



Basic Script Language

Release 2015

Disclaimer

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples are for illustration only and are fictitious. No real association is intended or inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes only.

Sample Code Warranty disclaimer

Microsoft Corporation disclaims any warranty regarding the sample code contained in this documentation, including the warranties of merchantability and fitness for a particular purpose.

License agreement

Use of this software is covered by a license agreement provided with it. If you have any questions, please call the Customer Assistance Department at 800-456-0025 (in the United States or Canada) or +1-701-281-6500.

Copyright

© 2014 Microsoft Corporation. All rights reserved.

Publication Date

September 2014

Contents

Introduction	1
Welcome to Basic Script Language.....	1
What is in this Documentation?	1
Quick Reference: Statements and Functions	1
Language Overview.....	1
BSL Language Reference	1
Microsoft Dynamics SL API Function Calls	1
Glossary of Terms	1
Appendices.....	1
Programming Conventions	2
Arguments	2
Named Arguments	2
Arrays.....	3
Comments	3
Line Continuation.....	3
Records	3
Typographic Conventions.....	4
Quick Reference	5
Statements and Functions	5
Arrays.....	5
Compiler Directives.....	5
Control Flow	5
Dates and Times	5
Declarations	6
Dialog Boxes.....	6
Dynamic Data Exchange (DDE).....	7
Environment Control	7
Errors	7
Files	8
Math Functions	8
Objects.....	9
ODBC	9
Screen Input/Output.....	10
Strings	10
Variants	11
Language Overview	13
Data Types.....	13
Arrays	13
Numbers	13
Records.....	13
Strings.....	14
Data Type Conversions	14
Dynamic Arrays.....	15
Variant Data Type.....	16
Dialog Boxes	17
Step 1: Define a dialog box	18
Step 2: Write a dialog box function	18
Step 3: Display the dialog box.....	18
Dialog Functions and Statements.....	19
Dynamic Data Exchange (DDE)	20

Expressions.....	20
Numeric Operators.....	20
String Operators.....	20
Comparison Operators (Numeric and String).....	20
Logical Operators.....	21
Object Handling.....	22
Step 1: Create an object variable to access the application.....	23
Step 2: Use methods and properties to act on objects.....	23
Error Trapping	25
Overview.....	25
Trapping Errors Returned by BSL.....	25
Option 1: Trap error within body of code.....	26
Option 2: Trap error using error handler.....	26
Trapping User-Defined (Non-BSL) Errors.....	27
BSL Language Reference	29
Introduction.....	29
Abs Function.....	29
AppActivate Statement.....	30
Asc Function.....	31
Assert Statement.....	31
Atn Function.....	32
Beep Statement.....	33
Begin Dialog...End Dialog Statement.....	34
Bitmap Viewer.....	37
Button Statement.....	38
ButtonGroup Statement.....	39
Call Statement.....	40
CancelButton Statement.....	42
Caption Statement.....	43
CCur Function.....	44
CDBl Function.....	45
ChDir Statement.....	46
ChDrive Statement.....	47
CheckBox Statement.....	48
Chr Function.....	50
CInt Function.....	51
Clipboard.....	52
CLng Function.....	53
Close Statement.....	54
ComboBox Statement.....	55
Command Function.....	57
Const Statement.....	58
Cos Function.....	59
CreateObject Function.....	60
CSng Function.....	61
CStr Function.....	62
\$CStrings Metacommand.....	63
CurDir Function.....	64
CVar Function.....	65
CDate Function.....	66
Date Function.....	67
Date Statement.....	68
DateSerial Function.....	69
DateValue Function.....	70
Day Function.....	71

DDEAppReturnCode Function	71
DDEExecute Statement	72
DDEInitiate Function	74
DDEPoke Statement	76
DDERequest Function.....	78
DDETerminate Statement.....	80
Declare Statement	81
Deftype Statement	83
Dialog Function	84
Dialog Statement	85
Dim Statement	86
Dir Function	89
DlgControlID Function	91
DlgEnable Function.....	94
DlgEnable Statement.....	96
DlgEnd Statement	98
DlgFocus Function.....	100
DlgFocus Statement.....	101
DlgListBoxArray Function	102
DlgListBoxArray Statement.....	104
DlgSetPicture Statement	106
DlgText Function.....	108
DlgText Statement.....	110
DlgValue Function	112
DlgValue Statement	114
DlgVisible Function.....	116
DlgVisible Statement.....	117
Do...Loop Statement	119
DoEvents Statement	120
DropComboBox Statement.....	121
DropListBox Statement.....	123
Environ Function	125
Eof Function	126
Erase Statement	127
Erl Function	129
Err Function	130
Err Statement	131
Error Function.....	132
Error Statement.....	133
Exit Statement.....	134
Exp Function	135
FileAttr Function	136
FileCopy Statement.....	137
FileDateTime Function.....	138
FileLen Function.....	139
Fix Function	140
For...Next Statement	141
Format Function	143
Formatting Numbers.....	144
Formatting Dates and Times.....	146
Formatting Strings	148
FreeFile Function.....	149
Function...End Function Statement	150
FV Function.....	152
Get Statement	153
GetAttr Function	155
GetField Function	157
GetObject Function	158

Global Statement	160
GoTo Statement.....	163
GroupBox Statement.....	164
Hex Function	165
Hour Function	166
If...Then...Else	167
'\$Include Metacommand	168
Input Function.....	169
Input Statement.....	170
InputBox Function	172
InStr Function	173
Int Function.....	175
IPmt Function.....	176
IRR Function	177
Is Operator	178
IsDate Function	179
IsEmpty Function	180
IsMissing Function.....	181
IsNull Function.....	182
IsNumeric Function	183
Kill Statement.....	184
LBound Function	186
LCase Function.....	187
Left Function.....	188
Len Function	189
Let (Assignment Statement)	190
Like Operator.....	191
Line Input Statement	192
ListBox Statement	193
Loc Function	195
Lock Statement	196
Lof Function	198
Log Function	199
Lset Statement	200
LTrim Function	201
Me	202
Mid Function	203
Mid Statement.....	204
Minute Function	205
MkDir Statement	206
Month Function	207
Msgbox Function	208
Msgbox Statement	210
Name Statement	212
New Operator.....	213
\$NoCStrings Metacommand	214
Nothing Function	215
Now Function.....	216
NPV Function	217
Null Function.....	218
Object Class	219
Oct Function.....	220
OKButton Statement.....	221
On...Goto Statement.....	222
On Error Statement	223
Open Statement	225
OptionButton Statement.....	227
OptionGroup Statement.....	228

Option Base Statement	229
Option Compare Statement.....	230
Option Explicit Statement	231
PasswordBox Function.....	232
Picture Statement	233
Pmt Function	234
PPmt Function	235
Print Statement	236
PushButton Statement	237
Put Statement	238
PV Function.....	240
Randomize Statement.....	241
Rate Function	242
ReDim Statement.....	244
Rem Statement	246
Reset Statement	247
Resume Statement	248
Right Function	249
Rmdir Statement	250
Rnd Function	251
Rset Statement	252
RTrim Function	253
Second Function	254
Seek Function.....	255
Seek Statement	256
Select Case Statement	258
SendKeys Statement	260
Set Statement	262
SetAttr Statement	264
SetField Function	266
Sgn Function.....	267
Shell Function.....	268
Sin Function.....	269
Space Function	270
Spc Function.....	271
SQLClose Function	272
SQLError Function	273
SQLExecQuery Function.....	274
SQLGetSchema Function.....	275
SQLOpen Function	277
SQLRequest Function	278
SQLRetrieve Function	279
SQLRetrieveToFile Function	281
Sqr Function	282
Static Statement	283
StaticComboBox Statement	284
Stop Statement	285
Str Function	286
StrComp Function	287
String Function	288
Sub...End Sub Statement	289
Tab Function.....	290
Tan Function.....	291
Text Statement	292
TextBox Statement.....	293
Time Function.....	294
Time Statement.....	295
Timer Function	296

TimeSerial Function	297
TimeValue Function.....	298
Trim Function.....	299
Type Statement	300
Typeof Function	301
UBound Function.....	302
UCase Function	303
Unlock Statement.....	304
Val Function	305
VarType Function.....	306
Weekday Function.....	308
While...Wend.....	309
Width Statement	311
With Statement.....	312
Write Statement	313
Year Function.....	314
API Function Calls	315
Microsoft Dynamics SL Kernel Functions Summary	315
AliasConstant Statement	318
ApplGetParms Function	319
ApplGetParmValue Function.....	320
ApplGetReturnParms Function	322
Syntax.....	322
<i>ParmValue</i> = ApplGetReturnParms().....	322
Remarks.....	322
ScreenExit Statement	323
ApplSetFocus Statement	323
ApplSetParmValue Statement	324
CallChks Function.....	326
DateCheck Function	327
DateCmp Function.....	328
DateMinusDate Function	329
DatePlusDays Statement.....	330
DatePlusMonthSetDay Statement	331
DateToStr Function	332
DateToStrSep Function	333
DispFields Statement.....	334
DispForm Statement	335
DParm Function.....	336
Edit_Cancel Statement	337
Edit_Close Statement.....	337
Edit_Delete Function.....	338
Edit_Finish Function.....	339
Edit_First Function	340
Edit_Last Function.....	341
Edit_New Function	342
Edit_Next Function	343
Edit_Prev Function	344
Edit_Save Statement.....	345
ExportCustom Function.....	346
FPAdd Function.....	348
FParm Function	349
FPDiv Function.....	350
FPMult Function	351
FPRnd Function	352
FPSub Function	353
GetBufferValue Statement.....	354

GetDelGridHandle Function.....	355
GetGridHandle Function	356
GetObjectValue Function	357
GetProp Function	358
GetSysDate Statement	360
GetSysTime Statement	361
HideForm Statement.....	361
ImportCustom Function	362
ImportField Function.....	365
IncrStrg Statement.....	366
IParm Function	367
Is_TI Function	368
Launch Function.....	369
MCallChks Function	372
MClear Statement.....	372
MClose Statement	373
MDelete Function.....	374
MDisplay Statement.....	375
Mess Statement.....	376
MessBox Statement.....	378
Messf Statement.....	379
MessResponse Function	381
MExtend Function	382
mFindControlName.....	383
MFirst Function	385
MGetLineStatus Function	386
MGetRowNum Function.....	386
MInsert Statement	387
MKey Statement	388
MKeyFind Function	389
MKeyFld Statement	391
MKeyHctl Statement.....	392
MKeyOffset Statement	393
MLast Function.....	396
MLoad Statement	397
MNext Function	399
MOpen Functions.....	400
MPrev Function	402
MRowCnt Function	403
MSet Statement	403
MSetLineStatus Function	404
MSetProp Statement	406
MSetRowNum Statement.....	407
MUpdate Statement.....	407
NameAltDisplay Function.....	408
PasteTemplate Function.....	409
PeriodCheck Function	410
PeriodMinusPeriod Function	411
PeriodPlusPerNum Function	412
SaveTemplate Statement.....	413
SDelete Statement.....	414
SDeleteAll Function.....	415
SetAddr Statement.....	416
SetBufferValue Statement.....	419
SetDefaults Statement	420
SetLevelChg Statement	422
SetObjectValue Function	423
SetProp Statement.....	424

SFetch Functions.....	425
SGroupFetch Functions.....	427
SInsert Statements	429
SParm Function	431
Sql Statement.....	432
SqlCursor Statement.....	433
SqlCursorEx.....	435
SqlErr Function	438
SqlErrException Statement.....	440
SqlExec Statement	440
SqlFetch Functions.....	442
SqlFree Statement	444
SqlSubst Statement	445
StrToDate Statement	446
StrToTime Statement	446
SUpdate Statements.....	447
TestLevelChg Function	450
TimeToStr Function	451
TranAbort Statement.....	452
TranBeg Statement	453
TranEnd Statement	454
TranStatus Function.....	455
BSL Coding Tips and Techniques	457
Declaring Variables	457
Including External Files	457
Levels and Events	458
Message Functions	458
Sample Date Functions.....	459
Structures	460
Transactional Flow of User Fields.....	461
Treating Character Fields Like Date Fields	461
Using SQL Statements	462
Working with Grids	463
Parameter Passing Methods	465
Overview	465
Temporary Parameter File	466
Building and Retrieving Parameters	466
Parameter Passing Example - Printing Reports.....	467
BSL Report Example: Simple WHERE Clause.....	467
BSL Report Example: Complex WHERE Clause	468
Parameter-Passing Example - Sending Parameters.....	468
Parameter-Passing Example - Receiving Parameters.....	468
Using Parameter Files with ApplGetParmValue.....	468
How BSL Compares to Other Versions of Basic	471
Differences Between BSL and Earlier Versions of Basic.....	471
Line Numbers and Labels.....	471
Subroutines and Modularity of the Language	471
Global Variables	471
Data Types.....	471
Dialog Box Handling.....	472
Financial Functions.....	472
Date and Time Functions.....	472
Object Handling.....	472
Environment Control	472
Differences Between BSL and Visual Basic.....	473

Functions and Statements Unique to BSL	473
Control-Based Objects	473
Dialog Box Capabilities and VBA	473
Differences Between BSL and Word Basic	473
Dialog Box Capabilities	473
Button vs. PushButton	473
Dialog Box Units	473
User Input Mechanisms	474
Appendix 1: Trappable Errors	475
Error Codes	475
Appendix 2: Derived Trigonometric Functions	477
Derived Trigonometric Functions	477
Glossary	479
Index	481

Introduction

Welcome to Basic Script Language

Basic Script Language (BSL) is a module of Microsoft Dynamics® SL designed to help you create scripts that automate a variety of tasks.

Attention Basic Script Language Users:

Microsoft Dynamics SL 2015 is the last major release of Microsoft Dynamics SL that supports using Basic Script Language (BSL) customizations. To use BSL in Microsoft Dynamics SL you must install it separately. You can find the BSL Components installation in the *AdditionalComponentInstallers* folder within the Microsoft Dynamics SL installation package.

BSL customizations will cause an error if you do not install the BSL components. You must use Visual Basic for Applications (VBA) to replace the current BSL customizations before the next release.

For more information about how to use Customization Manager with Basic Script Language, see the “Appendix B: Basic Script Language related content” topic in the Customization Manager Help or user’s guide.

What is in this Documentation?

This section contains technical support information and programming and typographic conventions. The rest of the documentation consists of the major topics listed below.

Quick Reference: Statements and Functions

Offers a summary of all the commands and functions, divided into functional areas. It also contains brief descriptions of each command and function. See “Statements and Functions” on page 5.

Language Overview

Describes the essential rules and components of BSL. See “Language Overview” on page 13.

BSL Language Reference

Contains a full listing of every command and function, including examples, in Basic Script Language. See “BSL Language Reference” on page 29.

Microsoft Dynamics SL API Function Calls

Provides a complete list of Microsoft Dynamics SL kernel functions available in the Basic Script Language and a brief summary of their purpose. See “Microsoft Dynamics SL Kernel Functions Summary” on page 315.

Glossary of Terms

Contains definitions of commonly-used words and terms.

Appendices

Offers information about trappable error codes and derived trigonometric functions. Also provides comparisons between BSL and other versions of Basic.

Programming Conventions

Arguments

Arguments to subroutines and functions are listed after the subroutine or function and might or might not be enclosed in parentheses. Whether you use parentheses depends on how you want to pass the argument to the subroutine or function, either by **value** or by **reference**:

- If an argument is passed by **value**, it means that the variable used for that argument retains its value when the subroutine or function returns to the caller.
- If an argument is passed by **reference**, it means that the variable's value might be (and probably will be) changed for the calling procedure.

For example, suppose you set the value of a variable, `x`, to 5 and pass `x` as an argument to a subroutine, named `mysub`. If you pass `x` by value to `mysub`, the value of `x` will always be 5 after `mysub` returns. If you pass `x` by reference to `mysub`, however, `x` could be 5 or any other value resulting from the actions of `mysub`.

To pass an argument by **value**, use one of the following syntax options:

```
Call mysub((x))
mysub(x)
y=myfunction(x)
Call myfunction(x)
```

To pass an argument by **reference**, use one of the following options:

```
Call mysub(x)
mysub x
y=myfunction(x)
Call myfunction(x)
```

Externally declared subroutines and functions (such as DLL functions) can be declared to take `byVal` arguments in their declaration. In that case, those arguments are always passed `byVal`.

Named Arguments

When you call a subroutine or function that takes arguments, you usually supply values for those arguments by listing them in the order shown in the syntax for the statement or function. For example, suppose you define a function this way:

```
myfunction(id, action, value)
```

From the above syntax, you know that the function called **myfunction** requires three arguments: `id`, `action`, and `value`. When you call this function, you supply those arguments in the order shown. If the function contains just a few arguments, it is fairly easy to remember the order of each of the arguments. However, if a function has several arguments, and you want to be sure the values you supply are assigned to the correct arguments, use named arguments.

Named arguments are arguments identified by name rather than by position in the syntax. To use a named argument, use the following syntax:

```
namedarg:= value
```

Using this syntax for `myfunction`, you get:

```
myfunction id:=1, action:="get", value:=0
```

The advantage of named arguments, though, is that you do not need to remember the original order as they were listed in the syntax, so the following function call is also correct:

```
myfunction action:="get", value:=0, id:=1
```

With named arguments, order is not important.

The other significant advantage to named arguments is when you call functions or subroutines that have a mix of required and optional arguments. Ordinarily, you need to use commas as placeholders in the syntax for the optional arguments that you do not use. With named arguments, however, you

can specify just the arguments you want to use and their values and forget about their order in the syntax.

For example, if myfunction is defined as:

```
myfunction(id, action, value, Optional counter)
```

you can use named arguments as follows:

```
myfunction id:="1", action:="get", value:="0"
```

or,

```
myfunction value:="0", counter:="10", action:="get", id:="1"
```

Note: Although you can shift the order of named arguments, you cannot omit required arguments.

All BSL functions and statements accept named arguments. The argument names are listed in their syntax for the statement and function.

Arrays

Array dimensions are enclosed in parentheses after the array name:

```
arrayname(a,b,c)
```

Comments

Comments are preceded by an apostrophe and can appear on their own line in a procedure or directly after a statement or function on the same line:

```
'this comment is on its own line
Dim i as Integer      'this comment is on the code line
```

Line Continuation

Long statements can be continued across more than one line by typing a space-underscore at the end of a line and continuing the statement on the next line. (You can add a comment after the underscore.)

```
Dim trMonth As Integer _      'month of transaction
    trYear As Integer        'year of transaction
```

Note: In the following sections, some example code may exceed one line. In this case, the symbol ↵ indicates a line continuation character. The entire line must be on a single line in the event code window.

Records

Elements in a record are identified using the following syntax:

```
record.element
```

where *record* is the previously defined record name and *element* is a member of that record.

Typographic Conventions

The following typographic conventions are used throughout this documentation:

To represent:	Syntax Is...
Statements and functions	Boldface ; initial letter uppercase: Abs Len (<i>variable</i>)
Arguments to statements or functions	All lowercase, italicized letters: <i>variable, rate, prompt</i> \$
Optional arguments and/or characters	Italicized arguments and/or characters in brackets: [, <i>caption</i> \$], [<i>type</i> \$], [\$]
Required choice for an argument from a list of choices	A list inside braces, with OR operator () separating choices: {Goto <i>label</i> Resume Next Goto 0}

Quick Reference

Statements and Functions

The topics below provide a list of statements and functions by functional group.

Arrays

Erase – Reinitialize contents of an array.

Lbound – Return the lower bound of an array's dimension.

ReDim – Declare dynamic arrays and reallocate memory.

Ubound – Return the upper bound of an array's dimension.

Compiler Directives

\$Cstrings – Treat backslash in string as an escape character as in 'C'.

\$Include – Tell the compiler to include statements from another file.

\$NoCStrings – Tell the compiler to treat a backslash as a normal character.

Line Continuation – Continuing a long statement across multiple lines.

Rem – Treat the remainder of the line as a comment.

Control Flow

Call – Transfer control to a subprogram.

Do...Loop – Control repetitive actions.

Exit – Cause the current procedure or loop structure to return.

For...Next – Loop a fixed number of times.

Goto – Send control to a line label.

If ... Then ... Else – Branch on a conditional value.

Let – Assign a value to a variable.

Lset – Left-align one string or a user-defined variable within another.

On...Goto – Branch to one of several labels depending upon value.

Rset – Right-align one string within another.

Select Case – Execute one of a series of statement blocks.

Set – Set an object variable to a value.

Stop – Stop program execution.

While ... Wend – Control repetitive actions.

With – Execute a series of statements on a specified variable.

Dates and Times

Date Function – Return the current date.

Date Statement – Set the system date.

DateSerial – Return the date value for year, month, and day specified.

DateValue – Return the date value for string specified.

Day – Return the day of month component of a date-time value.

Hour – Return the hour of day component of a date-time value.

IsDate – Determine whether a value is a legal date.

Minute – Return the minute component of a date-time value.

Month – Return the month component of a date-time value.

Now – Return the current date and time.

Second – Return the second component of a date-time value.

Time Function – Return the current time.

Time Statement – Set the current time.

Timer – Return the number of seconds since midnight.

TimeSerial – Return the time value for hour, minute, and second specified.

TimeValue – Return the time value for string specified.

Weekday – Return the day of the week for the specified date-time value.

Year – Return the year component of a date-time value.

Declarations

Const – Declare a symbolic constant.

Declare – Forward declare a procedure in the same module or in a dynamic link library.

Deftype – Declare the default data type for variables.

Dim – Declare variables.

Function ... End Function – Define a function.

Global – Declare a global variable.

Option Base – Declare the default lower bound for array dimensions.

Option Compare – Declare the default case sensitivity for string comparisons.

Option Explicit – Force all variables to be explicitly declared.

ReDim – Declare dynamic arrays and reallocate memory.

Static – Define a static variable or subprogram.

Sub ... EndSub – Define a subprogram.

Type – Declare a user-defined data type.

Dialog Boxes

Dialog Function – Display a dialog box and return the button pressed.

Dialog Statement – Display a dialog box.

DlgControlId – Return numeric ID of a dialog control.

DlgEnable Function – Tell whether a dialog control is enabled or disabled.

DlgEnable Statement – Enable or disable a dialog control.

DlgEnd – Close the active dialog box.

DlgFocus Function – Return ID of the dialog control having input focus.

DlgFocus Statement – Set focus to a dialog control.

DlgListBoxArray Function – Return contents of a list box or combo box.

DlgListBoxArray Statement – Set contents of a list box or combo box.

DlgSetPicture – Change the picture in the Picture control.

DlgText Function – Return the text associated with a dialog control.

DlgText Statement – Set the text associated with a dialog control.

DlgValue Function – Return the value associated with a dialog control.

DlgValue Statement – Set the value associated with a dialog control.

DlgVisible Function – Tell whether a control is visible or hidden.

DlgVisible Statement – Show or hide a dialog control.

Begin Dialog – Begin a dialog box definition.

Button – Define a button dialog box control.

ButtonGroup – Begin definition of a group of button dialog box controls.

CancelButton – Define a Cancel button dialog box control.

Caption – Define the title of a dialog box.

CheckBox – Define a check box dialog box control.

ComboBox – Define a ComboBox dialog box control.

DropComboBox – Define a drop-down combo box dialog box control.

DropListBox – Define a drop-down list box dialog box control.

GroupBox – Define a group box in a dialog box.

ListBox – Define a list box dialog box control.

OKButton – Define an OK button dialog box control.

OptionButton – Define an OptionButton dialog box control.

OptionGroup – Begin definition of a group of OptionButton dialog box controls.

Picture – Define a Picture control.

PushButton – Define a pushbutton dialog box control.

StaticComboBox – Define a static combo box dialog box control.

Text – Define a line of text in a dialog box.

TextBox – Define a text box in a dialog box.

Dynamic Data Exchange (DDE)

DDEAppReturnCode – Return a code from an application on a DDE channel.

DDEExecute – Send commands to an application on a DDE channel.

DDEInitiate – Open a dynamic data exchange (DDE) channel.

DDEPoke – Send data to an application on a DDE channel.

DDERequest – Return data from an application on a DDE channel.

DDETerminate – Close a DDE channel.

Environment Control

AppActivate – Activate another application.

Command – Return the command line specified when the MAIN sub was run.

Date Statement – Set the current date.

DoEvents – Let operating system process messages.

Environ – Return a string from the operating system's environment.

Randomize – Initialize the random-number generator.

SendKeys – Send keystrokes to another application.

Shell – Run an executable program.

Errors

Assert – Trigger an error if a condition is false.

Erl – Return the line number where a run-time error occurred.

Err Function – Return a run-time error code.

Err Statement – Set the run-time error code.

Error – Generate an error condition.

Error Function — Return a string representing an error.

On Error — Control run-time error handling.

Resume — End an error-handling routine.

Trappable Errors — Errors that can be trapped by BSL code.

Files

ChDir — Change the default directory for a drive.

ChDrive — Change the default drive.

Close — Close a file.

CurDir — Return the current directory for a drive.

Dir — Return a filename that matches a pattern.

Eof — Check for end of file.

FileAttr — Return information about an open file.

FileCopy — Copy a file.

FileDateTime — Return modification date and time of a specified file.

FileLen — Return the length of specified file in bytes.

FreeFile — Return the next unused file number.

Get — Read bytes from a file.

GetAttr — Return attributes of specified file, directory of volume label.

Input Function — Return a string of characters from a file.

Input Statement — Read data from a file or from the keyboard.

Kill — Delete files from a disk.

Line Input — Read a line from a sequential file.

Loc — Return current position of an open file.

Lock — Control access to some or all of an open file by other processes.

Lof — Return the length of an open file.

MkDir — Make a directory on a disk.

Name — Rename a disk file.

Open — Open a disk file or device for I/O.

Print — Print data to a file or to the screen.

Put — Write data to an open file.

Reset — Close all open disk files.

RmDir — Remove a directory from a disk.

Seek Function — Return the current position for a file.

Seek Statement — Set the current position for a file.

SetAttr — Set attribute information for a file.

Spc — Output given number of spaces.

Tab — Move print position to the given column.

Unlock — Control access to some or all of an open file by other processes.

Width — Set output-line width for an open file.

Write — Write data to a sequential file.

Math Functions

Abs — Return the absolute value of a number.

Atn — Return the arc tangent of a number.

Cos — Return the cosine of an angle.

Derived Functions – How to compute other numeric or trigonometric functions.

Exp – Return the value of e raised to a power.

Fix – Return the integer part of a number.

FV – Return future value of a cash flow stream.

Int – Return the integer part of a number.

Ipmt – Return interest payment for a given period.

IRR – Return internal rate of return for a cash flow stream.

IsNumeric – Determine whether a value is a legal number.

Log – Return the natural logarithm of a value.

NPV – Return net present value of a cash flow stream.

Pmt – Return a constant payment per period for an annuity.

PPmt – Return principal payment for a given period.

PV – Return present value of a future stream of cash flows.

Rate – Return interest rate per period.

Rnd – Return a random number.

Sgn – Return a value indicating the sign of a number.

Sin – Return the sine of an angle.

Sqr – Return the square root of a number.

Tan – Return the tangent of an angle.

Objects

Clipboard – Access the Windows Clipboard.

CreateObject – Create an OLE2 automation object.

GetObject – Retrieve an OLE2 object from a file or get the active OLE2 object for an OLE2 class.

Is – Determine whether two object variables refer to the same object.

Me – Get the current object.

New – Allocate and initialize a new OLE2 object.

Nothing – Set an object variable to not refer to an object.

Object – Declare an OLE2 automation object.

Typeof – Check the class of an object.

With – Execute statements on an object or a user-defined type.

ODBC

SQLClose – Close a data source connection.

SQLError – Return a detailed error message ODBC functions.

SQLExecQuery – Execute an SQL statement.

SQLGetSchema – Obtain information about data sources, databases, terminology, users, owners, tables, and columns.

SQLOpen – Establish a connection to a data source for use by other functions.

SQLRequest – Make a connection to a data source, execute an SQL statement, return the results.

SQLRetrieve – Return the results of a select that was executed by SQLExecQuery into a user-provided array.

SQLRetrieveToFile – Return the results of a select that was executed by SQLExecQuery into a user-specified file.

Screen Input/Output

Beep – Produce a short beeping tone through the speaker.

Input Function – Return a string of characters from a file.

Input Statement – Read data from a file or from the keyboard.

InputDialog – Display a dialog box that prompts for input.

MsgBox Function – Display a Windows message box.

MsgBox Statement – Display a Windows message box.

PasswordBox – Display a dialog box that prompts for input. Do not echo input.

Print – Print data to a file or to the screen.

Strings

Asc – Return an integer corresponding to a character code.

Ccur – Convert a value to currency.

CDbl – Convert a value to double-precision floating point.

Chr – Convert a character code to a string.

Cint – Convert a value to an integer by rounding.

CLng – Convert a value to a long by rounding.

CSng – Convert a value to single-precision floating point.

CStr – Convert a value to a string.

Cvar – Convert an number or string to a Variant.

CVDate – Convert a value to a variant date.

Format – Convert a value to a string using a picture format.

Val – Convert a string to a number.

GetField – Return a substring from a delimited source string.

Hex – Return the hexadecimal representation of a number, as a string.

InStr – Return the position of one string within another.

Lcase – Convert a string to lower case.

Left – Return the left portion of a string.

Len – Return the length of a string or size of a variable.

Like Operator – Compare a string against a pattern.

Ltrim – Remove leading spaces from a string.

Mid Function – Return a portion of a string.

Mid Statement – Replace a portion of a string with another string.

Oct – Return the octal representation of a number, as a string.

Right – Return the right portion of a string.

RTrim – Remove trailing spaces from a string.

SetField – Replace a substring within a delimited target string.

Space – Return a string of spaces.

Str – Return the string representation of a number.

StrComp – Compare two strings.

String – Return a string consisting of a repeated character.

Trim – Remove leading and trailing spaces from a string.

UCase – Convert a string to upper case.

Variants

IsEmpty – Determine whether a variant has been initialized.

IsNull – Determine whether a variant contains a NULL value.

Null – Return a null variant.

VarType – Return the type of data stored in a variant.

Language Overview

Data Types

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. BSL supports standard Basic numeric, string, record and array data. BSL also supports Dialog Box Records and Objects (which are defined by the application).

Arrays

Arrays are created by specifying one or more subscripts at declaration or **Redim** time. Subscripts specify the beginning and ending index for each dimension. If only an ending index is specified, the 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, (*arrayname(i,j,k)*). See the “ReDim Statement” on page 244 for more information.

Numbers

The five numeric types are:

Type	From	To
Integer	-32,768	32,767
Long	-2,147,483,648	2,147,483,647
Single	-3.402823e+38 0.0, 1.401298e-45	-1.401298e-45 (negative) 3.402823466e+38 (positive)
Double	-1.797693134862315d+308 0.0, 2.2250738585072014d-308	-4.94065645841247d-308 (negative) 1.797693134862315d+308 (positive)
Currency	-922,337,203,685,477.5808	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” (for example, &o177). To represent a hexadecimal value, precede the constant with “&H” or “&h” (for example, &H8001).

