viewing and working with reports; use presentation mode to check the appearance of a report as it will be printed.

Dynamic Data Access (DDA)

Controls which resources ReportSmith uses to store data, to recognize the size of data, and to determine the best strategy for transferring it into a report.

Dynamic Data Exchange (DDE)

A protocol for Windows-compatible applications, which enables two applications to communicate with each other, exchanging and dynamically updating data within each application.

event

Any action in ReportSmith that triggers the execution of a macro to which it is linked. Many events can occur at either the report-specific level or the application level, while other events may be unique to a particular level. For example, the *After Report Open* event occurs after you open a particular report (in which you have created the macro linked to this event), while the *On SQL Error* event is generated by an SQL error, and triggered only when an SQL error occurs.

field

A field contains a data value within a column.

field label

A column heading. By default, ReportSmith applies the field name from the source table as the field label within a report. However, you can change a ReportSmith field label by typing over it on the report surface, or by creating an alias for it.

Field Editing mode

Used to move *individual fields* (not entire columns) freely around the report page (horizontally and vertically). Use this mode to place values into headers and footers. When you move values in Field Editing mode, column labels do not automatically move along with their respective columns. Compare Column Editing mode.

form report

Displays data in free-form across the report page. Compare columnar report.

function

A procedure which returns a value.   In BASIC, the return value is specified by assigning a value to the name of the function, as if the function were a variable.

global macro

A macro linked to the ReportSmith application that runs each time you run ReportSmith. Used to automate global tasks, or to customize ReportSmith.

grid

A series of row and column coordinates used as a visual reference when placing objects.

group

A number of fields and values consolidated by user-specified criteria and presented together or considered as a unit. Recurrences of a group in a report are suppressed if duplicated.

IDAPI

An alternative (older) name for the Borland Database Engine.

join

To add more than one table to a report, you must *join* or *link* them. A successful join or link (the terms are used interchangeably within ReportSmith) consists of a field which both tables share in common. Typically, the common field shares the same name.

key column

The column or columns in a table that contains a unique value for every record.

label

A label identifies a position in the program at which to continue execution, usually as a result of executing a **GoTo** statement. To be recognized as a label, a name must begin in the first column, and must be immediately followed by a colon (":"). Reserved words are not valid labels.

link

To add more than one table to a report, you must *join* or *link* them. A successful join or link (the terms are used interchangeably within ReportSmith) consists of a field which both tables share in common. Typically, the common field shares the same name.

macro

An operator that causes the generation of a sequence of instructions to accomplish one or more tasks. For example, you can write a macro to open, run, print, and close a daily report. A macro is a user-defined "mini-program" which you build using the ReportBasic programming language. You can create and store a custom macro, or load macros provided by ReportSmith in the RPTSMITH\MACROS directory. You can also save macros as *.MAC files, so that they are then available for use in any report, or globally in all reports.

Macros can be stored locally, with a report, or stored globally and used with all reports.

macro-derived field

A derived field created using the ReportBasic language.

master/detail report

A report using more than one query. Generally, the master report contains one record per key value, while the detail report contains many records per key value. The report breaks data into groups, so that for each record in the "A" table, matching records in the "B" table print beneath it, followed by matching records in the "C' table. Each query in the report can come from a different data source.

metacommand

A metacommand is a command which gives the compiler instructions on how to build the program. In BASIC, metacommands are specified in comments which begin with a dollar sign ($).

method

A function or statement which performs actions on the DataSet object.

mode

A term applied to an operating condition that is one of two or more such conditions. For example, in ReportSmith you can activate Form Editing mode or Column Editing mode, and draft mode or presentation mode. Modes are mutually exclusive; you cannot operate two similar modes (such as Form Editing mode and Column Editing mode) at once.

name (ReportBasic)

A BASIC name must start with a letter (A through Z). The remaining part of a name can also contain numeric digits (0 through 9) or an underscore character (_). A name cannot be longer than 40 characters in length. Type characters are not considered part of a name.

named connection

A permanent connection you set up, name, and save, that lets you quickly access frequently-used databases and tables.

null password

A password with no value, which can be ignored in the interpretation of data.

on server mode

Allows ReportSmith to store a limited number of records (generally 100) on the client disk, while maintaining a temporary area on the server disk to store other records.

parameter

Used to pass information from one macro to another, and then in turn, pass information back from the second macro to the first. This newer term is interchangeable with the term “argument”; the two terms mean exactly the same thing.

point size

A measurement of the size of the text characters used in a report. There are 72 points per inch, 12 points per pica.

presentation mode

Used to view the final copy of a report. All components that will print on a report are displayed in the actual size and position used on the printed copy. Compare draft mode.

property

A data value similar to BASIC variables, that can have any of the BASIC variable types. Used with DataSet object.

query

A request for specific data from one or more tables.

report

A collection of data filtered and formatted according to unique and specific data-viewing requirements. Also, a ReportSmith file saved with the extension .RPT. It can contain objects, tables, links, and other information.

report event

An action in ReportSmith that triggers the execution of a macro linked to a specific report. An example of a report event is the Keystroke event, Ctrl+P. Each time a selected report macro encounters the keystroke Ctrl+P, the macro to which the report is linked is triggered.

report macro

A macro linked to a specific report which becomes part of that report. A report macro differs from a global macro in that a report macro applies only to a certain report, while a global macro runs each time a user starts ReportSmith.

ReportBasic

ReportSmith has licensed the Softbridge Basic Language (SBL) from Mystic River Software, Inc. to provide ReportSmith users with a complete high-level programming language. SBL contains similar commands to the Visual Basic programming language. ReportSmith has added specific reporting commands to SBL to assist you in creating reports. This combined command set is called ReportBasic, a complete programming language designed specifically for report creation.

Click here to view macro-language Help.

report variable

A report variable is a dialog box which prompts a user for specific information that changes the contents of the report to which it is linked. Based on the values that the user specifies the report then executes and displays the corresponding data.

row

A set of values from within each column that constitute a "set" of information.

SBL

The acronym for the Softbridge Basic Language (SBL), which forms the basis of the ReportBasic macro language, is pronounced as "Sybil".

self-join (self-link)

Linking a table or file to itself.

SQL

An acronym for Structured Query Language, an industry-standard means of creating queries to filter, sort, and extract data. When you use ReportSmith features to select the tables and fields that will make up your report, the necessary SQL query is generated "behind the scenes," but you can also choose to directly enter and maintain the SQL queries that make up your report.

Note: SQL is pronounced either "sequel" or "S-Q-L." ReportSmith documentation assumes the latter pronunciation.

SQL-derived field

A derived field (a new report field that does not exist in the original data sources, constructed by performing calculations upon existing report fields) created using Structured Query Language (SQL) to define the calculations performed.

string value

Any set of characters entered to stand for a numeric value.

subprogram

A procedure which does not return a value.

summary field

A calculated field such as a subtotal or count, summarizing values of other fields.

summary-only report

A report that presents only summarized values, omitting the details used to arrive at those values.

system field

A field that has been automatically created by ReportSmith for your use, such as text, date, time, page number, or report name. System fields do not originate with your database application.

table

An arrangement of words, numbers, or signs (usually in parallel columns) displaying a set of facts or relations in a compact or comprehensive form.

text mode

The application state in which ReportSmith accepts the placement of text.

type character

A special character used as a suffix to a name of a function, variable, or constant. The character defines the data type of the variable or function.   The characters are:

Dynamic **String**	$
Integer	%
Long integer	&
Single single precision floating point	!
Double double precision floating point	#
Currency exact fixed point	@

unit

An item in a scale of measurement. Can be inches or centimeters.

value label

Data field names automatically inserted into a crosstab report in addition to the defined row and column values. May be used or suppressed.

vartype

The internal tag used to identify the type of value currently assigned to a variant. One of the following:

Empty	0
Null	1
Integer	2
Long	3
Single	4
Double	5
Currency	6
Date	7
String	8

visual link

Linking objects by matching common fields.

ReportBasic Commands Reference

ReportSmith supports two types of macro commands:

- ReportBasic commands, written specifically for ReportSmith.
- Core Basic commands, common to the Basic language.

This Help system lists and describes both ReportBasic and core BASIC commands. You can select the command name in the Help Index (choose the Index button at the top of the Help window), then choose Display to view the topic. If you know the task you want to accomplish but are unsure of which macro command would be best suited, use the Find command: choose the Index button at the top of the Help window, then choose the Find tab and follow the instructions provided.

Note: Regardless of whether you use the Index or the Find command, Help will search only those topics pertaining to ReportBasic Macro Language Help. General ReportSmith Help topics can be found in ReportSmith 3.0 Help. (Check the title bar to quickly determine whether you are viewing Macro Help or ReportSmith 3.0 Help.) A thorough discussion of macro concepts and techniques can be found in *Creating Reports*.

ActiveTitle$( ) function

Syntax

```
ActiveTitle$()
```

Definition

With the ActiveTitle$() function you can get the window title of the currently active report.

Parameters

Not applicable.

Returns

Not applicable.

Comments

For saved reports, the title will contain the path name and the file name of the report file.

Example

```
ActiveReport$=ActiveTitle$()
Msgbox "Your report is called" + ActiveReport$
```

AddMenu Statement

Syntax

```
AddMenu MenuText$, Macro$, AfterMenu$, HelpText$
```

Definition

The AddMenu Statement allows you to add your own commands to the ReportSmith Main Menu to execute a macro when the command is chosen.

Parameters

Argument	Description
MenuText$	A string that specifies the text of the new menu item.
Macro$	A string that specifies which macro should be run when the menu item is selected by a user.
AfterMenu$	A string that specifies which existing menu item the new menu item should be placed after. You specify this menu name in the following format (omitting accelerator characters, underlines, and so forth): `(MenuName / SubMenuName)`
HelpText$	A string that specifies the text of a help line on the status bar.

Returns

Not applicable.

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.

When a macro that modifies the menu is executed, the current menu is modified. That menu is either the active *ReportSpecific* menu or the *NoReport* menu.

A special character in the first position of the AfterMenu parameter indicates that a menu ***other than the current menu*** will be modified. If the flag is a "!" (exclamation mark), the *DefaultReport* menu — and thus every ReportSpecific menu subsequently created — will be modified. The current menu will not be affected. If the flag is an "*" (asterisk), then the *NoReport* menu is modified. If this represents the current menu, it is redrawn.

Note: If one of these commands was entered in a global macro that was tied to the Startup of the application, then the menu would be customized each time ReportSmith is brought up.

Examples

```
AddMenu "Open Sales Reports", "LoadSales","!File | Open", "Opens all sales
  reports for the previous quarter"

AddMenu "Open Sales Reports","C:\INDIGO\MACRO\LSALES.MAC","File/Open", ""
```

CloseReport

Syntax

```
CloseReport Conditional%
```

Definition

The CloseReport command lets you close the active report.

Parameters

Argument	Description
conditional%	If the integer argument is 0 (FALSE), then the report closes unconditionally. If CloseReport is called with a non-zero value, then reports that were modified since last opened will prompt the user to save the report before closing it and allow the user to cancel the close.

Returns

This function returns 0 if the active report was closed successfully. It will return 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), it requires the return value, and you must enclose its arguments within parentheses.

Example

```
CloseReport 1
```

CloseRS

Syntax

`CloseRS Conditional%`

Definition

The CloseRS command closes ReportSmith.

Parameters

Argument	Description
conditional%	If the argument "*Conditional%*" is TRUE (non-zero), then ReportSmith brings up dialog boxes prompting the user to do certain tasks, such as save unsaved files. If the argument is FALSE (zero), then ReportSmith closes without bringing up dialog boxes or allowing the user to cancel the process.

Returns

This function returns non-zero if a conditional close was canceled.

Comments

When you use this command as a function (rather than a statement), it requires the return value, and you must enclose its arguments within parentheses.

Example

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

Connect Statement

Syntax

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

Definition

The Connect statement opens a connection to a database server.

Parameters

Argument	Description
Type$	The Type parameter can take on the following values:
	0 Named Connection
	1 Reserved
	2 DBASE
	3 EXCEL
	4 PARADOX
	5 ASCII
	6 SQLSRVR
	7 ORACLE
	8 DB2
	10 SYBASE
	11 BTRIEVE
	12 GUPTA
	13 INGRES
	16 TERADATA
	17 DB2/GUPTA
	19 UNIFY
	40 DBASE (via ODBC)
	41 EXCEL (via ODBC)
	42 PARADOX (via ODBC)
	48 BTRIEVE (via ODBC)
	55 Generic ODBC driver (use this for most ODBC connections)
	61 PARADOX (via IDAPI)
	62 DBASE (via IDAPI)
	67 INTERBASE (via IDAPI)
Server$	Identifies the server that will be used to make the connection.
UserId$	Identifies the user to make the connection under.
Password$	The password of the user to make the connection under.
Database$	The name of the database to connect to or the filename of a local database.

Returns

Not applicable.

Comments

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

Example

```
Connect 6, "sqlsvr","myuser","mypassword","mydb"
```

CreateDDEItem Statement

Syntax

```
CreateDdeItem TopicName$, ItemName$, RequestMacro$, Pokemacro$
```

Definition

The CreateDDEItem statement allows the user to create a new Dynamic Data Exchange item in ReportSmith's DDE server.

Parameters

Argument	Description
TopicName$	The name of a user-defined DDE topic.
ItemName$	The name of a user-defined DDE item.
RequestMacro$	The name of a macro to call before a DDE request on this topic is serviced.
Pokemacro$	The name of a macro to call after a DDE poke to this topic is serviced.

Returns

Not applicable.

Comments

The user can optionally specify the name of a macro to execute before data is requested from this item. The user can also optionally specify another name of a macro to be executed after data is poked into this item. Those macros could use other commands to retrieve or change the data in the DDE item. A Topic that matches the *TopicName$* parameter must already exist for this function to execute successfully.

This function will allow you to make duplicate item names. In this case only the first item of a given name can be accessed. At this time a DDE item remains in effect as long as ReportSmith is executing. Eventually, functions will be added to get a list of available DDE items and to remove existing items.

Example

```
CreateDdeItem "MyTopic","MyItem","RequestProc", "PokeProc"
```

CreateDDETopic Statement

Syntax

```
CreateDdeTopic TopicName$
```

Definition

The CreateDDETopic Statement allows you to create a new Dynamic Data Exchange topic in the ReportSmith DDE server. After a topic is created, DDE items can be added to it.

Parameters

Argument	Description
TopicName$	The name of a user defined DDE topic.

Returns

Not applicable.

Comments

This function will allow you to make duplicate topic names. In this case only the first topic of a given name can be accessed. At this time a DDE topic remains in effect as long as ReportSmith is executing. Eventually, functions will be added to get a list of available DDE topics and to remove existing topics.

Example

```
CreateDdeTopic "MyTopic"
```

Current

Syntax

```
Current()
```

Definition

The Current function returns the record number to which the data set (belonging to the active report) is pointing. In other words, it tells you what record number the Field$ function will return when executed.

Parameters

Not applicable.

Returns

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

Comments

None.

Example

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

CurrentPage( ) function

Syntax

```
CurrentPage()
```

Definition

The CurrentPage function returns the page number that is displayed in the currently active report.

Parameters

Not applicable.

Returns

This function 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

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

DateField( ) function

Syntax

`DateField (*Field$*, *Code%*)`

Definition

The DateField command gives you date values.

Parameters

Argument	Description
Field$	A date DateField in your report.
Code%	The number from the table in the Comments section below that indicates the portion of the date you want returned.

Returns

Not applicable.

Comments

To use this command, grab a date field from the Data Fields list box in the Edit Macro dialog box and add one of the following numbers. The number specifies the type of information you want about the date:

Number	Information
1	Gets the number of days since the date field (based on a Julian date)
2	Gets the month of the date as a number
3	Gets the day of the month
4	Gets the year
5	Gets the day of the week
6	Gets the day of the year out of 365 days

Note: Any other value returns a zero and indicates an error.

Example

```
Year_Hired = DateField ("Hire_Date",4)
```

DDEAppReturnCode Function

Returns a code received from an application on an open dynamic data exchange (DDE) channel.

Syntax DDEAppReturnCode()

Comments To open a DDE channel, use **DDEInitiate**. Use **DDEAppReturnCode** to check for error return codes from the server application after using **DDEExecute**, **DDEPoke** or **DDERequest**.

DDEExecute

Syntax

```
DDEExecute Service$, Topic$, Command$
```

Definition

The DDEExecute command lets you send a DDE execute command to a DDE server application.

Parameters

Argument	Description
Service$	A string that specifies the DDE service to which you send the Execute command. (This is usually the name of the application that receives the command.)
Topic$	A string that identifies the DDE Topic to which you send the command. (Most applications accept DDE Execute messages that are sent to their SYSTEM topics.)
Command$	A string that identifies the DDE Command itself. See the documentation of the application to which you're sending the DDE command for details of the DDE Execute commands it accepts.

Returns

This command returns 0 if the command is sent successfully, or nonzero on error.

Comments

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

Example

The following line of code will send a DDEExecute command to cause Excel to beep.

```
Success = DdeExecute( "Excel","System","[Beep(0)]" )
```

DDEInitiate Function

Opens a dynamic-data exchange (DDE) channel and returns the DDE channel number (1,2, etc.).

Syntax DDEInitiate(*appname$, topic$* **)**

where:	is:
appname$	A string or expression for the name of the DDE application to talk to.
topic$	A string or expression for the name of a topic recognized by *appname$.*

Comments If **DDEInitiate** is unable to open a channel, it returns zero (0).

Appname$ is usually the name of the application's .EXE file without the .EXE filename extension. If the application is not running, **DDEInitiate** cannot open a channel and returns an error. Use **Shell** to start an application.

Topic$* is usually an open filename. If *appname$ doesn't recognize *topic$*, **DDEInitiate** generates an error. Many applications that support DDE recognize a topic named **System**, which is always available and can be used to find out which other topics are available. For more information on the **System** topic, see **DDERequest**.

The maximum number of channels that can be open simultaneously is determined by the operating system and your system's memory and resources. If you aren't using an open channel, you should conserve resources by closing it using **DDETerminate**.

DDEPoke

Syntax

```
DDEPoke Service$, Topic$, Item$, Data$
```

Definition

The DDEPoke command lets you poke a string to a DDE server application.

Parameters

Argument	Description
Service$	A string that specifies the DDE service to which you're going to poke. (This is usually the name of the application receiving the command.)
Topic$	A string that identifies the DDE Topic to which you send the command. (Most applications use the topic to identify the document into which you want to poke the data.)
Item$	A string that identifies the DDE Item into which you want to poke data.
Data$	A string, which is actually the data poked into the other application. See the documentation of the application to which you're sending the DDE poke command for details on the accepted DDE poke commands.

Returns

This command returns 0 if the command is sent successfully, or nonzero on error.

Comments

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

Example

The following line of code puts the string "My Data" in the first cell of a default Excel spreadsheet.

```
Success = DDEPoke( "Excel","Sheet1","R1C1","My Data" )
```

DDERequest

Syntax

```
DDERequest( Service$, Topic$, Item$)
```

Definition

The DDERequest command gets data from a DDE Server application as a string.

Parameters

Argument	Description
Service$	A string that specifies the DDE service to which you're going to Poke. (This is usually the name of the application that receives the command.)
Topic$	A string that identifies the DDE Topic to which you're going to send the command. (Most applications use the topic to identify the document into which you want to poke the data.
Item$	A string that identifies the DDE Item into which you want to poke data.

Returns

This command returns a string that has the requested data, or the string "<ERROR>" if the operation is not performed successfully.

Comments

See the documentation of the application to which you're sending the DDE poke command for details on the accepted DDE poke commands.

Example

The following line of code puts the data in the first cell of a default Excel spreadsheet into a string variable called Data$.

```
Data$ = DDERequest( "Excel","Sheet1","R1C1" )
```

DDETerminate Statement

Closes the specified dynamic data exchange (DDE) channel.

Syntax DDETerminate *channel%*

where:	**is:**
channel%	An integer or expression for the open DDE channel number.

Comments To free system resources, you should close channels you aren't using. If *channel%* doesn't correspond to an open channel, an error occurs.

DerivedField

Syntax
`DerivedField *Value$*`

Definition
The DerivedField command sets the value of a derived field.

Parameters

Argument	Description
Value$	A quoted value or variable.

Returns
Not applicable.

Comments
The DerivedField command only affects macros you use as derived fields. It takes a string that is used for the derived field. If the value for the derived field is a number, you need to convert it to a string using the STR$ function.

Example
```
DerivedField "Bill Smith"
```

DoEvents( ) function

Syntax

```
DoEvents
```

Definition

The DoEvents function allows other Windows applications to process messages.

Parameters

Not applicable.

Returns

Not applicable.

Comments

Use this function when you want your Basic code to yield processor time to allow other applications to process messages.