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 (*varname.elementname*). Records can contain elements that are themselves 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 the “Begin Dialog...End Dialog Statement” on page 34 for more information.

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 that can be included in a string. For example, the character whose ANSI value is 0 can be embedded in strings.

Data Type 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 might 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 (for example., 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 that 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 might 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.

Dynamic Arrays

Dynamic arrays differ from fixed arrays in that you do not specify a subscript range for the array elements when you dimension the array. Instead, the subscript range is set using the **ReDim** statement. With dynamic arrays, you can set the size of the array elements based on other conditions in your procedure. For example, you might want to use an array to store a set of values entered by the user, but you do not know in advance how many values the user has. In this case, you dimension the array without specifying a subscript range and then execute a **ReDim** statement each time the user enters a new value. Or, you might want to prompt for the number of values a user has and execute one **ReDim** statement to set the size of the array before prompting for the values.

If you use **ReDim** to change the size of an array and want to preserve the contents of the array at the same time, be sure to include the **Preserve** argument to the **ReDim** statement.

If you **Dim** a dynamic array before using it, the maximum number of dimensions it can have is 8. To create dynamic arrays with more dimensions (up to 60), do not **Dim** the array at all; instead use just the **ReDim** statement inside your procedure.

The following procedure uses a dynamic array, *varray*, to hold cash flow values entered by the user:

```
Sub main
    Dim aprate as Single
    Dim varray() as Double
    Dim cflowper as Integer
    Dim msgtext
    Dim x as Integer
    Dim netpv as Double
    cflowper=InputBox("Enter number of cash flow periods")
    ReDim varray(cflowper)
    For x= 1 to cflowper
        varray(x)=InputBox("Enter cash flow amount for period #" & x & ":")
    Next x
    aprate=InputBox("Enter discount rate: ")
    If aprate>1 then
        aprate=aprate/100
    End If
    netpv=NPV(aprate,varray())
    msgtext="The net present value is: "
    msgtext=msgtext & Format(netpv, "Currency")
    MsgBox msgtext
End Sub
```

Variant Data Type

The variant data type can 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 can examine the tag by using the **VarType** function.

A variant can contain a value of any of the following types:

Type/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)	Jan 1st, 100 to Dec 31st, 9999
8 String	0 to ~64kbytes	0 to ~64k characters
9 Object	N/A	N/A

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 can 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 can 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 Boxes

To create and run a dialog box:

1. Define a dialog box record using the **Begin Dialog...End Dialog** statements and the dialog box definition statements, such as **TextBox**, **OKButton**.
2. Create a function to handle dialog box interactions using the **Dialog Functions and Statements**. (Optional)
3. Display the dialog box using either the **Dialog Function** or **Dialog Statement**.

The example code skeleton below illustrates these steps.

```

Declare Function myfunc(identifier$,action,suppvalue)
Sub Main
  Begin Dialog NEWDLG dimx, dimy, caption, myfunc
  ListBox.....
  ComboBox.....
  OKButton....
  CancelButton....
End Dialog

  Dim dlg as NEWDLG
  Dim response as Integer
  response=Dialog(dlg)
  If response=-1 then
    'clicked OK button
  ElseIf reponse= 0 then
    'clicked Cancel button
  ElseIf response> 0 then
    'clicked another command button
  End If
End Sub

Function myfunc(identifier$,action,suppvalue)
  '...code to handle dialog box actions
End Function

```

Step 1
Define the dialog box

Step 3
Display the dialog box

Step 2
Write a function to handle dialog box interaction

Step 1: Define a dialog box

The **Begin Dialog...End Dialog** statements define a dialog box. The last parameter to the **Begin Dialog** statement is the name of a function, prefixed by a period (.). This function handles interactions between the dialog box and the user.

The **Begin Dialog** statement supplies three parameters to your function:

- An **identifier** (a dialog control ID)
- The **action** taken on the control
- A **value** with additional action information.

Your function should have these three arguments as input parameters. See the “**Begin Dialog...End Dialog Statement**” on page 34 for more information.

Step 2: Write a dialog box function

This function defines dialog box behavior. For example, your function could disable a check box, based on a user's action. The body of the function uses the “Dig”-prefixed BSL statements and functions to define dialog box actions.

Define the function itself using the **Function...End Function** statement or declare it using the **Declare** statement *before* using the **Begin Dialog** statement. Enter the name of the function as the last argument to **Begin Dialog**. The function receives three parameters from **Begin Dialog** and returns a value. Return a non-zero value to leave the dialog box open after the user clicks a command button (such as Help).

Step 3: Display the dialog box

You use the **Dialog** function (or statement) to display a dialog box. The argument to **Dialog** is a variable name that you previously dimensioned as a dialog box record. The name of the dialog box record comes from the **Begin Dialog...End Dialog** statement. The return values for the **Dialog** function determine which key was pressed: -1 for OK, 0 for Cancel, >0 for a command button. If you use the **Dialog** statement, it returns an error if the user presses Cancel, which you can then trap with the **On Error** statement.

Dialog Functions and Statements

The function you create uses the “**Dig**” dialog functions and statements to manipulate the active dialog box. This is the *only* function that can use these functions and statements. The list of the “**Dig**” functions and statements is as follows:

DigControlId – Return numeric ID of a dialog control.

DigEnable Function – Tell whether a control is enabled or disabled.

DigEnable Statement – Enable or disable a dialog control.

DigFocus Function – Return ID of the dialog control having input focus.

DigFocus Statement – Set focus to a dialog control.

DigListBoxArray Function – Return contents of a list box or combo box.

DigListBoxArray Statement – Set contents of a list box or combo box.

DigText Function – Return the text associated with a dialog control.

DigText Statement – Set the text associated with a dialog control.

DigValue Function – Return the value associated with a dialog control.

DigValue Statement – Set the value associated with a dialog control.

DigVisible Function – Tell whether a control is visible or disabled.

DigVisible Statement – Show or hide a dialog control.

Most of these functions and statements take control ID as their first argument. For example, if a check box was defined with the following statement:

```
CheckBox 20, 30, 50, 15, "My check box", .Check1
```

Then, **DigEnable** “Check1”, 1 enables the check box, and **DigValue**(“Check1”) returns 1 if the check box is currently selected, 0 if not. Note that the IDs are case-sensitive and do not include the dot that appears before the ID. Dialog functions and statements can also work with numeric IDs. Numeric IDs depend on the order in which dialog controls are defined.

For example, if the check box that we considered was the first control defined in the dialog record, then **DigValue**(0) would be equivalent to **DigValue**(“Check1”). (The control numbering begins from 0, and the **Caption** control does not count.) Find the numeric ID using the **DigControlID** function.

Note that for some controls (such as buttons and texts) the last argument in the control definition, ID, is optional. If it is not specified, the text of the control becomes its ID. For example, the Cancel button can be referred as “Cancel” if its ID was not specified in the **CancelButton** statement.

Dynamic Data Exchange (DDE)

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 BSL 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.

Expressions

An expression is a collection of two or more terms that perform a mathematical or logical operation. The terms are usually either variables or functions that are combined with an operator to evaluate to a string or numeric result. You use expressions to perform calculations, manipulate variables, or concatenate strings. Expressions are evaluated according to precedence order. Use parentheses to override the default precedence order.

The precedence order (from high to low) for the operators is:

- Numeric Operators
- String Operators
- Comparison Operators
- Logical Operators

Numeric Operators

- ^** 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 can also be used for string concatenation.

String Operators

- &** String concatenation
- +** String concatenation

Comparison Operators (Numeric and String)

- >** Greater than

<	Less than
=	Equal to
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

For numbers, the operands are 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.

Logical Operators

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).

Object Handling

Objects are the end products of a software application, such as a spreadsheet, graph, or document. Each software application has its own set of properties and methods that change the characteristics of an object.

Properties affect how an object behaves. For example, width is a property of a range of cells in a spreadsheet, colors are a property of graphs, and margins are a property of word processing documents.

Methods cause the application to do something to an object. Examples are Calculate for a spreadsheet, Snap to Grid for a graph, and AutoSave for a document.

In BSL, you have the ability to access an object and use the originating software application to change properties and methods of that object. Before you can use an object in a procedure, however, you must access the software application associated with the object by assigning it to an object variable. Then you attach an object name (with or without properties and methods) to the variable to manipulate the object. The syntax for doing this is shown in the following example code.

```

Sub main
  Dim visio as Object
  Dim doc as Object
  Dim page as Object
  Dim i as Integer, doccount as Integer
  Set visio = GetObject("visio.application")
  If (visio Is Nothing) then
    Set visio = CreateObject("visio.application")
    If (visio Is Nothing) then
      MsgBox "Couldn't find visio!"
      Exit Sub
    End If
  End If
  doccount = visio.documents.count
  For i = 1 to doccount
    Set doc = visio.documents(i)
    If doc.name = "myfile.vsd" then
      Set page = doc.pages(1)
    End If
  Next i
  Set doc = visio.documents.open("myfile.vsd")
  Set page = doc.pages(1)
End Sub

```

Step 1
Create an object variable to access the application

Step 2
Use methods and properties to act on the objects

Note: The examples shown here are specific to Microsoft® Visio® software application. Object, property and method names vary from one application to another. You will need to refer to the software documentation for the application you want to access for the applicable names to use.

Step 1: Create an object variable to access the application

The **Dim** statement creates an object variable called “visio” and assigns the application, Microsoft Visio, to it. The **Set** statement assigns the Visio application to the variable using either **GetObject** or **CreateObject**. You use **GetObject** if the application is already open on the Windows desktop. Use **CreateObject** if the application is not open.

Step 2: Use methods and properties to act on objects.

To access an object, property or method, you use this syntax:

appvariable.object.property

appvariable.object.method

For example, **visio.document.count** is a value returned by the Count method of the Document object for the Visio application, which is assigned to the Integer variable doccount.

Alternatively, you can create a second object variable and assign the Document object to it using Visio's Document method, as the **Set** statement shows.

Error Trapping

Overview

BSL contains three error handling statements and functions for trapping errors in your program: **Err**, **Error**, and **On Error**. BSL returns a code for many of the possible runtime errors you might encounter. See "Appendix 1: Trappable Errors" on page 475 for a complete list of codes.

In addition to the errors trapped by BSL, you might want to create your own set of codes for trapping errors specific to your program. You would do this if, for example, your program establishes rules for file input and the user does not follow the rules. You can trigger an error and respond appropriately using the same statements and functions you would use for BSL-returned error codes.

Regardless of the error trapped, you have one of two methods to handle errors; one is to put error-handling code directly before a line of code where an error might occur (such as after a File Open statement), and the other is to label a separate section of the procedure just for error handling, and force a jump to that label if any error occurs. The On Error statement handles both options.

Trapping Errors Returned by BSL

This code example shows the two ways to trap errors. Option 1 places error-handling code directly before the line of code that could cause an error. Option 2 contains a labeled section of code that handles any error.

```

Sub main
  Dim userdrive, userdir, msgtext
  in1: userdrive=InputBox("Enter drive:","C:")
  Option 1
  Place error-
  handling code
  within the body
  of a procedure
  On Error Resume Next
  Err=0
  ChDrive userdrive
  If Err=68 then
    MsgBox "Invalid Drive. Try again."
    Goto in1
  End If

```



```

Option 2
  Place error-
  handling code
  at the end of
  a procedure
  and Goto it via
  a label
  On Error Goto Errhdr1
  in2: userdir=InputBox("Enter directory:")
  ChDir userdrive & "\" & userdir
  MsgBox "New default directory is: " & userdrive & "\" & userdir
  Exit Sub
  Errhdr1:
  Select Case Err
    Case 75
      msgtext="Path is invalid."
    Case 76
      msgtext="Path not found."
    Case Else
      msgtext="Error " & Err & ": " & Error$ & "occurred."
  End Select
  MsgBox msgtext & " Try again."
  Resume in2
End Sub

```

Option 1: Trap error within body of code

The **On Error** statement identifies the line of code to go to in case of an error. In this case, the Resume Next parameter means execution continues with the next line of code after the error. In this example, the line of code to handle errors is the **If** statement. It uses the **Err** statement to determine which error code is returned.

Option 2: Trap error using error handler

The **On Error** statement used here specifies a label to jump to in case of errors. The code segment is part of the main procedure and uses the **Err** statement to determine which error code is returned. To make sure your code does not accidentally fall through to the error handler, precede it with an **Exit** statement.

Trapping User-Defined (Non-BSL) Errors

These code examples show the two ways to set and trap user-defined errors. Both options use the **Error** statement to set the user-defined error to the value 30000. To trap the error, option 1 places error-handling code directly before the line of code that could cause an error. Option 2 contains a labeled section of code that handles any user-defined errors.

Option 1
Place error-handling code within the body of a procedure

```

Sub Main
  Dim custname as String
  On Error Resume Next
  in1: Err=0
  custname=InputBox$("Enter customer name:")
  If custname="" then
    Error 30000
    Select Case Err
      Case 30000
        MsgBox "You must enter a customer name."
        Goto in1
      Case Else
        MsgBox "Undetermined error. Try again."
        Goto in1
    End Select
  End If
  MsgBox "The name is: " & custname
End Sub

```

Option 2
Place error-handling code at the end of a procedure and Goto it via a label

```

Sub Main
  Dim custname as String
  On Error Goto Errhandler
  in1: Err=0
  custname=InputBox$("Enter customer name:")
  If custname="" then
    Error 30000
  End If
  MsgBox "The name is: " & custname
  Exit Sub
Errhandler:
  Select Case Err
    Case 30000
      MsgBox "You must enter a customer name."
    Case Else
      MsgBox "Undetermined error. Try again."
  End Select
  Resume in1
End Sub

```


BSL Language Reference

Introduction

This section lists the BSL commands and functions in alphabetical order, and indicates syntax, return value, comments, an example, and a list of related commands.

This information is also included in the Help system. Depending on the version you are using, the Help system may contain more recent material.

Abs Function

Returns the absolute value of a number.

Syntax

Abs(*number*)

Where:

Is:

number Any valid numeric expression.

Remarks

The data type of the return value matches the type of the *number*. If *number* is a variant string (vartype 8), the return value will be converted to vartype 5 (Double). If the absolute value evaluates to vartype 0 (Empty), the return value will be vartype 3 (Long).

Example

This example finds the difference between two variables, oldacct and newacct.

```
Sub main
Dim oldacct, newacct, count
oldacct=InputBox("Enter the oldacct number")
newacct=InputBox("Enter the newacct number")
count=Abs(oldacct-newacct)
MsgBox "The absolute value is: " &count
End Sub
```

See Also

Exp, Fix, Int, Log, Rnd, Sgn, Sqr

AppActivate Statement

Activates an application window.

Syntax

AppActivate *title*

Where: **Is:**

title A string expression for the title bar name of the application window to activate.

Remarks

Title must match the name of the window character for character, but comparison is not case-sensitive, for example, "File Manager" is the same as "file manager" or "FILE MANAGER". If there is more than one window with a name matching *title*, a window is chosen at random.

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

Example

This example opens the Windows bitmap file SETUP.BMP in Paint. (Paint must already be open before running this example. It must also not be minimized.)

```
Sub main
  MsgBox "Opening C:\WINDOWS\SETUP.BMP in Paint."
  AppActivate "untitled - Paint"
  DoEvents
  SendKeys "%FOC:\WINDOWS\SETUP.BMP{Enter}",1
  MsgBox "File opened."
End Sub
```

See Also

SendKeys, Shell

Asc Function

Returns an integer corresponding to the character code of the first character in the specified string.

Syntax

Asc(*string\$*)

Where: **Is:**

string\$ A string expression of one or more characters.

Remarks

To obtain the first byte of a string, use **AscB**.

To change a character code to a character string, use **Chr**.

Example

This example asks the user for a letter and returns its ASCII value.

```
Sub main
    Dim userchar
    userchar=InputBox("Type a letter:")
    MsgBox "The ASC value for " & userchar & " is: " & Asc(userchar)
End Sub
```

See Also

Chr

Assert Statement

Triggers a run-time error if the condition specified is FALSE.

Syntax

Assert *condition*

Where: **Is:**

condition A numeric or string expression that can evaluate to TRUE or FALSE.

Remarks

The **Assert** statement should be used by BSL clients to handle an application-specific error. An assertion error cannot be trapped by the **On Error** statement.

Use the **Assert** statement to ensure that a procedure is performing in the expected manner.

Note: This is a BSL extension. BSL offers a number of extensions that are not included in Microsoft® Visual Basic®.

Atn Function

Returns the angle (in radians) for the arc tangent of the specified number.

Syntax

Atn(*number*)

Where:

Is:

number

Any valid numeric expression.

Remarks

The **Atn** function assumes *number* is the ratio of two sides of a right triangle: the side opposite the angle to find and the side adjacent to the angle. The function returns a single-precision value for a ratio expressed as an integer, a currency, or a single-precision numeric expression. The return value is a double-precision value for a long, variant or double-precision numeric expression.

To convert radians to degrees, multiply by (180/PI). The value of PI is approximately 3.14159.

Example

This example finds the roof angle necessary for a house with an attic ceiling of 8 feet (at the roof peak) and a 16 foot span from the outside wall to the center of the house. The **Atn** function returns the angle in radians; it is multiplied by 180/PI to convert it to degrees.

```
Sub main
  Dim height, span, angle, PI
  PI=3.14159
  height=8
  span=16
  angle=Atn(height/span)*(180/PI)
  MsgBox "The angle is " & Format(angle, "##.##") & " degrees"
End Sub
```

See Also

Cos, Sin, Tan, Derived Trigonometric Functions

Beep Statement

Produces a tone through the computer speaker.

Syntax

Beep

Remarks

The frequency and duration of the tone depends on the hardware.

Example

This example beeps and displays a message in a box if the variable *balance* is less than 0. (If you have a set of speakers hooked up to your computer, you might need to turn them on to hear the beep.)

```
Sub main
  Dim expenses, balance, msgtext
  balance=InputBox("Enter your account balance")
  expenses=1000
  balance=balance-expenses
  If balance<0 then
    Beep
    MsgBox "I'm sorry, your account is overdrawn."
  Else
    MsgBox "Your balance minus expenses is: " &balance
  End If
End Sub
```

See Also

InputBox, MsgBox Statement, Print

Begin Dialog...End Dialog Statement

Begins and ends a dialog-box declaration.

Syntax

Begin Dialog *dialogName* [*x* , *y* ,] *dx* , *dy* [, *caption\$*] [, *.dialogfunction*]

' dialog box definition statements

End Dialog

Where:

Is:

<i>dialogName</i>	The record name for the dialog box definition.
<i>x</i> , <i>y</i>	The coordinates for the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the dialog box (relative to <i>x</i> and <i>y</i>).
<i>caption\$</i>	The title for the dialog box.
<i>.dialogfunction</i>	A Basic function to process user actions in the dialog box.

Remarks

To display the dialog box, you create a dialog record variable with the **Dim** statement, and then display the dialog box using the **Dialog function** or **Dialog statement** with the variable name as its argument. In the **Dim** statement, this variable is defined **As** *dialogName*.

The *x* and *y* coordinates are 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. For example, 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* argument is measured in 1/4 system-font character-width units. The *dy* argument is measured in 1/8 system-font character-width units. For example, to create a dialog box 80 characters wide, and 15 characters in height, enter 320, 120 for the *dx* , *dy* coordinates. If the *caption\$* argument is omitted, a standard default caption is used.

The optional *.dialogfunction* function must be defined (using the **Function** statement) or declared (using **Dim**) before being used in the **Begin Dialog** statement. Define the *dialogfunction* with the following three arguments:

Function *dialogfunction%* (*id\$* , *action%* , *suppvalue&*)
' function body

End Function

Where:

Is:

<i>id\$</i>	The text string that identifies the dialog control that triggered the call to the dialog function (usually because the user changed this control).
<i>action%</i>	An integer from 1 to 5 identifying the reason why the dialog function was called.
<i>suppvalue&</i>	Gives more specific information about why the dialog function was called.

As with any Basic function, these arguments can have different names. The arguments of the dialog function can also be variants.

In most cases, the return value of *dialogfunction* is ignored. The exceptions are a return value of 2 or 5 for *action%*. If the user clicks the OK button, Cancel button, or a command button (as indicated by an *action%* return value of 2 and the corresponding *id\$* for the button clicked), and the dialog function returns a non-zero value, the dialog box will *not* be closed.

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 definition statements 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.

Id\$ is the same value for the dialog control that you use in the definition of that control. For example, the *id\$* value for a text box is `Text1` if it is defined this way:

```
Textbox 271 , 78, 33, 18, .Text1
```

The following table summarizes the possible *action%* values and their meanings:

action%	Meaning
1	Dialog box initialization. This value is passed before the dialog box becomes visible.
2	Command button selected or dialog box control changed (except typing in a text box or combo box).
3	Change in a text box or combo box. This value is passed when the control loses the input focus: the user presses the TAB key or clicks another control.
4	Change of control focus. <i>Id\$</i> is the id of the dialog control gaining focus. <i>Suppvalue&</i> contains the numeric id of the control losing focus. A dialog function cannot display a message box or dialog box in response to an action value 4.
5	An idle state. As soon as the dialog box is initialized (<i>action%</i> = 1), the dialog function will be continuously called with <i>action%</i> = 5 if no other action occurs. If <i>dialog function</i> wants to receive this message continuously while the dialog box is idle, return a non-zero value. If 0 (zero) is returned, <i>action%</i> = 5 will be passed only while the user is moving the mouse. For this action, <i>Id\$</i> is equal to empty string ("") and <i>suppvalue&</i> is equal to the number of times action 5 was passed before.

If the user clicks a command button or changes a dialog box control, *action%* returns 2 or 3 and *suppvalue&* identifies the control affected. The value returned depends on the type of control or button the user changed or clicked.

The following table summarizes the possible values for *suppvalue&*:

Control	suppvalue&
List box	Number of the item selected, 0-based.
Check box	1 if selected, 0 if cleared, -1 if filled with gray.
Option button	Number of the option button in the option group, 0-based.
Text box	Number of characters in the text box.
Combo box	The number of the item selected (0-based) for action 2, the number of characters in its text box for action 3.
OK button	1
Cancel button	2

Example

This example defines and displays a dialog box with each type of item in it: list box, combo box, buttons, etc.

```
Sub main
    Dim ComboBox1() as String
    Dim ListBox1() as String
    Dim DropListBox1() as String
    ReDim ListBox1(0)
    ReDim ComboBox1(0)
    ReDim DropListBox1(3)
    ListBox1(0)="C:\"
    ComboBox1(0)=Dir("C:\*.*.*)
    For x=0 to 2
        DropListBox1(x)=Chr(65+x) & ":"
    Next x
    Begin Dialog UserDialog 274, 171, "BSL Dialog Box"
        ButtonGroup .ButtonGroup1
```

```
Text 9, 3, 69, 13, "Filename:", .Text1
DropComboBox 9, 14, 81, 119, ComboBox1(), .ComboBox1
Text 106, 2, 34, 9, "Directory:", .Text2
ListBox 106, 12, 83, 39, ListBox1(), .ListBox2
Text 106, 52, 42, 8, "Drive:", .Text3
DropListBox 106, 64, 95, 44, DropListBox1(), .DropListBox1
CheckBox 9, 142, 62, 14, "List .TXT files", .CheckBox1
GroupBox 106, 111, 97, 57, "File Range"
OptionGroup .OptionGroup2
    OptionButton 117, 119, 46, 12, "All pages", .OptionButton3
    OptionButton 117, 135, 67, 8, "Range of pages", .OptionButton4
Text 123, 146, 20, 10, "From:", .Text6
Text 161, 146, 14, 9, "To:", .Text7
TextBox 177, 146, 13, 12, .TextBox4
TextBox 145, 146, 12, 11, .TextBox5
OKButton 213, 6, 54, 14
CancelButton 214, 26, 54, 14
PushButton 213, 52, 54, 14, "Help", .Push1
End Dialog
Dim mydialog as UserDialog
On Error Resume Next
Dialog mydialog
If Err=102 then
    MsgBox "Dialog box canceled."
End If
End Sub
```

See Also

Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, Dialog, DropComboBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Bitmap Viewer

Displays a series of bitmaps in a dialog box

Example

```

Declare Sub GetWindowsDirectory Lib "kernel" (ByVal buf$, ByVal buflen%)
Dim fname$, WinDir$
Const IdleLoop = 5

'Dialog Box Function. Find and display next bitmap.
Function DlgFunc% (id$, action%, svalue&)
  If action = IdleLoop And (svalue Mod 800 = 799) Then
    fname = dir$
    If fname = "" Then
      SendKeys "{enter}"
      Exit Function
    End If
    ' load next picture
    DlgSetPicture "p1", WinDir & fname, 0
    DlgText DlgControlID("FileName"), fname
  End If
  If action = IdleLoop Then DlgFunc = 1
End Function

Sub Main
  Dim WinDirBuf as String * 150
  'Find Windows bitmap files
  Call GetWindowsDirectory (WinDirBuf, Len(WinDirBuf) )
  WinDir = Left(WinDirBuf, InStr(WinDirBuf, Chr$(0))-1) & "\"
  fname = Dir$(WinDir & "*.bmp")
  If (fname = "") Then Exit Sub
  Begin Dialog PictureBoxType 25, 25, 210, 240, "Picture" , .DlgFunc
    Picture      5, 5, 200, 200, WinDir & fname, 0, .p1
    Text         15, 225, 70, 15, fname, .FileName
    PushButton 145, 220, 45, 15, "Stop"
  End Dialog
  Dim PictureBox as PictureBoxType
  Dialog PictureBox
End Sub

```

Button Statement

Defines a custom push button.

Syntax A

Button *x*, *y*, *dx*, *dy*, *text\$* [, *.id*]

Syntax B

PushButton *x*, *y*, *dx*, *dy*, *text\$* [, *.id*]

Where:	Is:
<i>x</i> , <i>y</i>	The position of the button relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the button.
<i>text\$</i>	The name for the push button. If the width of this string is greater than <i>dx</i> , trailing characters are truncated.
<i>.id</i>	An optional identifier used by the dialog statements that act on this control.

Remarks

A *dy* value of 14 typically accommodates text in the system font. Use this statement to create buttons other than OK and Cancel. Use this statement in conjunction with the **ButtonGroup** statement. The two forms of the statement (**Button** and **PushButton**) are equivalent. Use the **Button** statement only between **Begin Dialog** and **End Dialog**.

Example

This example defines a dialog box with a combination list box and three buttons.

```
Sub main
  Dim fchoices as String
  fchoices="File1" & Chr(9) & "File2" & Chr(9) & "File3"
  Begin Dialog UserDialog 185, 94, "BSL Dialog Box"
    Text 9, 5, 69, 10, "Filename:", .Text1
    DropComboBox 9, 17, 88, 71, fchoices, .ComboBox1
    ButtonGroup .ButtonGroup1
    OKButton 113, 14, 54, 13
    CancelButton 113, 33, 54, 13
    Button 113, 57, 54, 13, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **DropComboBox**, **DropListBox**, **GroupBox**, **ListBox**, **OKButton**, **OptionButton**, **OptionGroup**, **Picture**, **StaticComboBox**, **Text**, **TextBox**

ButtonGroup Statement

Begins the definition of a group of custom buttons for a dialog box.

Syntax

ButtonGroup *.field*

Where: *.field*
Is: The field to contain the user's custom button selection.

Remarks

If **ButtonGroup** is used, it must appear before any **PushButton** (or **Button**) statement that creates a custom button (one other than OK or Cancel). Only one **ButtonGroup** statement is allowed within a dialog box definition.

Use the **ButtonGroup** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a group of three buttons.

```
Sub main
  Begin Dialog UserDialog 34,0,231,140, "BSL Dialog Box"
    ButtonGroup .bg
    PushButton 71,17,88,17, "&Button 0"
    PushButton 71,50,88,17, "&Button 1"
    PushButton 71,83,88,17, "&Button 2"
  End Dialog
  Dim mydialog as UserDialog
  Dialog mydialog
  MsgBox "Button " & mydialog.bg & " was pressed."
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, CancelButton, Caption, CheckBox, ComboBox, DropComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Call Statement

Transfers control to a subprogram or function.

Syntax A

Call *subprogram-name* [(*argumentlist*)]

Syntax B

subprogram-name *argumentlist*

Where:

Is:

subprogram-name The name of the subroutine or function to call.

argumentlist The arguments for the subroutine or function (if any).

Remarks

Use the **Call** statement to call a subprogram or function 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.

If a procedure accepts named arguments, you can use the names to specify the argument and its value. Order is not important. For example, if a procedure is defined as follows:

```
Sub mysub(aa, bb, optional cc, optional dd)
```

the following calls to this procedure are all equivalent:

```
call mysub(1, 2, , 4)
mysub aa := 1, bb := 2, dd :=4
call mysub(aa := 1, dd:=4, bb := 2)
mysub 1, 2, dd:=4
```

Note that the syntax for named arguments is as follows:

```
argname := argvalue
```

where *argname* is the name for the argument as supplied in the **Sub** or **Function** statement and *argvalue* is the value to assign to the argument when you call it. The advantage to using named arguments is that you do not have to remember the order specified in the procedure's original definition, and if the procedure takes optional arguments, you do not need to include commas (,) for arguments that you leave out.

The procedures that use named arguments include:

- All functions defined with the **Function** statement.
- All subprograms defined with the **Sub** statement.
- All procedures declared with **Declare** statement.
- Many built-in functions and statements (such as **InputBox**).
- Some externally registered DLL functions and methods.

Arguments are passed by reference to procedures written in Basic. If you pass a variable to a procedure that modifies its corresponding formal parameter, and you do not want to have your variable modified, enclose the variable in parentheses in the **Call** statement. This will tell BSL 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 that 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.)

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 data type specified in the declaration was **Any**.

Example

This example calls a subprogram named CREATEFILE to open a file, write the numbers 1 to 10 in it and leave it open. The calling procedure then checks the file's mode. If the mode is 1 (open for Input) or 2 (open for Output), the procedure closes the file.

```
Declare Sub createfile()
Sub main
  Dim filemode as Integer
  Dim attrib as Integer
  Call createfile
  attrib=1
  filemode=FileAttr(1,attrib)
  If filemode=1 or 2 then
    MsgBox "File was left open. Closing now."
    Close #1
  End If
  Kill "C:\TEMP001"
End Sub

Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
End Sub
```

See Also

Declare

CancelButton Statement

Sets the position and size of a Cancel button in a dialog box.

Syntax

CancelButton *x*, *y*, *dx*, *dy* [, *.id*]

Where:

Is:

<i>x</i> , <i>y</i>	The position of the Cancel button relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the button.
<i>.id</i>	An optional identifier for the button.