Example

```
DoEvents
```

EnableIcon( ) function

Syntax
`EnableIcon` *`GroupNo, ItemNo, EnableFlag`*

Definition
The EnableIcon function enables and disables icons and combo boxes on the toolbar and ribbon.

Parameters

Argument	Description
GroupNo	Index of the icon group that the icon or combo box belongs to. (See illustration in example.)
ItemNo	Index of the item within the group. (See illustration in example.)
EnableFlag	0 is to disable, 1 is to enable.

Returns
Not applicable.

Comments
None.

Example
Below command disables the Italic button so that it is not available.

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

EnableMenu

Syntax

```
EnableMenu Menu$, EnableCode%
```

Definition

The EnableMenu command enables or disables a menu command.

Parameters

Argument	Description
Menu$	The Menu name and menu subname you want to be affected.
EnableCode%	Specify 1 to *enable*, 0 to *disable*.

Returns

This command returns a 0 if a menu was removed successfully , and -1 if a menu of the given name was not found when used as a function.

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 our 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. 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), it requires the return value, and you must enclose its arguments within parentheses.

Example

The following line of code disables the ReportSmith File | New menu Item.

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

EnableRMenu

Syntax

`EnableRMenu ObjectType, EnableFlag`

Definition

Enables or disables right mouse menus by object type.

Parameters

Argument	Description	
ObjectType	A code that specifies the object type to disable right mouse menu items for.	
	1	Character
	2	Border
	3	Field Format
	4	Text Alignment
	5	Display as Picture
	6	Column Width
	7	Field Height
	8	Section Criteria
	9	Translate
EnableFlag	0 *disables* the menu; non-zero *enables* the menu.	

Returns

Not applicable.

Comments

This command can be used to check the state of the menu that will be used for new reports by default by placing an "!" before the menu name. This can be done whether the menu item is specified by command or relative location.

Example

The following example will disable the right mouse abilities for text fields.

```
EnableRMenu 1,0
```

ExecuteMenu

Syntax

`ExecuteMenu` *`Menu$`*

Definition

The ExecuteMenu command lets you simulate a user clicking one of the ReportSmith menu items.

Parameters

Argument	Description
Menu$	The menu name and/or submenu name that you want to execute.

Returns

This function returns a 0 if a menu was executed successfully, and -1 if a menu of the given name wasn't found.

Comments

It takes a string that specifies a menu item or a sum menu item. The string uses this format:

```
"MenuName|SubMenuName"
```

The names must match our 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.

When you use this command as a function (rather than a statement), it requires the return value, and you must enclose its arguments within parentheses.

Note: This command can be used to check the state of the menu that will be used by new reports by default by placing an "!" before the menu name. This can be done whether the menu item is specified by command or relative location.

Example

```
Execute Menu "View|Zoom"
```

ExportTable

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:

- 0—Reserved for named connections
- 1—Reserved
- 2—dBASE
- 3—Excel
- 4—Paradox
- 5—ASCII
- 6—SQLSrvr
- 7—Oracle
- 8—DB2 (MDI Gateway)
- 9—NetSQL
- 10—Sybase 4.x
- 11—Btrieve
- 12—SQLBase
- 13—Ingres
- 15—Ocelot
- 16—Teradata
- 17—DB2Gupta
- 21—Delphi 2.0
- 22—Sybase System 10 or later
- 55—Special (See “Data Source” parameter)
- 61—Paradox BDE connection
- 62—dBASE BDE connection
- 67—InterBase BDE connection
- 70—Informix BDE connection

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$( ) Function

Syntax

```
Field$(FieldName$)
```

Definition

The Field$ command 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.

Parameters

Argument	Description
FieldName$	A field in your report that you want the value of.

Returns

This function will return the value of the specified field as a string regardless of the retrieved field's data type. If necessary, you can use the Val command to convert these strings to numbers. If the specified field is not found, this function will return "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 "." followed by the field name. You can also use a field or table alias. An easy way to get the 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.

Note: Local database tables (such as dBASE and Excel) use the path to the local database file as a table name. If you use the Fix Report feature to run a report from a different table, you must change the path information in the macro code as well.

Example

```
NextId$ = Field$("Employee_Id")
```

FieldFont

Syntax

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

Definition

The FieldFont command 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.

Parameters

Argument	Description
Facename$	The font name.
PointSize	The point size.
Style	The style: 1 Text Fields 2 Sections 3 Draw Windows 4 Crosstabs 5 Crosstab Cells 6 General 7 Reserved
ForColor	The foreground text color.
BackColor	The background text color.

Returns

Not applicable.

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). For the fourth and fifth arguments, use a color value for the last two color arguments.

You can specify one of 16 million colors using the Rgb command. Windows substitutes the closest color to the one you select.

Use a negative one (-1) if you don't want to change the attribute, point size, or color.

Example

The following example would change the font to red, italic, 14 point Arial.

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

FieldText

Syntax

```
FieldText Text$
```

Definition

The FieldText command applies to macro fields that are linked to the display event of a data field object. It changes the text of the field.

Parameters

Argument	Description
Text	The text used to replace the data within a field.

Returns

Not applicable.

Comments

None.

Example

For example, suppose you wanted a field that has a person's name. And suppose you wanted to use their nickname instead. In this case, you would use the FieldText command similar to this one:

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

GetDDEItem( ) function

Syntax

```
GetDDEItem$(TopicName$, Item$, RequestMacro$, PokeMacro$)
```

Definition

The GetDDEItem$ function allows you to retrieve the data that a DDE client application would receive in response to a DDE request to ReportSmith.

Parameters

Argument	Description
TopicName$	The name of a user defined DDE topic.
ItemName$	The name of a user defined DDE item.
RequestMacro$	
PokeMacro$	

Returns

This function returns data in the DDE item.

Comments

A macro that is named as the poke response macro for a user defined DDE item can use this function to get the data that a DDE client has poked into the item. You can use this technique to create a DDE Item that will load a report when a DDE client application pokes the filename of that report into the item.

Example

```
TheData$=GetDDEItem("MyTopic","MyItem","MyData")
```

GetFieldName( ) function

Syntax

`GetFieldName()`

Definition

The GetFieldName function is a global filter, returning the column name of the data field that the filter macro is being called for.

Parameters

Not applicable.

Returns

The name of the field that a global data filter or conditional formatting macro was called for.

Comments

See also SetDataFilter.

Example

```
CurrentValue$=Field$(GetFieldName$0)
```

GetIncludepath$( ) Function

Syntax

```
GetIncludePath$()
```

Definition

The GetIncludePath$ command gets the default directory for macro include files.

Parameters

Not applicable.

Returns

This command returns a string that is the default path for macro include fields.

Comments

None.

Example

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

GetNext

Syntax
`GetNext`

Definition
The GetNext Command 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.

Parameters
Not applicable.

Returns
Not applicable.

Comments
None.

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

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

GetPrevious

Syntax
`GetPrevious`

Definition
The GetPrevious Command 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.

Parameters
Not applicable.

Returns
Not applicable.

Comments
The GetPrevious command can be used to change the current record of a dataset. The current record determines what data the Field$, SumField$ and DateField$ functions will 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 will be 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, GetRandom, Field$, SumField$, DateField$, Current, TotalRecords.

Example
```
`Position to the 3rd record
GetRandom 3
`Now go to 2
GetPrevious
```

GetRandom

Syntax
`GetRandom RecordNumber%`

Definition
The GetRandom command causes the data set in an active report to point to the record specified by the *RecordNumber%*, if that record number exists.

Parameters

Argument	Description
RecordNumber%	The number of the record in the data you want to navigate to.

Returns
Not applicable.

Comments
See also GetPrevious, GetNext, Field$, SumField$, DateField$, Current, and Total Records.

Example
```
`Point to the 23rd record in a set of data
GetRandom 23
```

GetRecordLimit( ) function

Syntax
`GetRecordLimit()`

Definition
The GetRecordLimit function will get the total number of records that ReportSmith will download for any loaded or created report.

Parameters
Not applicable.

Returns
Not applicable.

Comments
This limit is set with the function SetRecordLimit.

Example
```
TheLimit=GetRecordLimit()
```

GetRepVar( ) Function

Syntax

`GetRepVar(VariableName$)`

Definition

The GetRepVar command retrieves the value of a report variable in the active report. This command is only used as a function.

Parameters

Argument	Description
VariableName$	The name of a report variable in your report.

Returns

This function returns the value of the specified report variable as a string. It returns "<ERROR>" if a report variable of the specified name cannot be found in the active report.

Comments

This command takes a string argument that specifies the name of the report variable being retrieved. See also SetRepVar.

Example

```
Var_name$=GetRepVar("Rep_var_name")
```

GetSQL( ) function

Syntax
```
GetSQL()
```

Definition
The GetSQL function will return a string that is the text of the last SQL statement that ReportSmith executed.

Parameters
Not applicable.

Returns
Not applicable.

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 SWL string "On the Fly."

Example
The following stores the last generated SQL statement in a variable called The_SQL$.
```
The_SQL$ = GetSQL$()
```

hWin_Active( ) function

Syntax
```
hWin_Active()
```

Definition
The hWin_Active( ) function to get the window handle of the currently active report.

Parameters
Not applicable.

Returns
Not applicable.

Comments
This function can be used along with windows API functions that you make available to ReportBasic through the use of the declare function.

Example
```
`Force the active report to an Icon
Result=ShowWindow(hWin_Active(),2)
Declare function ShowWindow Lib "User"(ByVal hWnd As Integer, ByVal nCmdShow
  As Integer) As Integer
```

hWin_RS( ) function

Syntax
`hWin_RS()`

Definition
The hWin_RS( ) function allows you to get the ReportSmith main window handle.

Parameters
Not applicable.

Returns
Not applicable.

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()
```

IsMenuChecked

Syntax
`IsMenuChecked (Menu$)`

Definition
The IsMenuChecked command lets you determine if a given menu item has a check mark next to it. This command is only used as a function.

Parameters

Argument	Description
Menu$	The menu name and/or submenu name that you are interested in.

Returns
This function 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
It takes a string that specifies a menu item or a sum menu item. The string is of the form *"MenuName | SubMenuName."* The names must match our menu commands, not including keyboard accelerators and "..." characters. If you omit the pipe and submenu names, the routine assumes you're working with a top level menu.

Note: This function will not 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 will be used for new reports by default by placing an "!" before the menu name. This can be done whether the menu item is specified by command or relative location. See second example. See also EnableMenu, KillMenu, AddMenu, ExecuteMenu and IsMenuEnabled.

Examples
```
Success = IsMenuChecked ("View|Boundaries")

If IsMenuChecked ("!View|Boundaries") = 1 Then
ExecuteMenu "View|Boundaries"
End if
If IsMenuChecked ("!3,8") = 1 Then
ExecuteMenu "View|Boundaries"
End if
```

IsMenuEnabled

Syntax

`IsMenuEnabled (`*`Menu$`*`)`

Definitions

The IsMenuEnabled Command lets you determine if a given menu item is enabled or grayed. This command is only used as a function.

Parameters

Argument	Description
Menu$	The menu name and/or submenu name that you are interested in.

Returns

This function 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

It takes a string that specifies a menu item or a sum menu item. The string uses this format:

```
"MenuName|SubMenuName"
```

The names must match our menu commands, not including keyboard accelerators and "..." characters. If you omit the pipe and submenu name, then the routine assumes you're working with a top level menu.

Note: This command can be used to check the state of the menu that will be used for new reports by default by placing an "!" before the menu name. This can be done whether the menu item is specified by command or relative location. See second example. See also IsMenuChecked, EnableMenu, KillMenu, AddMenu and ExecuteMenu.

Examples

```
Success = IsMenuEnabled ("Edit|Cut")

`Check if the tables Menu is enabled
If IsMenuEnabled ("Tools|Tables") = 1 Then MsgBox
"Tables Menu Enables"
```

KillMenu

Syntax
`KillMenu Menu$`

Definition
The KillMenu Command removes one of the ReportSmith menu items.

Parameters

Argument	Description
Menu$	The menu name and/or submenu name that you are interested in.

Returns
This function returns a value of 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 sum menu item. The string uses this format:

```
"MenuName | SubMenuName"
```

The names must match our 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 change the state of the menu that will be used for new reports by default by placing an "!" before the menu name. This can be done whether the menu item is specified by command or relative location. "!" removes the menu item for all future reports for this session of ReportSmith, instead of just the active report.

When you use this command as a function (rather than a statement), it requires the return value, and you must enclose its arguments within parentheses.

Example
The following code fragment will remove the File | New menu Item from ReportSmith.

```
Success = KillMenu( "File|New" )
```

LastLoaded$( ) function

Syntax

```
LastLoaded$()
```

Definition

The LastLoaded$( ) function allows you to get the file name and path of the last .RPT file that was loaded into ReportSmith.

Parameters

Not applicable.

Returns

Not applicable.

Comments

None.

Example

```
MsgBox"The last report Loaded was:"+LastLoaded$
```

ListRepVar$( ) Function

Syntax

`ListRepVar$( Index% )`

Definition

ListRepVar returns the name of the report variable at the specified index in the active report. If no report variable exists at the given index then the command returns a NULL string. If this command is executed in a macro linked to the “Before Report Open” event of the Report Object, or the After Data Read event of the application object, the command will return the report variable names for the loading report. ListRepVar can be used to get the list of variables in a report and initialize them before they are prompted.

Parameters

Argument	Description
Index%	Index% specifies the report variable name. If there is no report variable at this index then the function will return a null string.

Example

```
‘ Put the names of the report variables in an array
dim Variables$(20 )
‘ allow for a maximum of 20 report variables
while ListRepVar( CurrentVar )<> ""
                  Variables$( CurrentVar )   = ListRepVar( CurrentVar )
                  CurrentVar = CurrentVar + 1
              wend
```

LoadMacro

Syntax

```
LoadMacro (FileName$,[MacroType%])
```

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, the macro will be loaded into the active report. If no reports are loaded then the macro will be loaded as a global macro. If the macro type parameter is equal to 1 then the macro will be loaded to into the global collection regardless of any open reports.

Parameters

Argument	Description
FileName$	The 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. The following error codes apply: 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

Syntax

```
LoadReport Filespec$, InitString$
```

Definitions

The LoadReport Command loads a report.

Parameters

Argument	Description
Filespec$	The directory path and name of the report (.rpt file) you would like to run
InitString$	You can use the InitString argument to set report variables before SQL is executed. Report variables set in this manner will not prompt the user to enter their values.

This is the format for the *InitString$* argument:

```
@Report_Variable1=<Value1>,@ReportVariable2=<Value2>, ...
```

Returns

This function returns non-zero on error.

Comments

Enter the full path name to the .rpt file in which you want the macro to load, and then make it the active report.

When you use this command as a function (rather than a statement), it requires the return value, and you must enclose its arguments within parentheses.

Example

```
LoadReport "c:\rptsmith\video\
  summary.rpt","@Repvar1=<40>,@Repvar2=<'Smith'>"
```

PrintReport

Syntax

```
PrintReport [StartingPage%], [EndingPage%], [Printer$], [Port$], [Driver$],
  [Copies%]
```

Definitions

The PrintReport command prints the specified pages of the active report to the specified printer.

Parameters

Argument	Description
StartingPage%	The page number of the report that you want to start the print job.
EndingPage%	The page number of the report that you want to end the print job.
Printer$	The name of the printer you would like to use.
Port$	The correct port specification.
Driver$	The correct drive specification.
Copies%	The number of copies to print.

Returns

This function 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$ arguments. To specify a printer, see the Devices section of your win.ini file. You'll see printers in this listed format:

```
Printer=Driver, Port1, Port2
```

When you use this command as a function (rather than a statement), it requires the return value, and you must enclose its arguments within parentheses.

This initial string cannot be used to initialize variables that belong to detail sections in master/detail reports.

Example

```
PrintReport 1,5, "HPLaserJet 4/4M","HPPCL5E","LPT3"
```

Recalc Statement

Syntax
`Recalc`

Definition
The Recalc Statement re-executes the currently active report so that the report surface will reflect changes made to report variables or changes made with an associated dataset control object. When this command is linked to the NewReportDialog Control, it affects only the report to which it is linked.

Parameters
Not applicable.

Returns
Not applicable.

Comments
None.

Example
The Following sets the starting date report variable to today and recalculates.

```
SetRepVar("StartDate",Date$)
Recalc
```

RecordCount

Syntax

`RecordCount()`

Definitions

The RecordCount command gives the total number of records in the data set that belongs to the currently active report.

Parameters

Not applicable.

Returns

This function returns the number of records in the currently active report or 0 if no active report exists.

Comments

It's useful when writing macros that step through data in a report.

Example

The following code fragment counts all of the customers from the city of Palo Alto in a customer database and display the result.

```
GetRandom 1
For X = 1 to RecordCount()
If Field$("CITY") = "Palo Alto" then Count =Count + 1
GetNext
Next X
MsgBox " The total number of customers located in Palo Alto are: " + Str$
  (Count)
```

ResumeEvent

Syntax

```
ResumeEvent ResumeCode%
```

Definition

The ResumeEvent command lets a macro which is linked to an event determine whether the event should be executed or aborted.

Parameters

Argument	Description	
ResumeCode%	Event	Meaning
	0	Abort the event to which this macro is linked.
	1	Perform the event as usual (default).

Returns

Not applicable.

Comments

This only applies to certain events. For most events, 0 means abort and 1 means proceed.

Example

The following stops an event.

```
Resume Event 0
```

Rgb

Syntax

```
Rgb (Red%, Blue%, Green%)
```

Definition

The Rgb command returns a value for the color that the FieldFont command uses for its color assignment.

Parameters

Argument	Description
Red%, *Blue%*, *Green%*	The intensity of red in color, blue in color, or green in color.

Returns

An integer representing the color you've specified.

Comments

This command uses three numbers from 0 to 255. The first number designates the intensity of red, the second number of blue, and the third of green. It provides 16.5 million color combinations. Windows then chooses the closest match to that color.

Example

For example, this 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 Control Panel to select a color. Take note of the Rgb settings for the color you select. Then use these values in the Rgb function to get the color you need.

RunMacro

Syntax

```
RunMacro Macro$, Arguments$
```

Definition

The RunMacro command lets you execute a macro from another macro.

Parameters

Argument	Description
Macro$	Specifies the macro being run.
Arguments$	The arguments defined in your macro if they exist. If your macro contains no arguments, just use ""

Returns

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

Comments

The macro language first looks for an active global macro that matches the name and then searches for active macros that belong to the active report. You can also specify the filename of a .mac file. To be safe and to make sure the correct .mac file is executed, specify the full path of the .mac file.

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

Examples

```
RunMacro "c:\rptsmith\macros\greeting.mac",""

RunMacro "two_arguments,","1,2"
```

SetDataFilter Statement

Syntax
`SetDataFilter` *`MacroName$`*

Definition
This statement allows you to specify a macro that will be 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.

Parameters

Argument	Description
MacroName$	The path and file name of a .mac file or name of an actual global macro to be used as the macro data filter.

Returns
Not applicable.

Comments
This function 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"
```

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

Argument	Description
FileName$	The name of the file to save the report under. If this Argument is omitted then the function will attempt to save the file under the same name it was last saved under. 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 will save. If it is omitted, ReportSmith will save 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

SelectReport Statement

Syntax
`SelectReport` *`ReportTitle$`*

Definition
This statement allows you to set the input focus to the report that has the indicated title.

Parameters

Argument	Description
ReportTitle$	The title of the report to set focus to.

Returns
Not applicable.

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

Example
```
SelectReport"c:\rptsmith\sales.rpt"
```

SetDDEItem Statement

Syntax

```
SetDDEItem TopicName$, ItemName$, Data$
```

Definition

This statement allows you to set or change the data in one of the ReportSmith DDE items. This could be done in response to a DDE request to the specified item.

See Also CreateDDETopic and CreateDDEITem

Parameters

Argument	Description
TopicName$	The name of a user defined DDE topic.
ItemName$	The name of a user defined DDE item.
Data$	The data to set into a DDE item.

Returns

Not applicable.

Comments

A macro that is named as the request response macro for a user defined DDE item can use this function to set the data that a DDE client will receive in response to a DDE request. This technique could be used to create a DDE item that would report the name of the currently active report or other information that might be useful to a DDE client application.

Example

```
SetDDEItem "MyTopic","MyItem","Mydata"
```

SetIncludePath

Syntax
```
SetIncludePath Path$
```

Definition
This command sets the default directory for macro include files.

Parameters

Argument	Description
Path$	The path where RS_BASIC expects to find include files.

Returns
This function returns a nonzero on error.

Comments
The macro compiler will first look in this directory for include files, then it will look in the standard search path. The beginning value is the default macro path in the rptsmith.ini file.

Example
```
SetIncludePath "c:\macros\include"
```

SetRecordLimit Statement

Syntax

`SetRecordLimit` *`Limit`*

Definition

The SetRecordLimit Statement will set the total number of records that ReportSmith will download for any loaded or created report. The limit can be removed by calling the function with a value of 0.