Remarks

A *dy* value of 14 can usually accommodate text in the system font. *.id* is used by the dialog statements that act on this control. If you use the **Dialog** statement to display the dialog box and the user clicks Cancel, the box is removed from the screen and an Error 102 is triggered. If you use the **Dialog** function to display the dialog box, the function will return 0 and no error occurs. Use the **CancelButton** statement only between a **Begin Dialog...End Dialog** statement.

Example

This example defines a dialog box with a combination list box and three buttons.

```
Sub main
  Dim fchoices as String
  fchoices="File1" & Chr(9) & "File2" & Chr(9) & "File3"
  Begin Dialog UserDialog 185, 94, "BSL Dialog Box"
    Text 9, 5, 69, 10, "Filename:", .Text1
    DropComboBox 9, 17, 88, 71, fchoices, .ComboBox1
    ButtonGroup .ButtonGroup1
    OKButton 113, 14, 54, 13
    CancelButton 113, 33, 54, 13
    PushButton 113, 57, 54, 13, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, ButtonGroup, Caption, CheckBox, ComboBox, DropComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Caption Statement

Defines the title of a dialog box.

Syntax

Caption *text*\$

Where: **Is:**

text\$ A string expression containing the title of the dialog box.

Remarks

Use the **Caption** statement only between a **Begin Dialog** and an **End Dialog** statement. If no **Caption** statement is specified for the dialog box, a default caption is used. The default caption for this statement includes the acronym *BSL*. If you do not want *BSL* to appear, change the default caption.

Example

This example defines a dialog box with a combination list box and three buttons. The **Caption** statement changes the dialog box title to "Example -Caption Statement".

```
Sub main
  Dim fchoices as String
  fchoices="File1" & Chr(9) & "File2" & Chr(9) & "File3"
  Begin Dialog UserDialog 185, 94
    Caption "Example-Caption Statement"
    Text 9, 5, 69, 10, "Filename:", .Text1
    DropComboBox 9, 17, 88, 71, fchoices, .ComboBox1
    ButtonGroup .ButtonGroup1
    OKButton 113, 14, 54, 13
    CancelButton 113, 33, 54, 13
    PushButton 113, 57, 54, 13, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, CancelButton, ButtonGroup, CheckBox, ComboBox, DropComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

CCur Function

Converts an expression to the data type Currency.

Syntax

CCur(*expression*)

Where: **Is:**

expression Any expression that evaluates to a number.

Remarks

CCur accepts any type of *expression*. Numbers that do not fit in the Currency data type result in an “Overflow” error. Strings that cannot be converted result in a “Type Mismatch” error. Variants containing null result in an “Illegal Use of Null” error.

Example

This example converts a yearly payment on a loan to a currency value with four decimal places. A subsequent **Format** statement formats the value to two decimal places before displaying it in a message box.

```
Sub main
Dim aprate, totalpay, loanpv
  Dim loanfv, due, monthlypay
  Dim yearlypay, msgtext
  loanpv=InputBox("Enter the loan amount: ")
  aprate=InputBox("Enter the annual percentage rate: ")
  If aprate >1 then
    aprate=aprate/100
  End If
  aprate=aprate/12
  totalpay=InputBox("Enter the total number of pay periods: ")
  loanfv=0
Rem Assume payments are made at end of month
  due=0
  monthlypay=Pmt(aprate,totalpay,-loanpv,loanfv,due)
  yearlypay=CCur(monthlypay*12)
  msgtext= "The yearly payment is: " & Format(yearlypay, "Currency")
  MsgBox msgtext
End Sub
```

See Also

CDBl, CInt, CLng, CSng, CStr, CVar, CVDate

CDbl Function

Converts an expression to the data type Double.

Syntax

CDbl(*expression*)

Where: **Is:**

expression Any expression that evaluates to a number.

Remarks

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

Example

This example calculates the square root of 2 as a double-precision floating point value and displays it in scientific notation.

```
Sub main
  Dim value
  Dim msgtext
  value=CDbl(Sqr(2))
  msgtext= "The square root of 2 is: " & Value
  MsgBox msgtext
End Sub
```

See Also

CCur, CInt, CLng, CSng, CStr, CVar, CVDate

ChDir Statement

Changes the default directory for the specified drive.

Syntax

ChDir *path*\$

Where:	Is:
<i>path</i> \$	A string expression identifying the new default directory.

Remarks

The syntax for *path*\$ is:

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

If the drive argument is omitted, **ChDir** changes the default directory on the current drive. The **ChDir** statement does not change the default drive. To change the default drive, use **ChDrive**.

Example

This example changes the current directory to C:\WINDOWS, if it is not already the default.

```
Sub main
    Dim newdir as String
    newdir="c:\windows"
    If CurDir <> newdir then
        ChDir newdir
    End If
    MsgBox "The default directory is now: " & newdir
End Sub
```

See Also

ChDrive, **CurDir**, **Dir**, **MkDir**, **Rmdir**

ChDrive Statement

Changes the default drive.

Syntax

ChDrive *drive*\$

Where **is**

drive\$ A string expression designating the new default drive.

Remarks

This drive must exist and must be within the range specified by the LASTDRIVE statement in the CONFIG.SYS file. If a null argument (" ") is supplied, the default drive remains the same. If the *drive*\$ 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**.

Example

This example changes the default drive to A:.

```
Sub main
  Dim newdrive as String
  newdrive="A:"
  If Left(CurDir,2) <> newdrive then
    ChDrive newdrive
  End If
  MsgBox "The default drive is now " & newdrive
End Sub
```

See Also

ChDir, **CurDir**, **Dir**, **MkDir**, **Rmdir**

CheckBox Statement

Creates a check box in a dialog box.

Syntax

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

Where...	is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the check box, relative to the upper left corner of the dialog box.
<i>dx</i>	The sum of the widths of the check box and <i>text\$</i> .
<i>dy</i>	The height of <i>text\$</i> .
<i>text\$</i>	The title shown to the right of the check box.
<i>.field</i>	The name of the dialog-record field that will hold the current check box setting. (0=unchecked, -1=grey, 1=checked).

Remarks

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 the “Begin Dialog...End Dialog Statement” on page 34 for more information.)

Because proportional spacing is used, the *dx* argument 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 check box.

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 check box and the accompanying text will move down within the dialog box.

If the width of the *text\$* field is greater than *dx*, trailing characters will be truncated. If you want to include underlined characters so that the check box selection can be made from the keyboard, precede the character to be underlined with an ampersand (&).

BSL treats any other value of *.field* the same as a 1. The *.field* argument is also used by the dialog statements that act on this control.

Use the **CheckBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a combination list box, a check box, and three buttons.

```
Sub main
  Dim ComboBox1() as String
  ReDim ComboBox1(0)
  ComboBox1(0)=Dir("C:\*.*.")
  Begin Dialog UserDialog 166, 76, "BSL Dialog Box"
    Text 9, 3, 69, 13, "Filename:", .Text1
    DropComboBox 9, 14, 81, 119, ComboBox1(), .ComboBox1
    CheckBox 10, 39, 62, 14, "List .TXT files", .CheckBox1
    OKButton 101, 6, 54, 14
    CancelButton 101, 26, 54, 14
    PushButton 101, 52, 54, 14, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, ButtonGroup, CancelButton, Caption, ComboBox, DropComboBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Chr Function

Returns the one-character string corresponding to a character code.

Syntax

Chr[\$](*charcode*)

Where...	Is...
----------	-------

<i>charcode</i>	An integer representing the character to be returned.
-----------------	---

Remarks

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).

To obtain a byte representing a given character, use **ChrB**.

Example

This example displays the character equivalent for an ASCII code between 65 and 122 typed by the user.

```
Sub main
  Dim numb as Integer
  Dim msgtext
  Dim out
  out=0
  Do Until out
    numb=InputBox("Type a number between 65 and 122:")
    If Chr$(numb)>="A" AND Chr$(numb)<="Z" OR Chr$(numb)>="a" AND _
      Chr$(numb)<="z" then
      msgtext="The letter for the number " & numb &" is: " & Chr$(numb)
      out=1
    ElseIf numb=0 then
      Exit Sub
    Else
      Beep
      msgtext="Does not convert to a character; try again."
    End If
    MsgBox msgtext
  Loop
End Sub
```

See Also

Asc, **CCur**, **CDbl**, **CLng**, **CSng**, **CStr**, **CVar**, **CVDate**, **Format**, **Val**

CInt Function

Converts an expression to the data type Integer by rounding.

Syntax

CInt(*expression*)

Where...	Is...
----------	-------

<i>expression</i>	Any expression that can evaluate to a number.
-------------------	---

Remarks

After rounding, the resulting number must be within the range of -32767 to 32767, or an error occurs.

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

Example

This example calculates the average of ten golf scores.

```
Sub main
    Dim score As Integer
    Dim x, sum
    Dim msgtext
    Let sum=0
    For x=1 to 10
        score=InputBox("Enter golf score #"&x &":")
        sum=sum+score
    Next x
    msgtext="Your average is: " & Format(CInt(sum/(x-1)),"General Number")
    MsgBox msgtext
End Sub
```

See Also

CCur, CDbl, CLng, CSng, CStr, CVar, CDate

Clipboard

The Windows Clipboard can be accessed directly in your program to enable you to get text from and put text into other applications that support the Clipboard.

Syntax

Clipboard.Clear

Clipboard.GetText()

Clipboard.SetText *string*\$

Clipboard.GetFormat()

Where...	Is...
<i>string</i> \$	A string or string expression containing the text to send to the Clipboard.

The Clipboard methods supported are:

Method	What it does
Clear	Clears the contents of the Clipboard.
GetText	Returns a text string from the Clipboard.
SetText	Puts a text string to the Clipboard.
GetFormat	Returns TRUE (non-0) if the format of the item on the Clipboard is text. Otherwise, returns FALSE (0).

Note: Data on the Clipboard is lost when another set of data of the same format is placed on the Clipboard (either through code or a menu command).

Example

This example places the text string "Hello, world." on the Clipboard.

```
Sub main
    Dim mytext as String
    mytext="Hello, world."
    Clipboard.Settext mytext
    MsgBox "The text: '" & mytext & "' added to the Clipboard."
End Sub
```

CLng Function

Converts an expression to the data type Long by rounding.

Syntax

CLng(*expression*)

Where...	Is...
----------	-------

<i>expression</i>	Any expression that can evaluate to a number.
-------------------	---

Remarks

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

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

Example

This example divides the United States debt by the number of people in the country or region to find the amount of money each person would have to pay to wipe it out. This figure is converted to a Long integer and formatted as Currency.

```
Sub main
    Dim debt As Single
    Dim msgtext
    Const Populace = 250000000
    debt=InputBox("Enter the current US debt:")
    msgtext="The $/citizen is: " & Format(CLng(Debt/Populace), "Currency")
    MsgBox msgtext
End Sub
```

See Also

CCur, CDbl, CInt, CSng, CStr, CVar, CDate

Close Statement

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

Syntax

Close [[#] *filename%* [, [#] *filename%* ...]]

Where...	Is...
----------	-------

<i>filename%</i>	An integer expression identifying the file to close.
------------------	--

Remarks

Filename% is the number assigned to the file in the **Open** statement and can be preceded by a pound sign (#). If this argument is omitted, all open files are closed. Once a **Close** statement is executed, the association of a file with *filename%* is ended, and the file can be reopened with the same or a 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.

Example

This example opens a file for Random access, gets the contents of one variable, and closes the file again. The subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram.

```
Declare Sub createfile()
Sub main
  Dim acctno as String*3
  Dim recno as Long
  Dim msgtext as String
  Call createfile
  recno=1
  newline=Chr(10)
  Open "C:\TEMP001" For Random As #1 Len=3
  msgtext="The account numbers are:" & newline & newline
  Do Until recno=11
    Get #1,recno,acctno
    msgtext=msgtext & acctno
    recno=recno+1
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub
```

```
Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also

Open, Reset, Stop

ComboBox Statement

Creates a combination text box and list box in a dialog box.

Syntax A**ComboBox** *x* , *y* , *dx* , *dy* , *text\$* , *.field***Syntax B****ComboBox** *x* , *y* , *dx* , *dy* , *stringarray\$* , *.field*

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the list box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the combo box in which the user enters or selects text.
<i>text\$</i>	A string containing the selections for the combo box.
<i>stringarray\$</i>	An array of dynamic strings for the selections in the combo box.
<i>.field</i>	The name of the dialog-record field that will hold the text string entered in the text box or chosen from the list box.

Remarks

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 the “Begin Dialog...End Dialog Statement” on page 34 for more information.)

The *text\$* argument must be defined, using a **Dim** Statement, before the **Begin Dialog** statement is executed. The arguments in the *text\$* string are tab delimited as shown in the following example:

```
dimname = "listchoice"+Chr$(9)+"listchoice"+Chr$(9)+"listchoice"...
```

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. The *field* argument is also used by the dialog statements that act on this control.

Use the **ComboBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a combination list and text box and three buttons.

```
Sub main
  Dim ComboBox1() as String
  ReDim ComboBox1(0)
  ComboBox1(0)=Dir("C:\*.*.")

  Begin Dialog UserDialog 166, 142, "BSL Dialog Box"
    Text 9, 3, 69, 13, "Filename:", .Text1
    ComboBox 9, 14, 81, 119, ComboBox1(), .ComboBox1
    OKButton 101, 6, 54, 14
    CancelButton 101, 26, 54, 14
    PushButton 101, 52, 54, 14, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, ButtonGroup, CancelButton, Caption, CheckBox, DropComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Command Function

Returns the command line specified when the MAIN subprogram was invoked.

Syntax

Command[\$]

Remarks

After the MAIN subprogram returns, further calls to the **Command** function yield an empty string. This function might not be supported in some implementations of BSL.

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

Example

This example opens the file entered by the user on the command line.

```
Sub main
  Dim filename as String
  Dim cmdline as String
  Dim cmdlength as Integer
  Dim position as Integer
  cmdline=Command
  If cmdline="" then
    MsgBox "No command line information."
    Exit Sub
  End If
  cmdlength=Len(cmdline)
  position=InStr(cmdline,Chr(32))
  filename=Mid(cmdline,position+1,cmdlength-position)
  On Error Resume Next
  Open filename for Input as #1
  If Err<>0 then
    MsgBox "Error loading file."
    Exit Sub
  End If
  MsgBox "File " & filename & " opened."
  Close #1
  MsgBox "File " & filename & " closed."
End Sub
```

See Also

AppActivate, DoEvents, Environ, SendKeys, Shell

Const Statement

Declares symbolic constants for use in a Basic program.

Syntax

[Global] Const *constantName* [As *type*]= *expression* [,*constantName* [As *type*]= *expression*] ...

Where...	Is...
<i>constantName</i>	The variable name to contain a constant value.
<i>type</i>	The data type of the constant (Number or String).
<i>expression</i>	Any expression that evaluates to a constant number.

Remarks

Instead of using the As clause, the type of the constant can be specified by using a type character as a suffix (# for numbers, \$ for strings) to the *constantName*. If no type character is specified, the type of the *constantName* is derived from the type of the expression.

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

Example

This example divides the United States debt by the number of people in the country or region to find the amount of money each person would have to pay to wipe it out. This figure is converted to a Long integer and formatted as Currency.

```
Sub main
    Dim debt As Single
    Dim msgtext
    Const Populace=250000000
    debt=InputBox("Enter the current US debt:")
    msgtext="The $/citizen is: " & Format(CLng(Debt/Populace), "Currency")
    MsgBox msgtext
End Sub
```

See Also

Declare, Deftype, Dim, Let, Type

Cos Function

Returns the cosine of an angle.

Syntax

Cos(*number*)

Where...	Is...
<i>number</i>	An angle in radians.

Remarks

The return value will be between -1 and 1. The return value is a single-precision number if the angle has a data type Integer, Currency, or is a single-precision value. The return value will be a double precision value if the angle has a data type Long, variant or is a double-precision value.

The angle can be either positive or negative. To convert degrees to radians, multiply by (PI/180). The value of PI is approximately 3.14159.

Example

This example finds the length of a roof, given its pitch and the distance of the house from its center to the outside wall.

```
Sub main
    Dim bwidth, roof, pitch
    Dim msgtext
    Const PI=3.14159
    Const conversion=PI/180
    pitch=InputBox("Enter roof pitch in degrees")
    pitch=Cos(pitch*conversion)
    bwidth=InputBox("Enter 1/2 of house width in feet")
    roof=bwidth/pitch
    msgtext="The length of the roof is " & Format(roof, "##.##") & " feet."
    MsgBox msgtext
End Sub
```

See Also

Atn, Sin, Tan, Derived Trigonometric Functions

CreateObject Function

Creates a new OLE2 automation object.

Syntax

CreateObject(*class*)

Where...	Is...
<i>class</i>	The name of the application, a period, and the name of the object to be used.

Remarks

To create an object, you first must declare an object variable, using **Dim**, and then **Set** the variable equal to the new object, as follows:

```
Dim OLE2 As Object
Set OLE2 = CreateObject("spoly.cpoly")
```

To refer to a method or property of the newly created object, use the syntax *objectvar.property* or *objectvar.method*, as follows:

```
OLE2.reset
```

Refer to the documentation provided with your OLE2 automation server application for correct application and object names.

Example

This example uses CreateObject to open Microsoft Visio (if it is not already open).

```
Sub main
    Dim visio as Object
    Dim doc as Object
    Dim i as Integer, doccount as Integer
    'Initialize Visio
    on error resume next
    Set visio = GetObject(,"visio.application")
    find Visio
    If (visio Is Nothing) then
        Set visio = CreateObject("visio.application") ' find Visio
        If (visio Is Nothing) then
            MsgBox "Couldn't find Visio!"
            Exit Sub
        End If
    End If
    MsgBox "Visio is open."
End Sub
```

See Also

GetObject, **Is**, **Me**, **New**, **Nothing**, **Object Class**, **Typeof**

CSng Function

Converts an expression to the data type Single.

Syntax

CSng(*expression*)

Where...	Is...
<i>expression</i>	Any expression that can evaluate to a number.

Remarks

The *expression* must have a value within the range allowed for the Single data type, or an error occurs.

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

Example

This example calculates the factorial of a number. A factorial (notated with an exclamation mark, !) is the product of a number and each integer between it and the number 1. For example, 5 factorial, or 5!, is the product of $5*4*3*2*1$, or the value 120.

```
Sub main
    Dim number as Integer
    Dim factorial as Double
    Dim msgtext
    number=InputBox("Enter an integer between 1 and 170:")
    If number<=0 then
        Exit Sub
    End If
    factorial=1
    For x=number to 2 step -1
        factorial=factorial*x
    Next x
    Rem If number =<35, then its factorial is small enough to be stored
    Rem as a single-precision number
    If number<35 then
        factorial=CSng(factorial)
    End If
    msgtext="The factorial of " & number & " is: " & factorial
    MsgBox msgtext
End Sub
```

See Also

CCur, **CDbl**, **CLnt**, **CLng**, **CStr**, **CVar**, **CVDate**

CStr Function

Converts an expression to the data type String.

Syntax

CStr(*expression*)

Where...	Is...
<i>expression</i>	Any expression that can evaluate to a number.

Remarks

The **CStr** statement accepts any type of *expression*:

<i>expression</i> is...	CStr returns...
Boolean	A String containing "True" or "False".
Date	A String containing a date.
Empty	A zero-length String ("").
Error	A String containing "Error", followed by the error number.
Null	A run-time error.
Other Numeric	A String containing the number.

Example

This example converts a variable from a value to a string and displays the result. Variant type 5 is Double and type 8 is String.

```
Sub main
    Dim var1
    Dim msgtext as String
    var1=InputBox("Enter a number:")
    var1=var1+10
    msgtext="Your number + 10 is: " & var1 & Chr(10)
    msgtext=msgtext & "which makes its Variant type: " & Vartype(var1)
    MsgBox msgtext
    var1=CStr(var1)
    msgtext="After conversion to a string," & Chr(10)
    msgtext=msgtext & "the Variant type is: " & Vartype(var1)
    MsgBox msgtext
End Sub
```

See Also

Asc, CCur, CDBl, Chr, CInt, CLng, CSng, CVar, CVDate, Format

\$CStrings Metacommand

Tells the compiler to treat a backslash character inside a string (\) as an escape character.

Syntax

\$CStrings [Save | Restore]

Where...	Means...
Save	Saves the current \$CStrings setting.
Restore	Restores a previously saved \$CStrings setting.

Remarks

This treatment of a backslash in a string is based on the 'C' language.

Save and Restore operate as a stack and allow the user to change the setting for a range of the program without impacting the rest of the program.

The supported special characters are:

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

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	<code>\ddd</code>
Hexadecimal Code	<code>\xdd</code>

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.

Note: BSL offers a number of extensions that are not included in Visual Basic.

Example

This example displays two lines, the first time using the C-language characters "\n" for a carriage return and line feed.

```
Sub main
  '$CStrings
  MsgBox "This is line 1\n This is line 2 (using C Strings)"
  '$NoCStrings
  MsgBox "This is line 1" +Chr$(13)+Chr$(10)+"This is line 2 (using Chr)"
End Sub
```

See Also

\$Include, \$NoCStrings, Rem

CurDir Function

Returns the default directory (and drive) for the specified drive.

Syntax

CurDir[\$] [(*drive*\$)]

Where...	Is...
<i>drive</i> \$	A string expression containing the drive to search.

Remarks

The drive must exist, and must be within the range specified in the LASTDRIVE statement of the CONFIG.SYS file. If a null argument (" ") is supplied, or if no *drive*\$ is indicated, the path for the default drive is returned.

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).

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

Example

This example changes the current directory to C:\WINDOWS, if it is not already the default.

```
Sub main
    Dim newdir as String
    newdir="c:\windows"
    If CurDir <> newdir then
        ChDir newdir
    End If
    MsgBox "The default directory is now: " & newdir
End Sub
```

See Also

ChDir, **ChDrive**, **Dir**, **MkDir**, **Rmdir**

CVar Function

Converts an expression to the data type variant.

Syntax

CVar(*expression*)

Where...	Is...
----------	-------

<i>expression</i>	Any expression that can evaluate to a number.
-------------------	---

Remarks

CVar accepts any type of *expression*.

CVar generates the same result as you would get by assigning the *expression* to a variant variable.

Example

This example converts a string variable to a variant variable.

```
Sub main
    Dim answer as Single
    answer=100.5
    MsgBox "'Answer' is DIM'ed as Single with the value: " & answer
    answer=CVar(answer)
    answer=Fix(answer)
    MsgBox "'Answer' is now a variant with a type of: " & VarType(answer)
End Sub
```

See Also

CCur, **CDbl**, **CLng**, **CSng**, **CStr**, **CVDate**

CVDate Function

Converts an expression to the data type Variant Date.

Syntax

CVDate(*expression*)

Where...	Is...
<i>expression</i>	Any expression that can evaluate to a number.

Remarks

CVDate 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 zero represents December 30, 1899. Times are represented as fractional days.

Example

This example displays the date for one week from the date entered by the user.

```
Sub main
Dim str1 as String
Dim nextweek
Dim msgtext
i: str1=InputBox$("Enter a date:")
answer=IsDate(str1)
If answer=-1 then
    str1=CVDate(str1)
    nextweek=DateValue(str1)+7
    msgtext="One week from the date entered is:
    msgtext=msgtext & "Format(nextweek,"dddddd")
    MsgBox msgtext
Else
    MsgBox "Invalid date or format. Try again."
    Goto i
End If
End Sub
```

See Also

Asc, CCur, CDbl, Chr, CInt, CLng, CSng, CStr, CVar, Format, Val

Date Function

Returns a string representing the current date.

Syntax

Date[\$]

Remarks

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

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).

Example

This example displays the date for one week from the today's date (the current date on the computer).

```
Sub main
    Dim nextweek
    nextweek=CVar(Date)+7
    MsgBox "One week from today is: " & Format(nextweek,"dddd")
End Sub
```

See Also

CVDate, Date Statement, Format, Now, Time Function, Time Statement, Timer, TimeSerial

Date Statement

Sets the system date.

Syntax

Date[\$] = *expression*

Where...	Is...
<i>expression</i>	A string in 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).

Remarks

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 the 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.

Example

This example changes the system date to a date entered by the user.

```
Sub main
  Dim userdate
  Dim answer
i: userdate=InputBox("Enter a date for the system clock:")
  If userdate="" then
    Exit Sub
  End If
  answer=IsDate(userdate)
  If answer=-1 then
    Date=userdate
  Else
    MsgBox "Invalid date or format. Try again."
    Goto i
  End If
End Sub
```

See Also

[Date Function](#), [Time Function](#), [Time Statement](#)

DateSerial Function

Returns a date value for year, month, and day specified.

Syntax

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

Where...	Is...
<i>year%</i>	A year between 100 and 9999, or a numeric expression.
<i>month%</i>	A month between 1 and 12, or a numeric expression.
<i>day%</i>	A day between 1 and 31, or a numeric expression.

Remarks

The **DateSerial** function returns a variant of vartype 7 (date) that represents a date from January 1, 100 through December 31, 9999. A value of zero represents December 30, 1899. Times are represented as fractional days.

A numeric expression can be used for any of the arguments to specify a relative date: a number of days, months, or years before or after a certain date.

Example

This example finds the day of the week New Year's day will be for the year 2000.

```
Sub main
  Dim newyearsday
  Dim daynumber
  Dim msgtext
  Dim newday as Variant
  Const newyear=2000
  Const newmonth=1
  Let newday=1
  newyearsday=DateSerial(newyear,newmonth, newday)
  daynumber=Weekday(newyearsday)
  msgtext="New Year's day 2000 falls on a " & Format(daynumber, "dddd")
  MsgBox msgtext
End Sub
```

See Also

DateValue, Day, Month, Now, TimeSerial, TimeValue, Weekday, Year

DateValue Function

Returns a date value for the string specified.

Syntax

DateValue(*date\$*)

Where...	Is...
<i>date\$</i>	A string representing a valid date.

Remarks

The **DateValue** function returns a variant of vartype 7 (date) that represents a date from January 1, 100 through December 31, 9999. A value of zero represents December 30, 1899. Times are represented as fractional days.

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

Example

This example displays the date for one week from the date entered by the user.

```
Sub main
    Dim str1 as String
    Dim nextweek
    Dim msgtext
i: str1=InputBox$("Enter a date:")
    answer=IsDate(str1)
    If answer=-1 then
        str1=CVDate(str1)
        nextweek=DateValue(str1)+7
        msgtext="One week from your date is: " & Format(nextweek,"dddddd")
        MsgBox msgtext
    Else
        MsgBox "Invalid date or format. Try again."
        Goto i
    End If
End Sub
```

See Also

DateSerial, Day, Month, Now, TimeSerial, TimeValue, Weekday, Year

Day Function

Returns the day of the month (1-31) of a date-time value.

Syntax

Day(*date*)

Where...

date

Is...

Any expression that can evaluate to a date.

Remarks

Day attempts to convert the input value of *date* to a date value. The return value is a variant of vartype 2 (integer). If the value of *date* is null, a Variant of vartype 1 (null) is returned.

Example

This example finds the month (1-12) and day (1-31) values for this Thursday.

```
Sub main
    Dim x, today, msgtext
    Today=DateValue(Now)
    Let x=0
    Do While Weekday(Today+x) <> 5
        x=x+1
    Loop
    msgtext="This Thursday is: " & Month(Today+x) & "/" & Day(Today+x)
    MsgBox msgtext
End Sub
```

See Also

[Date Function](#), [Date Statement](#), [Hour](#), [Minute](#), [Month](#), [Now](#), [Second](#), [Weekday](#), [Year](#)

DDEAppReturnCode Function

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

Syntax

DDEAppReturnCode()

Remarks

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**.

See Also

[DDEExecute](#), [DDEInitiate](#), [DDEPoke](#), [DDERequest\(\)](#), [DDETerminate](#)

DDEExecute Statement

Sends one or more commands to an application via a dynamic-data exchange (DDE) channel.

Syntax

DDEExecute *channel%*, *cmd\$*

Where...	Is...
<i>channel%</i>	An integer or expression for the channel number of the DDE conversation as returned by DDEInitiate .
<i>cmd\$</i>	One or more commands recognized by the application.

Remarks

If *channel* does not correspond to an open channel, an error occurs.

You can also use the format described under **SendKeys** to send specific key sequences. If the server application cannot perform the specified command, an error occurs.

In many applications that support DDE, *cmd\$* can be one or more statements or functions in the application's macro language. Note that some applications require that each command received through a DDE channel be enclosed in brackets and quotation marks.

You can use a single **DDEExecute** instruction to send more than one command to an application.

Many commands require arguments in the form of strings enclosed in quotation marks. Because quotation marks indicate the beginning and end of a string in BSL, you must use **Chr\$(34)** to include a quotation mark in a command string. For example, the following instruction tells Microsoft® Excel® to open MYFILE.XLS:

```
DDEExecute channelno, "[OPEN(" + Chr$(34) + "MYFILE.XLS" + Chr$(34) + ")]"
```

Example

This example opens Microsoft Word, uses **DDEPoke** to write the text "Hello, world" to the open document (Untitled) and uses **DDEExecute** to save the text to the file TEMPO01.

```
Sub main
    Dim channel as Integer
    Dim appname as String
    Dim topic as String
    Dim testtext as String
    Dim item as String
    Dim pcommand as String
    Dim msgtext as String
    Dim x as Integer
    Dim path as String
    appname="WinWord"
    path="c:\msoffice\winword\"
    topic="Document1"
    item="Page1"
    testtext="Hello, world."
    On Error Goto Errhandler
    x=Shell(path & appname & ".EXE")
    channel = DDEInitiate(appname, topic)
    If channel=0 then
        MsgBox "Unable to open Write."
        Exit Sub
    End If
    DDEPoke channel, item, testtext
```

```
pcommand="[FileSaveAs .Name = " & Chr$(34) & "C:\TEMP001" & Chr$(34) & "]"
DDEExecute channel, pcommand
pcommand="[FileClose]"
DDEExecute channel, pcommand
msgtext="The text: " & testtext & " saved to C:\TEMP001." & Chr$(13)
msgtext=msgtext & Chr$(13) & "Delete? (Y/N)"
answer=InputBox(msgtext)
If answer="Y" or answer="y" then
    Kill "C:\TEMP001.doc"
End If
DDETerminate channel
Exit Sub
Errhandler:
If Err<>0 then
    MsgBox "DDE Access failed."
End If
End Sub
```

See Also

DDEAppReturnCode(), **DDEInitiate()**, **DDEPoke**, **DDERequest()**, **DDETerminate**

DDEInitiate Function

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

Syntax

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

Where...

Is...

<i>appname\$</i>	A string or expression for the name of the DDE application to talk to.
<i>topic\$</i>	A string or expression for the name of a topic recognized by <i>appname\$</i> .

Remarks

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\$* does not 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 are not using an open channel, you should conserve resources by closing it using **DDETerminate**.

Example

This example uses **DDEInitiate** to open a channel to Microsoft Word. It uses **DDERequest** to obtain a list of available topics (using the System topic).