Parameters

Argument	Description
Limit	The maximum number of records that ReportSmith will download for a single report.

Returns

Not applicable.

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 will continue to work against the entire result set, such as selections, sorting, and summary fields, so that for these operations performance will not change.

Example

```
SetRecordLimit 100
```

SetRepVar

Syntax

`SetRepVar` *`ReportVariable$, Value$`*

Definition

The SetRepVar Command stores a value in a report variable in the active report.

Returns

This function returns non-zero on error.

Parameters

Argument	Description
ReportVariable$	A string argument that specifies the name of the report variable being set.
Value$	A string argument that specifies what value to set the report value to.

Returns

This function returns non-zero on error.

Comments

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

Example

```
SetRepVar "Repvar1", "Smith"
```

SetSQL Statement

Syntax

`SetSQL` *`SQL$`*

Definition

The SetSQL statement replaces the SQL string that would normally be generated by ReportSmith.

Parameters

Argument	Description
SQL$	A quoted, valid SQL statement.

Returns

Not applicable.

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 function as the string is not verified before it is executed. This statement can be used along with the GetSQL function in a macro that is linked to the "Before SQL is Executed" event to change the SQL string "On the Fly."

Example

```
SetSQL "Select ENAME, EMP_ID from SCOTT.EMP"
```

ShowRS

Syntax

```
ShowRS Code%
```

Definition

The ShowRS command lets you hide, show, minimize, and maximize ReportSmith.

Parameters

Argument	Description
Code%	A list of the valid values of code and the effect each value has when you send it with a ShowRS command.

Value	Effect
0	Hides ReportSmith and passes activation to another window
1	Activates and displays ReportSmith. If ReportSmith is minimized or maximized, Windows restores it to its original size and position.
2	Activates ReportSmith and displays it as an 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 activates the top-level window in the system's list.
7	Displays ReportSmith as an icon. The window that's currently active remains so.
8	Displays ReportSmith in its current state. The window that's currently active remains so.
9	Activates and displays ReportSmith. If ReportSmith is minimized or maximized, Windows restores it to its original size and position.

Returns

If ShowRS is called with a value of 0, ReportSmith becomes invisible.

Comments

There are some things you should be aware of. If a user minimizes ReportSmith after it opens a modal dialog box, the user won't be able to do anything with ReportSmith until it's brought back to normal size. This may only be possible by sending another ShowRS command from DDE.

It will be impossible for the user to switch to it or execute any of its commands except through macros and DDE. This may be desirable to many users. ReportSmith can still bring up dialog boxes and do any other activity, but because it's not visible, these actions can only be started by DDE commands or macros which are already running.

Example

```
' Force ReportSmith to be an icon
ShowRS 2
```

SumField

Syntax

```
SumField$ (Field$, Table$, GroupLevel, Operation$)
```

Definition

The SumField command gives you the value of a summary field.

Parameters

Argument	Description
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.

Returns

Not applicable.

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 Listbox and then double click on your Summary Field. ReportSmith will use the SumField$ command and fill in the parameters for you. This is how a Summary Field should be referenced in macros.

Example

```
my_var$ = SumField$(QTY_FIELD","OWNER.TABLE_NAME",0,"Count")
```

TotalPages( ) function

Syntax

```
TotalPages
```

Definition

This function returns the number of pages in the curently active report. This command can also be a property when used with the Report Control.

Parameters

Not applicable.

Returns

Not applicable.

Comments

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

Example

```
MsgBox "There are" + str$(Totalpages()) + "in the current report"
```

TotalRecords statement

Returns the total number of records in the active report.

Syntax

```
TotalRecords
```

Example

```
r = TotalRecords()
```

—or—

```
  r = TotalRecords
```

SetDirtyFlag statement

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 report is marked as "dirty" (modified), and the Save Report dialog box appears when the report is closed, prompting the report user to save changes.

Example

The following code will prompting the user to save changes to the report before closing it, whether or not the report has been modified.

```
SetDirtyFlag 1
```

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”
```

Macro Events

You can link a macros to an event. An event is an action in ReportSmith that can trigger the execution of a macro.

Note: You link macros to events and objects using the Macro Links dialog box.

This list describes ReportSmith events, categorized according to objects to which they can be linked.

Application Events

Report Events

Data Field Events

Header/Footer Events

Application Events

You can link a macro to an action, such as the keystroke event, CTRL+R, or to an event which coincides with report loading. You cannot link application macros to report events.

To link macros to specific reports, see Report Events.

The following list describes application events and shows you how to use them to solve reporting problems:

KeyStroke

Before New Report

After New Report

Application Start Up

Before Executing SQL

Before Report Print

Before Report Load

After Report Load

Before Report Save

After Report Save

Before Application Close

On SQL Error

New File Icon Click

SQL Icon Click

After Report Connects

Before Report Close

After Report Close

KeyStroke (Application event)

A macro linked to the KeyStroke event runs when you press the key or key combination you specify.

Note: If the active report has a report macro linked to the same keystroke, the report macro is executed first. If the report macro is intended to replace the application macro (which is also linked to the same keystroke), then the report macro should call the ResumeEvent command passing a value of zero (false). In effect, the ResumeEvent code tells ReportSmith it should not execute other processes normally tied to that event.

As an example, suppose you want to link a macro to CTRL + R to load a report you use often. Also, suppose that the report already has a macro linked to CTRL + R, which tells ReportSmith to print the report.

In this example, the report macro (which prints the report) should call the ResumeEvent command with a value of zero (false), so that the macro (which loads the report) will not load another copy of the same report. This is an example of a report macro overriding the application macro. When you close the report, the global macro that loads the report responds again to the CTRL + R keystroke.

Before New Report (Application event)

A macro linked to this event runs after you choose File | New . This event allows you perform tasks, such as verifying that a user has permission to create reports.

For example, a global macro can display a dialog box that prompts for a password. If the password is correct, the user can continue. If the password is incorrect, an error message appears and the operation is canceled.

You can also use this event to keep track of users attempting to create new reports. For example, a macro may execute a DDE command to notify Visual Basic when users select File | New

A macro linked to this event can abort the creation of a report by calling ResumeEvent 0.

After New Report (Application event)

A macro linked to this event runs when you choose File | New, after ReportSmith executes the report query. You can link to this event to change the default configuration for new reports. For example, you can specify the display mode (draft or presentation), turn the report boundaries on or off, and turn the grid on or off.

You can also link to this event to send DDE commands to another Windows application. For example, you can link a global macro to this event and send a DDE message to PowerBuilder informing it to save all new reports using the filenames it generates.

Application Start Up (Application event)

A macro linked to this event runs when you click the ReportSmith icon to open the application, after ReportSmith displays the splash screen. You can link a macro to this event to set the default ReportSmith environment: add new menu items, disable or remove existing menu items, load daily reports, or execute a menu command, such as File | New or File | Open .

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

A macro linked to this event can call ResumeEvent 0 to cancel the start up of the application.

Before Executing SQL (Application event)

This event occurs each time ReportSmith generates an SQL statement, before that statement executes. You can use it to allow a macro to obtain a copy of the ReportSmith SQL statement each time it changes.

Note: You can retrieve the last SQL statement using the GetSQL function.

In rare cases you might find it necessary to modify the ReportSmith SQL statement. To do this you can use the macro command, SetSQL, to replace the string generated by ReportSmith.

Important: Do not modify the query in the Before Executing SQL event. Instead, we recommend you use the DataSet Control methods (such as SetUserSQL, AddTable, IncludeFields). Changes made during the Before Executing SQL event are lost the next time you generate an SQL statement. Use the SetSQL command with caution. Use it to make small changes to the SQL statement to support servers which have unusual or non-standard SQL requirements.

Before Report Print (Application event)

A macro linked to this event runs after you select the File | Print , before the report is sent to the printer. You can link a macro to this event to perform print-related tasks.

For example, a macro can display a dialog box identifying the printer or display an interactive dialog box that lets the user enter printer parameters, such as margin specifications, number of copies, page orientation, and paper size.

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

Before Report Load (Application event)

A macro linked to this event runs after you select File | Open , after you select a report from the Open Report dialog box. It runs before ReportSmith actually opens and displays the corresponding report. Use this event to determine if the report can actually open.

For example, to insure that a user has only one report open at a given time, you can create a global macro and link it to this event. This saves and closes all active reports before allowing a new report to be opened.

After Report Load (Application event)

A macro linked to this event runs when you select a report using File | Open, after ReportSmith actually opens and displays the report.

For example, display all reports in Draft mode by creating a macro that activates Draft mode each time you open a report.

You can also log all of the report files opened by a certain end user. A macro linked to this event could use the last loaded commands, write the title and the time opened to an ASCII file.

Before Report Save (Application event)

A macro linked to this event runs after you select File | Save or File | Save As, before ReportSmith actually saves the active report. If this is the fist save or if you are saving an existing report under a different name, 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 certain files that you do not want modified. For example, to prevent users from modifying all reports created in the month of May, have the macro display a message indicating that the report can not be modified, and cancel the File | Save or File | Save As operation. Similarly, you can use this event to verify sufficient disk space, and then display a warning message if disk space is unavailable.

After Report Save (Application event)

A macro linked to this event runs after you select File | Save or File | Save As, after ReportSmith saves the active report. When you link a macro to this event, you can close reports immediately after ReportSmith saves them. You can also use this event to automatically create backup copies of saved reports in a backup directory.

Before Application Close (Application event)

A macro linked to this event runs after you select File | Exit, before ReportSmith actually closes. You can link a macro to this event to prevent 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 display a message informing the user to close the driving application first.

You can also use this event to keep a running log of the dates and times ReportSmith closes, and the names of the users attempting the close. For example, you can have a macro execute a DDE command to notify Visual Basic when a user selects File | Close .

Suppose the ReportSmith users in your company work on several reports simultaneously. You can link a macro to this event to automatically save all open reports before ReportSmith closes, instead of requiring ReportSmith to prompt the user to save each open report individually.

On SQL Error (Application event)

Each time an SQL execution error occurs, the On SQL Error event is triggered. A macro linked to this event can execute the LastError function to retrieve error text (usually from an ODBC driver). It can attempt to fix the error by changing the report query from which it originated. If that macro executes a ResumeEvent command with an argument of 0, the FixReport Dialog box will not appear and the SQL will re-execute.

Note: If the macro does not correct the error in FixReport, this event is called again. Use caution when using ResumeEvent 0 to avoid being caught in an endless loop.

New File Icon Click (Application event)

This event is called when the user presses the New Report button on the toolbar. A macro linked to this event can call ResumeEvent command passing a 0 (FALSE) value which would cause the regular new report processing to be aborted. This might be done to replace the standard new report behavior with custom behavior, defined by a macro.

SQL Icon Click (Application event)

This event is called when the user presses the SQL button on the toolbar. A macro linked to this event can call ResumeEvent command passing a 0 (FALSE) value which would cause the regular SQL processing to be aborted. This might be done to replace the standard SQL viewing/editing behavior with custom behavior defined by a macro.

After Report Connects (Application event)

This event is called after a report connects, before ReportSmith executes the report query. It is the application level version of the report event, "Before Report Open." This event is handy for macros whose purpose it to modify a report query before running the report. A control object could be associated with a loading report using the .setfrom loading command.

Before Report Close (Application event)

This event is called before a report closes.

After Report Close (Application event)

This event is called after a report closes.

Report Events

A report macro is linked to a specific report and becomes a part of that report. You link report macros to report events.

This list describes report events and shows you how to use them in common reports. Click on an item in the list to see a description.

KeyStroke

Before Report Open

After Report Open

Before SQL Execution

Before Print

Before Report Save

Before Report Close

On SQL Error

Selecting a Menu Item

KeyStroke (Report event)

You can link a macro to a 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 in the active report.

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

Note: For more detailed information and for examples of how you can use the KeyStroke event, see the Application Event KeyStroke

Before Report Open **(Report event)**

A macro linked to this event runs after you open a specific report, before ReportSmith executes the report query. You can use this event to automatically set report variables for the selection criteria.

Note: If the macro linked to this event calls the Set report variable command, it sets the report variables for the report being loaded, instead of the active report.

After Report Open (Report event)

A macro linked to this event runs after you open the specific report to which the macro is linked, 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 opens or require the opening of a specific report to trigger the loading of additional, related reports.

Use this event to force a certain 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, 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 when you open it.

Before SQL Execution (Report event)

This event is the same as the application level event, except that it only applies to the report to which the macro is linked.

Before Print (Report event)

A macro linked to this event runs after you select File | Print , before ReportSmith actually prints the specific report to which the macro is linked.

For example, if you have a specific report that is particularly large and you do not want to tie up the printer, you can link a macro to this event to automatically send that report to another printer.

You can also use this event to warn users that the specific report is particularly large, giving them the opportunity to cancel the print operation.

Before Report Save (Report event)

A macro linked to this event runs after you attempt to save a specific report to which the macro is linked, before ReportSmith actually saves it. If this is the first save, the macro runs before the dialog box prompts you for a filename.

Link a macro to this event to prevent a user from overriding a specific report that you do not want modified. The macro can display a message indicating that the report cannot be modified, and then cancel the save operation. You can also use this event to send a DDE command to through your company e-mail to notify other users that a new version of the report is available.

Before Report Close (Report event)

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

For example, suppose the macro that runs after you open a report changed the display mode from Presentation to Draft. You can link another macro to this event to restore the display mode to Presentation.

On SQL Error (Report event)

Each time an SQL execution error occurs, the On SQL Error event is triggered. A macro linked to this event can execute the LastError function to retrieve error text (usually from an ODBC driver). It can attempt to fix the error by changing the report query from which it originated. If that macro executes a ResumeEvent command with an argument of 0, the FixReport Dialog box will not appear and the SQL will re-execute.

This event applies only to the report to which the macro is linked.

Note: If the macro does not correct the error in FixReport, this event is called again. Use caution when using ResumeEvent 0 to avoid being caught in an endless loop.

Selecting a Menu Item (Report event)

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 linked, before the menu item task occurs. Use the ResumeEvent command to determine if the corresponding action is executed.

Suppose you have a help file for a specific report. You can link a macro to this event to replace the ReportSmith help file with the new help file, only for the specific report.

Data Field Events (Report events)

A macro linked to data field events runs when ReportSmith generates a value for the data field object to which the macro is linked.

The primary purpose of the Display event is to allow conditional formatting. To perform conditional formatting, it must be based on criteria to which values in report columns can correspond.

When creating a conditional formatting macro, use the FieldFont and FieldText macro commands to tell the macro how to format the values that fulfill the criteria.

Note: If more than one macro is linked to a particular data field, only the last macro linked will be run. Macro arguments are not valid for this link.

Header/Footer Events (Report events)

Macros can be linked to the Group Header or Group Footer objects. These objects currently have one event, the Creation Event.

When you link a macro to the Header/Footer Event you must choose the grouping level of the header or footer to which you want to link. In the Items Listbox of the Macro Links Dialog Box,you can view a list of the groups in the active report. For a macro to execute each time you create a Department group footer, select the Group Footer object, and the Department group, and press the Link button.

A macro linked to the Header/Footer Event can call ResumeEvent, passing an argument of 0 (FALSE) to suppress the creation of an individual group header or footer based on the data in the report. The current record in the report is set to the first record of the group when the macro is called. The current record of the dataset is the first record of the specified group. For example, if you group by department and want to know what department header or footer is being called, execute Field$ ("Dept") to retrieve the department name.

ReportBasic Programming Language

Introduction to Macros offers a brief tour of explanation of working with macros.

Overview of Creating Macros shows you how to create, load and link a simple macro, and how to save a macro to a .MAC file.

Using Control Objects shows you how to use control objects in your ReportBasic macros.

Command Reference lists all topics available for the ReportBasic Programming Language.

Macro Events lists all topics available for Macro Events.

What is ReportBasic? defines the ReportBasic programming language.

When to use ReportBasic lists common uses for ReportBasic.

Note: If you use the Index button at the top of this window (or the Find command within the Index), Help will search only those topics pertaining to ReportBasic Macro Language Help. General ReportSmith Help topics can be found in ReportSmith 3.0 Help. (Check the title bar to quickly determine whether you are viewing Macro Help or ReportSmith 3.0 Help.) A thorough discussion of macro concepts and techniques can be found in *Creating Reports*.

Using Control Objects

See Also

ReportBasic control objects are used by employing their properties and methods. ReportBasic provides three control objects:

- DataSet Control Object
- Report Control Object
- NewReportDialog Object

You can use the commands in the macro facility either as functions that require a return value, or as statements, which do not require a return value. You can use many command as both, but some can be used only as a function or a statement. Use a control object for various tasks such as making a connection, setting your selection criteria, or changing the current report page.

The Dataset Control Object allows you to define a description of report data. It is used by employing its properties and methods to:

- Make a connection
- Set a list of tables
- Set selection criteria
- Obtain a list of available tables
- Get a list of table owners
- Create a default columnar report

You can use a Report Control Object to:

- Check the total number of pages
- Change the page displayed
- Recalculate the report

You can use a NewReportDialog Control Object to:

- Prompt a user to choose a report type
- Grant users access to style editing

See Also

Creating a DataSet Control

Creating a Report Control Object

Creating a NewReportDialog Object

Properties and Methods

Creating a DataSet Control Object

You can create a control with Dim and Global statements.

For all subroutines in a macro to be able to use your control, place the Dim statement outside of all subroutines. A macro declared thus has modular scope, and is destroyed when the macro ends.

To use the control in a single function, place the Dim statement inside that function. This gives the control local scope. It will be created when the function is called, and destroyed when the function is complete. You could then use the same name for the local DataSet controls of other functions.

You can use the Global statement to create a DataSet control that is known to all macros in ReportSmith, called global scope. The Global statement must appear in all modules that refer to the global DataSet. Before the object is used it must be outside all functions and subroutines.

Properties and Methods

ReportBasic controls are used by employing their properties and methods. Use the following references to learn how to employ properties and methods.

Introduction to Macros

See Also

Macros allow you to save time, enhance productivity and add complex features to your reports. Macros are programs that you build using ReportBasic to automatically perform tasks. You can link a macro to an event which the macro can run.

For example, you can display a dialog box which prompts for options each time a user opens a report. In this case, the dialog box is the result of the macro, while the user opening the report is the event.

You can write a macro to perform many tasks, including the following:

- Automatically load and print reports.
- Customize ReportSmith by creating a custom dialog box, hiding or disabling menu items.
- Drive ReportSmith from PowerBuilder or VisualBasic.
- Create derived fields.
- Create customized prompts such as prompting a user for a password.
- Schedule report printing.
- Perform complex calculations.

Note: For more information, see the *User's Guide.*

See Also

Global and Report Macros

Using Macros With Other Windows Applications

What is ReportBasic?

Overview of Creating , Linking, and Loading a Macro

This section shows you how to create a simple macro, link a macro to an event, save a macro to a .MAC file, and load a macro.

Creating Macros

To create a macro:

1. Select Tools | Macro to open the Macro Commands dialog box.
2. Enter a name for your macro into the Macro Name text box and press New.

Note: The macro name cannot be the same as a Basic reserved word, or errors result.

3. In the Edit Macro dialog box, insert commands into the Macro Formula text box by dragging and dropping them from the upper list boxes. Once you complete the formula, press the Test button.
4. After you receive a success message, press OK to return to the Macro Commands dialog box. Press Run to see your macro.

Linking Macros

To link a macro to an event:

1. Select a macro in the Macro Commands dialog box and choose Links. Select an event to which to link your macro, set necessary options (such as choosing a keystroke sequence if the event to which you link it is a KeyStroke event).
2. If you like, you can link a macro to more than one event. Press the Link button each time you add an event. The number of links appears in the Link Number list.
3. Press OK to return to the Macro Commands dialog box.

Saving Macros

To save a macro to a MAC file:

When you save a macro in a .MAC file, you save only the macro code itself. Links to events and objects are not stored in the .MAC file since they might not be appropriate for other users.

1. In the Macro Commands dialog box, choose a macro from the list of active macros and choose Save As.
2. Use the Drives and Directories boxes to specify the path where you want to store the macro, and enter a name for your macro file into the File Name box, followed by the three character extension .MAC. Press OK.

Loading Macros

To load a macro:

Tip: To successfully load a macro, a macro with the same name in the first line of its routine must not appear in the Active Macros list box. If you attempt to do this, ReportSmith displays an error message. You must first remove the macro from the active list.

1. Select Tools | Macro to display the Macro Commands dialog box, and choose Load.
2. In the Load Macro dialog box, double-click the macro you want. This becomes the active macro in the Macro Commands dialog box.
3. To edit the macro, choose Edit to display the Edit Macro dialog box, make a valid change in the Macro Formula text box and press OK.

Global and Report Macros

See Also

You can create two kinds of macros:

- Global
- Report.

A global macro belongs to the ReportSmith application. It appears in the active list each time your run ReportSmith. A report macro belongs to a specific report. It becomes a part of that report. Global macros let you automate global tasks, while report macros let you automate tasks specific to a given report.

You can create global macros to customize ReportSmith. For example, you can combine loading, printing and closing a report into one step. To do this, create a global macro containing an argument string which takes a series of report names, separated by commas. This same macro can also display a dialog box that prompts for report names. When you run the macro, it opens, prints and closes each report specified in the dialog box.

Report macros let you automate tasks that are specific to a given report. For example, you can create a report macro that lets only a certain user access a confidential report. The macro can prompt for a password, or verify the value of a DOS environment variable. For example, you might have this line in your AUTOEXEC.BAT:

```
SET USER=John Doe
```

In this case, the macro opens the report only if the environment variable is set to John Doe.