```
Sub main
    Dim channel as Integer
    Dim appname as String
    Dim topic as String
    Dim item as String
    Dim msgtext as String
    Dim path as string
    appname="winword"
    topic="System"
    item="Topics"
    path="c:\msoffice\winword\"
    channel = -1
    x=Shell(path & appname & ".EXE")
    channel = DDEInitiate(appname, topic)
    If channel= -1 then
        msgtext="M/S Word not found -- please place on your path."
```

```
Else
  On Error Resume Next
  msgtext="The Word topics available are:" & Chr$(13)
  msgtext=msgtext & Chr$(13) & DDERequest(channel,item)
  DDETerminate channel
  If Err<>0 then
    msgtext="DDE Access failed."
  End If
End If
MsgBox msgtext
End Sub
```

See Also

DDEAppReturnCode, DDEExecute, DDEPoke, DDERequest(), DDETerminate

DDEPoke Statement

Sends data to an application on an open dynamic-data exchange (DDE) channel.

Syntax

DDEPoke *channel%*, *item\$*, *data\$*

Where...	Is...
<i>channel%</i>	An integer or expression for the open DDE channel number.
<i>item\$</i>	A string or expression for the name of an item in the currently opened topic.
<i>data\$</i>	A string or expression for the information to send to the topic.

Remarks

If *channel%* does not correspond to an open channel, an error occurs.

When you open a channel to an application using **DDEInitiate**, you also specify a topic, such as a filename, to communicate with. The *item\$* is the part of the topic you want to send data to. **DDEPoke** sends data as a text string; you cannot send text in any other format, nor can you send graphics.

If the server application does not recognize *item\$*, an error occurs.

Example

This example opens Microsoft Write, uses **DDEPoke** to write the text "Hello, world" to the open document (Untitled) and uses **DDEExecute** to save the text to the file TEMPO01.

```
Sub main
    Dim channel as Integer
    Dim appname as String
    Dim topic as String
    Dim testtext as String
    Dim item as String
    Dim pcommand as String
    Dim msgtext as String
    Dim x as Integer
    Dim path as String
    appname="WinWord"
    path="c:\msoffice\winword\"
    topic="Document1"
    item="Page1"
    testtext="Hello, world."
    On Error Goto Errhandler
    x=Shell(path & appname & ".EXE")
    channel = DDEInitiate(appname, topic)
    If channel=0 then
        MsgBox "Unable to open Write."
    Exit Sub
```

```
End If
DDEPoke channel, item, testtext
pcommand="[FileSaveAs .Name = "&Chr$(34) &"C:\TEMP001" &Chr$(34)&"]"
DDEExecute channel, pcommand
pcommand="[FileClose]"
DDEExecute channel, pcommand
msgtext="The text: " & testtext & " saved to C:\TEMP001." & Chr$(13)
msgtext=msgtext & Chr$(13) & "Delete? (Y/N)"
answer=InputBox(msgtext)
If answer="Y" or answer="y" then
    Kill "C:\TEMP001.doc"
End If
DDETerminate channel
Exit Sub
Errhandler:
If Err<>0 then
    MsgBox "DDE Access failed."
End If
End Sub
```

See Also**DDEAppReturnCode(), DDEExecute, DDEInitiate(), DDERequest(), DDETerminate**

DDERequest Function

Returns data from an application through an open dynamic data exchange (DDE) channel.

Syntax

DDERequest[\$] (*channel%*, *item\$*)

Where...	Is...
<i>channel%</i>	An integer or expression for the open DDE channel number.
<i>item\$</i>	A string or expression for the name of an item in the currently opened topic to get information about.

Remarks

If *channel%* does not correspond to an open channel, an error occurs.

If the server application doesn't recognize *item\$*, an error occurs.

If **DDERequest** is unsuccessful, it returns an empty string ("").

When you open a channel to an application using **DDEInitiate**, you also specify a topic, such as a filename, to communicate with. The *item\$* is the part of the topic whose contents you are requesting.

DDERequest returns data as a text string. Data in any other format cannot be transferred, nor can graphics.

Many applications that support DDE recognize a topic named System. Three standard items in the System topic are described in the following table:

Item	Returns
System	A list of all items in the System topic
Topics	A list of available topics
Formats	A list of all the Clipboard formats supported

Example

This example uses **DDEInitiate** to open a channel to Microsoft Word. It uses **DDERequest** to obtain a list of available topics (using the System topic).

```
Sub main
    Dim channel as Integer
    Dim appname as String
    Dim topic as String
    Dim item as String
    Dim msgtext as String
    Dim path as string
    appname="winword"
    topic="System"
    item="Topics"
    path="c:\msoffice\winword\"
    channel = -1
    x=Shell(path & appname & ".EXE")
    channel = DDEInitiate(appname, topic)
    If channel= -1 then
        msgtext="M/S Word not found -- please place on your path."
    Else
        On Error Resume Next
        msgtext="The Word topics available are:" & Chr$(13)
        msgtext=msgtext & Chr$(13) & DDERequest(channel,item)
        DDETerminate channel
        If Err<>0 then
            msgtext="DDE Access failed."
        End If
    End If
    MsgBox msgtext
End Sub
```

See Also

DDEAppReturnCode(), **DDEExecute**, **DDEInitiate()**, **DDEPoke**, **DDETerminate**

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.

Remarks

To free system resources, you should close channels you are not using. If *channel%* does not correspond to an open channel, an error occurs.

Example

This example uses **DDEInitiate** to open a channel to Microsoft Word. It uses **DDERequest** to obtain a list of available topics (using the System topic) and then terminates the channel using **DDETerminate**.

```
Sub main
  Dim channel as Integer
  Dim appname as String
  Dim topic as String
  Dim item as String
  Dim msgtext as String
  Dim path as string
  appname="winword"
  topic="System"
  item="Topics"
  path="c:\msoffice\winword\"
  channel = -1
  x=Shell(path & appname & ".EXE")
  channel = DDEInitiate(appname, topic)
  If channel= -1 then
    msgtext="M/S Word not found -- please place on your path."
  Else
    On Error Resume Next
    msgtext="The Word topics available are:" & Chr$(13)
    msgtext=msgtext & Chr$(13) & DDERequest(channel,item)
    DDETerminate channel
    If Err<>0 then
      msgtext="DDE Access failed."
    End If
  End If
  MsgBox msgtext
End Sub
```

See Also

DDEAppReturnCode(), **DDEExecute**, **DDEInitiate()**, **DDEPoke**, **DDERequest()**

Declare Statement

Declares a procedure in a module or dynamic link library (DLL).

Syntax A

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

Syntax B

Declare Function *name* [*libSpecification*] [(*parameter* [*As type*])] [*As functype*]

Where...	Is...
<i>name</i>	The subprogram or function procedure to declare.
<i>libSpecification</i>	The location of the procedure (module or DLL).
<i>parameter</i>	The arguments to pass to the procedure, separated by commas.
<i>type</i>	The type for the arguments.
<i>functype</i>	The type of the return value for a function procedure.

Remarks

A **Sub** procedure does not return a value. A **Function** procedure returns a value, and can be used in an expression. To specify the data type for the return value of a function, end the Function name with a type character or use the **As** *functype* clause shown above. If no type is provided, the function defaults to data type variant.

If the *libSpecification* is of the format:

```
BasicLib libName [ Alias "aliasname" ]
```

the procedure is 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. BSL will not automatically unload modules that are loaded in this fashion. BSL will detect errors of misdeclaration.

If the *libSpecification* is of the format:

```
Lib libName [ Alias [ " ]ordinal[ " ] ] or  
Lib libName [ Alias "aliasname" ]
```

the procedure is 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 is accessed by name. It is recommended that the *ordinal* be used whenever possible, since accessing functions by name might 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 defined. In this case, the **BasicLib**, **Lib** and **Alias** clauses are not used.

The data type of a parameter can be specified by using a type character or by using the **As** clause. Record parameters are declared by using an **As** clause and a *type* that 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** can also be used in the call. It is the external DLL procedure's responsibility to determine the type and size of the passed-in value.

BSL supports two different behaviors when an empty string ("") is passed **ByVal** to an external procedure. The implementer of BSL can specify which behavior by using the BSL API function **BSLSetInstanceFlags**. In any specific implementation that uses BSL, one of these two behaviors should be used consistently. We recommend the second behavior, which is compatible with Microsoft's Visual Basic language.

1. When an empty string ("") is passed **ByVal** to an external procedure, the external procedure will receive a NULL pointer. If you want to send a valid pointer to an empty string, use **Chr\$(0)**.
2. 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**.

Example

This example declares a function that is later called by the main subprogram. The function does nothing but set its return value to 1.

```
Declare Function BSL_exfunction()  
Sub main  
    Dim y as Integer  
    Call BSL_exfunction  
    y=BSL_exfunction  
    MsgBox "The value returned by the function is: " & y  
End Sub  
Function BSL_exfunction()  
    BSL_exfunction=1  
End Function
```

See Also

Call, Const, Deftype, Dim, Static, Type

Deftype Statement

Specifies the default data type for one or more variables.

Syntax

DefCur *varTypeLetters*

DefInt *varTypeLetters*

DefLng *varTypeLetters*

DefSng *varTypeLetters*

DefDbI *varTypeLetters*

DefStr *varTypeLetters*

DefVar *varTypeLetters*

Where...

Is...

varTypeLetters A first letter of the variable name to use.

Remarks

varTypeLetters can be a single letter, a comma-separated list of letters, or a range of letters. For example, a-d indicates the letters a, b, c and d. The case of the letters is not important, even in a letter range. The letter range a-z is treated as a special case: it denotes all alpha characters, including the international characters. The **Deftype** statement affects only the module in which it is specified. It must precede any variable definition within the module. Variables defined using the Global or Dim can override the **Deftype** statement by using an **As** clause or a type character.

Example

This example finds the average of bowling scores entered by the user. Since the variable *average* begins with A, it is automatically defined as a single-precision floating point number. The other variables will be defined as Integers.

```

DefInt c,s,t
DefSng a
Sub main
    Dim count
    Dim total
    Dim score
    Dim average
    Dim msgtext
    For count=0 to 4
        score=InputBox("Enter bowling score #" & count+1 &":")
        total=total+score
    Next count
    average=total/count
    msgtext="Your average is: " &average
    MsgBox msgtext
End Sub

```

See Also

Declare, Dim, Let, Type

Dialog Function

Displays a dialog box and returns a number for the button selected (-1= OK, 0=Cancel).

Syntax

Dialog (*recordName*)

Where... **Is...**

recordName A variable name declared as a dialog box record.

Remarks

If the dialog box contains additional command buttons (for example, Help), the **Dialog** function returns a number greater than 0. 1 corresponds to the first command button, 2 to the second, and so on.

The dialog box *recordName* must have been declared using the **Dim** statement with the **As** parameter followed by a dialog box definition name. This name comes from the name argument used in the Begin Dialog statement.

To trap a user's selections within a dialog box, you must create a function and specify it as the last argument to the Begin Dialog statement. See the "Begin Dialog...End Dialog Statement" on page 34 for more information.

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

Example

This example creates a dialog box with a drop down combo box in it and three buttons: OK, Cancel, and Help. The **Dialog** function used here enables the subroutine to trap when the user clicks on any of these buttons.

```
Sub main
  Dim cchoices as String
  cchoices="All"+Chr$(9)+"Nothing"
  Begin Dialog UserDialog 180, 95, "BSL Dialog Box"
    ButtonGroup .ButtonGroup1
    Text 9, 3, 69, 13, "Filename:", .Text1
    ComboBox 9, 17, 111, 41, cchoices, .ComboBox1
    OKButton 131, 8, 42, 13
    CancelButton 131, 27, 42, 13
    PushButton 132, 48, 42, 13, "Help", .Push1
  End Dialog
  Dim mydialogbox As UserDialog
  answer= Dialog(mydialogbox)
  Select Case answer
    Case -1
      MsgBox "You pressed OK"
    Case 0
      MsgBox "You pressed Cancel"
    Case 1
      MsgBox "You pressed Help"
  End Select
End Sub
```

See Also

Begin Dialog...End Dialog, Dialog Statement

Dialog Statement

Displays a dialog box.

Syntax

Dialog *recordName*

Where... **Is...**

recordName A variable name declared as a dialog box record.

Remarks

The dialog box *recordName* must have been declared using the **Dim** statement with the **As** parameter followed by a dialog box definition name. This name comes from the name argument used in the **Begin Dialog** statement.

If the user exits the dialog box by pushing the Cancel button, the run-time error 102 is triggered, which can be trapped using **On Error**. To trap a user's selections within a dialog box, you must create a function and specify it as the last argument to the **Begin Dialog** statement. See "Begin Dialog...End Dialog Statement" on page 34 for more information. The **Dialog** statement does not return until the dialog box is closed.

Example

This example defines and displays a dialog box defined as *UserDialog* and named *mydialogbox*. If the user presses the Cancel button, an error code of 102 is returned and is trapped by the **If...Then** statement listed after the **Dialog** statement.

```
Sub main
  Dim cchoices as String
  On Error Resume Next
  cchoices="All"+Chr$(9)+"Nothing"
  Begin Dialog UserDialog 180, 95, "BSL Dialog Box"
    ButtonGroup .ButtonGroup1
    Text 9, 3, 69, 13, "Filename:", .Text1
    ComboBox 9, 17, 111, 41, cchoices, .ComboBox1
    OKButton 131, 8, 42, 13
    CancelButton 131, 27, 42, 13
  End Dialog
  Dim mydialogbox As UserDialog
  Dialog mydialogbox
  If Err=102 then
    MsgBox "You pressed Cancel."
  Else
    MsgBox "You pressed OK."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog, **Dialog Function**

Dim Statement

Declares variables for use in a Basic program.

Syntax

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

Where...	Is...
<i>variableName</i>	The name of the variable to declare.
<i>type</i>	The data type of the variable.

Remarks

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

```
Dim my_1st_variable As String
```

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

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

Basic is a strongly typed language: all variables must be given a data type or they will be automatically assigned the data type variant. The available data types are:

- Arrays
- Numbers
- Objects
- Records
- Strings
- Variants

Variables can be shared across modules. A variable declared inside a procedure has scope Local to that procedure. A variable declared outside a procedure has scope Local to the module. If you declare a variable with the same name as a module variable, the module variable is not accessible. 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 a procedure. It has no effect.

It is considered good programming practice to declare all variables. 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 which mechanism you use to declare a variable, you can choose to use or omit the type character when referring to the variable in the rest of your program. The type suffix is not considered part of the variable name.

Arrays

The available data types for arrays are: numbers, strings, variants, objects and records. Arrays of arrays and dialog box records 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 that can 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.

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. Numeric variables are initialized to 0.

Objects

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

```
Dim OLE2 As Object
Set OLE2 = CreateObject("spoly.cpoly")
OLE2.reset
```

An object can 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
```

Records

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

```
Dim variableName As typeName
```

Records are made up of a collection of data elements called fields. These fields can be of any numeric, string, variant, or previously-defined record type. See the "Type Statement" on page 300 for details on accessing fields within a record.

You can also use the **Dim** statement to declare a dialog box record. In this case, *type* is specified as *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 box records have the same behavior as regular records; they differ only in the way they are defined. Some applications might provide a number of predefined dialog boxes.

Strings

BSL 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 32,767. 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
```

When initialized, fixed-length strings are filled with zeros. Dynamic strings are initialized as zero-length strings.

Variants

Declare variables as variants when the type of the variable is not known at the start of, or might change during, the procedure. For example, a variant is useful for holding input from a user when valid input can be either text or numbers. Use the following syntax to declare a variant:

```
Dim variableName or  
Dim variableName As Variant
```

Variant variables are initialized to vartype Empty.

Example

This example shows a Dim statement for each of the possible data types.

```
Rem Must define a record type before you can declare a record variable  
Type Testrecord  
    Custno As Integer  
    Custname As String  
End Type  
  
Sub main  
    Dim counter As Integer  
    Dim fixedstring As String*25  
    Dim varstring As String  
    Dim myrecord As Testrecord  
    Dim ole2var As Object  
    Dim F(1 to 10), A()  
    ' ... (code here) ...  
End Sub
```

See Also

Global, Option Base, ReDim, Set, Static, Type

Dir Function

Returns a filename that matches the specified pattern.

Syntax

Dir[\$] [(*pathname*\$ [,*attributes*%)]

Where...	Is...
<i>pathname</i> \$	A string expression identifying a path or filename.
<i>attributes</i> %	An integer expression specifying the file attributes to select.

Remarks

Pathname\$ can include a drive specification and wildcard characters ('?' and '*'). **Dir** returns the first filename that matches the *pathname*\$ argument. An empty string ("") passed as *pathname*\$ is interpreted as the current directory (same as "."). To retrieve additional matching filenames, call the **Dir** function again, omitting the *pathname*\$ and *attributes*% arguments. If no file is found, an empty string ("") is returned.

The default value for *attributes*% is 0. In this case, **Dir** returns only files without directory, hidden, system, or volume label attributes set.

Here are the possible values for *attributes*%:

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 added together to select multiple attributes. For example, to list hidden and system files in addition to normal files set *attributes*% to 6 (6=2+4).

If *attributes*% is set to 8, the **Dir** function returns the volume label of the drive specified in the *pathname*\$, or of the current drive if drive is not explicitly specified. If volume label attribute is set, all other attributes are ignored.

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).

Example

This example lists the contents of the diskette in drive A.

```
Sub main
  Dim msgret
  Dim directory, count
  Dim x, msgtext
  Dim A()
  msgret=MsgBox("Insert a disk in drive A.")
  count=1
  ReDim A(100)
  directory=Dir ("A:\*.*")
  Do While directory<>""
    A(count)=directory
    count=count+1
    directory=Dir
  Loop
  msgtext="Contents of drive A:\ is:" & Chr(10) & Chr(10)
  For x=1 to count
    msgtext=msgtext & A(x) & Chr(10)
  Next x
  MsgBox msgtext
End Sub
```

See Also

ChDir, ChDrive, CurDir, Mkdir, Rmdir

DlgControlID Function

Returns the numeric ID of a dialog box control with the specified *Id\$* in the active dialog box.

Syntax

DlgControlID (*Id\$*)

Where...	Is...
<i>Id\$</i>	The string ID for a dialog control.

Remarks

The **DlgControlID** function translates a string *Id\$* into a numeric ID. This function can only be used from within a dialog box function. The value of the numeric identifier is based on the position of the dialog box control with the dialog; it will be 0 (zero) for the first control, 1 (one) for the second control, and so on.

Given the following example, the statement **DlgControlID**("doGo") will return the value 1.

```
Begin Dialog newdlg 200, 200
  PushButton 40, 50, 80, 20, "&Stop", .doStop
  PushButton 40, 80, 80, 20, "&Go", .doGo
End Dialog
```

The advantage of using a dialog box control's numeric ID is that it is more efficient, and numeric values can sometimes be more easily manipulated.

Rearranging the order of a control within a dialog box will change its numeric ID. For example, if a PushButton control originally had a numeric value of 1, and a textbox control is added before it, the PushButton control's new numeric value will be 2. This is shown in the following example:

```
CheckBox 40, 110, 80, 20, "CheckBox", .CheckBox1
TextBox 40, 20, 80, 20, .TextBox1 ' this is the new added control
PushButton 40, 80, 80, 20, "&Go", .doGo
```

The string IDs come from the last argument in the dialog definition statement that created the dialog control, such as the **TextBox** or **ComboBox** statements. The string ID does not include the period (.) and is case-sensitive.

Use **DlgControlID** only while a dialog box is running. See "Begin Dialog...End Dialog Statement" on page 34 for more information.

Example

This example displays a dialog box similar to File Open.

```
Declare Sub ListFiles(str1$)
Declare Function FileDlgFunction(identifier$, action, suppvalue)

Sub main
  Dim identifier$
  Dim action as Integer
  Dim suppvalue as Integer
  Dim filetypes as String
  Dim exestr$()
  Dim button as Integer
  Dim x as Integer
  Dim directory as String
  filetypes="Program files (*.exe)+Chr$(9)+"All Files (*.*)"
  Begin Dialog newdlg 230, 145, "Open", .FileDlgFunction
    '$CStrings Save
    Text 8, 6, 60, 11, "&Filename:"
    TextBox 8, 17, 76, 13, .TextBox1
    ListBox 9, 36, 75, 61, exestr$(), .ListBox1
    Text 8, 108, 61, 9, "List Files of &Type:"
    DropListBox 7, 120, 78, 30, filetypes, .DropListBox1
    Text 98, 7, 43, 10, "&Directories:"
    Text 98, 20, 46, 8, "c:\windows"
    ListBox 99, 34, 66, 66, "", .ListBox2
    Text 98, 108, 44, 8, "Dri&ves:"
    DropListBox 98, 120, 68, 12, "", .DropListBox2
    OKButton 177, 6, 50, 14
    CancelButton 177, 24, 50, 14
    PushButton 177, 42, 50, 14, "&Help"
    '$CStrings Restore
  End Dialog
  Dim dlg As newdlg
  button = Dialog(dlg)
End Sub

Sub ListFiles(str1$)
  DlgText 1, str1$
  x=0
  Redim exestr$(x)
  directory=Dir$("c:\windows\" & str1$,16)
  If directory<>"" then
    Do
      exestr$(x)=LCase$(directory)
      x=x+1
      Redim Preserve exestr$(x)
      directory=Dir
    Loop Until directory=""
  End If
  DlgListBoxArray 2, exestr$()
```

```
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
  Select Case action
    Case 1
      str1$="*.exe"           'dialog box initialized
      ListFiles str1$
    Case 2
      'button or control value changed
      If DlgControlId(identifier$) = 4 Then
        If DlgText(4)="All Files (*.*)" then
          str1$="*.*"
        Else
          str1$="*.exe"
        End If
      ListFiles str1$
      End If
    Case 3
      'text or combo box changed
      str1$=DlgText$(1)
      ListFiles str1$
    Case 4
      'control focus changed

    Case 5
      'idle
  End Select
End Function
```

See Also

BeginDialog...End Dialog, **DlgEnable Function**, **DlgEnable Statement**, **DlgFocus Function**, **DlgFocus Statement**, **DlgListBoxArray Function**, **DlgListBoxArray Statement**, **DlgSetPicture Statement**, **DlgText Function**, **DlgText Statement**, **DlgValue Function**, **DlgValue Statement**, **DlgVisible Function**, **DlgVisible Statement**

DlgEnable Function

Returns the enable state for the specified dialog control (-1=enabled, 0=disabled).

Syntax

DlgEnable (*Id*)

Where...	Is...
----------	-------

<i>Id</i>	The control ID for the dialog control.
-----------	--

Remarks

If a dialog box control is enabled, it is accessible to the user. You might want to disable a control if its use depends on the selection of other controls.

Use the **DlgControlID** function to find the numeric ID for a dialog control, based on its string identifier.

Use **DlgEnable** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with two check boxes, one labeled Either, the other labeled Or. If the user clicks on Either, the Or option is grayed. Likewise, if Or is selected, Either is grayed. This example uses the **DlgEnable** statement to toggle the state of the buttons.

```
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92,"DlgEnable example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        CheckBox 34, 25, 75, 19, "Either", .CheckBox1
        CheckBox 34, 43, 73, 25, "Or", .CheckBox2
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 2 'button or control value changed
            If DlgControlId(identifier$) = 2 Then
                DlgEnable 3
            Else
                DlgEnable 2
            End If
        End Select
    End Function
```

See Also

BeginDialog...End Dialog, DigControlID Function, DigEnable Function, DigFocus Function, DigFocus Statement, DigListBoxArray Function, DigListBoxArray Statement, DigSetPicture Statement, DigText Function, DigText Statement, DigValue Function, DigValue Statement, DigVisible Function, DigVisible Statement

DlgEnable Statement

Enables, disables, or toggles the state of the specified dialog control.

Syntax

DlgEnable *Id* [, *mode*]

Where...	Is...
<i>Id</i>	The control ID for the dialog control to change.
<i>mode</i>	An integer representing the enable state (1=enable, 0=disable)

Remarks

If *mode* is omitted, the **DlgEnable** toggles the state of the dialog control specified by *Id*. If a dialog box control is enabled, it is accessible to the user. You might want to disable a control if its use depends on the selection of other controls.

Use the **DlgControlID** function to find the numeric ID for a dialog control, based on its string identifier. The string IDs come from the last argument in the dialog definition statement that created the dialog control, such as the **TextBox** or **ComboBox** statements.

Use **DlgEnable** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with one check box, labeled Show More, and a group box, labeled More, with two option buttons, Option 1 and Option 2. It uses the **DlgEnable** function to enable the More group box and its options if the Show More check box is selected.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgEnable example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        CheckBox 13, 6, 75, 19, "Show more", .CheckBox1
        GroupBox 16, 28, 94, 50, "More"
        OptionGroup .OptionGroup1
            OptionButton 23, 40, 56, 12, "Option 1", .OptionButton1
            OptionButton 24, 58, 61, 13, "Option 2", .OptionButton2
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
            DlgEnable 3,0
            DlgEnable 4,0
            DlgEnable 5,0
        Case 2
            'button or control value changed
            If DlgControlID(identifier$) = 2 Then

```

```
    If DlgEnable (3)=0 then
        DlgEnable 3,1
        DlgEnable 4,1
        DlgEnable 5,1
    Else
        DlgEnable 3,0
        DlgEnable 4,0
        DlgEnable 5,0
    End If
End If
End Select
End Function
```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Statement, **DlgFocus** Function, **DlgFocus** Statement, **DlgListBoxArray** Function, **DlgListBoxArray** Statement, **DlgSetPicture** Statement, **DlgText** Function, **DlgText** Statement, **DlgValue** Function, **DlgValue** Statement, **DlgVisible** Function, **DlgVisible** Statement

DlgEnd Statement

Closes the active dialog box.

Syntax

DlgEnd *exitCode*

Where...	Is...
<i>exitCode</i>	The return value after closing the dialog box (-1=OK, 0=Cancel).

Remarks

ExitCode contains a return value only if the dialog box was displayed using the **Dialog** function. That is, if you used the **Dialog** statement, *exitCode* is ignored.

If the dialog box contains additional command buttons (for example, Help), the **Dialog** function returns a number greater than 0. 1 corresponds to the first command button, 2 to the second, and so on.

Use **DlgEnd** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with the message “You have 30 seconds to cancel.” The dialog box counts down from 30 seconds to 0. If the user clicks OK or Cancel during the countdown, the dialog box closes. If the countdown reaches 0, however, the **DlgEnd** statement closes the dialog box.

```
Function timeout(id$,action%,suppvalue&)
  Static timeoutStart as Long
  Static currentSecs as Long
  Dim thisSecs as Long
  Select Case action%
    Case 1
      ' initialize the dialog box. Set the ticker value to 30
      ' and remember when we put up the dialog box
      DlgText "ticker", "30"
      timeoutStart = timer
      currentSecs = 30
    Case 5
      ' this is an idle message - set thisSecs to the number of
      ' seconds left until timeout
      thisSecs = timer
      If thisSecs < timeoutStart Then thisSecs = thisSecs + 24*60*60
      thisSecs = 30 - (thisSecs - timeoutStart)
      ' if there are negative seconds left, timeout!
      If thisSecs < 0 Then DlgEnd -1
      ' If the seconds left has changed since last time,
      ' update the dialog box
      If thisSecs <> currentSecs Then
        DlgText "ticker", trim$(str$(thisSecs))
        currentSecs = thisSecs
      End If
      ' make sure to return non-zero so we keep getting idle messages
      timeout = 1
  End Select
End Function
```

```
Sub main
  Begin Dialog newdlg 167, 78, "Do You Want to Continue?", .timeout
    '$CStrings Save
    OKButton 27, 49, 50, 14
    CancelButton 91, 49, 50, 14
    Text 24, 14, 119, 8, "This is your last chance to bail out."
    Text 27, 30, 35, 8, "You have"
    Text 62, 30, 13, 8, "30", .ticker
    Text 74, 30, 66, 8, "seconds to cancel."
    '$CStrings Restore
  End Dialog
  Dim dlgVar As newdlg
  If dialog(dlgvar) = 0 Then
    Exit Sub          ' abort
  End If
  ' do whatever it is we want to do
End Sub
```

See Also

BeginDialog...End Dialog, DigControlID Function, DigEnable Function, DigEnable Statement, DigFocus Function, DigFocus Statement, DigListBoxArray Function, DigListBoxArray Statement, DigSetPicture Statement, DigText Function, DigText Statement, DigValue Function, DigValue Statement, DigVisible Function, DigVisible Statement

DlgFocus Function

Returns the control ID of the dialog control having the input focus.

Syntax

DlgFocus[\$]()

Remarks

A control has focus when it is active and responds to keyboard input. Use **DlgFocus** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with a check box, labeled Check1, and a text box, labeled Text Box 1, in it. When the box is initialized, the focus is set to the text box. As soon as the user selects the check box, the focus goes to the OK button.

```
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub main
  Dim button as integer
  Dim identifier$
  Dim action as Integer
  Dim suppvalue as Integer
  Begin Dialog newdlg 186, 92, "DlgFocus Example", .FileDlgFunction
    OKButton 130, 6, 50, 14
    CancelButton 130, 23, 50, 14
    TextBox 15, 37, 82, 12, .TextBox1
    Text 15, 23, 57, 10, "Text Box 1"
    CheckBox 15, 6, 75, 11, "Check1", .CheckBox1
  End Dialog
  Dim dlg As newdlg
  button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
  Select Case action
    Case 1
      DlgFocus 2
    Case 2
      'user changed control or clicked a button
      If DlgFocus () <> "OKButton" then
        DlgFocus 0
      End If
    End Select
  End Function
```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Function, **DlgEnable** Statement, **DlgFocus** Statement, **DlgListBoxArray** Function, **DlgListBoxArray** Statement, **DlgSetPicture** Statement, **DlgText** Function, **DlgText** Statement, **DlgValue** Function, **DlgValue** Statement, **DlgVisible** Function, **DlgVisible** Statement

DlgFocus Statement

Sets the focus for the specified dialog control.

Syntax

DlgFocus *Id*

Where...	Is...
<i>Id</i>	The control ID for the dialog control to make active.

Remarks

Use the **DlgControlID** function to find the numeric ID for a dialog control, based on its string identifier. The string IDs come from the last argument in the dialog definition statement that created the dialog control, such as the **TextBox** or **ComboBox** statements.

Use **DlgFocus** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with a check box, labeled Check1, and a text box, labeled Text Box 1, in it. When the box is initialized, the focus is set to the text box. As soon as the user selects the check box, the focus goes to the OK button.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgFocus Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        TextBox 15, 37, 82, 12, .TextBox1
        Text 15, 23, 57, 10, "Text Box 1"
        CheckBox 15, 6, 75, 11, "Check1", .CheckBox1
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub
Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
            DlgFocus 2
        Case 2 'user changed control or clicked a button
            If DlgFocus() <> "OKButton" then
                DlgFocus 0
            End If
        End Select
    End Function

```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Function, **DlgEnable** Statement, **DlgFocus** Function, **DlgListBoxArray** Function, **DlgListBoxArray** Statement, **DlgSetPicture** Statement, **DlgText** Function, **DlgText** Statement, **DlgValue** Function, **DlgValue** Statement, **DlgVisible** Function, **DlgVisible** Statement

DlgListBoxArray Function

Returns the number of elements in a list or combo box.