```
' This code will abort the open event if the user is not John Doe
' Get the user name
USER$=ENVIRON$("USER")
  IF USER$ <> "John Doe" THEN
  RESUME EVENT 0
  ENDIF
```

See Also

Introduction to Macros

Using Macros With Other Windows Applications

Uses for Global Macros

Uses for Report Macros

Uses for Global Macros

- Prompting for Passwords
- Customizing ReportSmith

You can create a global macro that prompts for a password when a user opens ReportSmith. If the password is correct, the macro opens ReportSmith and loads specific confidential reports. If the password is incorrect, the macro denies access to those reports, and closes ReportSmith.

You can create a global macro that hides the toolbar, executes a certain menu command, or adds a menu item that will open and print standard reports. Or, add a DDE topic that will open reports in response to DDE pokes from another application.

Uses for Report Macros

- Creating Derived Fields
- Scheduling Events

You can create a report macro that steps through the fields in a report column and produces a summary or derived field. For example, suppose the Sales Department might have a sophisticated commission structure containing different multipliers for different quotas. You can write a macro that calculates commissions based on the commission structure, and place that value in a report.

You can create a report macro which behaves dynamically according to the information it receives. For example, for an unusually large report, you can create a macro that prevents a user from opening the report during peak time, and then displays a brief explanation. In this case, the macro behaves differently according to the time of day.

Using Macros with other Windows Applications

See Also

Report macros let you extend to other Windows applications. For example, suppose you have a report with a column listing part numbers, and an Excel spreadsheet with a table listing these part numbers and their corresponding part names. You can write a macro that changes the part number to the part name in your report. To do this, link the macro to the Display event and use the FieldText command to change the text of each record. You can then perform a DDE request to your Excel table to locate the corresponding part name for each part number.

You can also add C functions from a Windows DLL (Dynamic Link Library) into a report macro. If you have a Windows DLL containing functions that perform complicated financial computations, you can create a report macro that uses those functions to calculate the data in your report.

See Also

Introduction to Macros

Global and Report Macros

Using Dynamic Data Exchange (DDE) with ReportSmith

What is ReportBasic?

See Also

ReportSmith has licensed the Softbridge Basic Language (SBL) from Mystic River Software, Inc. to provide ReportSmith users with a complete high-level programming language. SBL contains similar commands to Microsoft's Visual Basic programming language. ReportSmith has also added commands to SBL that were designed with reporting in mind. This combined command set is called ReportBasic, a complete programming language designed specifically for report manipulation.

See Also

Examples Using ReportBasic

When to use ReportBasic

Because it is a complete programming language, ReportBasic can be used for a variety of tasks. The following is a list of the more common uses for ReportBasic:

Calculating Derived Fields With ReportBasic

Conditional Formatting With ReportBasic

Customizing User Interface With ReportBasic

Automatically Processing Reports With ReportBasic

Creating User-Defined Functions With ReportBasic

Using DDE With ReportBasic

Calculating Derived Fields with ReportBasic

With ReportBasic you can create derived fields that are not stored in the database, but are a by-product of a calculation using the values stored in your database fields.

Conditional Formatting with ReportBasic

ReportBasic lets you highlight a value if it meets a certain criteria in a report. For example, you might want to display all sales transactions in a report that are greater than a stated quota to print in a bolder, larger font than those that fall below the quota. ReportBasic can also be used to warn a report reader that invalid data exists in a database. Suppose a mailing list database contains a number of null values in the state field. ReportBasic can print the value "NO STATE CODE" in place of the null state code value.

Customizing User Interface with ReportBasic

You can use ReportBasic to create custom user interfaces such as dialog boxes that contain user entry fields, command buttons and check boxes. A report user can build a complex SQL query with a few mouse clicks, with the help of the report designer who is familiar with ReportBasic. ReportBasic can also be used to disable ReportSmith menus, menu items and toolbar icons. A report designer may want to do this to prevent a user from modifying a report, or to add specific menu items that run specific macros.

Automatically Processing Reports with ReportBasic

ReportBasic can be used to load a number of reports, send them to a printer and close those reports once they have been printed. A user can cause such a macro to execute in response to a keystroke, or another ReportSmith event. Or, a macro might be programmed to run reports at midnight, when user traffic is low.

With ReportBasic, you can create your own DDE topics and items. Macros can be assigned to run in response to a change in one of the items, or in response to a request for data from another application.

Creating User-defined Functions with ReportBasic

Advanced users can create ReportBasic functions that can be called by other ReportBasic scripts or a ReportSmith event. Users can also call *.DLL functions, created in other development environments, that can be used within ReportBasic.

Dynamic Data Exchange (DDE)

See Also

Dynamic data exchange (DDE) is a process by which two applications communicate and exchange data. One application can be your Basic program. To "talk" to another application and send it data, you need to open a connection, called a DDE channel, using the statement, **DDEInitiate**. The application must already be running before you can open a DDE channel. To start an application, use the **Shell** command.

DDEInitiate requires two arguments: the DDE application name and a topic name. The DDE application name is usually the name of the .EXE file used to start the application, without the .EXE extension. For example, the DDE name for Microsoft Word is "WINWORD". The topic name is usually a filename to get or send data to, although there are some reserved DDE topic names, such as **System**. Refer to the documentation for the application, to get a list of the available topic names.

After you have opened a channel to an application, you can get text and numbers (**DDERequest**), send text and numbers (**DDEPoke**) or send commands (**DDEExecute**). When you have finished communicating with the application, you should close the DDE channel using **DDETerminate**. Because you have a limited number of channels available at once (depending on the operating system in use and the amount of memory you have available), it is a good idea to close a channel as soon as you finish using it.

The other DDE command available in ReportBasic is **DDEAppReturnCode**, which you use for error checking purposes. After getting or sending text, or executing a command, you might want to use **DDEAppReturnCode** to make sure the application performed the task as expected. If an error did occur, your program can notify the user of the error.

See Also

DDEAppReturnCode

DDEInitiate

DDEExecute

DDEPoke

DDERequest

DDETerminate

Using DDE with ReportBasic

ReportSmith, along with ReportBasic, can serve as either a DDE server or client. A developer may want to build a number of reports with ReportSmith and build DDE commands within the application that calls ReportSmith or ReportSmith Runtime to print those reports when the user clicks a button or selects a menu option. This can easily be done from within applications developed with Visual Basic or PowerBuilder. You can perform DDE executes, pokes and requests and create your own DDE topics and items.

DataSet Methods Reference

Methods are functions or statements that perform actions on the control.

The following list offers a brief description of each method. To view the syntax, definition and parameters, and to see returns, comments and specific examples of a method, double-click on it.

Method	Description
AddGroup	Adds grouping criteria.
AddSort	Adds a sorting criteria.
AddSummary	Adds a summary field.
AddTable	Adds a table.
Connect	Replaces previous connection information with new information.
CreateReport	Allows a DataSet control with a defined query to create one of four different report types.
Disconnect	Removes a connection previously set with the Connect Method.
Field$	Returns the value of the specified data field for the current method.
GetAllField$	Returns a list of fields available for the specified field in the specified table.
GetColumnAlias	Returns the alias for the specified field in the specified table.
GetDataSources	Returns a list of all available data sources.
GetFieldList$	Returns a list of fields in the given table.
GetGroup$	Returns a string providing information about grouping.
GetSort$	Returns a string indicating the sorting criteria at the given level.
GetSQL$	Returns the last SQL statement executed for the dataset control object.
GetSummary$	Returns a string providing information about a summary field.
GetTable$	Returns a string describing the table at the specified index.
GetTableAlias$	Returns the alias for the specified table.
GetTableLink$	Returns a string providing information about the table link.
Include Field$	Set the list of fields that should be included in a table.
LinkMacro	Links a macro in an active list to the specified object, even and item.
Load	Replaces connection information with that specified by the *Filename$* parameter.
LoadMacro	Allows a macro to be loaded into the active macro list from a .MAC file.
Recalc	Execute the SQL for a DataSet.
RemoveGroup	Removes a grouping criteria from a report.
RemoveSort	Removes the sorting criteria at the given level.
RemoveSummary	Removes a summary field from a report.
RemoveTable	Removes a table at the specified index.
RemoveTableLink	Removes the table link at the specified index from the dataset control object.
ReplaceTable	Replaces one table in a report with another.
Save	Store the object in a file.
SetColumnAlias	Sets or changes the Alias for a column in a table.
SetFromActive	Associates the object with the active report.
SetFromLoading	Associates the dataset control object with a report being loaded.

SetTableAlias	Sets or changes the Alias for a table in a report.
SetTableLink	Sets a logical link between two tables that are part of this DataSet.
SetUserSQL	Places the dataset control object into user entered SQL mode.
TestSelection$	Returns a string telling how many records would be selected with the current selection criteria.

AddGroup Method

Syntax

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

Definition

The AddGroup Method adds grouping criteria at the specified level.

Parameters

Argument	Description
Table$*, *DataBase$, *Field$*	The Table$, DataBase$ and Field$ serve to identify the field to be grouped upon.
Level	The level argument specifies the grouping level that you want information about. If an invalid index is specified, a null string will be returned and the error$ property will be set to indicate the error. The level argument 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.
Type	Type specifies the type of grouping: 0 Same value 1 Every n records (*n* is the *NumRecs* argument) 2 Daily 3 Monthly 4 Weekly 5 Annually 6 Quarterly 7 Hourly 8 Every minute 9 Every second 10 Second/10 11 Second/100 12 Second/100 Types 2-12 are only valid for date and/or time fields.
NumRecs	When grouping by every *n* records, this specifies how many records per group.

Returns

This function will return a zero on success, a non-zero on error.

Comments

If a group exists at the given level then 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
```

AddSort Method

Syntax

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

Definition

This AddSort Method adds a sorting criteria to the current dataset at the current level.

Parameters

Argument	Description
Table$	The path and file name for local databases. For database servers it takes the form: `Owner.TableName` Or for Local databases like DBase and Excel, the *Table$* parameter is the File Name of the local database file.
Database$	For local databases or servers that don't require that a database be specified, the *database$* parameter should be set to a null string.
Column$	The field that is being sorted.
Ascending	The *Ascending* argument should be set to zero to sort from largest to smallest, non-zero to sort from smallest to largest.
Level	The level indicates its priority among other sorting criteria for this dataset. Valid values for this argument are 1 to the number of current sorting criteria +1.

Returns

If an invalid index is specified the function will fail and return an error. This function will return a zero on success, a non-zero on error.

Comments

Valid levels are 1 to 1+ the current number of sorting criteria.

Example

```
MyData.AddSort "dbo.emp","hr","DEPT_ID",1,1
```

AddSummary Method

Syntax

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

Definition

The AddSummary Method adds a summary field to the specified grouping level and index.

Parameters

Argument	Description	
Table$*, *DataBase$, *Field$*	These arguments identify the field to be summed.	
Level	The level argument specifies the grouping level for creating a summary field.	
	0	Entire report group.
	1	Primary grouping criteria
	2	Secondary grouping criteria
Type	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

If an invalid index is specified, a null string will be returned and the error$ property will be set to indicate the error.

Comments

Valid values for levels are zero to the number of groups defined.

Example

```
MyData.AddSummary"dbo.emo" ,"database" , "SALARY", 1, 1
```

AddTable Method

Syntax
`[object].AddTable Table$, DBase$`

Definition
The AddTable Method adds a table to a dataset control object.

Parameters

Argument	Description
Table$	The *Table$* parameter defines a string of the form: Owner.TableName. (For local databases like DBase and Excel, the *Table$* parameter is the File Name of the local database file.)
DBase$	The *DBase$* parameter is for data servers that require databases. For servers that don't require a database, it should be set to a NULL string. In 2.0 + this argument can now take the path to a local database file or both path and filename can be combined in *Table$* as before.

Returns
This function will return a zero on success and a non-zero on error.

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

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

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.*

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 Method

Syntax

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

Definition

The Connect Method replaces any previous connection information in a DataSet control with the supplied connection information.

Parameters

Argument	Description
Type$	The Type parameter can take on the following values:

0	Named Connection
1	Reserved
2	DBASE
3	EXCEL
4	PARADOX
5	ASCII
6	SQLSRVR
7	ORACLE
8	DB2
10	SYBASE
11	BTRIEVE
12	GUPTA
13	INGRES
16	TERADATA
17	DB2/GUPTA
19	UNIFY
40	DBASE (via ODBC)
41	EXCEL (via ODBC)
42	PARADOX (via ODBC)
48	BTRIEVE (via ODBC)
55	Generic ODBC driver (use this for most ODBC connections)
61	PARADOX (via IDAPI)
62	DBASE (via IDAPI)
67	INTERBASE (via IDAPI)

Argument	Description
Server$	Name of the data server or local data file name.
UserID$	Name of the user to make the connections that require a user id.
Pswrd$	The user's password.
DBase$	The database name for connections that require a database. (Null for Oracle.)

Returns

The return code for this function should be 0 on success. If it is not zero, then the Error$ property will contain text that describes the error.

Comments

Note that for Oracle, *DBase$* should be set to a Null String.

Using the connect function will clear any previously defined tables or table columns.

Example

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

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 (the insert field dialog will be brought up).
2	A crosstab report (the crosstab dialog will be brought up).
3	A form report (will use default form layout).

Returns

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

Disconnect Method

Syntax

`[object].Disconnect`

Definition

The Disconnect method allows you to remove a connection that was previously set with the Connect method.

Parameters

Not applicable.

Returns

The function will return error if an active report is using the connection. The return code for this function should be 0 on success. If it is not zero, then the Error$ property will contain text that describes the error.

Comments

This function will only execute successfully if there are no other DataSet controls or reports using the same connection.

Note: This command will return 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, we are assuming that we have a server called *X:ORASRV*, 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.AddTableSBL "SCOTT.DEPARTMENT"," "

  `Create the Report Object
NewData.Commit
PrintReport 0, 0" ", " "

  CloseReport 0
NewData.Disconnect
End Sub
```

Field$ Method

Syntax
```
[object].Field$(FieldName$)
```

Definition
The Field$ Method returns the value of the specified data field for the current dataset record. You can set the current record with the dataset record property.

Parameters

Argument	Description
FieldName$	The name of the field to reference data for.

Returns
The Field$ Method 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"))
```

GetAllField$ Method

Syntax
```
[object].GetAllField$(Table$,DBase$)
```

Definition
The GetAllField$ Method returns a list of all fields that are available in the given table.

Parameters

Argument	Description
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.

Returns
The GetAllField$ Method returns a list of all fields that are available in the given table.

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 left blank.

See the GetField command to get an individual field out of the list of fields.

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

GetColumnAlias$ Method

Syntax

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

Definition

The GetColumnAlias$ Method returns the alias for the specified field in the specified table.

Parameters

Argument	Description
Table$	The *Table$* parameter defines a string of the form: `Owner. TableName` Or, for local databases like DBase and Excel, the *Table$* parameter is the File Name of the local database file.
Database$	The name of the database that contains the table for connections that have databases.
Column$	The column to set an alias for.

Returns

The GetColumnAlias$ Method 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")
```

GetDataSources$ Method

Syntax

`[`*`object`*`].GetDataSources`

Definition

The GetDataSources$ Method returns a list of all of the datasources available to ReportSmith, including ODBC Sources, separated by commas.

Parameters

Not applicable.

Returns

The GetDataSources$ Method returns a list of all of the datasources available to ReportSmith.

Comments

None.

Example

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

GetFieldList$ Method

Syntax
`[object].GetFieldList$(Table$,Dbase$)`

Definition
The GetFieldList$ Method returns a list of all fields that have been included in the given table.

Parameters

Argument	Description
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.

Returns
The GetFieldList$ Method 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 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 left blank.

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

GetGroup$ Method

Syntax
[*object*].GetGroup$(*level*)

Definition
The GetGroup$ Method returns a string that provides information about grouping at the specified level.

Parameters

Argument	Description
level	The level argument 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.

Returns
The GetGroup$ Method returns a string that provides information about grouping at the specified level.

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

Example
```
PrimaryGroup$=MyData.GetGroup(1)
```

GetSort$ Method

Syntax

```
[object].GetSort$(Level)
```

Definition

The GetSort$ Method returns a string that indicates the sorting criteria at the given level if one exists.

Parameters

Argument	Description
level	The level argument specifies the sorting level that you want information about where 0 is the entire report, 1 is the primary sorting criterion, 2 is the secondary sorting criterion, and so forth.

Returns

The GetSort$ Method returns a string that indicates the sorting criteria at the given level if one exists.

Comments

Valid values for the level argument are 1 to the number of current sorting criteria. If an invalid index is specified, a null string will be returned and the Error$ property will be set to "Invalid Index."

Example

```
PrimarySort$=MyData.GetSort$(1)
```

GetSummary$ Method

Syntax

[*object*].GetSummary$(*Level*, *Index*)

Definition

The GetSummary$ Method returns a string that provides information about a summary field at the specified grouping level and index.

Parameters

Argument	Description
level	The level argument 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	The Index matches the order in which the tables are originally added.

Returns

If an invalid index is specified a null string will be returned and the error$ property will be set to indicate the error.

Comments

None.

Example

```
SecondSummary$=MyData.GetSummary(1,2)
```

GetSQL$ Method

Syntax
`[object].GetSQL$()`

Definition
The GetSQL$ Method returns the last SQL statement that was executed for this dataset control object.

Parameters
Not applicable.

Returns
The GetSQL$ Method returns the last SQL statement that was executed for this dataset control object.

Comments
None.

Example
```
MySQL$=MyData.GetSQL$()
```

GetTable$ Method

Syntax

`[object].GetTable$(Index)`

Definition

The GetTable$ Method returns a string that describes the table at the specified index if possible.

Parameters

Argument	Description
Index	The Index matches the order in which the tables are originally added.

Returns

The GetTable$ Method returns a string that describes the table the specified index if possible.

Comments

If an invalid index is given, this function will return a null string and the Error$ property will be set to an appropriate error message.

Example

```
SecondTable$=MyData.GetTable$(2)
```

GetTableAlias$ Method

Syntax

```
[object].GetTableAlias$(Table$, DBase$)
```

Definition

The GetTableAlias$ Method returns the alias for the specified table.

Parameters

Argument	Description
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.

Returns

The GetTableAlias$ Method returns the alias for the specified table.

Comments

For local databases or servers that don't require that a databases be specified, the Database parameter should be set to a null string.

Example

```
TableAlias$=MyData.GetTableAlias$("dbo.emp","hr")
```

GetTableLink$

Syntax
`[object].GetTableLink$(Index)`

Definition
The GetTableLink$ Method returns a string that provides information about the table link at the given index if one exists.

Parameters

Argument	Description
Index	The index of the link to retrieve information for.

Returns
The GetTableLink$ Method 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 will be returned and the Error$ property will be set to indicate the error.

Example
```
SecondTableLink$ = MyData.GetTableLink(2)
```

IncludeFields$ Method

Syntax

`[`*`object`*`].IncludeFields$,`*`Table$, DBase$, IncludeList$`*

Definition

The IncludeFields$ Method adds columns from a table to your dataset.

Parameters

Argument	Description
Table$	The *Table$* parameter defines a string of the form: `Owner. TableName` Or, for local databases like DBase and Excel, the *Table$* parameter is the File Name of the local database file.
DBase$	The database that contains the table for connections that have databases.
IncludeList$	A list of fields in a table to include as part of a table.

Returns

Not applicable.

Comments

The field list should be one string with the names of the fields to be included separated by commas. The names should be provided exactly as they appear in our dialog boxes. It is case sensitive.

Example

```
MyData.IncludeField$ "dbo.emp","Pubs","First_Name,
Last_Name, Dept, Emp_Id"
```

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 then the macro must be in the list of active global macros. Otherwise, it must appear in the list of macros for the report that the Dataset object is associated with.

Parameters

Argument	Parameters
MacroName$	The *MacroName$* parameter defines the name of a macro in the list of active macros for which you want to define a link.
Object%	A number that specifies the object to link the macro to.
Event %	A number that specifies the event to link the macro to.
IgnoreDialog%	If *IgnoreDialog%* is specified and is non-zero then the macro dialog will not be updated by this method.
Item$	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
```

Load Method

Syntax

`[object].Load Filename$`

Definition

The Load Method replaces any previous connection information in a DataSet control with information from the file that is specified by the *Filename$* parameter. Note that the file must have been created with the save method and can have any extension.