Syntax

DlgListBoxArray (*Id* [, *Array\$*])

Where...	Is...
<i>Id</i>	The control ID for the list or combo box.
<i>Array\$</i>	The entries in the list box or combo box returned.

Remarks

Array\$ is a one-dimensional array of dynamic strings. If *array\$* is dynamic, its size is changed to match the number of strings in the list or combo box. If *array\$* is not dynamic and it is too small, an error occurs. If *array\$* is omitted, the function returns the number of entries in the specified dialog control.

Use the **DigControlID** function to find the numeric ID for a dialog control, based on its string identifier. The string IDs come from the last argument in the dialog definition statement that created the dialog control, such as the **TextBox** or **ComboBox** statements.

Use **DlgListBoxArray** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with a check box, labeled “Display List”, and an empty list box. If the user selects the check box, the list box is filled with the contents of the array called “myarray”. The **DlgListBoxArray** function makes sure the list box is empty.

```
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgListBoxArray Example", .FileDlgFunction
        '$CStrings Save
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        ListBox 19, 26, 74, 59, "", .ListBox1
        CheckBox 12, 4, 86, 13, "Display List", .CheckBox1
        '$CStrings Restore
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Dim myarray$(3)
    Dim msgtext as Variant
    Dim x as Integer
    For x= 0 to 2
        myarray$(x)=Chr$(x+65)
    Next x
    Select Case action
        Case 1
```

```
Case 2                                'user changed control or clicked a button
  If DlgControlID(identifier$)=3 then
    If DlgListBoxArray(2)=0 then
      DlgListBoxArray 2, myarray$()
    End If
  End If
End Select
End Function
```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Function, **DlgEnable** Statement, **DlgFocus** Function, **DlgFocus** Statement, **DlgListBoxArray** Statement, **DlgSetPicture** Statement, **DlgText** Function, **DlgText** Statement, **DlgValue** Function, **DlgValue** Statement, **DlgVisible** Function, **DlgVisible** Statement

DlgListBoxArray Statement

Fills a list or combo box with an array of strings.

Syntax

DlgListBoxArray *Id*, *Array\$*

Where...	Is...
<i>Id</i>	The control ID for the list or combo box.
<i>Array\$</i>	The entries for the list box or combo box.

Remarks

Array\$ has to be a one-dimensional array of dynamic strings. One entry appears in the list box for each element of the array. If the number of strings changes depending on other selections made in the dialog box, you should use a dynamic array and **ReDim** the size of the array whenever it changes.

Use **DlgListBoxArray** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box similar to File Open.

```
Declare Sub ListFiles(str1$)
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub main
  Dim identifier$
  Dim action as Integer
  Dim suppvalue as Integer
  Dim filetypes as String
  Dim exestr$()
  Dim button as Integer
  Dim x as Integer
  Dim directory as String
  filetypes="Program files (*.exe)+Chr$(9)+"All Files (*.*)"
  Begin Dialog newdlg 230, 145, "Open", .FileDlgFunction
    '$CStrings Save
    Text 8, 6, 60, 11, "&Filename:"
    TextBox 8, 17, 76, 13, .TextBox1
    ListBox 9, 36, 75, 61, exestr$(), .ListBox1
    Text 8, 108, 61, 9, "List Files of &Type:"
    DropListBox 7, 120, 78, 30, filetypes, .DropListBox1
    Text 98, 7, 43, 10, "&Directories:"
    Text 98, 20, 46, 8, "c:\\windows"
    ListBox 99, 34, 66, 66, "", .ListBox2
    Text 98, 108, 44, 8, "Dri&ves:"
    DropListBox 98, 120, 68, 12, "", .DropListBox2
    OKButton 177, 6, 50, 14
    CancelButton 177, 24, 50, 14
    PushButton 177, 42, 50, 14, "&Help"
    '$CStrings Restore
  End Dialog
  Dim dlg As newdlg
  button = Dialog(dlg)
End Sub
```

```

Sub ListFiles(str1$)
  DlgText 1,str1$
  x=0
  Redim exestr$(x)
  directory=Dir$("c:\windows\" & str1$,16)
  If directory<>"" then
    Do
      exestr$(x)=LCase$(directory)
      x=x+1
      Redim Preserve exestr$(x)
      directory=Dir
    Loop Until directory=""
  End If
  DlgListBoxArray 2,exestr$()
End Sub

Function FileDlgFunction(identifier$, action, supvalue)
  Select Case action
    Case 1
      str1$="*.exe"           'dialog box initialized
      ListFiles str1$
    Case 2                   'button or control value changed
      If DlgControlId(identifier$) = 4 Then
        If DlgText(4)="All Files (*.*)" then
          str1$="*.*"
        Else
          str1$="*.exe"
        End If
      ListFiles str1$
      End If
    Case 3                   'text or combo box changed
      str1$=DlgText$(1)
      ListFiles str1$
    Case 4                   'control focus changed
    Case 5                   'idle
  End Select
End Function

```

See Also

BeginDialog...End Dialog, DlgControlID Function, DlgEnable Function, DlgFocus Function, DlgFocus Statement, DlgListBoxArray Function, DlgEnable, DlgSetPicture Statement, DlgText Function, DlgText Statement, DlgValue Function, DlgValue Statement, DlgVisible Function, DlgVisible Statement

DlgSetPicture Statement

Changes the picture in a picture dialog control for the current dialog box.

Syntax

DlgSetPicture *id*, *filename\$*, *type*

Where...	Is...
<i>id</i>	The control ID for the picture dialog control.
<i>filename\$</i>	The name of the bitmap file (.BMP) to use.
<i>type</i>	An integer representing the location of the file (0= <i>filename\$</i> , 3=Clipboard)

Remarks

Use the **DlgControlID** function to find the numeric ID for a dialog control, based on its string identifier. The string IDs come from the last argument in the dialog definition statement that created the dialog control, such as the **TextBox** or **ComboBox** statements.

Use **DlgListBoxArray** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information. See the “Picture Statement” on page 233 for more information about displaying pictures in dialog boxes.

Example

This example displays a picture in a dialog box and changes the picture if the user selects the check box labeled “Change Picture”.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgSetPicture Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        Picture 43, 28, 49, 31, "C:\WINDOWS\CIRCLES.BMP", 0
        CheckBox 30, 8, 62, 15, "Change Picture", .CheckBox1
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub
Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
        Case 2 'user changed control or clicked a button
            If DlgControlID(identifier$)=3 then
                If suppvalue=1 then
                    DlgSetPicture 2, "C:\WINDOWS\TILES.BMP",0
                Else
                    DlgSetPicture 2, "C:\WINDOWS\CIRCLES.BMP",0
                End If
            End If
        End Select
    End Function

```

See Also

BeginDialog...End Dialog, DlgControlIID Function, DlgEnable Function, DlgEnable Statement, DlgFocus Function, DlgFocus Statement, DlgListBoxArray Function, DlgListBoxArray Statement, DlgText Function, DlgText Statement, DlgValue Function, DlgValue Statement, DlgVisible Function, DlgVisible Statement

DlgText Function

Returns the text associated with a dialog control for the current dialog box.

Syntax

DlgText[\$] (*Id*)

Where...	Is...
<i>Id</i>	The control ID for a dialog control.

Remarks

If the control is a text box or a combo box, **DlgText** function returns the text that appears in the text box. If it is a list box, the function returns its current selection. If it is a text box, **DlgText** returns the text. If the control is a command button, option button, option group, or a check box, the function returns its label. Use **DlgText** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box similar to File Open. It uses **DlgText** to determine what group of files to display.

```
Declare Sub ListFiles(str1$)
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub main
  Dim identifier$
  Dim action as Integer
  Dim suppvalue as Integer
  Dim filetypes as String
  Dim exestr$()
  Dim button as Integer
  Dim x as Integer
  Dim directory as String
  filetypes="Program files (*.exe)+"&Chr$(9)+"All Files (*.*)"
  Begin Dialog newdlg 230, 145, "Open", .FileDlgFunction
    '$CStrings Save
    Text 8, 6, 60, 11, "&Filename:"
    TextBox 8, 17, 76, 13, .TextBox1
    ListBox 9, 36, 75, 61, exestr$(), .ListBox1
    Text 8, 108, 61, 9, "List Files of &Type:"
    DropListBox 7, 120, 78, 30, filetypes, .DropListBox1
    Text 98, 7, 43, 10, "&Directories:"
    Text 98, 20, 46, 8, "c:\\windows"
    ListBox 99, 34, 66, 66, "", .ListBox2
    Text 98, 108, 44, 8, "Dri&ves:"
    DropListBox 98, 120, 68, 12, "", .DropListBox2
    OKButton 177, 6, 50, 14
    CancelButton 177, 24, 50, 14
    PushButton 177, 42, 50, 14, "&Help"
    '$CStrings Restore
  End Dialog
  Dim dlg As newdlg
  button = Dialog(dlg)
End Sub
```

```

Sub ListFiles(str1$)
  DlgText 1,str1$
  x=0
  Redim exestr$(x)
  directory=Dir$("c:\windows\" & str1$,16)
  If directory<>"" then
    Do
      exestr$(x)=LCase$(directory)
      x=x+1
      Redim Preserve exestr$(x)
      directory=Dir
    Loop Until directory=""
  End If
  DlgListBoxArray 2,exestr$()
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
  Select Case action
    Case 1
      str1$="*.exe"           'dialog box initialized
      ListFiles str1$
    Case 2                   'button or control value changed
      If DlgControlId(identifier$) = 4 Then
        If DlgText(4)="All Files (*.*)" then
          str1$="*.*"
        Else
          str1$="*.exe"
        End If
      End If
      ListFiles str1$
    End If
    Case 3                   'text or combo box changed
      str1$=DlgText$(1)
      ListFiles str1$
    Case 4                   'control focus changed
    Case 5                   'idle
  End Select
End Function

```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Function, **DlgEnable** Statement, **DlgFocus** Function, **DlgFocus** Statement, **DlgListBoxArray** Function, **DlgListBoxArray** Statement, **DlgSetPicture**, **DlgText** Statement, **DlgValue** Function, **DlgValue** Statement, **DlgVisible** Function, **DlgVisible** Statement

DlgText Statement

Changes the text associated with a dialog control for the current dialog box.

Syntax

DlgText *id*, *text*\$

Where...	Is...
<i>id</i>	The control ID for a dialog control.
<i>text</i> \$	The text to use for the dialog control.

Remarks

If the dialog control is a text box or a combo box, **DlgText** sets the text that appears in the text box. If it is a list box, a string equal to *text*\$ or beginning with *text*\$ is selected. If the dialog control is a text control, **DlgText** sets it to *text*\$. If the dialog control is a command button, option button, option group, or a check box, the statement sets its label. The **DlgText** statement does not change the identifier associated with the control. Use **DlgText** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

Uses the **DlgText** statement to display the list of files in the Filename list box.

```
Declare Sub ListFiles(str1$)
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub main
  Dim identifier$
  Dim action as Integer
  Dim suppvalue as Integer
  Dim filetypes as String
  Dim exestr$()
  Dim button as Integer
  Dim x as Integer
  Dim directory as String
  filetypes="Program files (*.exe)+Chr$(9)+"All Files (*.*)"
  Begin Dialog newdlg 230, 145, "Open", .FileDlgFunction
    '$CStrings Save
    Text 8, 6, 60, 11, "&Filename:"
    TextBox 8, 17, 76, 13, .TextBox1
    ListBox 9, 36, 75, 61, exestr$(), .ListBox1
    Text 8, 108, 61, 9, "List Files of &Type:"
    DropListBox 7, 120, 78, 30, filetypes, .DropListBox1
    Text 98, 7, 43, 10, "&Directories:"
    Text 98, 20, 46, 8, "c:\windows"
    ListBox 99, 34, 66, 66, "", .ListBox2
    Text 98, 108, 44, 8, "Dri&ves:"
    DropListBox 98, 120, 68, 12, "", .DropListBox2
    OKButton 177, 6, 50, 14
    CancelButton 177, 24, 50, 14
    PushButton 177, 42, 50, 14, "&Help"
    '$CStrings Restore
  End Dialog
  Dim dlg As newdlg
  button = Dialog(dlg)
End Sub
Sub ListFiles(str1$)
  DlgText 1, str1$
  x=0
  Redim exestr$(x)
  directory=Dir$("c:\windows\" & str1$,16)
  If directory<>"" then
    Do
      exestr$(x)=LCase$(directory)
      x=x+1
      Redim Preserve exestr$(x)
      directory=Dir
    Loop Until directory=""
  End If
End Sub
```

```
    DlgListBoxArray 2,exestr$()
End Sub
Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
    Case 1
        str1$="*.exe"           'dialog box initialized
        ListFiles str1$
    Case 2
        'button or control value changed
        If DlgControlId(identifier$) = 4 Then
            If DlgText(4)="All Files (*.*)" then
                str1$="*.*"
            Else
                str1$="*.exe"
            End If
            ListFiles str1$
        End If
    Case 3
        'text or combo box changed
        str1$=DlgText$(1)
        ListFiles str1$
    Case 4
        'control focus changed
    Case 5
        'idle
    End Select
End Function
```

See Also

BeginDialog...End Dialog, DlgControlID Function, DlgEnable Function, DlgEnable Statement, DlgFocus Function, DlgFocus Statement, DlgListBoxArray Function, DlgListBoxArray Statement, DlgSetPicture, DlgText Function, DlgValue Function, DlgValue Statement, DlgVisible Function, DlgVisible Statement

DlgValue Function

Returns a numeric value for the state of a dialog control for the current dialog box.

Syntax

DlgValue (*Id*)

Where...

Id The control ID for a dialog control.

Is...

Remarks

The values returned depend on the type of dialog control:

Control	Value returned
Checkbox	1 = Selected, 0=Cleared, -1=Grayed
Option Group	0 = 1st button selected, 1 = 2nd button selected, etc.
Listbox	0 = 1st item, 1= 2nd item, etc.
Combobox	0 = 1st item, 1 = 2nd item, etc.
Text, Textbox, Button	Error occurs

Use **DlgValue** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example changes the picture in the dialog box if the check box is selected and changes the picture to its original bitmap if the check box is turned off.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgSetPicture Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        Picture 43, 28, 49, 31, "C:\WINDOWS\CIRCLES.BMP", 0
        CheckBox 30, 8, 62, 15, "Change Picture", .CheckBox1
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
        Case 2 'user changed control or clicked a button
            If DlgControlID(identifier$)=3 then
                If DlgValue(3)=1 then
                    DlgSetPicture 2, "C:\WINDOWS\TILES.BMP",0
                Else
                    DlgSetPicture 2, "C:\WINDOWS\CIRCLES.BMP",0
                End If
            End If
        End If
    End Select
End Function

```

```
End Select  
End Function
```

See Also

BeginDialog...End Dialog, DigControlID Function, DigEnable Function, DigEnable Statement, DigFocus Function, DigFocus Statement, DigListBoxArray Function, DigListBoxArray Statement, DigSetPicture, DigText Function, DigText Statement, DigValue Statement, DigVisible Function, DigVisible Statement

DlgValue Statement

Changes the value associated with the dialog control for the current dialog box.

Syntax

DlgValue *Id*, *value%*

Where...	Is...
<i>Id</i>	The control ID for a dialog control.
<i>value%</i>	The new value for the dialog control.

Remarks

The values you use to set the control depend on the type of the control:

Control	Value Returned
Checkbox	1 = Select, 0=Clear, -1=Gray.
Option Group	0 = Select 1st button, 1 = Select 2nd button.
Listbox	0 = Select 1st item, 1= Select 2nd item, etc.
Combobox	0 = Select 1st item, 1 = Select 2nd item, etc.
Text, Textbox, Button	Error occurs

Use **DlgValue** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays a dialog box with a check box labeled Change Option, and a group box with two option buttons, labeled Option 1 and Option 2. When the user clicks Change Option, Option 2 is selected.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgValue Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        CheckBox 30, 8, 62, 15, "Change Option", .CheckBox1
        GroupBox 28, 34, 79, 47, "Group"
        OptionGroup .OptionGroup1
            OptionButton 41, 47, 52, 10, "Option 1", .OptionButton1
            OptionButton 41, 62, 58, 11, "Option 2", .OptionButton2
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
        Case 2 'user changed control or clicked a button
            If DlgControlID(identifier$)=2 then
                If DlgValue(2)=1 then

```

```
        DlgValue 4,1
    Else
        DlgValue 4,0
    End If
End If
End Select
End Function
```

See Also

BeginDialog...End Dialog, **DlgControlID** Function, **DlgEnable** Function, **DlgEnable** Statement, **DlgFocus** Function, **DlgFocus** Statement, **DlgListBoxArray** Function, **DlgListBoxArray** Statement, **DlgSetPicture**, **DlgText** Function, **DlgText** Statement, **DlgValue** Function, **DlgVisible** Function, **DlgVisible** Statement

DlgVisible Function

Returns -1 if a dialog control is visible, 0 if it is hidden.

Syntax

DlgVisible (*Id*)

Where...	Is...
<i>Id</i>	The control ID for a dialog control.

Remarks

Use **DlgVisible** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays Option 2 in the Group box if the user selects the check box labeled “Show Option 2”. If the user selects the box again, Option 2 is hidden.

```
Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgVisible Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        CheckBox 30, 8, 62, 15, "Show Option 2", .CheckBox1
        GroupBox 28, 34, 79, 47, "Group"
        OptionGroup .OptionGroup1
            OptionButton 41, 47, 52, 10, "Option 1", .OptionButton1
            OptionButton 41, 62, 58, 11, "Option 2", .OptionButton2
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

Function FileDlgFunction(identifier$, action, suppvalue)
    Select Case action
        Case 1
            DlgVisible 6,0
        Case 2 'user changed control or clicked a button
            If DlgControlID(identifier$)=2 then
                If DlgVisible(6)<>1 then
                    DlgVisible 6
                End If
            End If
    End Select
End Function
```

See Also

BeginDialog...End Dialog, **DlgControlID Function**, **DlgEnable Function**, **DlgEnable Statement**, **DlgFocus Function**, **DlgFocus Statement**, **DlgListBoxArray Function**, **DlgListBoxArray Statement**, **DlgSetPicture**, **DlgText Function**, **DlgText Statement**, **DlgValue Function**, **DlgValue Statement**, **DlgVisible Statement**

DlgVisible Statement

Hides or displays a dialog control for the current dialog box.

Syntax

DlgVisible *Id* [, *mode*]

Where...	Is...
<i>Id</i>	The control ID for a dialog control.
<i>mode</i>	Value to use to set the dialog control state: 1 = Display a previously hidden control. 0 = Hide the control.

Remarks

If you omit the *mode*, the dialog box state is toggled between visible and hidden.

Use **DlgVisible** only while a dialog box is running. See the “Begin Dialog...End Dialog Statement” on page 34 for more information.

Example

This example displays Option 2 in the Group box if the user selects the check box. labeled “Show Option 2”. If the user selects the box again, Option 2 is hidden.

```

Declare Function FileDlgFunction(identifier$, action, suppvalue)
Sub Main
    Dim button as integer
    Dim identifier$
    Dim action as Integer
    Dim suppvalue as Integer
    Begin Dialog newdlg 186, 92, "DlgVisible Example", .FileDlgFunction
        OKButton 130, 6, 50, 14
        CancelButton 130, 23, 50, 14
        CheckBox 30, 8, 62, 15, "Show Option 2", .CheckBox1
        GroupBox 28, 34, 79, 47, "Group"
        OptionGroup .OptionGroup1
            OptionButton 41, 47, 52, 10, "Option 1", .OptionButton1
            OptionButton 41, 62, 58, 11, "Option 2", .OptionButton2
    End Dialog
    Dim dlg As newdlg
    button = Dialog(dlg)
End Sub

```

```
Function FileDlgFunction(identifier$, action, suppvalue)
  Select Case action
    Case 1
      DlgVisible 6,0
    Case 2 'user changed control or clicked a button
      If DlgControlID(identifier$)=2 then
        If DlgVisible(6)<>1 then
          DlgVisible 6
        End If
      End If
    End Select
  End Function
```

See Also

BeginDialog...End Dialog, DlgControlID Function, DlgEnable Function, DlgEnable Statement, DlgFocus Function, DlgFocus Statement, DlgListBoxArray Function, DlgListBoxArray Statement, DlgSetPicture, DlgText Function, DlgText Statement, DlgValue Function, DlgVisible Function

Do...Loop Statement

Repeats a series of program lines as long as (or until) an expression is TRUE.

Syntax A

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

Loop

Syntax B

```
Do
    [ statementblock ]
    [ Exit Do ]
    [ statementblock ]

Loop [ { While | Until } condition ]
```

Where...	Is...
<i>Condition</i>	Any expression that evaluates to TRUE (nonzero) or FALSE (0).
<i>Statementblock</i>	Program lines to repeat while (or until) <i>condition</i> is TRUE.

Remarks

When an **Exit Do** statement is executed, control goes to the statement after the Loop statement. When used within a nested loop, an **Exit Do** statement moves control out of the immediately enclosing loop.

Example

This example lists the contents of the diskette in drive A.

```
Sub main
Dim msgret
    Dim directory, count
    Dim x, msgtext
    Dim A()
    msgret=MsgBox("Insert a disk in drive A.")
    count=1
    ReDim A(100)
    directory=Dir ("A:\*.*")
    Do While directory<>""
        A(count)=directory
        count=count+1
        directory=Dir
    Loop
    msgtext="Directory of drive A:\ is:" & Chr(10)
    For x=1 to count
        msgtext=msgtext & A(x) & Chr(10)
    Next x
    MsgBox msgtext
End Sub
```

See Also

Exit, For...Next, Stop, While...Wend

DoEvents Statement

Yields execution to Windows for processing operating system events.

Syntax

DoEvents

Remarks

DoEvents 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 BSL yields control to the operating system at regular intervals, **DoEvents** should only be used to force BSL to allow other applications to run at a known point in the program.

Example

This example activates the Windows 95 Phone Dialer application, dials the number and then allows the operating system to process events.

```
Sub main
    Dim phonenumber, msgtext
    Dim x
    phonenumber=InputBox("Type telephone number to call:")
    x=Shell("Dialer.exe",1)
    For i = 1 to 5
        DoEvents
    Next i
    AppActivate "Phone Dialer"
    SendKeys phonenumber & "{Enter}",1
    msgtext="Dialing..."
    MsgBox msgtext
    DoEvents
End Sub
```

See Also

AppActivate, SendKeys, Shell

DropComboBox Statement

Creates a combination of a drop-down list box and a text box.

Syntax A

DropComboBox *x* , *y* , *dx* , *dy* , *text\$* , *.field*

Syntax B

DropComboBox *x* , *y* , *dx* , *dy* , *stringarray\$()* , *.field*

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the list box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the combo box in which the user enters or selects text.
<i>text\$</i>	A string containing the selections for the combo box.
<i>stringarray\$</i>	An array of dynamic strings for the selections in the combo box.
<i>.field</i>	The name of the dialog-record field that will hold the text string entered in the text box or chosen from the list box.

Remarks

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 the "Begin Dialog...End Dialog Statement" on page 34 for more information.

The *text\$* argument 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 the following example:

```
dimname = "listchoice"+Chr$(9)+"listchoice"+Chr$(9)+"listchoice" ...
```

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. The *field* argument is also used by the dialog statements that act on this control.

You use a drop combo box when you want the user to be able to edit the contents of the list box (such as filenames or their paths). You use a drop list box when the items in the list should remain unchanged.

Use the **DropComboBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a drop combo box and the OK and Cancel buttons.

```
Sub main
  Dim cchoices as String
  On Error Resume Next
  cchoices="All"+Chr$(9)+"Nothing"
  Begin Dialog UserDialog 180, 95, "BSL Dialog Box"
    ButtonGroup .ButtonGroup1
    Text 9, 3, 69, 13, "Filename:", .Text1
    DropComboBox 9, 17, 111, 41, cchoices, .ComboBox1
    OKButton 131, 8, 42, 13
    CancelButton 131, 27, 42, 13
  End Dialog
  Dim mydialogbox As UserDialog
  Dialog mydialogbox
  If Err=102 then
    MsgBox "You pressed Cancel."
  Else
    MsgBox "You pressed OK."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

DropListBox Statement

Creates a drop-down list of choices.

Syntax A

DropListBox *x* , *y* , *dx* , *dy* , *text\$* , *.field*

Syntax B

DropListBox *x* , *y* , *dx* , *dy* , *stringarray\$()* , *.field*

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the list box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the combo box in which the user enters or selects text.
<i>text\$</i>	A string containing the selections for the combo box.
<i>stringarray\$</i>	An array of dynamic strings for the selections in the combo box.
<i>.field</i>	The name of the dialog-record field that will hold the text string entered in the text box or chosen from the list box.

Remarks

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 the "Begin Dialog...End Dialog Statement" on page 34 for more information.

The *text\$* argument 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 the following example:

```
dimname = "listchoice"+Chr$(9)+"listchoice"+Chr$(9)+"listchoice" ...
```

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. The *field* argument is also used by the dialog statements that act on this control.

A drop list box is different from a list box. The drop list box only displays its list when the user selects it; the list box also displays its entire list in the dialog box.

Use the **DropListBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a drop list box and the OK and Cancel buttons.

```
Sub main
  Dim DropListBox1() as String
  ReDim DropListBox1(3)
  For x=0 to 2
    DropListBox1(x)=Chr(65+x) & ":"
  Next x
  Begin Dialog UserDialog 186, 62, "BSL Dialog Box"
    Text 8, 4, 42, 8, "Drive:", .Text3
    DropListBox 8, 16, 95, 44, DropListBox1(), .DropListBox1
    OKButton 124, 6, 54, 14
    CancelButton 124, 26, 54, 14
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog Statement, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, DropComboBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Environ Function

Returns the string setting for a keyword in the operating system's environment table.

Syntax A

Environ[\$](*environment-string*\$)

Syntax B

Environ[\$](*numeric expression*%)

Where...	Is...
<i>Environment-string</i> \$	The name of a keyword in the operating system environment.
<i>Numeric expression</i> %	A number for the position of the string in the environment table. (1st, 2nd, 3rd, etc.)

Remarks

If you use the *environment-string*\$ parameter, enter it in uppercase, or **Environ** returns a null string (""). The return value for Syntax A is the string associated with the keyword requested.

If you use the *numeric expression*% parameter, the numeric expression is automatically rounded to a whole number, if necessary. The return value for Syntax B is a string in the form "keyword=value."

Environ returns a null string if the specified argument cannot be found.

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).

Example

This example lists all the strings from the operating system environment table.

```

Sub main
  Dim str1(100)
  Dim msgtext
  Dim count, x
  Dim newline
  newline=Chr(10)
  x=1
  str1(x)= Environ(x)
  Do While Environ(x)<>""
    str1(x)= Environ(x)
    x=x+1
    str1(x)=Environ(x)
  Loop
  msgtext="The Environment Strings are:" & newline & newline
  count=x
  For x=1 to count
    msgtext=msgtext & str1(x) & newline
  Next x
  MsgBox msgtext
End Sub

```

Eof Function

Returns the value -1 if the end of the specified open file has been reached, 0 otherwise.

Syntax

Eof(*filename%*)

Where...	Is...
----------	-------

<i>filename%</i>	An integer expression identifying the open file to use.
------------------	---

Remarks

See the “Open Statement” on page 225 for more information about assigning numbers to files when they are opened.

Example

This example uses the **Eof** function to read records from a Random file, using a **Get** statement. The **Eof** function keeps the **Get** statement from attempting to read beyond the end of the file. The subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram.

```
Declare Sub createfile()
Sub main
  Dim acctno
  Dim msgtext as String
  newline=Chr(10)
  Call createfile
  Open "C:\temp001" For Input As #1
  msgtext="The account numbers are:" & newline
  Do While Not Eof(1)
    Input #1,acctno
    msgtext=msgtext & newline & acctno & newline
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub
Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also

Get, Input Function, Input Statement, Line Input, Loc, Lof, Open

Erase Statement

Reinitializes the contents of a fixed array or frees the storage associated with a dynamic array.

Syntax

Erase *Array* [, *Array*]

Where...	Is...
<i>Array</i>	The name of the array variable to re-initialize.

Remarks

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, for example, numeric members have their value set to zero, etc.
object	Each element is set to the special value Nothing

Example

This example prompts for a list of item numbers to put into an array and clears array if the user wants to start over.

```

Sub main
  Dim msgtext
  Dim inum(100) as Integer
  Dim x, count
  Dim newline
  newline=Chr(10)
  x=1
  count=x
  inum(x)=0
  Do
    inum(x)=InputBox("Enter item #" & x & " (99=start over;0=end):")
    If inum(x)=99 then
      Erase inum()
      x=0
    ElseIf inum(x)=0 then
      Exit Do
    End If
    x=x+1
  Loop
  count=x-1
  msgtext="You entered the following numbers:" & newline
  For x=1 to count
    msgtext=msgtext & inum(x) & newline
  Next x
  MsgBox msgtext
End Sub

```

See Also

Dim, ReDim, LBound, UBound

Erl Function

Returns the line number where an error was trapped.

Syntax

Erl

Remarks

If you use a **Resume** or **On Error** statement after **Erl**, the return value for **Erl** is reset to 0. To maintain the value of the line number returned by **Erl**, assign it to a variable.

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

Example

This example prints the error number using the **Err** function and the line number using the **Erl** statement if an error occurs during an attempt to open a file. Line numbers are automatically assigned, starting with 1, which is the **Sub main** statement.

```
Sub main
  Dim msgtext, userfile
  On Error GoTo Debugger
  msgtext="Enter the filename to use:"
  userfile=InputBox$(msgtext)
  Open userfile For Input As #1
  MsgBox "File opened for input."
  ' ....etc....
  Close #1
done:
  Exit Sub
Debugger:
  msgtext="Error number " & Err & " occurred at line: " & Erl
  MsgBox msgtext
  Resume done
End Sub
```

See Also

Err Function, Err Statement, Error Function, Error Statement, On Error, Resume, Trappable Errors

Err Function

Returns the run-time error code for the last error trapped.

Syntax

Err

Remarks

If you use a **Resume** or **On Error** statement after **Erl**, the return value for **Err** is reset to 0. To maintain the value of the line number returned by **Erl**, assign it to a variable.

The value of the **Err** function can be set directly through the **Err** statement, and indirectly through the **Error** statement.

See “Error Codes” on page 475 for more information.