Parameters

Argument	Description
Filename$	The name of a file that the data set control object should be read from.

Returns

The return code for this function should be 0 on success. If it is not zero, then the Error$ property will contain text that describes the error.

Comments

Note that the file must have been created with the Save method and can have any extension.

Example

```
MyData.Load("c:\RPTSMITH\MyData.DSC)
```

LoadMacro Method

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 weather 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

Argument	Description
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

Recalc Method

Syntax
`[`*`object`*`].Recalc`

Definition
The Recalc Method re-executes the SQL for this DataSet Control Object.

Parameters
Not applicable.

Returns
Not applicable.

Comments
Because this command is associated with a dataset control, any changes to the result will not be reflected in associated reports until a report level recalc is performed. This command is generally used with dataset control functions as stand alone queries that do not have associated reports.

Example
`MyData.Recalc.`

RemoveGroup Method

Syntax
`[object].RemoveGroup level`

Definition
The RemoveGroup Method removes a grouping criteria from a report.

Parameters

Argument	Description
level	The level argument specifies the grouping level that you want to remove where 0 is the entire report group. 1 is the primary grouping criteria, and so forth.

Returns
This function will return a zero on success, a non-zero on error.

Comments
If an invalid index is specified, a null string will be returned and the Error$ property will be set to indicate the error.

Example
```
MyData.RemoveGroup 1
```

RemoveSort Method

Syntax

`[object].RemoveSort Level`

Definition

The RemoveSort Method removes the sorting criteria at the given level if one exists.

Parameters

Argument	Description
level	The level argument specifies the grouping level that you want to remove where 0 is the entire report group, 1 is the primary grouping criteria, 2 is the secondary grouping criteria, and so forth.

Returns

If an invalid index is specified the function will return non-zero. This function will return a zero on success, a non-zero on error.

Comments

Valid values for the level argument are 1 to the number of current sorting criteria.

Example

```
MyData.RemoveSort 1
```

RemoveSummary Method

Syntax

`[object].RemoveSummary level, Index`

Definition

The RemoveSummary Method returns a summary field from a report.

Parameters

Argument	Description
level	The level argument 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	The Index matches the order in which the tables were originally added.

Returns

If an invalid index is specified a null string will be returned and the Error$ property will be set to indicate the error.

Comments

None.

Example

```
MyData.RemoveSummary 1, 2
```

RemoveTable Method

Syntax

`[object].RemoveTable Index`

Definition

The RemoveTable Method removes the table at the specified index if possible.

Parameters

Argument	Description
Index	The Index matches the order in which the tables were originally added.

Returns

This function will return a zero 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 Method

Syntax

`[object].RemoveTableLink level, Index`

Definition

The RemoveTableLink Method removes the table link at the specified index from the dataset control object.

Parameters

Argument	Description
level	The level argument specifies the grouping level that the summary is operating on where 0 is the entire report group, 1 is the primary grouping criteria, 2 is the secondary grouping criteria, and so forth.
Index	The index of the link to retrieve information for.

Returns

Not applicable.

Comments

If an invalid index is specified, a null string will be returned and the Error$ property will be set to indicate the error.

Example

```
MyData.RemoveTableLink 1, 2
```

ReplaceTable Method

Syntax
```
[object].ReplaceTable (Table$, Database$, NewTable$, NewDataBase$)
```

Definition
The ReplaceTable Method replaces one table in a report with another.

Parameters

Argument	Description
Table$	The *Table$* parameter is the path and file name for local databases.
Database$	The database that contains the old table.
NewTable$	The name of the replacement table.
NewDataBase$	The database that contains the replacement table.

Returns
This function will return a zero on success, a non-zero on error.

Comments
Any fields that don't have a direct match in the old table will be excluded from the report and those fields on the report surface should be 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.

Important: Database and table names ***must*** be entered entirely in upper-case characters.

Example
```
MyData.ReplaceTable "DBO.EMP","HR","DBO.EMP2", "NEW_HR"
```

Save Method

Syntax

`[object].Save Filename$`

Definition

The Save Method replaces any previous connection information in a DataSet control with information from the file that is specified by the *Filename$* parameter.

Parameters

Argument	Description
Filename$	The file to save to.

Returns

The return code for this function should be 0 on success. If it is not zero, then the Error$ property will contain text that describes the error.

Comments

None.

Example

```
MyData.Save "c:\MyData.Dat"
```

SetColumnAlias Method

Syntax
```
[object].SetColumnAlias Table$, Database$, Column$
```

Definition
The SetColumnAlias Method sets or changes the Alias for a column in a report's table.

Parameters

Argument	Description
Table$	Path and file name for local databases.
Database$	Name of the database that contains the table
Column$	Name of the field to set an alias for.
Alias$	The new alias for the field.

Returns
This function will return a zero 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 a set to a null string.

Example
```
MyData.SetColumnAlias"dbo.emp", "hr", "DEPT_ID", "Departments"
```

SetFromActive Method

Syntax

`[object].SetFromActive`

Definition

The SetFromActive Method replaces any previous connection information in a DataSet control with a reference to the data description for the currently active report. When this command is used with the NewReportDialog Control, it is associated with the report and not the query of the report.

Parameters

Not applicable

Returns

Not applicable.

Comments

The SetFromActive Method can be useful to change a currently active report. It can also be used to save DataSet information for a report which could later 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 Method

Syntax
`[`*`object`*`].SetFromLoading`

Definition
The SetFromLoading Method associates the dataset control object with a report that is being loaded (before the SQL is executed for this report).

Parameters
Not applicable.

Returns
This function will return a zero on success, a non-zero on error.

Comments
This function is only valid when used in the Before Opening the Report event.

Example
`MyData.SetFromLoading`

SetTableAlias Method

Syntax

```
[object].SetTableAlias Table$, Database$, Alias$
```

Definition

The SetTableAlias Method sets or changes the Alias for a table in a report.

Parameters

Argument	Description
Table$	Path and file name for local databases.
Database$	Name of the database that contains the table.
Alias$	The new alias for the table.

Returns

This function will return a zero 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.SetTableAlias "dbo.emp", "hr", "Employees"
```

SetTableLink Method

Syntax

```
[object].SetTableLink
  Table1$,DBase1$,Field1$,Table2$,DBase2$,Field2$,Operation, JoinType
```

Definition

The SetTable Link Method defines a link between two tables.

Parameters

Argument	Description
Table1$	The first table to link.
DBase1$	The database that contains Table 1.
Field1$	The field to link on.
Table2$	The second table to link.
DBase2$	The database that contains Table2.
Field2$	The field to link on.
Operation	The relation between the linked fields. 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. 0 Inner Join 1 Left Outer Join 2 Right Outer Join 3 Full Outer Join

Returns

Not applicable.

Comments

Before a table link can be defined both tables must be added to the DataSet control using the AddTableSBL function.

Example

This example links the emp table to the dept table by the department id excluding all unmatched records.

```
`The following needs to be one line in Basic"
SetTableLink "dbo.emp","Indigo","Dept_Id","dbo.dept","Indigo","Dept_Id",0,0
```

SetUserSQL Method

Syntax

[*object*].SetUserSQL$

Definition

The SetUserSQL Method places the DataSet control object into user entered SQL mode with the provided SQL.

Parameters

Argument	Description
SQL$	The complete SQL string to be used for this dataset's query.

Returns

This function will return a zero on success, a non-zero on error.

Comments

You must have a connection to the appropriate server (or local database that matches the SQL you generate in order for the SetUserSQL Method to work properly.

Example

```
MyData.SetUserSQL"SELECT dbo.emp.First_Name,
dbo.emp.Last_Name FROM dbo.emp"
```

TestSelection$ Method

Syntax

```
[object].TestSelection$
```

Definition

The TestSelection$ Method returns a string that tells how many records would be selected or else an error message about the selection criteria.

Parameters

Not applicable.

Returns

Not applicable.

Comments

In order to set and test a selection criteria, you must have a connection and at least one table.

Example

```
MyData.Selection$="Salary>40000"
Msgbox MyData.Test Selection$0
```

DataSet Properties Reference

Properties are variables that belong to an object. You can access these variables in the same way as you access object methods, using the object name followed by the property name. Some properties are read-only while others can be both read and written.

Some ways that a property may be used:

Use it as a function	`msgbox MyData.name$`
Use it as an expression	`total_recs=data1.recordcount+data2.recordcount`
Assign values to it	`MyData Name$="My Name"`

The following list briefly describes the function of each property. To see the syntax, definition, comments and examples of a property, double-click on it.

Property	Description
AllDataBases$	Returns all the databases available under the current connection.
AllOwners$	Returns all owners available under the current connection.
AllTables	Returns a list of all owners for connections that have owners.
DataBases$	Returns the current database for the current connection.
Error$	Contains a string describing the error that occurred in the last dataset control method executed.
Id Property	Used to store an integer value.
Name$	Returns the current name for the current situation.
Owner$	Returns the current owner for the current situation.
Record	Returns the current record in the dataset.
RecordCount	Returns the total number of records in the dataset.
Selection$	Can be used to get or set the selection criteria for a dataset control object.
Tables	Returns a list of tables included in a report.

AllDataBases$ Property

Syntax

`[`*`object`*`].AllDataBases$`

Definition

The AllDataBases$ Property returns all the databases available under the current connection, separated by commas.

Comments

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

Example

```
ListofDataBases$=AllDataBases$
```

AllOwners$ Property

Syntax

```
[*object*].AllOwners$
```

Definition

The AllOwners$ Property returns all owners available under the current connection, separated by commas.

Comments

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

Example

```
OwnerList$ = MyData.AllOwners$
```

AllTables$ Property

Syntax

```
[object].Selection$ [=stringexpression]
```

Definition

The AllTable$ Property returns a list of all tables, separated by commas, for a connection.

Comments

The Table in the String is separated by commas and access to individual tables can be achieved by using the GetField command.

Example

```
Msgbox " All Tables: " + MyData.AllTables$
```

Databases$ Property

Syntax

`[object].Database$`

Definition

Database$ Property returns the current database for the current connection.

Comments

Before a current Database can be retrieved from a dataset object, a connection that has Databases must be made.

Example

```
CurrentDatabase$=MyData.DataBase$
```

Error$ Property

Syntax
`[object].Selection$ [=stringexpression]`

Definition
The Error$ Property contains a string that describes the error that occurred in the last dataset control method executed. This command may also be associated with the Report and NewReportDialog controls.

Comments
Use *MyData.PS* or any other DataSet name but be consistent.

Example
```
X=DS.ADDTABLESBL("INVALID.TABLE","BOGUS")
If x<>0 then msgbox DS.ERROR$
End If
```

Id Property

Syntax

```
[object].Id [=integerexpression]
```

Definition

A property that can be used to store an integer value. This command is valid with the DataSet, Report and NewReportDialog controls.

Comments

The Id Property function might be used to keep track of some information related to the DataSet, Report or NewReportDialog controls. As with the Name$ property this value has no meaning to ReportBasic and it is completely up to the Basic programmer how it is to be used. Read/write at run time.

Example

```
MyData.Id=127
```

Name$ Property

Syntax

```
[object].Name$ [=stringexpression]
```

Definition

The Name$ Property returns the current name for the current situation. This command is valid for the DataSet, Report and NewReportDialog controls.

Comments

Read/Write at run time.

Example

```
MyData.Name$="Susan's Data"
```

Owner$

Syntax

```
[object].Owner$
```

Definition

The Owner$ Property returns the current owner for the current situation.

Comments

Before the current owner can be retrieved from a dataset object, a connection that has owners must be made.

Example

```
CurrentOwner$=MyData.Owner$
```

Record Property

Syntax

`[object].Record`

Definition

The Record Property returns the current record in the dataset.

Comments

Before the number of records can be removed from a dataset object a connection must be made, links must be set and a commit or recalc must be successfully performed.

Example

```
If MyData.Record = 1 then MsgBox "We are at the beginning"
```

RecordCount

Syntax
```
[object].RecordCount()
```

Definition
The RecordCount Property returns the total number of records in the dataset.

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()
```

Selection$ Property

Syntax

```
[object].Selection$ [=stringexpression]
```

Definition

The Selection$ Property can be used to get or set the selection criteria for a DataSet Control Object.

Comments

A table must be added to the Data Set before the selection criteria may be written or read. A change in Selection$ will not change the data until a commit method or a recalc command is executed. Read/write at run time.

You can get the individual tables from the list by using the GetField$ Function.

Example

```
MyData.Selection$="Salary>40000"
```

Tables Property

Syntax

```
[object].Table$
```

Definition

The Table$ Property returns a list of tables included in a report separated by commas.

Comments

By using the GetField$ Function you can get the individual tables from the list.

Example

```
SecondTable$=GetField$(MyData.DataBase$, 2,",")
```

NewReportDialog Control Object

The NewReportDialog Control object is a ReportBasic object that allows you to bring up the ReportSmith Create A New Report dialog box and pass the information it gathers to the ReportBasic macro language. The Create Report Command will take this information and use it in creating new reports.

You can create these objects in the same way that you create a DataSet control object.

```
‘ Local scope
Dim MyDialog as NewReportDialog
‘ Global Scope
Global MyDialog as NewReportDialog
MyDialog.RunDialog
SelectedStyle$ = MyDialog.Style $
TheType = MyDialog.ReportType
```

The following macro example shows you what information the object returns to BASIC in response to the dialog that appears.

```
Sub Test_New_Object()

  ‘ Create a Dialog Control Object
  dim MyDialog as NewReportdialog
  MyDialog.rundialog
  ‘View information in the message boxes
  msgbox MyDialog.error,0,"Error"
  msgbox str$( MyDialog.ReportType ),0,"ReportType"
  msgbox MyDialog.style,0,"Style"
  msgbox MyDialog.crosstabstyle,0,"Crosstab Style"
  msgbox str$( MyDialog.DraftMode ),0,"DraftMode"
  msgbox str$( MyDialog.ReturnCode ),0,"ReportType"

End Sub
```

NewReportDialog Methods

Methods are functions or statements that perform actions on the control.

RunDialog.

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
```

NewReportDialog Properties

You can access the properties of a control by using the name of the control, followed by a period and the name of the property. Properties are variables that belong to an object. You can access these variables in the same way as you access object methods by using the object name followed by the property name. You use this just like any other variable. Some properties are read-only while others can be both read and written.

Properties	Description
Name$	A string that can be used to hold a name for this object.
Id Property	An integer that can be used to hold an ID for this object.
Error$	A string that indicates the last error generated by this object. This property is read only.
ReportType	Indicates the type of report selected. This property is read only.
Style$	The style name selected. May be a NULL string. This property is read only.
CrosstabStyle$	A string that holds the last crosstab style name selected. This property is read only.
DraftMode	A flag. If it is non-zero it indicates that the used checked the Draft Mode check-box.
ReturnCode	Indicates if the user selected OK or Cancel to exit the dialog. This property is read only.

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
```

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
```

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
```

DraftMode Property

Syntax
`[object].DraftMode`

Definition
A flag. If it is non-zero it indicates that the used 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
```

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
```

Report Control Object

The Report Control object is used to control reports in the same manner that the DataSet Control object is used to control a query. At this time the Report Control has only a limited number of methods and properties.

Create a report control object in the same manner as a dataset control.

```
Sub ReportControlEx()
  ' Create a Report Control object with local scope
  dim MyReport as Report

  ' Associates the ReportObject we created with the active report
  MyReport.SetFromActive

  ' Get information about the report an put it into strings
   ActivePage$ = Str$( MyReport.Page )
   NumberOfPages$ = str$( MyReport.TotalPages )

   ' Use the object to return information about the Report
 Msgbox "You're on pg" + ActivePage$+ "In rpt with "+ NOP$ + " pages"

End Sub
```

Report Control Properties Reference

Properties	Description
Name$	A string that identifies this object. Null by default.
Id	A number that identifies the object. Null by default.
Page	Gets or changes the current page of the report.
TotalPages	Returns the total number of pages in the report (read only).
Error$	Last error returned by a method of this object.

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
```

Report Control Methods Reference

Methods are functions or statements that perform actions on the control.

Methods	Description
Recalc	Identical to the general Recalc Command but only affects the associated Report.
SetFromActive	Associates a Report Control Object with the currently active report.

Abs function

Syntax **Abs(** *numeric-expression* **)**

Returns The **Abs** function returns the absolute value of the specified *numeric expression*.

The return type matches the type of the *numeric expression*. This includes **variant** expressions which will return a result of the same vartype as input except vartype 8 (string) will be returned as vartype 5 (double) and vartype 0 (empty) will be returned as vartype 3 (long).

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

Asc function

Syntax **Asc(** *string-expression$* **)**

Returns The **Asc** function returns an integer corresponding to the ANSI code of the first character in the specified string.   See **Chr$**.

Assert statement

Syntax	**Assert** *condition*
Comment	The **Assert** statement 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.

Atn function

Syntax	**Atn(** *numeric-expression* **)**
Returns	The **Atn** function returns the angle (in radians) corresponding to the arc tangent of the specified numeric expression.
Comment	The return value is single-precision for an integer, currency, or single-precision numeric expression, double precision for a long, variant or double-precision numeric expression.

Beep statement

Syntax	**Beep**
Comment	The **Beep** statement produces a single short beeping tone through the computer speaker.

Begin Dialog ... End Dialog statement

Syntax

Begin Dialog *dialogName* [*x, y*,] *dx, dy*
' dialog box definition statements
End Dialog

Comments

The **Begin Dialog** statement is used to start the dialog-box declaration for a user-defined dialog box.

The *x* and *y* arguments 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* argument is measured in units that are 1/4 the average width of the system font. The *y* argument is measured in units 1/8 the height of the system font. (E.g., to position a dialog box 20 characters in, and 15 characters down from the upper left hand corner, enter 80, 120 as the *x, y* coordinates.) If these arguments are omitted, the dialog box is centered in the client area of the parent window.

The *dx* and *dy* arguments specify the width and height of the dialog box (relative to the *x* and *y* coordinates). The *dx* argument is measured in 1/4 system-font character-width units. The *dy* argument is measured in 1/8 system-font character-width units. (I.e., to create a dialog box 80 characters wide, and 15 characters in height, enter 320, 120 as the *dx, dy* coordinates.)

The **Begin Dialog** statement assumes that if only two arguments are given, they are the *dx* (width) and *dy* (height) arguments.

Unless the **Begin Dialog** statement is followed by at least one other dialog-box definition statement and the **End Dialog** statement, an error will result. The other definition statement must include an **OkButton**, **CancelButton** or **Button** statement. If this statement is left out, there will be no way to close the dialog box, and the procedure will be unable to continue executing.

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.

Button statement

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

Comments The **Button** statement is used to define 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.

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

The *x* and *y* arguments set the position of the button relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the button. A *dy* value of 14 typically accommodates text in the system font.

The *text$* field contains a message that will be contained in the push button. If the width of this string is greater than *dx*, trailing characters will be truncated.

ButtonGroup statement

Syntax **ButtonGroup** *.field*

Comments The **ButtonGroup** statement begins definition of the buttons when custom buttons are to be used. **ButtonGroup** establishes the dialog-record field that will contain the user's selection. If **ButtonGroup** is used, it must appear before any **Button** statement which creates a pushbutton. Only one **ButtonGroup** statement is allowed within a dialog box definition.

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

Call statement

Syntax A	**Call** *subprogram-name* [**(** *argumentlist* **)**]
Syntax B	*subprogram-name argumentlist*
Syntax C	**Call** *app-dialog* **(** *recordName* **)**
Syntax D	*App-dialog* { *recordName* | *dotList* }

Comments

The Call statement is used to transfer control to a subprogram procedure or application-defined dialog box. The Call statement can be 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 arguments to the subprogram must match the parameters as specified in the definition of the subprogram. The arguments may be either variables or expressions.

Arguments 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 will tell BASIC to pass a copy of the variable. Note that this will be less efficient, and should not be done unless necessary.

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

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 **Dim**'ed 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, e.g.:

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

When calling an external DLL procedure, arguments 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.

CancelButton statement

Syntax **CancelButton** *x, y, dx, dy*

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

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

The *x* and *y* arguments set the position of the cancel button relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the button. A *dy* value of 14 can usually accommodate text in the system font.

If the CancelButton is pushed at runtime, the dialog box will be removed from the screen and an Error 102 will be triggered.

Caption statement

Syntax	**Caption** *text$*
Comments	The **Caption** statement defines the text to be used as the title of a dialog-box. The **Caption** statement can only be used between a **Begin Dialog** and an **End Dialog** statement. If no **Caption** statement is specified for the dialog box, a default caption will be used.

CCur function

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 function

Syntax **CDbl(** *expression* **)**

Returns The **CDbl** function converts an *expression* to a double-precision floating point.

Comments **CDbl** accepts any type of *expression.* 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 expression to a different data type, see **CCur**, **CInt**, **CLng**, **CSng**, **CStr**, **CVDate** and **CVar**.

ChDir statement

Syntax **ChDir** *pathname$*

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**.)

Pathname$* is a string expression identifying the new default directory. The syntax for *pathname$ is:

[*drive*:] [\] *directory* [*directory*]

The drive argument is optional. If omitted, **ChDir** changes the default directory on the current drive.

ChDrive statement

Syntax **ChDrive** *drivename$*

Comments The **ChDrive** statement changes the default drive.

Drivename$* is a string expression designating the new default drive. This drive must exist, and must be within the range specified in the CONFIG.SYS file. If a null argument (" ") is supplied, the default drive remains the same. If the *drivename$ argument is a string, **ChDrive** uses the first letter only. If the argument is omitted, an error message is produced. (To change the current directory on a drive, use **ChDir**.)

CheckBox statement

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

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

The *x* and *y* arguments 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* argument is measured in 1/4 system-font character-width units. The *y* argument is measured in 1/8 system-font character-height units. (See **Begin Dialog**.)

The *dx* argument is the combined width of the check box and the text$ field. Because proportional spacing is used, the width will vary with the characters used. To approximate the width, multiply the number of characters in the *text$* field (including blanks and punctuation) by 4 and add 12 for the checkbox.

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

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

The *.field* argument is the name of the dialog-record field that will hold the current check box setting. If its value is 0, the box is unchecked; if its value is -1 the box will be gray; if its value is 1, the box will be checked. SBL will treat any other value of *.field* the same as a 1.

Chr$ function

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 function

Syntax	**CInt(** *expression* **)**
Returns	The **CInt** function converts the value of *expression* to an integer by rounding.
Comments	**CInt** accepts any type of expression. After rounding, the resulting number must be within the range of -32767 to 32767, or an error will occur. Strings that cannot be converted to an integer 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 **CCur**, **CDbl**, **CLng**, **CSng**, **CStr**, **CVDate** and **CVar**.

CLng function

Syntax **CLng(** *expression* **)**

Returns The **CLng** function converts the value of *expression* to a long by rounding.

Comments **CLng** accepts any type of expression. After rounding, the resulting number must be within the range of -2,147,483,648 to 2,147,483,647, or an error will occur.

Strings that cannot be converted to a long 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 **CCur**, **CDbl**, **CInt**, **CSng**, **CStr**, **CVDate** and **CVar**.

Close statement

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

Comments The **Close** statement closes a file, concluding input/output to that file.

Filenumber% is an integer expression identifying the file to close. It is the number used in the **Open** statement for the file. If this argument is omitted, all open files are closed. 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 will flush its buffers to disk.

ComboBox statement

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

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

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

The *x* and *y* arguments give the coordinates that position the upper left corner of the list box, relative to the upper left corner of the dialog box. The *x* argument is measured in 1/4 system-font character-width units. The *y* argument is measured in 1/8 system-font character-width units. (See **Begin Dialog**.)

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

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

The *.field* argument is the name of the dialog-record field that will hold the text string entered in the text box or chosen from the list box. The string in the text box will be recorded in the field designated by the *.field* argument when the OK button (or any pushbutton other than CANCEL) is pushed.

Command$ function

Syntax **Command**[**$**]

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

Const statement

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

Comments You use the **Const** statement to declare symbolic constants for use in a BASIC program. BASIC is a strongly typed language. The available data types for constants are numbers and strings.

The type of the constant may be specified by using a type character as a suffix to the *constantName*. If no type character is specified, the type of the *constantName* will be 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 will fail if a mismatch is found. This is useful as a mechanism for detecting version mismatches between modules.

Cos function

Syntax	**Cos(** *angle* **)**
Returns	The **Cos** function returns the cosine of an angle. The return value will be between -1 and 1. The return value is single-precision if the angle is an integer, currency, or single-precision value, double precision for a long, variant or double-precision value.
Comments	The *angle* is specified in radians, and can be either positive or negative.

CreateObject function

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

CSng function

Syntax **CSng(** *expression* **)**

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

Comments **CSng** accepts any type of *expression*. The *expression* must have a value within the range allowed for the **Single** data type, or an error will occur.

Strings that cannot be converted to an integer 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 **CCur**, **CDbl**, **CInt**, **CLng**, **CStr**, **CVDate** and **CVar**.

CStr function

Syntax **CStr(** *expression* **)**

Returns The **CStr** function converts the value of *expression* to a string.

Comments **CStr** accepts any type of *expression*.

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

$CStrings Metacommand

Syntax **'$CSTRINGS**

Comments 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.

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

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
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.

CurDir$ function

Syntax **CurDir**[**$**] [**(** *drivename$* **)**]

Returns The **CurDir$** function 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 will return a **variant** of vartype 8 (string).

Comments *Drivename$* is a string expression identifying the drive to return the default directory of. This drive must exist, and must be within the range specified in the CONFIG.SYS file. If a null argument (" ") is supplied, or if no *drivename* is indicated, the path for the default drive is returned.

To change the current drive, use **ChDrive**; to change the current directory, use **ChDir**.

CVar function

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**, and **CStr**. To convert a value to a different variant type, see **CVar**.

Date$ function

Syntax	**Date**[**$**]
Returns	The **Date$** function returns a string representing the current date. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will return a **variant** of vartype 8 (string).
Comments	The **Date$** function returns a ten character string.

Date$ statement

Syntax	**Date**[**$**] **=** *expression*

Comments

The **Date$** statement is used to set the current system date.

When **Date$** (with the dollar sign '**$**') is used, the *expression* must evaluate to a string of one of the following forms:

mm-dd-yy

mm-dd-yyyy

mm/dd/yy

mm/dd/yyyy

where *mm* denotes a month (01-12), *dd* denotes a day (01-31), and *yy* or *yyyy* denotes a year (1980-2099).

If the dollar sign '**$**' is omitted, *expression* can be a string containing a valid date, a **variant** of vartype 7 (date), or a **variant** of vartype 8 (string).

If *expression* is not already a **variant** of vartype 7 (date), **Date** attempts to convert it to a valid date from January, 1, 1980 through December, 31, 2099. **Date** uses Short Date format in the International section of Windows Control Panel to recognize day, month, and year if a string contains three numbers delimited by valid date separators. In addition, **Date** recognizes month names in either full or abbreviated form.

DateSerial function

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, 100 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 function

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 function

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 will attempt to convert the input value to a date value.

Declare statement

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

Syntax B **Declare function** *name* **[** *libSpecification* **]** **[(** *parameter* **[As** *type* **])]** **[As** *functype* **]**

Comments 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.

If the *libSpecification* is of the format:

BasicLib *libName* **[Alias "***aliasname***"]**

the procedure is to be found in another BASIC module named *libName*. The **Alias** keyword specifies that the procedure in *libName* is called *aliasname*. The other module will be loaded on demand whenever the procedure is called. SBL will not automatically unload modules which are loaded in this fashion. SBL will detect errors of mis-declaration.

If the *libSpecification* is of the format:

Lib *libName* **[Alias ["]***ordinal***["]]**

or

Lib *libName* **[Alias "***aliasname***"]**

the procedure is to be found in a Dynamic Link Library (DLL) named *libName*. The *ordinal* argument specifies the ordinal number of the procedure within the external DLL. Alternatively, *aliasname* specifies the name of the procedure within the external DLL. If neither *ordinal* nor *aliasname* is specified, the DLL function will be accessed by name. It is recommended that the *ordinal* be used whenever possible, since accessing functions by name may cause the module to load more slowly.

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.

A **Sub** procedure does not return a value. **function** returns a value, and can be used in an expression. Either a **function** name can end with a type character or the **As** *functype* clause can be supplied. This provides the type of the return value for the function. If no type is provided, the function will default to type variant. The *name* argument names the **Sub** or **function** being declared.

The *parameters* are specified as a comma-separated list of parameter names. The data type of a parameter may 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*. Array dimensions are not specified in the **Declare** statement.

External DLL procedures are called with the PASCAL calling convention (the actual arguments are pushed on the stack from left to right). By default, the actual arguments 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 argument used in calls to the procedure is disabled (although other arguments not declared as type **Any**

are fully type-safe). The actual argument 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** may 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.

SBL supports two different behaviors when an empty string ("") is passed **ByVal** to an external procedure. The implementor of SBL can specify which behavior by using the SBL API function **SblSetInstanceFlags**. In any specific implementation which uses SBL, 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. This paragraph, and one of the two following paragraphs should be removed for the final documentation.

When an empty string ("") is passed **ByVal** to an external procedure, the external procedure will receive a NULL pointer. If you wish to send a valid pointer to an empty string, use **Chr$(0)**.

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

Def*type* statement

Syntax

DefCur *varTypeLetters*
DefInt *varTypeLetters*
DefLng *varTypeLetters*
DefSng *varTypeLetters*
DefDbl *varTypeLetters*
DefStr *varTypeLetters*
DefVar *varTypeLetters*

Comments

The **Def*type*** statement specifies the default data type of a variable which begins with one of the letters specified in *varTypeLetters*.

The *varTypeLetters* are specified as a comma-separated list of letters; a range of letters may 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 **Def*type*** statement only affects the module in which it is specified.   It must precede any variable definition within the module.

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

Dialog statement

Syntax **Dialog** *recordName*

Comments The **Dialog** statement displays a dialog box. 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 pushing the Cancel button, a run-time error will be triggered which can be trapped using **On Error**.

Dim statement

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

Comments You use the **Dim** statement to declare variables for use in a BASIC program. BASIC is a strongly typed language. The available data types are: numbers, strings, variants, records, arrays, dialog boxes and Application Data Types (ADTs).

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

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 may 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**. The dialog record variable can then be used in a **Dialog** statement.

Dialog records have the same behavior as regular 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 may 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 will be allocated when the variable is used. Note: The class **object** does not support the **new** operator.

Dim *variableName* **As New** *className*
variableName.methodName

Object variables may be declared when the class name is a reserved Basic keyword by enclosing the class name in square brackets.

Dim *variableName* **As [***For***]**

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 may be specified in an array definition is 60. The maximum total size for an array is only limited 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 may 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 used to declare a variable, you may chose 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.

Dir$ function

Syntax **Dir**[**$**] [**(** *filespec$* [, *attrib%*]**)**]

Returns The **Dir$** function returns a filename that matches the specified pattern.

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 *Filespec$* is a string expression identifying a path or filename. This argument may include a drive specification. It may also include the "wildcard" characters '?' and '*'. **Dir$** will return the first filename that matches the *filespec$* argument. To retrieve additional filenames that match the *filespec$*, call the **Dir$** function again, omitting the *filespec$* and *attrib%* arguments. If no file is found, an empty string ("") is returned.

Attrib% is an integer expression specifying the filenames that need to be added to the list. The default value for *attrib%* is 0. In this case, **Dir$** returns only files without directory, hidden, system, or volume label attributes set.

Here are the possible values for *attrib%*:

Value	**Meaning**
0	return normal files
2	add hidden files
4	add system files
8	return volume label
16	add directories

The values in the table can be combined by using addition. For example, to list hidden and system files in addition to normal files set *attrib%* to 6 (6=2+4)

If *attrib%* is set to 8, the **Dir$** function returns the volume label of the drive specified in the *filespec$*, or of the current drive if drive is not explicitly specified. If volume label attribute is set, all other attributes are ignored.

Do...While statement

Syntax A **Do** [{ **While** | **Until** } *condition*]

[*statementblock*]

[**Exit Do**]

    [*statementblock*]

Loop

Syntax B **Do**

[*statementblock*]

[**Exit Do**]

[*statementblock*]

Loop [{ **While** | **Until** } *condition*]

Comments *Condition* is any expression that BASIC can determine to be TRUE (nonzero) or FALSE (0). BASIC will repeat the program lines contained in the *statementblock(s)* as long as a **While** condition is true or until an **Until** condition is true.

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

DoEvents statement

Syntax	**DoEvents**
Comments	**DoEvents** statement is used to yield execution so that Windows can process events. It does not return until Windows has finished processing all events in the queue and all keys sent by SendKeys statement. **DoEvents** should not be used if other tasks can interact with the running program in unforeseen ways. Since BASIC yields control to the operating system at regular intervals, **DoEvents** should only be used to force BASIC to allow other applications to run at a known point in the program.

Environ$ function

Syntax A **Environ**[**$**]**(** *environment-string$* **)**

Syntax B **Environ**[**$**]**(** *numeric expression%* **)**

Returns The **Environ$** function returns a string from the operating system's environment table.

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 argument of the **Environ$** function may be either a string (*environment-string$*) or an integer (*numeric expression%*).

Environment-string$ is the name of a keyword in the operating system environment. If this argument is given, it must be entered in uppercase, or it will return a null string. The value associated with the keyword will be returned.

Numeric expression% represents one of the strings from the operating system environment parameters. This may be any numeric expression, but it will be rounded to a whole number by **Environ$**. If this argument is used, **Environ$** will return the *n*th string from the environment table. This string will be in the form "keyword = value."

A null string will be returned if the specified argument cannot be found.

Eof function

Syntax **Eof(** *filenumber%* **)**

Returns **Eof** 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. The *filenumber%* is the number used in the **Open** statement of the file.

Erase statement

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 the 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 function

Syntax **Erl**

Returns **Erl** returns the line number where an error was trapped.

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

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

Err function

Syntax **Err**

Returns **Err** returns the run-time error code for the last error that was trapped.

Comments Using the **Resume** or **On Error** statements will reset the **Err** value to 0. If you wish 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.

The Trappable Errors are listed in an appendix.

Err statement

Syntax **Err** = *n%*

Comments The argument *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). The **Err** statement is used to send error information between procedures.

Error statement

Syntax **Error** *errorcode%*

Comments **Error** *errcode%* simulates the occurrence of a SBL or user-defined error. The *errorcode %* argument, which represents the error code, must be an integer between 1 and 32,767. If an *errcode%* is one which SBL already uses, the **Error** statement will simulate an occurrence of that error.

User-defined error codes should employ values greater than those used for standard SBL error codes. To help ensure that non-SBL 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, SBL produces an error message and halts program execution. If an **Error** statement specified an error code not used by SBL, the message "User-defined error" is displayed.

Error$ function

Syntax **Error**[**$**] [**(** *errorcode%* **)**]

Returns **Error$** returns the error message that corresponds to the specified error 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 The *errorcode%* argument, which represents the error code, must be an integer between 1 and 32,767. If this argument 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, "" is returned.

The Trappable Errors are listed in an appendix.

Exit statement

Syntax **Exit** {**Do** | **For**| **function** | **Sub**}

Comments **Exit Do** and **Exit For** terminate loop statements. **Exit Do** can only be used within a **Do...Loop** statement. **Exit For** can only be used 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 only be used to exit from a **Sub** procedure.

Exp function

Syntax	**Exp(** *numeric-expression* **)**
Returns	The **Exp** function returns the value *e* raised to the *numeric-expression* power.
Comment	The return value is single-precision for an integer, currency or single-precision numeric expression, double precision for a long, variant or double-precision numeric expression.

FileAttr function

Syntax **FileAttr(** *filenumber%, attribute%* **)**

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.

Comments The argument *filenumber%* is the number used in the **Open** statement to open the file. The argument *attribute%* is either a 1 or 2. The following table lists the return values and corresponding file modes if *attribute%* is 1:

Value	**Mode**
1	**Input**
2	**Output**
8	**Append**

If *attribute%* is 2, **FileAttr** returns the operating system handle for the file.

FileCopy statement

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 function

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 function

Syntax **Fix (** *numeric-expression* **)**

Returns **Fix** returns the integer part of a *numeric-expression*.

The return type matches the type of the numeric expression. This includes **variant** expressions which will return a result of the same vartype as input except vartype 8 (string) will be returned as vartype 5 (double) and vartype 0 (empty) will be returned as vartype 3 (long).

Comments The argument given is any *numeric-expression.* For both positive and negative *numeric-expressions,* **Fix** removes the fractional part of the expression and returns the integer part only. For example, **Fix** (6.2) returns 6; **Fix** (-6.2) returns -6.

See **CInt** and **Int**.

For...Next statement

Syntax **For** *counter* = *start* **TO** *end* [**STEP** *increment***]**

[*statementblock*]

[**Exit For**]

[*statementblock*]

Next [*counter*]

Comments The **For...Next** statement repeats the statement block a fixed number of times, determined by the values of *start*, *end*, and *step*.

Argument	**Description**
counter	Variable used as the loop counter.
start	Beginning value of the counter.
end	Ending value of the counter.
increment	The amount by which the counter is changed each time the loop is run through. (The default is one.)
statementblock	BASIC functions, statements, or methods to be executed.

In order for a **For...Next** loop to execute, 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. BASIC compares the sign of (*end-start*) with the sign of **Step**. If the signs are the same, and *end* does not equal *start*, the **For...Next** loop is entered. 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 executes 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* will make programs more difficult to maintain 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 may be used as an alternative exit from **For...Next** loops.