Example

This example prints the error number using the **Err** function and the line number using the **Erl** statement if an error occurs during an attempt to open a file. Line numbers are automatically assigned, starting with 1, which is the **Sub main** statement.

```
Sub main
    Dim msgtext, userfile
    On Error GoTo Debugger
    msgtext="Enter the filename to use:"
    userfile=InputBox$(msgtext)
    Open userfile For Input As #1
    MsgBox "File opened for input."
'    ....etc....
    Close #1
done:
    Exit Sub
Debugger:
    msgtext="Error number " & Err & " occurred at line: " & Erl
    MsgBox msgtext
    Resume done
End Sub
```

See Also

Erl, Err Statement, Error Function, Error Statement, On Error, Resume, Trappable Errors

Err Statement

Sets a run-time error code.

Syntax

Err = *n%*

Where...	Is...
<i>n%</i>	An integer expression for the error code (between 1 and 32,767) or 0 for no run-time error.

Remarks

The **Err** statement is used to send error information between procedures.

Example

This example generates an error code of 10000 and displays an error message if a user does not enter a customer name when prompted for it. It uses the **Err** statement to clear any previous error codes before running the loop the first time and it also clears the error to allow the user to try again.

```
Sub main
  Dim custname as String
  On Error Resume Next
  Do
    Err=0
    custname=InputBox$("Enter customer name:")
    If custname="" then
      Error 10000
    Else
      Exit Do
    End If
    Select Case Err
      Case 10000
        MsgBox "You must enter a customer name."
      Case Else
        MsgBox "Undetermined error. Try again."
    End Select
  Loop Until custname<>""
  MsgBox "The name is: " & custname
End Sub
```

See Also

[Erl](#), [Err Function](#), [Error Function](#), [Error Statement](#), [On Error](#), [Resume](#), [Trappable Errors](#)

Error Function

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

Syntax

Error[\$] [(*errornumber%*)]

Where...

Is...

errornumber% An integer between 1 and 32,767 for the error code.

Remarks

If this argument is omitted, BSL returns the error message for the run-time error that has occurred most recently.

If no error message is found to match the errorcode, "" (a null string) is returned.

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).

Trappable Errors are listed in "Appendix 1: Trappable Errors" on page 475.

Example

This example prints the error number, using the **Err** function, and the text of the error, using the **Error\$** function, if an error occurs during an attempt to open a file.

```
Sub main
  Dim msgtext, userfile
  On Error GoTo Debugger
  msgtext="Enter the filename to use:"
  userfile=InputBox$(msgtext)
  Open userfile For Input As #1
  MsgBox "File opened for input."
'   ....etc....
  Close #1
done:
  Exit Sub
Debugger:
  msgtext="Error " & Err & ": " & Error$
  MsgBox msgtext
  Resume done
End Sub
```

See Also

Err, **Err** Function, **Err** Statement, **Error** Statement, **On Error**, **Resume**, **Trappable Errors**

Error Statement

Simulates the occurrence of a BSL or user-defined error.

Syntax

Error *errornumber%*

Where... Is...

errornumber% An integer between 1 and 32,767 for the error code.

Remarks

If an *errornumber%* is one that BSL 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 BSL error codes. To help ensure that non-BSL 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, BSL produces an error message and halts program execution. If an **Error** statement specifies an error code not used by BSL, the message "User-defined error" is displayed.

Example

This example generates an error code of 10000 and displays an error message if a user does not enter a customer name when prompted for it.

```
Sub main
  Dim custname as String
  On Error Resume Next
  Do
    Err=0
    custname=InputBox$("Enter customer name:")
    If custname="" then
      Error 10000
    Else
      Exit Do
    End If
  Select Case Err
    Case 10000
      MsgBox "You must enter a customer name."
    Case Else
      MsgBox "Undetermined error. Try again."
  End Select
  Loop Until custname<>""
  MsgBox "The name is: " & custname
End Sub
```

See Also

[Erl](#), [Err Function](#), [Err Statement](#), [Error Function](#), [On Error, Resume](#), [Trappable Errors](#)

Exit Statement

Terminates Loop statements or transfers control to a calling procedure.

Syntax

Exit {Do | For| Function | Sub}

Remarks

Use **Exit Do** inside a **Do...Loop** statement. Use **Exit For** inside a **For...Next** statement. When the **Exit** statement is executed, control transfers to the statement after the **Loop** or **Next** statement. When used within a nested loop, an **Exit** statement moves control out of the immediately enclosing loop.

Use **Exit Function** inside a **Function...End Function** procedure. Use **Exit Sub** inside a **Sub...End Sub** procedure.

Example

This example uses the **On Error** statement to trap run-time errors. If there is an error, the program execution continues at the label "Debugger". The example uses the **Exit** statement to skip over the debugging code when there is no error.

```
Sub main
    Dim msgtext, userfile
    On Error GoTo Debugger
    msgtext="Enter the filename to use:"
    userfile=InputBox$(msgtext)
    Open userfile For Input As #1
    MsgBox "File opened for input."
'   ....etc....
    Close #1
done:
    Exit Sub
Debugger:
    msgtext="Error " & Err & ": " & Error$
    MsgBox msgtext
    Resume done
End Sub
```

See Also

Do...Loop, For...Nextfor, Function...End Function, Stop, Sub...End Sub

Exp Function

Returns the value e (the base of natural logarithms) raised to a power.

Syntax

Exp(*number*)

Where...	Is...
<i>number</i>	The exponent value for e .

Remarks

If the variable to contain the return value has a data type Integer, Currency, or Single, the return value is a single-precision value. If the variable has a data type of Long, Variant, or Double, the value returned is a double-precision number.

The constant e is approximately 2.718282.

Example

This example estimates the value of a factorial of a number entered by the user. A factorial (notated with an exclamation mark, !) is the product of a number and each integer between it and the number 1. For example, 5 factorial, or 5!, is the product of $5*4*3*2*1$, or the value 120.

```
Sub main
    Dim x as Single
    Dim msgtext, PI
    Dim factorial as Double
    PI=3.14159
    i: x=InputBox("Enter an integer between 1 and 88: ")
    If x<=0 then
        Exit Sub
    ElseIf x>88 then
        MsgBox "The number you entered is too large. Try again."
        Goto i
    End If
    factorial=Sqr(2*PI*x)*(x^x/Exp(x))
    msgtext="The estimated factorial is: " & Format(factorial, "Scientific")
    MsgBox msgtext
End Sub
```

See Also

Abs, Fix, Int, Log, Rnd, Sgn, Sqr

FileAttr Function

Returns the file mode or the operating system handle for the open file.

Syntax

FileAttr(*filenumber%* , *returntype*)

Where...	Is...
<i>filenumber%</i>	An integer expression identifying the open file to use.
<i>returntype</i>	1=Return file mode, 2=Return operating system handle

Remarks

The argument *filenumber%* is the number used in the **Open** statement to open the file.

The table below lists the return values and corresponding file modes if *returntype* is 1:

Value	Mode
1	Input
2	Output
8	Append

Example

This example closes an open file if it is open for Input or Output. If open for Append, it writes a range of numbers to the file. The second subprogram, CREATEFILE, creates the file and leaves it open.

```
Declare Sub createfile()
Sub main
  Dim filemode as Integer
  Dim attrib as Integer
  Call createfile
  attrib=1
  filemode=FileAttr(1,attrib)
  If filemode=1 or 2 then
    MsgBox "File was left open. Closing now."
    Close #1
  Else
    For x=11 to 15
      Write #1, x
    Next x
    Close #1
  End If
  Kill "C:\TEMP001"
End Sub
```

```
Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
End Sub
```

See Also

GetAttr, Open, SetAttr

FileCopy Statement

Copies a file.

Syntax**FileCopy** *source\$* , *destination\$***Where...****Is...**

<i>source\$</i>	A string expression for the name (and path) of the file to copy.
<i>destination\$</i>	A string expression for the name (and path) for the copied file.

Remarks

Wildcards (* or ?) are not allowed for either the *source\$* or *destination\$*. The *source\$* file cannot be copied if it is opened by BSL for anything other than **Read** access.

Example

This example copies one file to another. Both filenames are specified by the user.

```

Sub main
  Dim oldfile, newfile
  On Error Resume Next
  oldfile= InputBox("Copy which file?")
  newfile= InputBox("Copy to?")
  FileCopy oldfile,newfile
  If Err<>0 then
    msgtext="Error during copy. Rerun program."
  Else
    msgtext="Copy successful."
  End If
  MsgBox msgtext
End Sub

```

See Also

FileAttr, FileDateTime, GetAttr, Kill, Name

FileDateTime Function

Returns the last modification date and time for the specified file.

Syntax

FileDateTime(*pathname\$*)

Where...	Is...
----------	-------

<i>pathname\$</i>	A string expression for the name of the file to query.
-------------------	--

Remarks

Pathname\$ can contain path and disk information, but cannot include wildcards (* and ?).

Example

This example writes data to a file if it has not been saved within the last two minutes.

```
Sub main
    Dim tempfile
    Dim filetime, curtime
    Dim msgtext
    Dim acctno(100) as Single
    Dim x, I
    tempfile="C:\TEMP001"
    Open tempfile For Output As #1
    filetime=FileDateTime(tempfile)
    x=1
    I=1
    acctno(x)=0
    Do
        curtime=Time
        acctno(x)=InputBox("Enter an account number (99 to end):")
        If acctno(x)=99 then
            For I=1 to x-1
                Write #1, acctno(I)
            Next I
            Exit Do
        ElseIf (Minute(filetime)+2)<=Minute(curtime) then
            For I=I to x
                Write #1, acctno(I)
            Next I
        End If
        x=x+1
    Loop
    Close #1
    x=1
    msgtext="Contents of C:\TEMP001 is:" & Chr(10)
    Open tempfile for Input as #1
    Do While Eof(1)<>-1
        Input #1, acctno(x)
        msgtext=msgtext & Chr(10) & acctno(x)
        x=x+1
    Loop
    MsgBox msgtext
```

```

    Close #1
    Kill "C:\TEMP001"
End Sub

```

See Also

FileLen, GetAttr

FileLen Function

Returns the length of the specified file.

Syntax

FileLen(*pathname\$*)

Where...	Is...
<i>pathname\$</i>	A string expression that contains the name of the file to query.

Remarks

Pathname\$ can contain path and disk information, but cannot include wildcards (* and ?).

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

Example

This example returns the length of a file.

```

Sub main
    Dim length as Long
    Dim userfile as String
    Dim msgtext
    On Error Resume Next
    msgtext="Enter a filename:"
    userfile=InputBox(msgtext)
    length=FileLen(userfile)
    If Err<>0 then
        msgtext="Error occurred. Rerun program."
    Else
        msgtext="The length of " & userfile & " is: " & length
    End If
    MsgBox msgtext
End Sub

```

See Also

FileDateTime, GetAttr, Lof

Fix Function

Returns the integer part of a number.

Syntax

Fix (*number*)

Where...

Is...

number Any valid numeric expression.

Remarks

The return value's data type matches the type of the numeric expression. This includes variant expressions, unless the numeric expression is a string (vartype 8) that evaluates to a number, in which case the data type for its return value is vartype 5 (double). If the numeric expression is vartype 0 (empty), the data type for the return value is vartype 3 (long).

For both positive and negative *numbers*, **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.

Example

This example returns the integer portion of a number provided by the user.

```
Sub main
    Dim usernum
    Dim intvalue
    usernum=InputBox("Enter a number with decimal places:")
    intvalue=Fix(usernum)
    MsgBox "The integer portion of " & usernum & " is: " & intvalue
End Sub
```

See Also

Abs, **Clnt**, **Exp**, **Int**, **Log**, **Rnd**, **Sgn**, **Sqr**

For...Next Statement

Repeats a series of program lines a fixed number of times.

Syntax

For *counter* = *start* TO *end* [STEP *increment*]

[*statementblock*]

[Exit For]

[*statementblock*]

Next [*counter*]

Where...	Is...
<i>counter</i>	A numeric variable for the loop counter.
<i>start</i>	The beginning value of the counter.
<i>end</i>	The ending value of the counter.
<i>increment</i>	The amount by which the counter is changed each time the loop is run. (The default is one.)
<i>statementblock</i>	Basic functions, statements, or methods to be executed.

Remarks

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. BSL compares the sign of (*start*-*end*) with the sign of *increment*. If the signs are the same, and *end* does not equal *start*, the **For...Next** loop is started. 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 <i>counter</i> is less than or equal to <i>end</i> , the Step value is added to <i>counter</i> . Control returns to the statement after the For statement and the process repeats. If <i>counter</i> is greater than <i>end</i> , the loop is exited; execution resumes with the statement following the Next statement.
Negative	The loop repeats until <i>counter</i> is less than <i>end</i> .
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 can 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, BSL 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
```

Example

This example calculates the factorial of a number. A factorial (notated with an exclamation mark, !) is the product of a number and each integer between it and the number 1. For example, 5 factorial, or 5!, is the product of $5*4*3*2*1$, or the value 120.

```
Sub main
    Dim number as Integer
    Dim factorial as Double
    Dim msgtext
    number=InputBox("Enter an integer between 1 and 170:")
    If number<=0 then
        Exit Sub
    End If
    factorial=1
    For x=number to 2 step -1
        factorial=factorial*x
    Next x
    Rem If number<= 35, then its factorial is small enough
    Rem to be stored as a single-precision number
    If number<35 then
        factorial=CSng(factorial)
    End If
    msgtext="The factorial of " & number & " is: " & factorial
    MsgBox msgtext
End Sub
```

See Also

Do...Loop, Exit, While...Wend

Format Function

Returns a formatted string of an expression based on a given format.

Syntax

Format[\$](*expression* [, *format*])

Where...	Is...
<i>expression</i>	The value to be formatted. It can be a number, variant, or string.
<i>format</i>	A string expression representing the format to use. Select one of the topics below for a detailed description of format strings.

Remarks

Format formats the *expression* as a number, date, time, or string depending upon the *format* argument. 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). As with any string, you must enclose the *format* argument in quotation marks ("").

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

Both numeric values and variants can 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.

Strings are formatted by transferring one character at a time from the input *expression* to the output string. For more information, see the Related Topic below:

Example

This example calculates the square root of 2 as a double-precision floating point value and displays it in scientific notation.

```
Sub main
    Dim value
    Dim msgtext
    value=Cdbl(Sqr(2))
    msgtext= "The square root of 2 is: " & Format(Value,"Scientific")
    MsgBox msgtext
End Sub
```

See Also

Asc, **CCur**, **Cdbl**, **Chr**, **Clnt**, **CLng**, **CSng**, **CStr**, **CVar**, **CVDate**, **Str**

Formatting Numbers

The predefined numeric formats with their meanings are as follows:

Format	Description
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.

To create a user-defined numeric format, follow these guidelines:

For a simple numeric format, use one or more digit characters and (optionally) a decimal separator. The two format digit characters provided are 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	Format	Result
1234.56	#	1235
1234.56	###	1234.56
1234.56	##	1234.6
1234.56	#####	1234.56
1234.56	00000.000	01234.560
0.12345	###	.12
0.12345	0.###	0.12

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

Number	Format	Result
1234567.8901	#,###	1,234,567.89
1234567.8901	#,#####	1,234,567.8901

Note: Although a comma and period are used in the **format** to denote separators for thousands and decimals, the output string will contain the appropriate character, based upon the current international settings for your machine.

Numbers can be scaled either by inserting one or more commas before the decimal separator or by including a percent sign in the *format* 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 *format*.

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

Characters can be inserted into the output string by being included in the *format* specification. The following characters will be automatically inserted in the output string in a location matching their position in the *format* specification:

- + \$ () space : /

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

Number	Format	Result
1234567.89	\$#,0.00	\$1,234,567.89
1234567.89	"TOTAL:" \$#,#.00	TOTAL: \$1,234,567.89
1234	\=>#,\<=<	=>1,234<=

You can use the BSL **\$CSTRINGS** metacommand or the **Chr** function if you need to embed quotation marks in a format specification. The character code for a quotation mark is 34.

Numbers can be formatted in scientific notation by including one of the following exponent strings in the *format* 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. *Format* 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 *format* will cause a sign to always precede the exponent in the output.

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

A numeric *format* 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	Format	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 Dates and Times

As with numeric formats, there are several predefined formats for formatting dates and times:

Format	Description
General Date	If the number has both integer and real parts, display both date and time. (for example, 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 *format* specification contains a series of tokens. Each token is replaced in the output string by its appropriate value.

A complete date can be output using the following tokens:

Token	Output
c	The date time as if the <i>format</i> was: "dddd tttt". See the definitions below.
dddd	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.
tttt	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 *format* 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).

Token	Output
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 *format* 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 can be inserted into the output by enclosing them in double quotes. Any single character can be inserted by preceding it with a backslash, “\”. See “Formatting Numbers” on page 144 for more details.

Formatting Strings

By default, string formatting transfers characters from left to right. The exclamation point, “!”, when added to the *format* specification causes 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 can 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 *format* 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 *format* 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

Returns the lowest unused file number.

Syntax

FreeFile

Remarks

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 that is already in use.

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

Example

This example opens a file and assigns to it the next file number available.

```
Sub main
    Dim filenumber
    Dim filename as String
    filenumber=FreeFile
    filename=InputBox("Enter a file to open: ")
    On Error Resume Next
    Open filename For Input As filenumber
    If Err<>0 then
        MsgBox "Error loading file. Re-run program."
        Exit Sub
    End If
    MsgBox "File " & filename & " opened as number: " & filenumber
    Close #filenumber
    MsgBox "File now closed."
End Sub
```

See Also

Open

Function...End Function Statement

Defines a function procedure.

Syntax

```
[ Static ] [ Private ] Function name [ ( [ Optional ] parameter [ As type ] ... ) ] [ As functype ]
    name= expression
```

End Function

Where...	Is...
<i>name</i>	A function name.
<i>parameter</i>	The argument(s) to pass to the function when it is called.
<i>type</i>	The data type for the function arguments.
<i>func</i> <i>type</i>	The data type for the return value.
<i>name</i> = <i>expression</i>	The expression that sets the return value for the function.

Remarks

The purpose of a function is to produce and return a single value of a specified type. Recursion is supported.

The data type of *name* determines the type of the return value. Use a type character as part of the *name*, or use the *As func**type* clause to specify the data type. If omitted, the default data type is variant. When calling the function, you need not specify the type character.

The *parameters* are specified as a comma-separated list of variable names. The data type of a parameter can be specified by using a type character or by using the *As* clause. Record parameters are declared using an *As* clause and a *type* that has previously been defined using the **Type** statement. Array parameters are indicated by using empty parentheses after the *parameter*. The array dimensions are not specified in the **Function** statement. All references to an array parameter within the body of the function must have a consistent number of dimensions.

You specify the return value for the function name using the *name*=*expression* assignment, where *name* is the name of the function and *expression* evaluates to a return value. If omitted, the value returned is 0 for numeric functions and an empty string ("") for string functions and vartype 0 (Empty) is returned for a return type of variant. The function returns to the caller when the **End Function** statement is reached or when an **Exit Function** statement is executed.

If you declare a parameter as **Optional**, a procedure can omit its value when calling the function. Only parameters with variant data types can be declared as optional, and all optional arguments must appear after all required arguments in the **Function** statement. The function **IsMissing** must be used to check whether an optional parameter was omitted by the user or not. Named parameters are described under the **Call** statement heading, but they can be used when the function is used in an expression as well.

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.

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 with no return value.

Example

This example declares a function that is later called by the main subprogram. The function does nothing but set its return value to 1.

```
Declare Function BSL_exfunction()  
Sub main  
    Dim y as Integer  
    Call BSL_exfunction  
    y=BSL_exfunction  
    MsgBox "The value returned by the function is: " & y  
End Sub  
  
Function BSL_exfunction()  
    BSL_exfunction=1  
End Function
```

See Also

Call, Dim, Global, IsMissing, Option Explicit, Static, Sub...End Sub

FV Function

Returns the future value for a constant periodic stream of cash flows as in an annuity or a loan.

Syntax

FV (*rate* , *nper* , *pmt* , *pv* , *due*)

Where...	Is...
<i>rate</i>	Interest rate per period.
<i>nper</i>	Total number of payment periods.
<i>pmt</i>	Constant periodic payment per period.
<i>pv</i>	Present value or the initial lump sum amount paid (as in the case of an annuity) or received (as in the case of a loan).
<i>due</i>	An integer value for when the payments are due (0=end of each period, 1= beginning of the period).

Remarks

The given interest rate is assumed constant over the life of the annuity. If payments are on a monthly schedule and the annual percentage rate on the annuity or loan is 9%, the *rate* is 0.0075 (.0075=.09/12).

Example

This example finds the future value of an annuity, based on terms specified by the user.

```
Sub main
  Dim aprate, periods
  Dim payment, annuitypv
  Dim due, futurevalue
  Dim msgtext
  annuitypv=InputBox("Enter present value of the annuity: ")
  aprate=InputBox("Enter the annual percentage rate: ")
  If aprate >1 then
    aprate=aprate/100
  End If
  periods=InputBox("Enter the total number of pay periods: ")
  payment=InputBox("Enter the initial amount paid to you: ")
  Rem Assume payments are made at end of month
  due=0
  futurevalue=FV(aprate/12,periods,-payment,-annuitypv,due)
  msgtext= "The future value is: " & Format(futurevalue, "Currency")
  MsgBox msgtext
End Sub
```

See Also

IPmt, **IRR**, **NPV**, **Pmt**, **PPmt**, **PV**, **Rate**

Get Statement

Reads data from a file opened in Random or Binary mode and puts it in a variable.

Syntax

Get [#] *filename%*, [*recnumber&*], *varname*

Where...	Is...
<i>filename%</i>	An integer expression identifying the open file to use.
<i>recnumber&</i>	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.
<i>varname</i>	The name of the variable into which Get reads file data. <i>Varname</i> can be any variable except Object or Array variables (single array elements can be used).

Remarks

For more information about how files are numbered when they are opened, see the “Open Statement” on page 225.

Recnumber& is in the range 1 to 2,147,483,647. If omitted, the next record or byte is read.

Note: The commas before and after the *recnumber&* are required, even if you do not supply a *recnumber&*.

For Random mode, the following rules 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 *varname* is smaller than the record length, the additional data is discarded. If the size of *varname* 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 *varname*.
- 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 *varname*. Note that variants containing strings contain two bytes of data 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 is equal to the length of *varname*.

Example

This example opens a file for Random access, gets its contents, and closes the file again. The second subprogram, CREATEFILE, creates the C:\TEMP001 file used by the main subprogram.

```
Declare Sub createfile()
Sub main
  Dim acctno as String*3
  Dim recno as Long
  Dim msgtext as String
  Call createfile
  recno=1
  newline=Chr(10)
  Open "C:\TEMP001" For Random As #1 Len=3
  msgtext="The account numbers are:" & newline
  Do Until recno=11
    Get #1,recno,acctno
    msgtext=msgtext & acctno
    recno=recno+1
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub

Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also

Open, Put, Type

GetAttr Function

Returns the attributes of a file, directory or volume label.

Syntax

GetAttr(*pathname\$*)

Where...	Is...
<i>pathname\$</i>	A String expression for the name of the file, directory, or label to query.

Remarks

Pathname\$ cannot contain wildcards (* and ?).

The file attributes returned by **GetAttr** are as follows:

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

Example

This example tests the attributes for a file and if it is hidden, changes it to a non-hidden file.

```

Sub main
  Dim filename as String
  Dim attribs, saveattribs as Integer
  Dim answer as Integer
  Dim archno as Integer
  Dim msgtext as String
  archno=32
  On Error Resume Next
  msgtext="Enter name of a file:"
  filename=InputBox(msgtext)
  attribs=GetAttr(filename)
  If Err<>0 then
    MsgBox "Error in filename. Re-run Program."
    Exit Sub
  End If
  saveattribs=attribs
  If attribs>= archno then
    attribs=attribs-archno
  End If
  Select Case attribs
    Case 2,3,6,7
      msgtext=" File: " &filename & " is hidden." & Chr(10)
      msgtext=msgtext & Chr(10) & " Change it?"
      answer=Msgbox(msgtext,308)
      If answer=6 then
        SetAttr filename, saveattribs-2
      End If
    End Case
  End Select
End Sub

```

```
        MsgBox "File is no longer hidden."  
    Exit Sub  
End If  
MsgBox "Hidden file not changed."  
Case Else  
    MsgBox "File was not hidden."  
End Select  
End Sub
```

See Also

FileAttr, SetAttr

GetField Function

Returns a substring from a source string.

Syntax

GetField[\$](*string*\$, *field_number*% , *separator_chars*\$)

Where...	Is...
<i>string</i> \$	A list of fields, divided by separator characters.
<i>field_number</i> %	The number of the field to return, starting with 1.
<i>separator_chars</i> \$	The characters separating each field.

Remarks

Multiple separator characters can be specified. If *field_number* is greater than the number of fields in the string, an empty string ("") is returned.

Note: BSL offers a number of extensions that are not included in Visual Basic.

Example

This example finds the third value in a string, delimited by plus signs (+).

```
Sub main
    Dim teststring,retvalue
    Dim msgtext
    teststring="9+8+7+6+5"
    retvalue=GetField(teststring,3,"+")
    MsgBox "The third field in: " & teststring & " is: " & retvalue
End Sub
```

See Also

Left, **LTrim**, **Mid Function**, **Mid Statement**, **Right**, **RTrim**, **SetField**, **StrComp**, **Trim**

GetObject Function

Returns an OLE2 object associated with the file name or the application name.

Syntax A

GetObject(*pathname*)

Syntax B

GetObject(*pathname* , *class*)

Syntax C

GetObject(, *class*)

Where...	Is...
<i>pathname</i>	The path and file name for the object to retrieve.
<i>class</i>	A string containing the class of the object.

Remarks

Use **GetObject** with the **Set** statement to assign a variable to the object for use in a Basic procedure. The variable used must first be dimensioned as an Object.

Syntax A of **GetObject** accesses an OLE2 object stored in a file. For example, the following two lines dimension the variable, FILEOBJECT as an Object and assign the object file "PAYABLES" to it. PAYABLES is located in the subdirectory SPREDSHT:

```
Dim FileObject As Object
Set FileObject = GetObject("\spredsht\payables")
```

If the application supports accessing component OLE2 objects within the file, you can append an exclamation point and a component object name to the file name, as follows:

```
Dim ComponentObject As Object
Set ComponentObject = GetObject("\spredsht\payables!R1C1:R13C9")
```

Syntax B of **GetObject** accesses an OLE2 object of a particular class that is stored in a file. *Class* uses the syntax: "*appname.objtype*", where *appname* is the name of the application that provides the object, and *objtype* is the type or class of the object. For example:

```
Dim ClassObject As Object
Set ClassObject = GetObject("\spredsht\payables", "turbosht.spreadsheet")
```

The third form of **GetObject** accesses the active OLE2 object of a particular class. For example:

```
Dim ActiveSheet As Object
Set ActiveSheet = GetObject( , "turbosht.spreadsheet")
```

Example

This example displays a list of open files in the software application, Microsoft Visio. It uses the `GetObject` function to access Visio. To see how this example works, you need to start Visio and open one or more documents.

```
Sub main
    Dim visio as Object
    Dim doc as Object
    Dim msgtext as String
    Dim i as Integer, doccount as Integer

'Initialize Visio
    Set visio = GetObject(,"visio.application") ' find Visio
    If (visio Is Nothing) then
        MsgBox "Couldn't find Visio!"
        Exit Sub
    End If

'Get # of open Visio files
    doccount = visio.documents.count          'OLE2 call to Visio
    If doccount=0 then
        msgtext="No open Visio documents."
    Else
        msgtext="The open files are: " & Chr$(13)
        For i = 1 to doccount
            Set doc = visio.documents(i) ' access Visio's document method
            msgtext=msgtext & Chr$(13)& doc.name
        Next i
    End If
    MsgBox msgtext
End Sub
```

See Also

CreateObject, Is, Me, New, Nothing, Object Class, Typeof

Global Statement

Declare Global variables for use in a Basic program.

Syntax

Global *variableName* [As *type*] [,*variableName* [As *type*]] ...

Where...	Is...
<i>variableName</i>	A variable name.
<i>type</i>	The data type for a variable.

Remarks

Global data is shared across all loaded modules. If an attempt is made to load a module that has a global variable declared that has a different data type than an existing global variable of the same name, the module load will fail.

Basic is a strongly typed language: all variables must be given a data type or they will be automatically assigned a type of variant.

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

Regardless of which mechanism you use to declare a global variable, you can choose to use or omit the type character when referring to the variable in the rest of your program. The type suffix is not considered part of the variable name.

The available data types are discussed in the topics below:

Arrays

The available data types for arrays are: numbers, strings, variants and records. Arrays of arrays and dialog box records, 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 that can 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.

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.

Records

Record variables are declared by using an **As** clause and a *type* that 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 can be of any numeric, string, variant or previously-defined record type. See **Type** for details on accessing fields within a record.

You cannot use the **Global** statement to declare a dialog record.

Strings

BSL 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*

Variants

Declare variables as variants when the type of the variable is not known at the start of, or might change during, the procedure. For example, a variant is useful for holding input from a user when valid input can be either text or numbers. Use the following syntax to declare a variant:

Global *variableName* **or**
Global *variableName* **As** *Variant*

Variant variables are initialized to vartype Empty.

Example

This example contains two subroutines that share the variables TOTAL and ACCTNO, and the record GRECORD.

```
Type acctrecord
  acctno As Integer
End Type
Global acctno as Integer
Global total as Integer
Global grecord as acctrecord
Declare Sub createfile
Sub main
  Dim msgtext
  Dim newline as String
  newline=Chr$(10)
  Call createfile
  Open "C:\TEMP001" For Input as #1
  msgtext="The new account numbers are: " & newline
  For x=1 to total
    Input #1, grecord.acctno
    msgtext=msgtext & newline & grecord.acctno
  Next x
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub
Sub createfile
```

```
Dim x
x=1
grecord.acctno=1
Open "C:\TEMP001" For Output as #1
Do While grecord.acctno<>0
    grecord.acctno=InputBox("Enter 0 or new account #" & x & ":")
    If grecord.acctno<>0 then
        Print #1, grecord.acctno
        x=x+1
    End If
Loop
total=x-1
Close #1
End Sub
```

See Also**Const, Dim, Option Base, ReDim, Static, Type**

GoTo Statement

Transfers program control to the label specified.

Syntax

GoTo { *label* }

Where...

Is...

label A name in the first column of a line of code ending with a colon (:).

Remarks

A *label* has the same format as any other Basic name. Reserved words are not valid labels.

GoTo cannot be used to transfer control out of the current Function or Subprogram.

Example

This example displays the date for one week from the date entered by the user. If the date is invalid, the Goto statement sends program execution back to the beginning.

```
Sub main
  Dim str1 as String
  Dim nextweek
  Dim msgtext
i: str1=InputBox$("Enter a date:")
  answer=IsDate(str1)
  If answer=-1 then
    str1=CVDate(str1)
    nextweek=DateValue(str1)+7
    msgtext="One week from the date entered is:"
    msgtext=msgtext & Format(nextweek,"dddddd")
    MsgBox msgtext
  Else
    MsgBox "Invalid date or format. Try again."
    Goto i
  End If
End Sub
```

See Also

Do...Loop, For...Next, If...Then...Else, Select Case, While...Wend

GroupBox Statement

Defines and draws a box that encloses sets of dialog box items, such as option boxes and check boxes.

Syntax

GroupBox *x*, *y*, *dx*, *dy*, *text\$* [, *.id*]

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the group box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the group box.
<i>text\$</i>	A string containing the title for the top border of the group box.
<i>.id</i>	The optional string ID for the groupbox, used by the dialog statements that act on this control.

Remarks

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 the "Begin Dialog...End Dialog Statement" on page 34 for more information.

If *text\$* is wider than *dx*, the additional characters are truncated. If *text\$* is an empty string (""), the top border of the group box will be a solid line.

Use the **GroupBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Note: BSL offers a number of functions that are not included in Visual Basic.

Example

This example creates a dialog box with two group boxes.

```
Sub main
  Begin Dialog UserDialog 242, 146, "Print Dialog Box"
    '$CStrings Save
    GroupBox 115, 14, 85, 57, "Page Range"
    OptionGroup .OptionGroup2
      OptionButton 123, 30, 46, 12, "All Pages", .OptionButton1
      OptionButton 123, 50, 67, 8, "Current Page", .OptionButton2
    GroupBox 14, 12, 85, 76, "Include"
    CheckBox 26, 17, 54, 25, "Pictures", .CheckBox1
    CheckBox 26, 36, 54, 25, "Links", .CheckBox2
    CheckBox 26, 58, 63, 25, "Header/Footer", .CheckBox3
    PushButton 34, 115, 54, 14, "Print"
    PushButton 136, 115, 54, 14, "Cancel"
    '$CStrings Restore
  End Dialog
  Dim mydialog as UserDialog
  Dialog mydialog
End Sub
```

See Also

Begin Dialog...End Dialog, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, Dialog, DropComboBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Hex Function

Returns the hexadecimal representation of a number, as a string.

Syntax

Hex[\$](*number*)

Where...

number

Is...

Any numeric expression that evaluates to a number.

Remarks

If *number* is an integer, the return string contains up to four hexadecimal digits; otherwise, the value will be converted to a **Long** Integer, and the string can contain up to 8 hexadecimal digits.

To represent a hexadecimal number directly, precede the hexadecimal value with **&H**. For example, &H10 equals decimal 16 in hexadecimal notation.

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).

Example

This example returns the hex value for a number entered by the user.

```
Sub main
    Dim usernum as Integer
    Dim hexvalue
    usernum=InputBox("Enter a number to convert to hexidecimal:")
    hexvalue=Hex(usernum)
    MsgBox "The HEX value is: " & hexvalue
End Sub
```

See Also

Oct, Format

Hour Function

Returns the hour of day component (0-23) of a date-time value.

Syntax

Hour(*time*)

Where...

Is...

time

Any numeric or string expression that can evaluate to a date and time.

Remarks

Hour accepts any type of *time* including strings and attempts to convert the input value to a date value. The return value is a variant of vartype 2 (integer). If the value of *time* is Null, a variant of vartype 1 (null) is returned. *Time* is a double-precision value. The numbers to the left of the decimal point denote the date and the decimal value denotes the time (from 0 to .99999). Use the **TimeValue** function to obtain a specific time.

Example

This example extracts just the time from a file's last modification date and time.

```
Sub main
    Dim filename as String
    Dim ftime
    Dim hr, min
    Dim sec
    Dim msgtext as String
i: msgtext="Enter a filename:"
    filename=InputBox(msgtext)
    If filename="" then
        Exit Sub
    End If
    On Error Resume Next
    ftime=FileDateTime(filename)
    If Err<>0 then
        MsgBox "Error in file name. Try again."
        Goto i:
    End If
    hr=Hour(ftime)
    min=Minute(ftime)
    sec=Second(ftime)
    MsgBox "The file's time is: " & hr & ":" & min & ":" & sec
End Sub
```

See Also

DateSerial, DateValue, Day, Minute, Month, Now, Second, Time Function, Time Statement, TimeSerial, TimeValue, Weekday, Year

If...Then...Else

Executes alternative blocks of program code based on one or more expressions.

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

Where...

Is...

<i>condition</i>	Any expression that evaluates to TRUE (non-zero) or FALSE (zero).
<i>then_statement</i>	Any valid single expression.
<i>else_statement</i>	Any valid single expression.
<i>expression</i>	Any expression that evaluates to TRUE (non-zero) or FALSE (zero).
<i>statement_block</i>	0 or more valid expressions, separated by colons (:), or on different lines.

Remarks

When multiple statements are required in either the **Then** or **Else** clauses, use the block version (Syntax B) of the **If** statement.

Example

This example checks time and day of the week, and returns an appropriate message.

```
Sub main
  Dim h, m, m2, w
  h = hour(now)
  If h > 18 then
    m= "Good evening, "
  Elseif h >12 then
    m= "Good afternoon, "
  Else
    m= "Good morning, "
  End If
  w = weekday(now)
  If w = 1 or w = 7 then m2 = "the office is closed." else m2 = "
  please hold for company operator."
  MsgBox m & m2
End Sub
```

See Also

Do...Loop, For...Next, Goto, On...Goto, Select Case, While...Wend

'\$Include Metacommand

Includes statements from the specified file.

Syntax

'\$Include: "*filename*"

Where...	Is...
<i>filename</i>	The name and location of the file to include.

Remarks

It is recommended (although not required) that you use a file extension of .SBH for *filename*.

All metacommands must begin with an apostrophe (') and are recognized by the compiler only if the command starts at the beginning of a line. For compatibility with other versions of Basic, you can enclose the *filename* in single quotation marks (').

If no directory or drive is specified, the compiler will search for *filename* on the source file search path.

Note: BSL offers a number of extensions that are not included in Visual Basic.

Example

This example includes a file containing the list of global variables, called GLOBALS.SBH. For this example to work correctly, you must create the GLOBALS.SBH file with at least the following statement:

```
Dim gtext as String.
```

The Option Explicit statement is included in this example to prevent BSL from automatically dimensioning the variable as a variant.

```
Option Explicit
Sub main
    Dim msgtext as String
    '$Include: "c:\globals.sbh"
    gtext=InputBox("Enter a string for the global variable:")
    msgtext="The variable for the string '"
    msgtext=msgtext & gtext & "' was DIM'ed in GLOBALS.SBH."
    MsgBox msgtext
End Sub
```

See Also

\$CStrings, \$NoCStrings

Input Function

Returns a string containing the characters read from a file.

Syntax

Input[\$](*number%* , [#]*filename%*)

Where...	Is...
<i>number%</i>	The number of characters to be read from the file.
<i>filename%</i>	An integer expression identifying the open file to use.

Remarks

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. The dollar sign, "\$", in the function name is optional. If specified the return type is string. If omitted the function returns a variant of vartype 8 (string). To return a given number of bytes from a file, use the **InputB** function.

Example

This example opens a file and prints its contents to the screen.

```

Sub main
  Dim fname
  Dim fchar()
  Dim x as Integer
  Dim msgtext
  Dim newline
  newline=Chr(10)
  On Error Resume Next
  fname=InputBox("Enter a filename to print:")
  If fname="" then
    Exit Sub
  End If
  Open fname for Input as #1
  If Err<>0 then
    MsgBox "Error loading file. Re-run program."
    Exit Sub
  End If
  msgtext="The contents of " & fname & " is: " & newline & newline
  Redim fchar(Lof(1))
  For x=1 to Lof(1)
    fchar(x)=Input(1,#1)
    msgtext=msgtext & fchar(x)
  Next x
  MsgBox msgtext
  Close #1
End Sub

```

See Also

Get, **Input Statement**, **Line Input**, **Open**, **Write**

Input Statement

Reads data from a sequential file and assigns the data to variables.

Syntax A

Input [#] *filename%* , *variable* [, *variable*]...

Syntax B

Input [*prompt\$*,] *variable* [, *variable*]...

Where...	Is...
<i>filename%</i>	An integer expression identifying the open file to read from.
<i>variable</i>	The variable(s) to contain the value(s) read from the file.
<i>prompt\$</i>	An optional string that prompts for keyboard input.

Remarks

The *filename%* is the number used in the **Open** statement to open the file. The list of *variables* is separated by commas.

If *filename%* is not specified, the user is prompted for keyboard input, either with *prompt\$* or with a "?", if *prompt\$* is omitted.

Example

This example prompts a user for an account number, opens a file, searches for the account number and displays the matching letter for that number. It uses the Input statement to increase the value of x and at the same time get the letter associated with each value. The second subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram.

```
Declare Sub createfile()
Global x as Integer
Global y(100) as String

Sub main
    Dim acctno as Integer
    Dim msgtext
    Call createfile
i: acctno=InputBox("Enter an account number from 1-10:")
    If acctno<1 Or acctno>10 then
        MsgBox "Invalid account number. Try again."
        Goto i:
    End if
    x=1
    Open "C:\TEMP001" for Input as #1
    Do Until x=acctno
        Input #1, x,y(x)
    Loop
        msgtext="The letter for account number " & x & " is: " & y(x)
    Close #1
    MsgBox msgtext
    Kill "C:\TEMP001"
End Sub

Sub createfile()
' Put the numbers 1-10 and letters A-J into a file
    Dim startletter
```

```
Open "C:\TEMP001" for Output as #1
startletter=65
For x=1 to 10
    y(x)=Chr(startletter)
    startletter=startletter+1
Next x
For x=1 to 10
    Write #1, x,y(x)
Next x
Close #1
End Sub
```

See Also**Get, Input Function, Line Input, Open, Write**

InputBox Function

Displays a dialog box containing a prompt and returns a string entered by the user.

Syntax

InputBox[\$](*prompt\$* , [*title\$*] , [*default\$*] , [*xpos%* , *ypos%*])

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

Where...	Is...
<i>prompt\$</i>	A string expression containing the text to show in the dialog box.
<i>title\$</i>	The caption to display in the dialog box's title bar.
<i>default\$</i>	The string expression to display in the edit box as the default response.
<i>xpos%</i> , <i>ypos%</i>	Numeric expressions, specified in dialog box units, that determine the position of the dialog box.

Remarks

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. If either *prompt\$* or *default\$* is omitted, nothing is displayed.

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. If you want 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. If 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 ("").

Example

This example uses InputBox to prompt for a filename and then prints the filename using MsgBox.

```
Sub main
  Dim filename
  Dim msgtext
  msgtext="Enter a filename:"
  filename=InputBox$(msgtext)
  MsgBox "The file name you entered is: " & filename
End Sub
```

See Also

Dialog Boxes, Input Function, Input Statement, MsgBox Function, MsgBox Statement, PasswordBox

InStr Function

Returns the character position of the first occurrence of one string within another string.

Syntax A

InStr([*start%*,] *string1\$*, *string2\$*)

Syntax B

InStr(*start* , *string1\$*, *string2\$*[, *compare*])

Where...	Is...
<i>start%</i>	The position in <i>string1\$</i> to begin the search. (1=first character in string.)
<i>string1\$</i>	The string to search.
<i>string2\$</i>	The string to find.
<i>compare</i>	An integer expression for the method to use to compare the strings. (0=case-sensitive, 1=case-insensitive.)

Remarks

If not specified, the search starts at the beginning of the string (equivalent to a *start%* of 1). *string1\$* and *string2\$* can be of any type. They will be converted to strings.

InStr returns a zero under the following conditions:

1. *start%* is greater than the length of *string2\$*.
2. *string1\$* is a null string.
3. *string2\$* is not found.

If either *string1\$* or *string2\$* is a null variant , **Instr** returns a null variant.

If *string2\$* is a null string (""), **Instr** returns the value of *start%*.

If *compare* is 0, a case-sensitive comparison based on the ANSI character set sequence is performed. If *compare* is 1, a case-insensitive comparison is done based upon the relative order of characters as determined by the settings for your system in the Regional and Language Options of your Windows Control Panel.

To obtain the byte position of the first occurrence of one string within another string, use the **InStrB** function.

Example

This example generates a random string of characters then uses `InStr` to find the position of a single character within that string.

```
Sub main
  Dim x as Integer
  Dim y
  Dim str1 as String
  Dim str2 as String
  Dim letter as String
  Dim randomvalue
  Dim upper, lower
  Dim position as Integer
  Dim msgtext, newline
  upper=Asc("z")
  lower=Asc("a")
  newline=Chr(10)
  For x=1 to 26
    Randomize timer() + x*255
    randomvalue=Int(((upper - (lower+1)) * Rnd) +lower)
    letter=Chr(randomvalue)
    str1=str1 & letter
  'Need to waste time here for fast processors
  For y=1 to 1000
    Next y
  Next x
  str2=InputBox("Enter a letter to find")
  position=InStr(str1,str2)
  If position then
    msgtext="The position of " & str2 & " is: " & position & newline
    msgtext=msgtext & "in string: " & str1
  Else
    msgtext="The letter: " & str2 & " was not found in: " & newline
    msgtext=msgtext & str1
  End If
  MsgBox msgtext
End Sub
```

See Also

GetField, Left, Mid Function, Mid Statement, Option Compare, Right, Str, StrComp

Int Function

Returns the integer part of a *number*.

Syntax

Int(*number*)

Where...	Is...
<i>number</i>	Any numeric expression.

Remarks

For positive *numbers*, **Int** removes the fractional part of the expression and returns the integer part only. For negative *numbers*, **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 that 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).

Example

This example uses **Int** to generate random numbers in the range between the ASCII values for lowercase a and z (97 and 122). The values are converted to letters and displayed as a string.

```
Sub main
  Dim x as Integer
  Dim y
  Dim str1 as String
  Dim letter as String
  Dim randomvalue
  Dim upper, lower
  Dim msgtext, newline
  upper=Asc("z")
  lower=Asc("a")
  newline=Chr(10)
  For x=1 to 26
    Randomize timer() + x*255
    randomvalue=Int((upper - (lower+1)) * Rnd) +lower)
    letter=Chr(randomvalue)
    str1=str1 & letter
    For y=1 to 1500
      Next y
    Next x
    msgtext="The string is:" & newline
    msgtext=msgtext & str1
  MsgBox msgtext
End Sub
```

See Also

Exp, Fix, Log, Rnd, Sgn, Sqr

IPmt Function

Returns the interest portion of a payment for a given period of an annuity.

Syntax

IPmt(*rate* , *per* , *nper* , *pv* , *fv* , *due*)

Where...	Is...
<i>rate</i>	Interest rate per period.
<i>per</i>	Particular payment period in the range 1 through <i>nper</i> .
<i>nper</i>	Total number of payment periods.
<i>pv</i>	Present value of the initial lump sum amount paid (as in the case of an annuity) or received (as in the case of a loan).
<i>fv</i>	Future value of the final lump sum amount required (as in the case of a savings plan) or paid (0 as in the case of a loan).
<i>due</i>	0 if payments are due at the end of each payment period, and 1 if they are due at the beginning of the period.

Remarks

The given interest rate is assumed constant over the life of the annuity. If payments are on a monthly schedule, then *rate* will be 0.0075 if the annual percentage rate on the annuity or loan is 9%.

Example

This example finds the interest portion of a loan payment amount for payments made in last month of the first year. The loan is for \$25,000 to be paid back over 5 years at 9.5% interest.

```
Sub main
  Dim aprate, periods
  Dim payperiod
  Dim loanpv, due
  Dim loanfv, intpaid
  Dim msgtext
  aprate=.095
  payperiod=12
  periods=120
  loanpv=25000
  loanfv=0
  Rem Assume payments are made at end of month
  due=0
  intpaid=IPmt(aprate/12,payperiod,periods,-loanpv,loanfv,due)
  msgtext="For a loan of $25,000 @ 9.5% for 10 years," & Chr(10)
  msgtext=msgtext+ "the interest paid in month 12 is: "
  msgtext=msgtext + Format(intpaid, "Currency")
  MsgBox msgtext
End Sub
```

See Also

FV, IRR, NPV, Pmt, PPmt, PV, Rate

IRR Function

Returns the internal rate of return for a stream of periodic cash flows.

Syntax

IRR(*valuearray*(), *guess*)

Where...	Is...
<i>valuearray</i> ()	An array containing cash flow values.
<i>guess</i>	A ballpark estimate of the value returned by IRR .

Remarks

valuearray() must have at least one positive value (representing a receipt) and one negative value (representing a payment). All payments and receipts must be represented in the exact sequence. The value returned by **IRR** will vary with the change in the sequence of cash flows. In general, a *guess* value of between 0.1 (10 percent) and 0.15 (15 percent) would be a reasonable estimate. **IRR** is an iterative function. It improves a given *guess* over several iterations until the result is within 0.00001 percent. If it does not converge to a result within 20 iterations, it signals failure.

Example

This example calculates an internal rate of return (expressed as an interest rate percentage) for a series of business transactions (income and costs). The first value entered must be a negative amount, or **IRR** generates an "Illegal Function Call" error.

```
Sub main
  Dim cashflows() as Double
  Dim guess, count as Integer
  Dim i as Integer
  Dim intnl as Single
  Dim msgtext as String
  guess=.15
  count=InputBox("How many cash flow amounts do you have?")
  ReDim cashflows(count+1)
  For i=0 to count-1
    cashflows(i)=InputBox("Enter income value for month " & i+1 & ":")
  Next i
  intnl=IRR(cashflows(),guess)
  msgtext="The IRR for your cash flow amounts is: "
  msgtext=msgtext & Format(intnl, "Percent")
  MsgBox msgtext
End Sub
```

See Also

FV, **IPmt**, **NPV**, **Pmt**, **PPmt**, **PV**, **Rate**

Is Operator

Compares two object expressions and returns -1 if they refer to the same object, 0 otherwise.

Syntax

objectExpression **Is** *objectExpression*

Where...	Is...
----------	-------

<i>objectexpression</i>	Any valid object expression.
-------------------------	------------------------------

Remarks

Is can also be used to test if an object variable has been Set to Nothing.

Example

This example displays a list of open files in the software application, Microsoft Visio. It uses the **Is** operator to determine whether Visio is available. To see how this example works, you need to start Visio and open one or more documents.

```
Sub main
    Dim visio as Object
    Dim doc as Object
    Dim msgtext as String
    Dim i as Integer, doccount as Integer
    'Initialize Visio
    Set visio = GetObject("visio.application")    'find Visio
    If (visio Is Nothing) then
        MsgBox "Couldn't find Visio!"
        Exit Sub
    End If
    'Get # of open Visio files
    doccount = visio.documents.count            'OLE2 call to Visio
    If doccount=0 then
        msgtext="No open Visio documents."
    Else
        msgtext="The open files are: " & Chr$(13)
        For i = 1 to doccount
            Set doc = visio.documents(i) 'access Visio's document method
            msgtext=msgtext & Chr$(13)& doc.name
        Next i
    End If
    MsgBox msgtext
End Sub
```

See Also

Create Object, Get Object, Me, Nothing, Object, Typeof

IsDate Function

Returns -1 (TRUE) if an expression is a legal date, 0 (FALSE) if it is not.

Syntax

IsDate(*expression*)

Where...	Is...
<i>expression</i>	Any valid expression.

Remarks

IsDate returns -1 (TRUE) if the expression is of vartype 7 (date) or a string that can be interpreted as a date.

Example

This example accepts a string from the user and checks to see if it is a valid date

```
Sub main
    Dim theDate
    theDate = InputBox( "Enter a date:" )
    If IsDate(theDate)=-1 then
        MsgBox "The new date is: " & Format(CVDate(theDate), "dddddd")
    Else
        MsgBox "The date is not valid."
    End If
End Sub
```

See Also

CVDate, IsEmpty, IsNull, IsNumeric, VarType

IsEmpty Function

Returns -1 (TRUE) if a variant has been initialized. 0 (FALSE) otherwise.

Syntax

IsEmpty(*expression*)

Where...	Is...
<i>expression</i>	Any expression with a data type of variant.

Remarks

IsEmpty returns -1 (TRUE) if the variant is of vartype 0 (empty). 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.

Example

This example prompts for a series of test scores and uses **IsEmpty** to determine whether the maximum allowable limit has been hit. (**IsEmpty** determines when to exit the **Do...Loop**.)

```
Sub main
    Dim arrayvar(10)
    Dim x as Integer
    Dim tscore as Single
    Dim total as Integer
    x=1
    Do
        tscore=InputBox("Enter test score #" & x & ":")
        arrayvar(x)=tscore
        x=x+1
    Loop Until IsEmpty(arrayvar(10))<>-1
    total=x-1
    msgtext="You entered: " & Chr(10)
    For x=1 to total
        msgtext=msgtext & Chr(10) & arrayvar(x)
    Next x
    MsgBox msgtext
End Sub
```

See Also

IsDate, **IsNull**, **IsNumeric**, **VarType**

IsMissing Function

Returns -1 (TRUE) if an optional parameter was not supplied by the user, 0 (FALSE) otherwise.

Syntax

IsMissing(*argname*)

Where...	Is...
<i>argname</i>	An optional argument for a subprogram, function, BSL statement, or BSL function.

Remarks

IsMissing is used in procedures that have optional arguments to find out whether the argument's value was supplied or not.

Example

This example prints a list of letters. The number printed is determined by the user. If the user wants to print all letters, the Function `myfunc` is called without any argument. The function uses **IsMissing** to determine whether to print all the letters or just the number specified by the user.

```
Sub myfunc(Optional arg1)
  If IsMissing(arg1)=-1 then
    arg1=26
  End If
  msgtext="The letters are: " & Chr$(10)
  For x= 1 to arg1
    msgtext=msgtext & Chr$(x+64) & Chr$(10)
  Next x
  MsgBox msgtext
End sub
Sub main
  Dim arg1
  arg1=InputBox("How many letters do you want to print? (0 for all)")
  If arg1=0 then
    myfunc
  Else
    myfunc arg1
  End If
End Sub
```

See Also

Function...End Function

IsNull Function

Returns -1 (TRUE) if a variant expression contains the Null value, 0 (FALSE) otherwise.

Syntax

IsNull(*expression*)

Where...	Is...
----------	-------

<i>expression</i>	Any expression with a data type of variant.
-------------------	---

Remarks

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.

Example

This example asks for ten test score values and calculates the average. If any score is negative, the value is set to Null. Then **IsNull** is used to reduce the total count of scores (originally 10) to just those with positive values before calculating the average.

```
Sub main
  Dim arrayvar(10)
  Dim count as Integer
  Dim total as Integer
  Dim x as Integer
  Dim tscore as Single
  count=10
  total=0
  For x=1 to count
    tscore=InputBox("Enter test score #" & x & ":")
    If tscore<0 then
      arrayvar(x)=Null
    Else
      arrayvar(x)=tscore
      total=total+arrayvar(x)
    End If
  Next x
  Do While x<>0
    x=x-1
    If IsNull(arrayvar(x))=-1 then
      count=count-1
    End If
  Loop
  msgtext="The average (excluding negative values) is: " & Chr(10)
  msgtext=msgtext & Format (total/count, "##.##")
  MsgBox msgtext
End Sub
```

See Also

IsDate, **IsEmpty**, **IsNumeric**, **VarType**

IsNumeric Function

Returns -1 (TRUE) if an expression has a data type of **Numeric**, 0 (FALSE) otherwise.

Syntax

IsNumeric(*expression*)

Where...	Is...
<i>expression</i>	Any valid expression.

Remarks

IsNumeric returns -1 (TRUE) if the expression is of vartypes 2-6 (numeric) or a string that can be interpreted as a number.

Example

This example uses **IsNumeric** to determine whether a user selected an option (1-3) or typed "Q" to quit.

```
Sub main
    Dim answer
    answer=InputBox("Enter a choice (1-3) or type Q to quit")
    If IsNumeric(answer)=-1 then
        Select Case answer
            Case 1
                MsgBox "You chose #1."
            Case 2
                MsgBox "You chose #2."
            Case 3
                MsgBox "You chose #3."
        End Select
    Else
        MsgBox "You typed Q."
    End If
End Sub
```

See Also

IsDate, **IsEmpty**, **IsNull**, **VarType**

Kill Statement

Deletes files from a disk drive.

Syntax

Kill *pathname*\$

Where...

Is...

pathname\$ A String expression that specifies a valid DOS file specification.

Remarks

The *pathname*\$ specification can contain paths and wildcards. **Kill** deletes files only, not directories. Use the **Rmdir** function to delete directories.

Example

This example prompts a user for an account number, opens a file, searches for the account number and displays the matching letter for that number. The second subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram. After processing is complete, the first subroutine uses **Kill** to delete the file.

```
Declare Sub createfile()
Global x as Integer
Global y(100) as String

Sub main
  Dim acctno as Integer
  Dim msgtext
  Call createfile
i: acctno=InputBox("Enter an account number from 1-10:")
  If acctno<1 Or acctno>10 then
    MsgBox "Invalid account number. Try again."
    Goto i:
  End if
  x=1
  Open "C:\TEMP001" for Input as #1
  Do Until x=acctno
    Input #1, x,y(x)
  Loop
  msgtext="The letter for account number " & x & " is: " & y(x)
  Close #1
  MsgBox msgtext
  Kill "C:\TEMP001"
End Sub

Sub createfile()
' Put the numbers 1-10 and letters A-J into a file
  Dim startletter
  Open "C:\TEMP001" for Output as #1
  startletter=65
  For x=1 to 10
    y(x)=Chr(startletter)
    startletter=startletter+1
  Next x
```

```
For x=1 to 10
  Write #1, x,y(x)
Next x
Close #1
End Sub
```

See Also

FileAttr, FileDateTime, GetAttr, RmDir

LBound Function

Returns the lower bound of the subscript range for the specified array.

Syntax

LBound(*arrayname* [, *dimension*])

Where...	Is...
<i>arrayname</i>	The name of the array to use.
<i>dimension</i>	The dimension to use.

Remarks

The dimensions of an array are numbered starting with 1. If the *dimension* is not specified, 1 is used as a default.

LBound can be used with **UBound** to determine the length of an array.

Example

This example resizes an array if the user enters more data than can fit in the array. It uses **LBound** and **UBound** to determine the existing size of the array and **ReDim** to resize it. Option Base sets the default lower bound of the array to 1.

```
Option Base 1
Sub main
    Dim arrayvar() as Integer
    Dim count as Integer
    Dim answer as String
    Dim x, y as Integer
    Dim total
    total=0
    x=1
    count=InputBox("How many test scores do you have?")
    ReDim arrayvar(count)
start:
    Do until x=count+1
        arrayvar(x)=InputBox("Enter test score #" &x & ":")
        x=x+1
    Loop
    answer=InputBox$("Do you have more scores? (Y/N)")
    If answer="Y" or answer="y" then
        count=InputBox("How many more do you have?")
        If count<>0 then
            count=count+(x-1)
            ReDim Preserve arrayvar(count)
            Goto start
        End If
    End If
    x=LBound(arrayvar,1)
    count=UBound(arrayvar,1)
    For y=x to count
        total=total+arrayvar(y)
    Next y
    MsgBox "The average of " & count & " scores is: " & Int(total/count)
End Sub
```

See Also

Dim, Global, Option Base, ReDim, Static, UBound

LCase Function

Returns a copy of a string, with all uppercase letters converted to lowercase.

Syntax

LCase[\$](*string*\$)

Where...

Is...

string\$ A string, or an expression containing the string to use.

Remarks

The translation is based on the country or region specified in the Windows Control Panel. **LCase** accepts expressions of type String. **LCase** accepts any type of argument and will convert the input value to a 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 *string*\$ is NULL, a variant of vartype 1 (Null) is returned.

Example

This example converts a string entered by the user to lowercase.

```
Sub main
  Dim userstr as String
  userstr=InputBox$("Enter a string in upper and lowercase letters")
  userstr=LCase$(userstr)
  MsgBox "The string now is: " & userstr
End Sub
```

See Also

UCase

Left Function

Returns a string of a specified number of characters copied from the beginning of another string.

Syntax

Left[\$](*string*\$, *length*%)

Where...	Is...
<i>string</i> \$	A string or an expression containing the string to copy.
<i>length</i> %	The number of characters to copy.

Remarks

If *length*% is greater than the length of *string*%, **Left** returns the whole string.

Left accepts expressions of type String. **Left** accepts any type of *string*%, including numeric values, and will convert the input value to a 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 *string*\$ is NULL, a variant of vartype 1 (Null) is returned.

To obtain a string of a specified number of bytes, copied from the beginning of another string, use the **LeftB** function.

Example

This example extracts a user's first name from the entire name entered.

```
Sub main
    Dim username as String
    Dim count as Integer
    Dim firstname as String
    Dim charspace
    charspace=Chr(32)
    username=InputBox("Enter your first and last name")
    count=InStr(username,charspace)
    firstname=Left(username,count)
    MsgBox "Your first name is: " &firstname
End Sub
```

See Also

GetField, **Len**, **LTrim**, **Mid Function**, **Mid Statement**, **Right**, **RTrim**, **Str**, **StrComp**, **Trim**

Len Function

Returns the length of a string or variable.

Syntax A

`Len(string$)`

Syntax B

`Len(varname)`

Where...	Is...
<i>string\$</i>	A string or an expression that evaluates to a string.
<i>varname</i>	A variable that contains a string.

Remarks

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 data type or user-defined type is returned.

If syntax B is used, and *varname* is a variant containing a NULL, **Len** will return a Null variant.

To return the number bytes in a string, use the **LenB** function.

Note: It is critical to use **LenB()** instead of **Len()** for *all* non-null “table length” parameters.

Example

This example returns the length of a name entered by the user (including spaces).

```
Sub main
    Dim username as String
    username=InputBox("Enter your name")
    count=Len(username)
    MsgBox "The length of your name is: " &count
End Sub
```

See Also

Instr

Let (Assignment Statement)

Assigns an expression to a Basic variable.

Syntax

[**Let**] *variable* = *expression*

Where...	Is...
<i>variable</i>	The name of a variable to assign to the <i>expression</i> .
<i>expression</i>	The expression to assign to the variable.

Remarks

The keyword **Let** is optional.

The **Let** statement can be used to assign a value or expression to a variable with a data type of 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.

Let differs from **Set** in that Set assigns a variable to an OLE object. For example,

Set o1 = o2 will set the object reference.

Let o1 = o2 will set the value of the default member.

Example

This example uses the Let statement for the variable sum. The subroutine finds an average of 10 golf scores.

```
Sub main
    Dim score As Integer
    Dim x, sum
    Dim msgtext
    Let sum=0
    For x=1 to 10
        score=InputBox("Enter your last ten golf scores #" & x & ":")
        sum=sum+score
    Next x
    msgtext="Your average is: " & CInt(sum/(x-1))
    MsgBox msgtext
End Sub
```

See Also

Const, Lset, Set

Like Operator

Returns the value -1 (TRUE) if a string matches a pattern, 0 (FALSE) otherwise.