If the variable is left out of a **Next** statement, the **Next** statement will match the most recent **For** statement. If a **Next** statement occurs prior to its corresponding **For** statement, BASIC will return 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. For 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 will return a **variant** of vartype 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 will cause 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, while period, ".", is always used in the *fmt* to denote the decimal separator, the output string will contain the appropriate character based upon the current international settings for your machine. Likewise, while comma is always used in the *fmt* specification, the output will contain 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) will scale (divide) the number by 1000. The commas will not appear in the output string. The percent sign will cause the number to be multiplied by 100. The percent sign will appear 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 will be 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, "\".

Number	Fmt	Result
1234567.89	$#,0.00	$1,234,567.89
1234567.89	"TOTAL:" $#,#.00	TOTAL: $1,234,567.89
1234	\=\>#,#\<\=	=>1,234<=

You may wish to use the SBL **$CSTRINGS** metacommand or the **Chr** 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 will result in an upper case E in the output. Those containing a lower case e will result in a lower case e in the output. A minus sign following the E will cause negative exponents in the output to be preceded by a minus sign. A plus sign in the *fmt* will cause 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 will be output if the preceding token was an hour; otherwise, the month will be output.
mm	The month or the year or the minute of the hour as a two digit number. The minute will be output if the preceding token was an hour; otherwise, the month will be 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. Less than forces output characters to be in lowercase. Greater than 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 function

Syntax	**FreeFile**
Returns	The **FreeFile** function returns the lowest unused file number.
Comments	The **FreeFile** function is 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. The value returned can be used in a subsequent **Open** statement.

function ... End function statement

Syntax **[Static] [Private] function** *name* **[(** *parameter* [**As** *type*] ... **)]** **[As** *functype* **]**

name **=** *expression*

End function

Comments The **function..End function** structure defines a function procedure. The purpose of a function is to produce and return a single value of a specified type. Recursion is supported.

The parameters are specified as a comma-separated list of parameter names. The data type of a parameter may 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.

The **Static** keyword specifies that all the variables declared within the function will retain their values as long as the program is running, regardless of the way the variables are declared.

The **Private** keyword specifies that the function will not be accessible to functions and subprograms from other modules. Only procedures defined in the same module will have access to a **Private** function.

In the **function** statement, the *name* of the function may end with a type character or the **As** *functype* clause may be used to specify the type of value that the function will return. If neither a type character or a *functype* is provided the function will default to returning a **variant**. When calling the function, you need not specify the type character.

You specify the return value by assigning to the function name as if it were a variable or parameter. If no such assignment occurs, the value returned will be 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.

BASIC procedures use the call by reference convention. This means that if a procedure assigns a value to a parameter, it will modify 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.

Get statement

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 variable is smaller than the record length, the additional data is discarded. If the size of 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.

GetAttr function

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.

GetCurValues statement

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.

GetField$ function

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

Returns The **GetField$** function returns a substring from a source string.

Comments The source *string* is considered to be divided into fields by separator characters. Multiple separator characters may 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.

Global statement

Syntax **Global** *variableName* [**As** *type*] [,*variableName* [**As** *type*]] **...**

Comments You use the **Global** statement to declare global variables for use in a BASIC program. BASIC is a strongly typed language. The available data types are: numbers, strings, variants, records, arrays, dialog boxes and Application Data Types (ADTs).

Global data is shared across all loaded modules. If an attempt is made to load a module which has a global variable declared which has a different data type than an existing global variable of the same name, the module load will fail.

If the **As** clause is not used, the type of the global variable may be specified by using a type character as a suffix to the *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 may chose 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.

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:

Global *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:

Global *variableName$*
or **Global** *variableName* **As String**

Records

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

Global *variableName* **As** *typeName*

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

You can not use the **Global** statement to declare a dialog record.

Arrays

The available data types for arrays are: numbers, strings, variants 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:

Global *variable*([*subscriptRange*, ...]) [**As** *typeName*]

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 may be specified in an array definition is 60.

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.

GoTo statement

Syntax **GoTo** *label*

Comments **GoTo** sends control to a *label*.

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.

GroupBox statement

Syntax **GroupBox** *x, y, dx, dy, text$*

Comments The **GroupBox** statement is used to set 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.

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

The *x* and *y* arguments set the position of the group box relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the box.

The *text$* field contains a title that will be embedded in the top border of the group box. Trailing characters will be truncated in text$ is wider than dx. If the text$ argument is an empty string (""), the top border of the group box will be a solid line.

Hex$ function

Syntax **Hex**[**$**]**(** *numeric-expression* **)**

Returns The **Hex$** function returns a hexadecimal representation of a *numeric-expression*, as a string. If the numeric expression is an integer, the string will contain up to four hexadecimal digits; otherwise, the expression will be converted to a long integer, and the string may contain up to 8 hexadecimal digits.

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).

Hour function

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.

If ... Then ... Else

Syntax A **If** *condition* **Then** *then_statement* **[Else** *else_statement* **]**

Syntax B **If** *condition* **Then**

statement_block

[ElseIf *expression* **Then**

*statement_block***]...**

[Else

statement_block **]**

End If

Comments The **If ... Then ... Else** structure allows you to organize alternative actions into separate blocks of code. 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, the *then_statement* and *else_statement* 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.

In the block version of the **If** statement, the *statement_blocks* can be made up of zero or more statements, separated by colons (:) or on different lines.

$Include Metacommand

Syntax **'$Include: "***filename***"**

Comments Tells the compiler to include statements from another file.

Comments which include metacommands will only be recognized at the beginning of a line.   For compatibility with other versions of BASIC, you may use single quotes (') to enclose the *filename*.

A file extension of .SBH is suggested for SBL include files.   This is only a recommendation, and any other valid file extension may be used.

If no directory or drive is specified, the compiler will search for *filename* on the source file search path.

Input$ function

Syntax **Input**[$]**(** *numchars%,* [#]*filenumber%***)**

Returns The **Input$** function returns a string containing the characters read.

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 *numchars%* argument contains the number of characters (bytes) to read from the file. The *filenumber%* argument is an integer expression identifying the open file to read from.

The file pointer is advanced the number of characters read. Unlike the **Input #** statement, **Input$** returns all characters it reads, including carriage returns, line feeds, and leading spaces.

Input # statement

Syntax A **Input** [#] *filenumber%, variable* [*, variable*]...

Syntax B **Input** [*prompt$*,] *variable* [, *variable*]...

Comments The **Input #** statement reads data from a sequential file and assigns the data to variables. The *filenumber%* argument is an integer expression identifying the open file to read from. This is the number used in the **Open** statement to open the file. *Prompt$* is an optional string that can be used to prompt for keyboard input. The *variable* arguments list the variables that are assigned the values read from the file. The list of *variables* is separated by commas.

If *filenumber* is not specified, the user is prompted for keyboard input with a "?", unless *prompt$* is specified.

InputBox$ function

Syntax **InputBox**[**$**]**(***prompt$* [,[*title$*] [,[*default$*] [,*xpos%, ypos%*]]]**)**

Returns The **InputBox$** function returns a string entered by the user.

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 **InputBox$** 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 that will be 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, **InputBox$** returns the text contained in the input box. If the user selects Cancel, the **InputBox$** function returns a null string.

InStr function

Syntax A	**InStr(** [*position%*,] *string$*, *substring$* **)**
Syntax B	**InStr(** *position%*, *string$*, *substring$,*[, *comparetype%*]**)**
Returns	The **InStr** function returns an integer representing the position of the first occurrence of a *substring* within another *string*.

Comments

The *position%* argument indicates the index of the character within *string$* where the search should start. If not specified, the search starts at the beginning of the string (equivalent to a *position%* of 1). *String$* is the string being searched. *Substring$* is the string being looked for. These arguments may be of any type. They will be converted to strings.

If the *position%* argument is greater than the length of the substring; if the *string$* argument is a null string; or if the substring$ cannot be located, **InStr** will return a zero. If the *substring$* argument is a null string, then the *position%* argument will be returned.

The index of the first character in a string is 1.

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.

Instr returns a null variant if either *string$* or *substring$* is a null variant.

Int function

Syntax **Int(** *numeric-expression* **)**

Returns The **Int** function returns the integer part of a *numeric-expression*.

Comments The argument given is any *numeric -expression*. 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. For example, **Int** (6.2) returns 6; **Int**(-6.2) returns -7.

The return type matches the type of the numeric expression. This includes **variant** expressions which will return a result of the same vartype as input except vartype 8 (string) will be returned as vartype 5 (double) and vartype 0 (empty) will be returned as vartype 3 (long).

See **CInt** and **Fix**.

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 function

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 function

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.

IsNull function

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 function

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 statement

Syntax **Kill** *filespec$*

Comments **Kill** deletes files from disk. The argument *filespec$* is a string expression that specifies a valid DOS file specification. This specification can contain paths and wildcards. **Kill** deletes files only, not directories. Use the **RmDir** function to delete directories.

LBound function

Syntax **LBound(** *arrayVariable* [, *dimension*] **)**

Returns The LBound function 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. 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.

LCase$ function

Syntax	**LCase**[**$**]**(** *expression* **)**
Returns	The **LCase$** function returns a copy of the source *string*, with all upper case letters converted to lower case. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will typically return a **variant** of vartype 8 (string). If the value of *expression* is null a variant of vartype 1 (null) is returned
Comments	The translation is based on the country specified in the Windows Control Panel. **LCase$** accepts expressions of type string. **LCase** accepts any type of *expression* including numeric values and will convert the input value to a string.

Left$ function

Syntax	**Left**[**$**]**(** *expression*, *length%***)**
Returns	The **Left$** function returns a string of a specified *length* copied from the beginning of the source *string*. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will typically return a **variant** of vartype 8 (string). If the value of *expression* is null a variant of vartype 1 (null) is returned
Comments	If the length of *string$* is less than *length%*, **Left$** returns the whole string. **Left$** accepts expressions of type string. **Left** accepts any type of *expression* including numeric values and will convert the input value to a string.

Len function

Syntax A **Len(** *string expression$* **)**

Syntax B **Len(** *variable* **)**

Returns The **Len** function returns the length of the argument. The argument can be of any type; if the argument is a string, the number of characters in the string is returned. If the argument is a variant variable, **Len** returns the number of bytes required to represent its value as a string; otherwise, the length of the built-in datatype or user-defined type is returned.

If the syntax B is used and *variable* is a variant containing a null, **Len** will return a null variant.

Let (Assignment statement)

Syntax **[Let]** *variable* **=** *expression*

Comments An assignment statement stores a value in a BASIC *variable*. The keyword **Let** is optional.

The **Let** statement can be used to assign to a numeric, string, variant or record variable. You can also use the **Let** 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.

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 than 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 # statement

Syntax A **Line Input** [#] *filenumber%, variable$*

Syntax B **Line Input** [*prompt$*,] *variable$*

Comments The **Line Input** statement reads a line from a sequential file into a string variable. The *filenumber%* argument is an integer expression identifying the open file to read from. If specified, this is the number used in the **Open** statement to open the file. If filenumber% is not provided, the line will be read from the keyboard.

Prompt$ is an optional string that can be used to prompt for keyboard input. If not provided, a prompt of "?" will be used.

The *variable$* argument is a string variable into which the line is read.

ListBox statement

Syntax **ListBox** *x, y, dx, dy, text$, .field*

Comments The ListBox statement is used to create a list of choices.

The *x* and *y* arguments give the coordinates of the upper left corner of the list box, relative to the upper left corner of the dialog box. The *x* argument is measured in 1/4 system-font character-width units. The *y* argument is measured in 1/8 system-font character-width units. (See **Begin Dialog**.)

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

The *dx* and *dy* arguments specify the width and height of the list box.

The *text$* argument is a string containing the selections for the list box. This string must be defined, using a **Dim** statement, before the **Begin Dialog** statement is executed. The arguments in the *text$* string are entered as shown in this example:

dimname = "*listchoice*"+Chr$(9)+"*listchoice*"+Chr$(9)
+"*listchoice*"+Chr$(9)...

Where *dimname* is the name of a **String** variable defined in a **Dim** statement, *listchoice* is the text that will appear as a selection in the list box, and **Chr$(**9**)** is the function call that produces a tab character. Note that multiple selections may be specified in *text$* by separating the *listchoices* with tab characters, as shown above.

The *.field* argument is the name of the dialog-record field that will hold 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* argument. The numbers begin at 0. If no item is selected, it is -1.

Loc function

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 function

Syntax	**Lof(** *filenumber%* **)**
Returns	The **Lof** function returns the length in bytes of the file specified by *filenumber%.*
Comments	*Filenumber%* is an integer expression identifying the open file from which the file length will be read. The *filenumber%* is the number used in the **Open** statement of the file.

Log function

Syntax **Log(** *numeric-expression* **)**

Returns The **Log** function returns the natural logarithm of *numeric-expression*.

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

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$ function

Syntax	**LTrim**[**$**]**(** *expression* **)**
Returns	The **LTrim$** function returns a copy of the source *string*, with all leading space characters removed. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will typically return a **variant** of vartype 8 (string). If the value of *expression* is null a variant of vartype 1 (null) is returned.
Comment	**LTrim$** accepts expressions of type string. **LTrim** accepts any type of *expression* including numeric values and will convert the input value to a string.

Me

Syntax **Me**

Comment Some Basic modules are attached to application objects and Basic subroutines are invoked when that application object encounters events. A good example is a user visible button that triggers a Basic routine when the user clicks the mouse on the button.

Subroutines in such contexts, may use the variable **Me** to refer to the object which triggered the event (I.e. which button was clicked). The programmer may use **Me** in all the same ways as any other object variable except that **Me** may not be **Set**.

Mid statement

Syntax	**Mid (** *string$***,** *position%*[**,** *length%*] **) =** *subst-string$*
Returns	The **Mid$** statement replaces the specified substring in *string$* with *subst-string$*.
Comment	If the *length%* argument is omitted, or if there are fewer characters in a string than specified in *length%*, then **Mid$** will replace 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.

Mid$ function

Syntax **Mid**[**$**]**(** *expression***,** *position%*[**,** *length%*] **)**

Returns The **Mid$** function returns a substring of a specified *length%* from a source *expression*, 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 will typically return a **variant** of vartype 8 (string). If the value of *expression* is null a variant of vartype 1 (null) is returned.

Comment If the *length%* argument is omitted, or if there are fewer characters in a string than specified in *length%*, then **Mid$** will return all the characters from the *position%* to the end of the string. If *position%* is larger than the number of characters in the indicated *expression*, then **Mid$** returns a null string.

The index of the first character in a string is 1.

Mid$ accepts expressions of type string. **Mid** accepts any type of *expression* including numeric values and will convert the input value to a string.

To modify a portion of a string value, see **Mid statement**

Minute function

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 statement

Syntax **MkDir** *pathname$*

Comments The **MkDir** function makes a new directory.

Pathname$* is a string expression identifying the new default directory. The syntax for *pathname$ is:

[*drive*:] [\] *directory* [*directory*]

The *drive* argument is optional. If *drive* is omitted, **MkDir** makes a new directory on the current drive. The *directory* argument is any directory name.

Month function

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 will attempt to convert the input value to a date value.

Msgbox function

Syntax **Msgbox(** *message$*[,[*type%*][, *caption$*]] **)**

Returns **Msgbox** returns an integer value indicating which button the user selected.

Comments The **Msgbox** function displays a *message* in a dialog box.(The message displayed must be no more than 1024 characters long. A message string greater than 255 characters without intervening spaces will be truncated after the 255th character.) Once the user has selected a pushbutton, **Msgbox** returns a value indicating the user's choice.

The *type%* argument governs the icons and buttons that will be displayed in the dialog box. This argument is the sum of values describing the number and type of buttons which will appear, the icon style, and the default button. One selection should be made from each group.

	Value	Description
Group 1	0	OK only
Buttons	1	OK, Cancel
	2	Abort, Retry, Ignore
	3	Yes, No, Cancel
	4	Yes, No
	5	Retry, Cancel
Group 2	16	Critical Message (STOP)
Icons	32	Warning Query (?)
	48	Warning Message (!)
	64	Information Message (i)
Group 3	0	First button
Defaults	256	Second button
	512	Third button

If *type%* is omitted, a single OK button will appear.

Caption% is a string expression that will appear in the dialog box's title bar.

The return values for the Msgbox function are:

Value	Button Pressed
1	OK
2	Cancel
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

Msgbox statement

Syntax **MsgBox** *message$*[,[*type%*][, *caption$*]]

Comments The **Msgbox** statement displays a *message* in a dialog box. A message can be no longer than 1024 characters in length. Messages longer than 255 characters which contain no intervening spaces will be truncated after the 255th character.

The *type%* argument governs the icons and buttons that will be displayed in the dialog box. This argument is the sum of values describing the number and type of buttons which will appear, the icon style, and the default button. One selection should be made from each group.

	Value	Description
Group 1	0	OK only
Buttons	1	OK, Cancel
	2	Abort, Retry, Ignore
	3	Yes, No, Cancel
	4	Yes, No
	5	Retry, Cancel
Group 2	16	Critical Message (STOP)
Icons	32	Warning Query (?)
	48	Warning Message (!)
	64	Information Message (i)
Group 3	0	First button
Defaults	256	Second button
	512	Third button

If *type%* is omitted, a single OK button will appear.

Caption% is a string expression that will appear in the dialog box's title bar.

Name statement

Syntax **Name** *oldfilename$* **As** *newfilename$*

Comments The **Name** statement renames a file. It can also be used to move a file from one directory to another.

Oldfilename$* and *newfilename$ are string expressions that designate, respectively, the file to rename and the new name for that file. A path may be part of the *filename$*. If the paths are different, the file is moved to the new directory.

A file must be closed in order to be renamed. If the file *oldfilename$* is open, BASIC generates an error message. If the file *newfilename$* already exists, BASIC will also generate an error message.

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.

$NoCStrings Metacommand

Syntax **'$NOCSTRINGS**

Comments The **$NoCStrings** metacommand tells the compiler to treat a backslash inside a string as a normal character.   This is the default.

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

Nothing function

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 function

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 function

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$ function

Syntax **Oct$(** *numeric-expression* **)**

Returns The **Oct$** function returns an octal representation of a *numeric-expression*, as a string. If the numeric expression is an integer, the string will contain up to six octal digits; otherwise, the expression will be converted to a long, and the string may 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 will return a **variant** of vartype 8 (string).

OkButton statement

Syntax **OK** *x, y, dx, dy*

Comments The **OkButton** statement determines the position and size of an OK button.

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

The *x* and *y* arguments set the position of the OK button relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the button. A *dy* value of 14 typically accommodates text in the system font.

On Error statement

Syntax	**ON [Local] Error** {**GoTo** *label* [**Resume Next**] **GoTo 0**}

Comments

The **On Error** statement enables an error-handling routine, specifying the location of that routine within procedure. **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, i.e., SBL will terminate the execution of the program.

An **On Error** statement is composed of the following parts:

<u>Part</u>	**Definition**
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 will generate an error message.
Resume Next	This 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.

When it is referenced by an **On Error GoTo** *label* statement, an error-handler is enabled. Once this enabling occurs, a run-time error will result 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, SBL will search for an error handler in the procedure which called the current procedure (if this fails, SBL will look for a handler belonging to the caller's caller, ...). If a handler is found, the current procedure will terminate, and the error handler in the calling procedure will be 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.

On Goto statement

Syntax **ON** *numeric-expression* **GoTo** *label1* [,*label2, ...*]

Comments The **On GoTo** statement sends the control to any one of the locations specified by *label1, label2, etc.* depending on the value of *numeric-expression.* If *numeric expression* evaluates to 1, the flow is transferred to *label1.* If the *numeric-expression* evaluates to 2, the flow is transferred to *label*2, and etc. If *numeric expression* evaluates to 0 or to a number greater than the number of labels following **GoTo**, the program continues at the next statement. If *numeric-expression* evaluates to a number less than 0 or greater than 255, an **"I**llegal function call" error is issued

Open statement

Syntax **Open** *filename$* [**For** *mode*] [**Access** *access*] [*lock*] **As [#]** *filenumber%* [**Len =** *reclen*]

Comments The **Open** statement enables I/O to a file or a device. A file must be opened before any input/output operation can be performed on it.

The argument *filename$* is a string expression specifying the file to open. If *file* does not exist, it is created when opened in **Append**, **Binary**, **Output** or **Random** modes.

Mode is a keyword that specifies one of the following: **Input** (sequential input mode), **Output** (sequential output mode), **Append** (sequential output mode), **Random** (random access mode) or **Binary** (binary I/O mode). If *mode* is not specified, it defaults to **Random**.

Access is a keyword that indicates the operations that are permitted on the open file. It is one of **Read**, **Write**, or **Read Write**. A value of **Read Write** for *Access* is only valid for files opened for **Random**, **Binary** or **Append**. If *Access* is not specified for **Random** or **Binary** modes, *Access* is attempted in the following order: **Read Write**, **Write**, **Read**.

Lock specifies the operations that other processes are allowed to perform when accessing *filename$*. Allowed values are **Shared** (any process can read or write the file)**, Lock Read** (other processes cannot perform read operations)**, Lock Write** (other processes cannot perform write operations on the file)**, Lock Read Write** (other processes cannot perform read or write operations on the file). If *lock* is not specified, *filename$* can be opened by other processes that do not specify a *lock*, although that process cannot perform any file operations on the file while the original process still has the file open.

Filenumber% is an integer expression with a value between 1 and 255. The **FreeFile** function can be used to return the next available filenumber.

Reclen specifies the record length for files opened in **Random** mode. It is ignored for all other modes

See also: Get, Input$ function, Input statement, Loc, Print, Put, Seek function, Seek, Width, Write.

Option Base statement

Syntax **Option Base** *lowerBound%*

Comments The **Option Base** statement specifies the default lower bound to be used for array subscripts. The *lowerBound* must be either 0 or 1. If no **Option Base** statement is specified, the default lower bound for array subscripts will be 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.

OptionButton statement

Syntax **OptionButton** *x, y, dx, dy, text$*

Comments The **OptionButton** statements --there must be at least two -- are used to define the position and text associated with an option button. They are used in conjunction with the **OptionGroup** statement.

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

The *x* and *y* arguments set the position of the button relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the button. A *dy* value of 12 typically accommodates text in the system font.

The *text$* field 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 will be truncated. If you wish to include underlined characters so that the option selection can be made from the keyboard, the character must be preceded with an ampersand (&).

Option Compare statement

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 statement

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 statement

Syntax **OptionGroup** *.field*

Comments The **OptionGroup** statement is used in conjunction with **OptionButton** statements to set up a series of related options. The **OptionGroup** statement begins definition of the option buttons and establishes the dialog-record field that will contain the current option selection. *.Field* will contain 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, etc.

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

PasswordBox$ function

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 that will be 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 statement

Syntax **Print** [# *filenumber%,*] *expressionlist* [{ ; | , }]

Comments The **Print** statement outputs data to the specified *filenumber%*. If the *expressionlist* is omitted, a blank line is written to the file.

Filenumber% is optional. If this argument is omitted, the **Print** statement outputs data to the screen. If provided *Filenumber%* is an integer expression identifying the print destination. See the **Open** statement for more details.

Expressionlist is a list of values that are to be printed. *Expressionlist* can contain numeric, string, and variant expressions. Expressions are separated by either a semi-colon (";") or a comma (",") . A semi-colon 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 [{;|,}] argument at the end of the **Print** statement determines where output for the next **Print** statement to the same output file should begin. A semi-colon will place output immediately after the output from this **Print** statement on the current line; a comma will start output at the next print zone on the current line. If neither separator is specified, a CR-LF pair will be generated and the next **Print** statement will print to the next line.

Special functions **Spc** and **Tab** can be used inside **Print** statement to insert a given number of spaces and to move the print position to a desired column.

The **Print** statement supports only elementary BASIC data types. See **Input** for more information on parsing this statement.

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 statement

Syntax **Randomize** [*numeric-expression%*]

Comments The **Randomize** function seeds the random number generator. The (optional) argument *numeric-expression%* is an integer value between -32768 and 32767. If no *numeric-expression%* argument is given, BASIC uses the Timer function to initialize the random number generator.

ReDim statement

Syntax **ReDim** [**Preserve**] *variableName* **(** *subscriptRange, ...* **)** [**As** [**New**] *type*] , ...

Comments You use the **ReDim** statement to change the upper and lower bounds of a dynamic array's dimensions. Memory for the dynamic array will be reallocated to support the specified dimensions, and the array elements may be reinitialized. **ReDim** can not 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 will be set to zero. Strings and variants will be 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 may 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, variants, records and objects. Arrays of arrays, dialog box records, and ADTs are not supported.

If the **As** clause is not used, the type of the variable may 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 only to change 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.

Care should be taken to avoid **ReDim**'ing an array in a procedure that has received a reference to an element in the array in an argument; the result is unpredictable.

The *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.

Rem statement

Syntax **Rem** *arbitrary text*

Comment **Rem** is used by the BASIC programmer to insert a comment in a BASIC program. Everything from **Rem** to the end of the line is ignored.

The single quote (') can also be used to initiate a comment. Metacommands (e.g., **CSTRINGS**) must be preceded by the single quote comment form.

Reset statement

Syntax	**Reset**
Comments	The **Reset** statement closes all disk files that are open, and writes any data still remaining in the operating system buffers to disk.

Resume statement

Syntax A	**Resume Next**
Syntax B	**Resume** *label*
Syntax C	**Resume** [**0**]

Comments The **Resume** statement halts an error-handling routine

When the **Resume Next** statement is used, control is passed to the statement which immediately follows the statement in which the error occurred.

When the **Resume** *label* statement is used, control is passed to the statement which immediately follows the specified label.

When the **Resume [0]** statement is used, control is passed to the statement in which the error occurred.

The location of the error handler which has caught the error determines where execution will resume. If an error is trapped in the same procedure as the error handler, program execution will resume 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.

Right$ function

Syntax **Right**[**$**]**(** *expression*, *length%* **)**

Returns The **Right$** function returns a string of a specified *length* copied from the rightmost length characters from the string *string$*.

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 If the length of *expression* is less than *length%*, **Right$** returns the whole string.

Right$ accepts expressions of type string. **Right** accepts any type of *expression* including numeric values and will convert the input value to a string.

RmDir statement

Syntax **RmDir** *pathname$*

Comments **RmDir** removes a directory.

Pathname$* is a string expression identifying the directory to remove. The syntax for *pathname$ is:

[*drive*:] [\] *directory* [*directory*]

The *drive* argument is optional. The *directory* argument is a directory name.

The directory to be removed must be empty, except for the working (.) and parent (..) directories.

Rnd function

Syntax	**Rnd** [**(** *number!* **)**]
Returns	The **Rnd** function returns a single precision random number between 0 and 1.
Comment	The same sequence of random numbers is generated whenever the program is run, unless the random number generator is re-initialized by the **Randomize** statement.

Rset statement

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$ function

Syntax	**RTrim**[**$**]**(** *expression* **)**
Returns	The **RTrim$** function returns a copy of the source *expression*, with all trailing space characters removed. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will typically return a **variant** of vartype 8 (string). If the value of *expression* is null a variant of vartype 1 (null) is returned
Comments	**RTrim$** accepts expressions of type string. **RTrim** accepts any type of *expression* including numeric values and will convert the input value to a string.

Second function

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 will attempt to convert the input value to a date value.

Seek function

Syntax **Seek(** *filenumber%* **)**

Returns The **Seek** 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, etc. The return value is a **Long**.

Comments *Filenumber%* is an integer expression identifying an open file to query for position. See the **Open** statement for more details.

Seek statement

Syntax	**Seek** [#] *filenumber%, position&*
Comments	The **Seek** statement sets the position within a file for the next read or write. If you write to a file after seeking beyond the end of the file, the file's length is extended. BASIC will return an error message if a **Seek** operation is attempted which specifies a negative or zero position.

Filenumber% is an integer expression identifying the open file to **Seek** in. See the **Open** statement for more details.

Position& is a numeric expression in the range 1 to 2,147,483,647 that indicates where the next write or read will occur. For files opened in **Random** mode, *position* is a record number, for all other modes, *position* is a byte offset. The first byte or record in the file is at position 1, the second is at position 2, etc.

Select Case statement

Syntax

Select Case *testexpression*

[**Case** *expressionlist*

[*statement_block*]]

[**Case** *expressionlist*

[*statement_block*]]

.

.

[**Case Else**

[*statement_block*]]

End Select

Comments

The **Select Case** statement is used to execute one of a series of statement blocks, depending on the value of an expression.

The *testexpression* can be any numeric, string, or variant expression that you wish to test. Each *statement_block* can contain any number of statements on any number of lines.

The *expressionlist(s)* may 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: <, >, =, <=, >=, <>.

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
^	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 application before sending keys unless is already active.

SendKeys cannot be used to send keys to an application which was not designed to run under Windows.

Set statement

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 statement

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

SetField$ function

Syntax **SetField$(** *string$*, *field_number%*, *field$*, *separator_chars$* **)**

Returns The **SetField$** function 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 may 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 will be 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 will be used as the separator character.

It is legal for the new field value to be a different size than the old field value.

Sgn function

Syntax	**Sgn(***numeric-expression***)**
Returns	**Sgn** returns a value indicating the sign of the *numeric-expression*.
Comments	The value that the **Sgn** function returns depends on the sign of the numeric-expression: For *numeric-expressions* > 0, **Sgn** (*numeric-expression*) returns 1. For *numeric-expressions* = 0, **Sgn** (*numeric-expression*) returns 0. For *numeric-expressions* < 0, **Sgn** (*numeric-expression*) returns -1.

Shell function

Syntax **Shell(** *commandstring$,* [*windowstyle%*] **)**

Returns The **Shell** function returns a Task ID, a unique number that identifies the running program.

Comments The **Shell** function runs an executable program. *Commandstring$* is the name of the program to execute. It may be the name of any valid .COM, .EXE., .BAT, or .PIF file. Arguments or command line switches can also be included. If *commandstring$* is not a valid executable file name, or if **Shell** cannot start the program, an error message will be generated.

Windowstyle% is the style of the window in which the program is to be executed. It may be one of the following:

Value	Window Style
1	Normal window with focus
2	Minimized with focus
3	Maximized with focus
4	Normal window without focus
7	Minimized without focus

If *windowstyle%* is not specified, the default of *windowstyle%* = 1 is assumed (normal window with focus).

Sin function

Syntax **Sin**(*angle*)

Returns The **Sin** function returns the sine of an angle. The return value will be between -1 and 1. The return value is single-precision if the angle is an integer, currency or single-precision value, double precision for a long, variant or double-precision value.

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

Space$ function

Syntax	**Space**[**$**]**(** *numeric-expression* **)**
Returns	The **Space$** function returns a string of spaces. 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 argument numeric-expression indicates the number of spaces which the returned string will contain. Any numeric data type can be used, but the number will be rounded to an integer. The numeric-expression must be between 0 and 32,767.

Spc function

Syntax **Spc (** *numeric-expression* **)**

Comments The **Spc** function can be used only inside **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 function

Syntax **Sqr(** *numeric-expression* **)**

Returns The **Sqr** function returns the square root of *numeric-expression*.

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

Static statement

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 statement

Syntax **Stop**

Comments The **Stop** statement halts program execution.

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.

Str$ function

Syntax **Str**[**$**]**(** *numeric-expression* **)**

Returns The **Str$** function returns a string representation of a *numeric-expression*.

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).

Comment The precision in the returned string is single-precision for an integer or single-precision numeric expression, double precision for a long or double-precision numeric expression, and currency precision for currency. Variants return the precision of their underlying vartype.

StrComp function

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$ function

Syntax A **String**[**$**]**(** *numeric-expression, charcode%* **)**

Syntax B **String**[**$**] **(***numeric-expression, string-expression$* **)**

Returns The **String$** function returns a string consisting of a repeated character.

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* specifies the length of the string to be returned. This number must be between 0 and 32,767.

Charcode% is a decimal ANSI code of the character that will be used to create the string. It is a numeric expression that BASIC will evaluate as an integer between 0 and 255.

String-expression$ is a string argument, the first character of which becomes the repeated character.

Sub ... End Sub statement

Syntax **[Static] [Private] Sub** *name* **[(** *parameter* **[As** *type***] , ...)]**

End Sub

Comments The **Sub..End Sub** structure defines a subprogram procedure. A call to a subprogram stands alone as a separate statement. (See the **Call** statement). Recursion is supported.

The parameters are specified as a comma-separated list of parameter names. The data type of a parameter may 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 **Sub** statement. All references to an array parameter within the body of the subprogram must have a consistent number of dimensions.

The procedure returns to the caller when the **End Sub** statement is reached or when an **Exit Sub** statement is executed.

The **Static** keyword specifies that all the variables declared within the subprogram will retain their values as long as the program is running, regardless of the way the variables are declared.

The **Private** keyword specifies that the procedures will not be accessible to functions and subprograms from other modules. Only procedures defined in the same module will have access to a **Private** subprogram.

BASIC procedures use the call by reference convention. This means that if a procedure assigns a value to a parameter, it will modify the variable passed by the caller.

The MAIN subprogram has a special meaning. In many implementations of BASIC, MAIN will be called when the module is "run". The MAIN subprogram is not allowed to take arguments.

Use **function** to define a procedure which has a return value.

Tab function

Syntax **Tab (** *numeric-expression* **)**

Comments The **Tab** function can be used only inside **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 equals 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.*

Tan function

Syntax	**Tan**(*angle*)
Returns	The **Tan** function returns the tangent of an angle. The return value is single-precision if the angle is an integer, currency or single-precision value, double precision for a long, variant or double-precision value.
Comments	The *angle* is specified in radians, and can be either positive or negative.

Text statement

Syntax **Text** *x, y, dx, dy, text$*

Comments The **Text** statement is used to place line(s) of text in a dialog box.

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

The *x* and *y* arguments set the position of the upper left hand corner of the text area relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the text area.

The *text$* field contains the text that will appear 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 will continue as long as the height of the text area established by *dy* is not exceeded. Excess characters will be truncated.

By preceding an underlined character in *text$* with an ampersand (&), you enable a user to press the underlined character on the keyboard and position the cursor in the combo or text box defined in the statement immediately following the **Text** statement.

TextBox statement

Syntax **TextBox [NoEcho]** *x, y, dx, dy, .field*

Comments The **TextBox** statement is used to create a box, within a dialog box, in which the user can enter and edit text .

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

The **NoEcho** keyword is often used for passwords. It displays all characters entered as asterisks.

The *x* and *y* arguments set the position of the upper left hand corner of the text box relative to the upper left corner of the dialog box. *Dx* and *dy* set the width and height of the text area. A *dy* value of 12 will usually accommodate text in the system font.

The *.field* argument is the name of the dialog-record field that will hold the text entered in the text box. When the user selects the OK button, or any pushbutton other than cancel, the text string entered in the text box will be recorded in the *.field* field.

Time$ function

Syntax	**Time**[**$**]
Returns	The **Time$** function returns a string representing the current time. The dollar sign, "$", in the function name is optional. If specified the return type is string. If omitted the function will return a **variant** of vartype 8 (string).
Comments	The **Time$** 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.

Time$ statement

Syntax	Time[**$**] **=** *expression*

Comments

The **Time$** statement is used to set the current system time.

When **Time$** (with the dollar sign '**$**') is used, the *expression* must evaluate to a string of one of the following forms:

hh	Set the time to *hh* hours 0 minutes and 0 seconds
hh:mm	Set the time to *hh* hours *mm* minutes and 0 seconds.
hh:mm:ss	Set the time to *hh* hours *mm* minutes and *ss* seconds

Time$ uses 24-hour clock. Thus, 6:00 P.M. must be entered as 18:00:00

If the dollar sign '**$**' is omitted, *expression* can be a string containing a valid date, a **variant** of vartype 7 (date), or a **variant** of vartype 8 (string).

If *expression* is not already a **variant** of vartype 7 (date), **Time** attempts to convert it to a valid time. It recognizes time separator defined in the International section of Windows Control Panel. Both 12 and 24 hour clocks are accepted.

Timer function

Syntax	**Timer**
Returns	The **Timer** function 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.

TimeSerial function

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 function

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.

Trim$ function

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 statement

Syntax

Type *userType*
field1 **As** *type1*
field2 **As** *type2*
' ...
End Type

Comments

The **Type** statement declares a user-defined type which can then be used in the DIM statement to declare a record variable. A user-defined type is sometimes referred to as a *record type* or a *structure type*.

Between the **Type** and **Type End** you may define a number of elements known as *fields*. Each field may be of the following type: string (either dynamic or fixed), number (integer, long, single, double, or currency), variant, or a previously-defined record type. A field may not be an array. However, arrays of records are allowed.

The **Type** statement is not valid inside of 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

Typeof function

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 function

Syntax **UBound(** *arrayVariable* [, *dimension*] **)**

Returns The **UBound** function 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.

UCase$ function

Syntax **UCase$(** *expression* **)**

Returns The **UCase$** function returns a copy of the source *expression*, with all lower case letters converted to upper case.

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 The translation is based on the country specified in the Windows Control Panel.

UCase$ accepts expressions of type string. **UCase** accepts any type of *expression* including numeric values and will convert the input value to a string.

Val function

Syntax **Val(** *string expression$* **)**

Returns The **Val** function returns a numeric value corresponding to the first number found in the specified *string*.

Comments Spaces in the source string are ignored. If no number is found, or if the number *is not in the first position of the string*, 0 is returned.

VarType function

Syntax **VarType(** *variant* **)**

Returns The **VarType** function returns the ordinal number representing the type of data currently stored in the *variant*.

Comments The value returned by **VarType** is one of the following:

Ordinal	Representation
0	(Empty)
1	**Null**
2	**Integer**
3	**Long**
4	**Single**
5	**Double**
6	**Currency**
7	Date
8	**String**

Weekday function

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 will attempt to convert the input value to a date value.

While ... Wend

Syntax

While *condition*

statementblock

Wend

Comments

The **While...Wend** structure controls a repetitive action. The *condition* is tested, and if non-zero (True), the *statementblock* is executed This process is repeated until *condition* becomes 0 (False).

The **While** statement is included in SBL for compatibility with older versions of Basic. The **Do** statement is a more general and powerful flow control statement.

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 statement

Syntax **Write** [**#**] *filenumber%* [,*expressionlist*]

Comments The **Write** statement writes data to a sequential file. The file must be opened in output or append mode. See the **Open** statement for more information.

The *filenumber%* is an integer expression identifying the open file to write to. *Expressionlist* specifies one or more values to be written to the file. If the expressionlist argument is omitted, the **Write** statement writes a blank line to the file. (See **Input** for more information.)

Year function

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 will attempt to convert the input value to a date value.

Expressions

Expressions are evaluated according to precedence order. Operators with higher precedence are evaluated before operators with lower precedence. Operators with equal precedence are evaluated from left to right. Parentheses can be used to override the default precedence. The following table lists the operator in precedence order from high to low.

^	Exponentiation
-,+	Unary minus and plus.
*, /	Numeric multiplication or division. For division, the result is a **Double**.
\	Integer division. The operands can be **Integer** or **Long**.
Mod	Modulus or Remainder. The operands can be **Integer** or **Long**.
-, +	Numeric addition and subtraction. The + operator is also used for string concatenation.
>, <, =, <=, >=, <>	Numeric or String comparison. For numbers, the operands will be widened to the least common type (**Integer** is preferred over **Long**, which is preferred over **Single**, which is preferred over **Double**). For **Strings**, the comparison is case-sensitive, and based on the collating sequence used by the language specified by the user using the Windows Control Panel. The result is 0 for FALSE and -1 for TRUE.
Not	Unary Not - operand can be **Integer** or **Long**. The operation is performed bitwise (one's complement).
And	And - 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.
Xor	Exclusive Or - operands can be **Integer** or **Long**. The operation is performed bitwise.
Eqv	Equivalence - operands can be **Integer** or **Long**. The operation is performed bitwise. (A **Eqv** B) is the same as (**Not** (A **Xor** B)).
Imp	Implication - operands can be **Integer** or **Long**. The operation is performed bitwise. (A **Imp** B) is the same as ((**Not** A) OR B).
.	Record member - the left operand must be a record variable, and the right operand must be the name of a field.
( )	Array element

Data Types and Type Conversion

BASIC is a strongly-typed language. Variables can be declared implicitly on first reference by using a type character; if no type character is present, the default type of **Variant** is assumed. Alternatively, the type of a variable can be declared explicitly with the **Dim** statement. In either case, the variable can only contain data of the declared type. Variables of user-defined type must be explicitly declared. SBL supports standard Basic numeric, string, record and array data. SBL also supports Dialog Box Records and Application Data Types (which are defined by the application).

Numbers

The five numeric types are:

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
Currency	from -922,337,203,685,477.5808 to 922,337,203,685,477.5807

Numeric values are always signed.

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 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).

Strings

BASIC strings can be either fixed or dynamic. Fixed strings have a length specified when they are defined, and the length cannot be changed. Fixed strings cannot be of 0 length. Dynamic strings have no specified length. Any string 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.

Records

A record, or record variable, is a data structure containing one or more elements, each of which has a value. Before declaring a record variable, a **Type** must be defined. Once the **Type** is defined, the variable can be declared to be of that type. The variable name should not have a type character suffix.

Record elements are referenced using dot notation, e.g., *varname.elementname*. Records can contain elements which are themselves records.

Arrays

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

See the **Dim** statement for more information.

Conversions

BASIC will automatically convert data between any two numeric types. When converting from a larger type to a smaller type (for example **Long** to **Integer**), a runtime numeric overflow may occur. This indicates that the number of the larger type is too large for the target data type. Loss of precision is not a runtime error (e.g., when converting from **Double** to **Single**, or from either float type to either integer type).

BASIC will also automatically convert 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 will be created. When converting from a dynamic string to a fixed string, some adjustment may be required. If the dynamic string is shorter than the fixed string, the resulting fixed string will be extended with spaces. If the dynamic string is longer than the fixed string, the resulting fixed string will be a truncated version of the dynamic string. No runtime errors are caused by string conversions.

BASIC will automatically convert between any data type and **variants**. BASIC will convert variant strings to numbers when required. A type mismatch error will occur if the variant string does not contain a valid representation of the required number.

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

Application Data Types (ADTs)

Application Data Types are specific to each application that embeds SBL. 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 will cause application-specific actions to occur. ADT variables and arrays are declared just like any other variable, using the **Dim** or **Global** statements.

Variant Data Type

The variant data type may be used to define variables that contain any type of data. A tag is stored with the variant data to identify the type of data that it currently contains. You may examine the tag by using the **VarType** function.

A variant may contain a value of any of the following types:

	Name	**Size of Data**	**Range**
0	(Empty)	0	N/A
1	Null	0	N/A
2	Integer	2 bytes (short)	-32768 to 32767
3	Long	4 bytes (long)	-2.147E9 to 2.147E9
4	Single	4 bytes (float)	-3.402E38 to -1.401E-45 (negative) 1.401E-45 to 3.402E38 (positive)
5	Double	8 bytes (double)	-1.797E308 to -4.94E-324 (negative) 4.94E-324 to 1.797E308 (positive)
6	Currency	8 bytes (fixed)	-9.223E14 to 9.223E14
7	(Date)	8 bytes (double)	January 1st, 0100 to December 31st, 9999
8	String	0 to ~64kbytes	0 to ~64k characters

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. You may test whether a variant is uninitialized (empty) with the **IsEmpty** function.

Null variants have no associated data and serve only to represent invalid or ambiguous results. You may test whether a variant contains a null value with the **IsNull** function. Null is not the same as Empty, which indicates that a variant has not yet been initialized.

Dialog Box Records

Dialog box records look like any other user-defined data type. Elements are referenced using the same *recname.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. See **Begin Dialog** statement for more information.

Trappable Errors

The following table lists the runtime errors which SBL returns. These errors can be trapped by **On Error**. The **Err** function can be used to query the error code, and the **Error$** function can be used to query the error text.

Error code	Error Text
5	Illegal function call
6	Overflow
7	Out of memory
9	Subscript out of range
10	Duplicate definition
11	Division by zero
13	Type Mismatch
14	Out of string space
19	No Resume
20	Resume without error
28	Out of stack space
35	Sub or Function not defined
48	Error in loading DLL
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
58	File already exists
61	Disk full
62	Input past end of file
63	Bad record number
64	Bad file name
68	Device unavailable
71	Disk not ready
74	Can't rename with different drive
75	Path/File access error
76	Path not found
94	Illegal use of NULL
102	Command failed
901	Input buffer would be larger than 64K
902	Operating system error
903	External procedure not found
904	Global variable type mismatch
905	User-defined type mismatch

906	External procedure interface mismatch
907	Pushbutton required
908	Module has no MAIN
910	Dialog box not declared

Class List

Following is a list of classes that may be used in a **Dim** statement, a **Typeof** expression or with the **New** operator:

Object	Provides access to Ole2 automation.

Objects

Objects provide access to software functionality outside of Basic. Object variables are always **Dim**ed as a particular class. One such class is named **Object** and provides access to Ole2 automation.

See also: **Class List**, **Set** , **Nothing**, **Typeof** , **Is**.