Syntax

string\$ **LIKE** *pattern*\$

Where...	Is...
<i>string</i> \$	Any string expression.
<i>pattern</i> \$	Any string expression to match to <i>string</i> \$.

Remarks

pattern\$ can include the following special characters:

Character	Matches
?	A single character
*	A set of zero or more characters
#	A single digit character (0-9)
[<i>chars</i>]	A single character in <i>chars</i>
[! <i>chars</i>]	A single character not in <i>chars</i>
[<i>schar</i> - <i>echar</i>]	A single character in range <i>schar</i> to <i>echar</i>
[! <i>schar</i> - <i>echar</i>]	A single character not in range <i>schar</i> to <i>echar</i>

Both ranges and lists can appear within a single set of square brackets. Ranges are matched according to their ANSI values. In a range, *schar* must be less than *echar*.

If either *string*\$ or *pattern*\$ is NULL then the result value is NULL.

The **Like** operator respects the current setting of **Option Compare**.

For more information on operators, see "Expressions" on page 20.

Example

This example tests whether a letter is lowercase.

```
Sub main
  Dim userstr as String
  Dim revalue as Integer
  Dim msgtext as String
  Dim pattern
  pattern="[a-z]"
  userstr=InputBox$("Enter a letter:")
  revalue=userstr LIKE pattern
  If revalue=-1 then
    msgtext="The letter " & userstr & " is lowercase."
  Else
    msgtext="Not a lowercase letter."
  End If
  MsgBox msgtext
End Sub
```

See Also

Expressions, Instr, Option Compare, StrComp

Line Input Statement

Reads a line from a sequential file or keyboard into a string variable.

Syntax A

Line Input [#] *filename%*, *varname\$*

Syntax B

Line Input [*prompt\$*,] *varname\$*

Where...

Is...

<i>filename%</i>	An integer expression identifying the open file to use.
<i>prompt\$</i>	An optional string that can be used to prompt for keyboard input; it must be a literal string.
<i>varname\$</i>	A string variable to contain the line read.

Remarks

If specified, the *filename%* is the number used in the **Open** statement to open the file. If *filename%* is not provided, the line is read from the keyboard.

If *prompt\$* is not provided, a prompt of "?" is used.

Example

This example reads the contents of a sequential file line by line (to a carriage return) and displays the results. The second subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram.

```
Declare Sub createfile()
Sub main
  Dim testscore as String
  Dim x
  Dim y
  Dim newline
  Call createfile
  Open "c:\temp001" for Input as #1
  x=1
  newline=Chr(10)
  msgtext= "The contents of c:\temp001 is: " & newline
  Do Until x=Lof(1)
    Line Input #1, testscore
    x=x+1
    y=Seek(1)
    If y>Lof(1) then
      x=Lof(1)
    Else
      Seek 1,y
    End If
    msgtext=msgtext & testscore & newline
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub

Sub createfile()
  Rem Put the numbers 1-10 into a file
```

```

Dim x as Integer
Open "C:\TEMP001" for Output as #1
For x=1 to 10
    Write #1, x
Next x
Close #1
End Sub

```

See Also

DialogBoxes, Get, Input Function, Input Statement, InputBox, Open

ListBox Statement

Defines a list box of choices for a dialog box.

Syntax A

ListBox *x*, *y*, *dx*, *dy*, *text\$*, *.field*

Syntax B

ListBox *x*, *y*, *dx*, *dy*, *stringarray\$()*, *.field*

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the list box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the list box.
<i>text\$</i>	A string containing the selections for the list box.
<i>stringarray\$</i>	An array of dynamic strings for the selections in the list box.
<i>.field</i>	The name of the dialog-record field that will hold a number for the choice made in the list box.

Remarks

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.

The *text\$* argument 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 the following example:

```
dimname = "listchoice" + Chr$(9) + "listchoice" + Chr$(9) + "listchoice"...
```

A number representing the selection's position in the *text\$* string is recorded in the field designated by the *.field* argument when the OK button (or any pushbutton other than Cancel) is pushed. The numbers begin at 0. If no item is selected, it is -1. The *field* argument is also used by the dialog statements that act on this control.

Use the **ListBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with list box and two buttons.

```
Sub main
  Dim ListBox1() as String
  ReDim ListBox1(0)
  ListBox1(0)="C:\"
  Begin Dialog UserDialog 133, 66, 171, 65, "BSL Dialog Box"
    Text 3, 3, 34, 9, "Directory:", .Text2
    ListBox 3, 14, 83, 39, ListBox1(), .ListBox2
    OKButton 105, 6, 54, 14
    CancelButton 105, 26, 54, 14
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin...End Dialog, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, Dialog, DialogBoxes, DropComboBox, GroupBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Loc Function

Returns the current offset within an open file.

Syntax

Loc(*filenumber%*)

Where... **Is...**

filenumber% An integer expression identifying the open file to query.

Remarks

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

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.

Example

This example creates a file of account numbers as entered by the user. When the user finishes, the example displays the offset in the file of the last entry made.

```
Sub main
  Dim filepos as Integer
  Dim acctno() as Integer
  Dim x as Integer
  x=0
  Open "c:\TEMP001" for Random as #1
  Do
    x=x+1
    Redim Preserve acctno(x)
    acctno(x)=InputBox("Enter account #" & x & " or 0 to end:")
    If acctno(x)=0 then
      Exit Do
    End If
    Put #1,, acctno(x)
  Loop
  filepos=Loc(1)
  Close #1
  MsgBox "The offset is: " & filepos
  Kill "C:\TEMP001"
End Sub
```

See Also

Eof, **Lof**, **Open**

Lock Statement

Controls access to an open file.

Syntax

Lock [#]filename% [, [start&] [To end&]]

Where...	Is...
filename%	An integer expression identifying the open file.
start&	Number of the first record or byte offset to lock/unlock.
end&	Number of the last record or byte offset to lock/unlock.

Remarks

The *filename%* is the number used in the **Open** statement of the file.

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.

Example

This example locks a file that is shared by others on a network, if the file is already in use. The second subprogram, CREATEFILE, creates the file used by the main subprogram.

```
Declare Sub createfile
Sub main
  Dim btngrp, icongrp
  Dim defgrp
  Dim answer
  Dim noaccess as Integer
  Dim msgabort
  Dim msgstop as Integer
  Dim acctname as String
  noaccess=70
  msgstop=16
  Call createfile
  On Error Resume Next
  btngrp=1
  icongrp=64
  defgrp=0
  answer=MsgBox("Open the account file?" & Chr(10), btngrp+icongrp+defgrp)
  If answer=1 then
    Open "C:\TEMP001" for Input as #1
    If Err=noaccess then
      msgabort=MsgBox("File Locked",msgstop,"Aborted")
    Else
      Lock #1
      Line Input #1, acctname
      MsgBox "The first account name is: " & acctname
      Unlock #1
    End If
  End If
```

```
        Close #1
    End If
    Kill "C:\TEMP001"
End Sub

Sub createfile()
    Rem Put the letters A-J into the file
    Dim x as Integer
    Open "C:\TEMP001" for Output as #1
    For x=1 to 10
        Write #1, Chr(x+64)
    Next x
    Close #1
End Sub
```

See Also**Open, Unlock**

Lof Function

Returns the length in bytes of an open file.

Syntax

Lof(*filename%*)

Where... **Is...**

filename% An integer expression identifying the open file.

Remarks

The *filename%* is the number used in the **Open** statement of the file.

Example

This example opens a file and prints its contents to the screen.

```
Sub main
  Dim fname
  Dim fchar()
  Dim x as Integer
  Dim msgtext
  Dim newline
  newline=Chr(10)
  fname=InputBox("Enter a filename to print:")
  On Error Resume Next
  Open fname for Input as #1
  If Err<>0 then
    MsgBox "Error loading file. Re-run program."
    Exit Sub
  End If
  msgtext="The contents of " & fname & " is: " & newline & newline
  Redim fchar(Lof(1))
  For x=1 to Lof(1)
    fchar(x)=Input(1,#1)
    msgtext=msgtext & fchar(x)
  Next x
  MsgBox msgtext
  Close #1
End Sub
```

See Also

Eof, FileLen, Loc, Open

Log Function

Returns the natural logarithm of a number.

Syntax

Log(*number*)

Where...	Is...
<i>number</i>	Any valid numeric expression.

Remarks

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.

Example

This example uses the **Log** function to determine which number is larger: 999^{1000} (999 to the 1000 power) or 1000^{999} (1000 to the 999 power). Note that you cannot use the exponent (^) operator for numbers this large.

```
Sub main
    Dim x
    Dim y
    x=999
    y=1000
    a=y*(Log(x))
    b=x*(Log(y))
    If a>b then
        MsgBox "999^1000 is greater than 1000^999"
    Else
        MsgBox "1000^999 is greater than 999^1000"
    End If
End Sub
```

See Also

Exp, Fix, Int, Rnd, Sgn, Sqr

Lset Statement

Copies one string to another, or assigns a user-defined type variable to another.

Syntax A

Lset *string\$* = *string-expression*

Syntax B

Lset *variable1* = *variable2*

Where...	Is...
<i>string\$</i>	A string or string expression to contain the copied characters.
<i>string-expression</i>	An expression containing the string to copy.
<i>variable1</i>	A variable with a user-defined type to contain the copied variable.
<i>variable2</i>	A variable with a user-defined type to copy.

Remarks

If *string\$* is shorter than *string-expression*, **Lset** copies the 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.

In Syntax B, 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.

Example

This example puts a user's last name into the variable LASTNAME. If the name is longer than the size of LASTNAME, then the user's name is truncated. If you have a long last name and you get lots of junk mail, you have probably already seen how this works.

```
Sub main
  Dim lastname as String
  Dim strlast as String*8
  lastname=InputBox("Enter your last name")
  Lset strlast=lastname
  msgtext="Your last name is: " &strlast
  MsgBox msgtext
End Sub
```

See Also

Rset

LTrim Function

Returns a copy of a string with all leading space characters removed.

Syntax

LTrim[\$](*string*\$)

Where...	Is...
<i>string</i> \$	A string or expression containing a string to copy.

Remarks

LTrim accepts any type of *string*\$, including numeric values, and will convert the input value to a string.

The dollar sign, "\$", in the function name is optional. If specified, the return type is string. If omitted, the function typically returns a variant of vartype 8 (string). If the value of *string*\$ is NULL, a variant of vartype 1 (Null) is returned.

Example

This example trims the leading spaces from a string padded with spaces on the left.

```
Sub main
    Dim userInput as String
    Dim numsize
    Dim str1 as String*50
    Dim strsize
    strsize=50
    userInput=InputBox("Enter a string of characters:")
    numsize=Len(userInput)
    str1=Space(strsize-numsize) & userInput
    ' Str1 has a variable number of leading spaces.
    MsgBox "The string is: " &str1
    str1=LTrim$(str1)
    ' Str1 now has no leading spaces.
    MsgBox "The string now has no leading spaces: " & str1
End Sub
```

See Also

GetField, **Left**, **Mid Function**, **Mid Statement**, **Right**, **RTrim**, **Trim**

Me

Refers to the currently used OLE2 automation object.

Syntax

Me

Remarks

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 can use the variable **Me** to refer to the object that triggered the event (which button was clicked). The programmer can use **Me** in all the same ways as any other object variable except that **Me** cannot be **Set**.

Note: **Me** is applicable only if your application works with instantiable modules. If your application does not, you should not use this entry.

Example

(None)

See Also

Create Object, Get Object, New, Nothing, Object, Typeof

Mid Function

Returns a portion of a string, starting at a specified character position within the string.

Syntax

Mid[\$](*string*\$, *start*%[, *length*%])

Where... Is...

<i>string</i> \$	A string or expression that contains the string to change.
<i>start</i> %	The starting position in <i>string</i> \$ to begin replacing characters.
<i>length</i> %	The number of characters to replace.

Remarks

Mid accepts any type of *string*\$, including numeric values, and will convert the input value to a string. If the *length*% argument is omitted, or if *string*\$ is smaller than *length*%, then **Mid** returns all characters in *string*\$. If *start*% is larger than *string*%, then **Mid** returns a null string ("").

The index of the first character in a string is 1.

The dollar sign, "\$", in the function name is optional. If specified, the return type is string. If omitted, the function typically returns a variant of vartype 8 (string). If the value of *string*\$ is Null, a variant of vartype 1 (Null) is returned. **Mid**\$ requires the string argument to be of type string or variant. **Mid** allows the string argument to be of any datatype.

To return a specified number of bytes from a string, use the **MidB** function. With the **MidB** function, *start*% specifies a byte position and *length*% specifies a number of bytes.

To modify a portion of a string value, see "Mid Statement" on page 204.

Example

This example uses the Mid statement to replace the last name in a user-entered string to asterisks(*).

```
Sub main
    Dim username as String
    Dim position as Integer
    Dim count as Integer
    Dim uname as String
    Dim replacement as String
    username=InputBox("Enter your full name:")
    uname=username
    replacement="*"
    Do
        position=InStr(username, " ")
        If position=0 then
            Exit Do
        End If
        username=Mid(username,position+1)
        count=count+position
    Loop
    For x=1 to Len(username)
        count=count+1
        Mid(uname,count)=replacement
    Next x
    MsgBox "Your name now is: " & uname
End Sub
```

See Also

GetField, Left, Len, LTrim, Mid Statement, Right, RTrim, Trim

Mid Statement

Replaces part (or all) of one string with another, starting at a specified location.

Syntax

Mid (*stringvar*\$, *start*%[, *length*%]) = *string*\$

Where...	Is...
<i>stringvar</i> \$	The string to change.
<i>start</i> %	An expression for the position to begin replacing characters.
<i>length</i> %	An expression for the number of characters to replace.
<i>string</i> \$	The string to place into another string.

Remarks

If the *length*% argument is omitted, or if there are fewer characters in *string*\$ than specified in *length*%, then **Mid** replaces all the characters from the *start*% to the end of the *string*\$. If *start*% is larger than the number of characters in the indicated *stringvar*\$, then **Mid** appends *string*% to *stringvar*\$.

If *length*% is greater than the length of *string*\$, then *length*% is set to the length of *string*\$. If *start*% is greater than the number of characters in *stringvar*\$, an illegal function call error will occur at runtime. If *length*% plus *start*% is greater than the length of *stringvar*\$, then only the characters up to the end of *stringvar*\$ are replaced.

Mid never changes the number of characters in *stringvar*\$.

The index of the first character in a string is 1.

To replace a specified number of bytes in a string, with those from another string, use the **MidB** statement. With the **MidB** statement, *start*% specifies a byte position and *length*% specifies a number of bytes.

Example

This example uses the Mid function to find the last name in a string entered by the user.

```
Sub main
  Dim username as String
  Dim position as Integer
  username=InputBox("Enter your full name:")
  Do
    position=InStr(username, " ")
    If position=0 then
      Exit Do
    End If
    position=position+1
    username=Mid(username,position)
  Loop
  MsgBox "Your last name is: " & username
End Sub
```

See Also

GetField, LCase, Left, Len, LTrim, Mid Function, Right, RTrim, Trim

Minute Function

Returns an integer for the minute component (0-59) of a date-time value.

Syntax

Minute(*time*)

Where...	Is...
<i>time</i>	Any expression that can evaluate to a date-time value.

Remarks

Minute accepts any type of *time*, including strings, and will attempt to convert the input value to a date value. The return value is a variant of vartype 2 (Integer). If the value of *time* is null, a variant of vartype 1 (null) is returned.

Example

This example extracts the hour, minute, and second from a file's last modification date and time.

```
Sub main
    Dim filename as String
    Dim ftime
    Dim hr, min
    Dim sec
    Dim msgtext as String
i: msgtext="Enter a filename:"
    filename=InputBox(msgtext)
    If filename="" then
        Exit Sub
    End If
    On Error Resume Next
    ftime=FileDateTime(filename)
    If Err<>0 then
        MsgBox "Error in file name. Try again."
        Goto i:
    End If
    hr=Hour(ftime)
    min=Minute(ftime)
    sec=Second(ftime)
    MsgBox "The file's time is: " & hr & ":" & min & ":" & sec
End Sub
```

See Also

DateSerial, DateValue, Day, Hour, Month, Now, Second, Time Function, Time Statement, TimeSerial, TimeValue, Weekday, Year

MkDir Statement

Creates a new directory.

Syntax

MkDir *path*\$

Where...

Is...

path\$ A string expression identifying the new default directory to create.

Remarks

The syntax for *path*\$ 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.

Example

This example makes a new temporary directory in C:\ and then deletes it.

```
Sub main
    Dim path as String
    On Error Resume Next
    path=CurDir(C)
    If path<>"C:\" then
        ChDir "C:\"
    End If
    MkDir "C:\TEMP01"
    If Err=75 then
        MsgBox "Directory already exists"
    Else
        MsgBox "Directory C:\TEMP01 created"
        MsgBox "Now removing directory"
        Rmdir "C:\TEMP01"
    End If
End Sub
```

See Also

ChDir, **ChDrive**, **CurDir**, **Dir**, **Rmdir**

Month Function

Returns an integer for the month component (1-12) of a date-time value.

Syntax

Month(*date*)

Where...	Is...
<i>date</i>	Any expression that evaluates to a date-time value.

Remarks

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

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

Example

This example finds the month (1-12) and day (1-31) values for this Thursday.

```
Sub main
    Dim x, today
    Dim msgtext
    Today=DateValue(Now)
    Let x=0
    Do While Weekday(Today+x) <> 5
        x=x+1
    Loop
    msgtext="This Thursday is: " & Month(Today+x) & "/" & Day(Today+x)
    MsgBox msgtext
End Sub
```

See Also

Date Function, Date Statement, DateSerial, DateValue, Day, Hour, Minute, Now, Second, TimeSerial, TimeValue, Weekday, Year

Msgbox Function

Displays a message dialog box and returns a value (1-7) indicating which button the user selected.

Syntax

Msgbox(*prompt\$* , [*buttons%*] , *title\$*)

Where... Is...

<i>prompt\$</i>	The text to display in a dialog box.
<i>buttons%</i>	An integer value for the buttons, the icon, and the default button choice to display in a dialog box.
<i>title\$</i>	A string expression containing the title for the message box.

Remarks

prompt\$ must be no more than 1,024 characters long. A message string greater than 255 characters without intervening spaces will be truncated after the 255th character.

buttons% is the sum of three values, one from each of the following groups:

	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 *buttons%* is omitted, **Msgbox** displays a single OK button. After the user clicks a button, **Msgbox** returns a value indicating the user's choice. 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

Example

This example displays one of each type of message box.

```
Sub main
  Dim btngrp as Integer
  Dim icongrp as Integer
  Dim defgrp as Integer
  Dim msgtext as String
  icongrp=16
  defgrp=0
  btngrp=0
  Do Until btngrp=6
    Select Case btngrp
      Case 1, 4, 5
        defgrp=0
      Case 2
        defgrp=256
      Case 3
        defgrp=512
    End Select
    msgtext=" Icon group = " & icongrp & Chr(10)
    msgtext=msgtext + " Button group = " & btngrp & Chr(10)
    msgtext=msgtext + " Default group = " & defgrp & Chr(10)
    msgtext=msgtext + Chr(10) + " Continue?"
    answer=MsgBox(msgtext, btngrp+icongrp+defgrp)
    Select Case answer
      Case 2,3,7
        Exit Do
    End Select
    If icongrp<>64 then
      icongrp=icongrp+16
    End If
    btngrp=btngrp+1
  Loop
End Sub
```

See Also

Dialog Boxes, InputBox, MsgBox Statement, PasswordBox

Msgbox Statement

Displays a prompt in a message dialog box.

Syntax

MsgBox *prompt\$* , [*buttons%*][, *title\$*]

Where... Is...

<i>prompt\$</i>	The text to display in a dialog box.
<i>buttons%</i>	An integer value for the buttons, the icon, and the default button choice to display in a dialog box.
<i>title\$</i>	A string expression containing the title for the message box.

Remarks

Prompt\$ must be no more than 1,024 characters long. A message string greater than 255 characters without intervening spaces will be truncated after the 255th character.

buttons% is the sum of three values, one from each of the following groups:

	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 *buttons%* is omitted, **Msgbox** displays a single OK button.

Example

This example finds the future value of an annuity, whose terms are defined by the user. It uses the **MsgBox** statement to display the result.

```
Sub main
    Dim aprate, periods
    Dim payment, annuitypv
    Dim due, futurevalue
    Dim msgtext
    annuitypv=InputBox("Enter present value of the annuity: ")
    aprate=InputBox("Enter the annual percentage rate: ")
    If aprate >1 then
        aprate=aprate/100
    End If
    periods=InputBox("Enter the total number of pay periods: ")
    payment=InputBox("Enter the initial amount paid to you: ")
    Rem Assume payments are made at end of month
    due=0
    futurevalue=FV(aprate/12,periods,-payment,-annuitypv,due)
    msgtext="The future value is: " & Format(futurevalue, "Currency")
    MsgBox msgtext
End Sub
```

See Also

InputBox, **MsgBox** Function, **PasswordBox**

Name Statement

Renames a file or moves a file from one directory to another.

Syntax

Name *oldfilename*\$ **As** *newfilename*\$

Where...	Is...
----------	-------

<i>oldfilename</i> \$	A string expression containing the file to rename.
-----------------------	--

<i>newfilename</i> \$	A string expression containing the name for the file.
-----------------------	---

Remarks

A path can be part of either filename argument. 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 or if the file *newfilename*\$ already exists, Basic generates an error message.

Example

This example creates a temporary file, C:\TEMP001, renames the file to C:\TEMP002, then deletes them both. It calls the subprogram, CREATEFILE, to create the C:\TEMP001 file.

```
Declare Sub createfile()
Sub main
  Call createfile
  On Error Resume Next
  Name "C:\TEMP001" As "C:\TEMP002"
  MsgBox "The file has been renamed"
  MsgBox "Now deleting both files"
  Kill "TEMP001"
  Kill "TEMP002"
End Sub
Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Dim y()
  Dim startletter
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also

FileAttr, **FileCopy**, **GetAttr**, **Kill**

New Operator

Allocates and initializes a new OLE2 object of the named class.

Syntax

Set *objectVar* = **New** *className*

Dim *objectVar* **As New** *className*

Where...	Is...
<i>objectVar</i>	The OLE2 object to allocate and initialize.
<i>className</i>	The class to assign to the object.

Remarks

In the **Dim** statement, **New** marks *objectVar* so that a new object will be allocated and initialized when *objectVar* is first used. If *objectVar* 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 *objectVar* is Set to Nothing and referenced again.

See Also

Dim, Global, Set, Static

\$NoCStrings Metacommand

Tells the compiler to treat a backslash (\) inside a string as a normal character.

Syntax

'\$NoCStrings [Save]

Where...	Means...
----------	----------

Save	Saves the current '\$CStrings setting before restoring the treatment of the backslash (\) to a normal character.
------	---

Remarks

Use the **'\$CStrings Restore** command to restore a previously saved setting. Save and Restore operate as a stack and allow the user to change the **'\$CStrings** setting for a range of the program without impacting the rest of the program.

Use the **'\$CStrings** metacommand to tell the compiler to treat a backslash (\) inside of a string as an Escape character.

Note: BSL offers a number of extensions that are not included in Visual Basic.

Example

This example displays two lines, the first time using the C-language characters “\n” for a carriage return and line feed.

```
Sub main
  '$CStrings
  MsgBox "This is line 1\n This is line 2 (using C Strings)"
  '$NoCStrings
  MsgBox "This is line 1" +Chr$(13)+Chr$(10)+"This is line 2 (using Chr)"
End Sub
```

See Also

'\$CStrings, **'\$Include**, **Rem**

Nothing Function

Returns an object value that does not refer to an object.

Syntax

Set *variableName* = Nothing

Where... Is...

variableName The name of the object variable to set to nothing.

Remarks

Nothing is the value object variables have when they do not refer to an object, either because they have not been initialized yet or because they were explicitly **Set** to **Nothing**. For example:

```

If Not objectVar Is Nothing then
    objectVar.Close
    Set objectVar = Nothing
End If

```

Example

This example displays a list of open files in the software application, Microsoft Visio. It uses the **Nothing** function to determine whether Visio is available. To see how this example works, you need to start Visio and open one or more documents.

```

Sub main
    Dim visio as Object
    Dim doc as Object
    Dim msgtext as String
    Dim i as Integer, doccount as Integer

    'Initialize Visio
    Set visio = GetObject("visio.application") ' find Visio
    If (visio Is Nothing) then
        MsgBox "Couldn't find Visio!"
        Exit Sub
    End If

    'Get # of open Visio files
    doccount = visio.documents.count          'OLE2 call to Visio
    If doccount=0 then
        msgtext="No open Visio documents."
    Else
        msgtext="The open files are: " & Chr$(13)
        For i = 1 to doccount
            Set doc = visio.documents(i) ' access Visio's document method
            msgtext=msgtext & Chr$(13)& doc.name
        Next i
    End If
    MsgBox msgtext
End Sub

```

See Also

Is, **New**

Now Function

Returns the current date and time.

Syntax

Now()

Remarks

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.

Example

This example finds the month (1-12) and day (1-31) values for this Thursday.

```
Sub main
    Dim x, today
    Dim msgtext
    Today=DateValue (Now)
    Let x=0
    Do While Weekday(Today+x) <> 5
        x=x+1
    Loop
    msgtext="This Thursday is: " &Month(Today+x) &"/"&Day(Today+x)
    MsgBox msgtext
End Sub
```

See Also

Date Function, Date Statement, Day, Hour, Minute, Month, Second, Time Function, Time Statement, Weekday, Year

NPV Function

Returns the net present value of a investment based on a stream of periodic cash flows and a constant interest rate.

Syntax

NPV (*rate* , *valuearray* ())

Where...	Is...
<i>rate</i>	Discount rate per period.
<i>valuearray</i> ()	An array containing cash flow values.

Remarks

Valuearray() must have at least one positive value (representing a receipt) and one negative value (representing a payment). All payments and receipts must be represented in the exact sequence. The value returned by **NPV** will vary with the change in the sequence of cash flows. If the discount rate is 12% per period, *rate* is the decimal equivalent, 0.12. **NPV** uses future cash flows as the basis for the net present value calculation. If the first cash flow occurs at the beginning of the first period, its value should be added to the result returned by **NPV** and must not be included in *valuearray*().

Example

This example finds the net present value of an investment, given a range of cash flows by the user.

```
Sub main
  Dim aprate as Single
  Dim varray() as Double
  Dim cflowper as Integer
  Dim x as Integer
  Dim netpv as Double
  cflowper=InputBox("Enter number of cash flow periods")
  ReDim varray(cflowper)
  For x= 1 to cflowper
    varray(x)=InputBox("Enter cash flow amount for period #" & x & ":")
  Next x
  aprate=InputBox("Enter discount rate: ")
  If aprate>1 then
    aprate=aprate/100
  End If
  netpv=NPV(aprate,varray())
  MsgBox "The net present value is: " & Format(netpv, "Currency")
End Sub
```

See Also

FV, **IPmt**, **IRR**, **Pmt**, **PPmt**, **PV**, **Rate**

Null Function

Returns a variant value set to NULL.

Syntax

Null

Remarks

Null is used to set a variant to the Null value explicitly, as follows:

```
variableName = Null
```

Variants are initialized to the empty value, which is different from the null value.

Example

This example asks for ten test score values and calculates the average. If any score is negative, the value is set to Null. Then IsNull is used to reduce the total count of scores (originally 10) to just those with positive values before calculating the average.

```
Sub main
  Dim arrayvar(10)
  Dim count as Integer
  Dim total as Integer
  Dim x as Integer
  Dim tscore as Single
  count=10
  total=0
  For x=1 to count
    tscore=InputBox("Enter test score #" & x & ":")
    If tscore<0 then
      arrayvar(x)=Null
    Else
      arrayvar(x)=tscore
      total=total+arrayvar(x)
    End If
  Next x
  Do While x<>0
    x=x-1
    If IsNull(arrayvar(x))=-1 then
      count=count-1
    End If
  Loop
  msgtext="The average (excluding negative values) is: " & Chr(10)
  msgtext=msgtext & Format (total/count, "##.##")
  MsgBox msgtext
End Sub
```

See Also

IsEmpty, IsNull, VarType

Object Class

A class that provides access to OLE2 automation objects.

Syntax

Dim *variableName* **As Object**

Where... **Is...**

variableName The name of the object variable to declare.

Remarks

To create a new object, first dimension a variable, using the **Dim** statement, then **Set** the variable to the return value of **CreateObject** or **GetObject**, as follows:

Dim OLE2 As Object

SetOLE2 = CreateObject("spoly.cpoly")

To refer to a method or property of the newly created object, use the syntax: *objectvar.property* or *objectvar.method*, as follows:

OLE2.reset

Example

This example displays a list of open files in the software application, Microsoft Visio. It uses the Object class to declare the variables used for accessing Visio and its document files and methods.

```
Sub main
  Dim visio as Object
  Dim doc as Object
  Dim msgtext as String
  Dim i as Integer, doccount as Integer

  'Initialize Visio
  Set visio = GetObject("visio.application") ' find Visio
  If (visio Is Nothing) then
    MsgBox "Couldn't find Visio!"
    Exit Sub
  End If
  'Get # of open Visio files
  doccount = visio.documents.count          'OLE2 call to Visio
  If doccount=0 then
    msgtext="No open Visio documents."
  Else
    msgtext="The open files are: " & Chr$(13)
    For i = 1 to doccount
      Set doc = visio.documents(i)' access Visio's document method
      msgtext=msgtext & Chr$(13)& doc.name
    Next i
  End If
  MsgBox msgtext
End Sub
```

See Also

Create Object, Get Object, New, Nothing, Typeof

Oct Function

Returns the octal representation of a number, as a string.

Syntax

Oct[\$](*number*)

Where...	Is...
<i>number</i>	A numeric expression to be converted to octal.

Remarks

If the numeric expression has a data type of Integer, the string contains up to six octal digits; otherwise, the expression will be converted to a data type of Long, and the string can contain up to 11 octal digits.

To represent an octal number directly, precede the octal value with `&O`. For example, `&O10` equals decimal 8 in octal notation.

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

Example

This example prints the octal values for the numbers from 1 to 15.

```
Sub main
  Dim x,y
  Dim msgtext
  Dim nofspace
  msgtext="Octal numbers from 1 to 15:" & Chr(10)
  For x=1 to 15
    nofspace=10
    y=Oct(x)
    If Len(x)=2 then
      nofspace=nofspace-2
    End If
    msgtext=msgtext & Chr(10) & x & Space(nospace) & y
  Next x
  MsgBox msgtext
End Sub
```

See Also

Hex

OKButton Statement

Determines the position and size of an OK button in a dialog box.

Syntax

OKButton *x, y, dx, dy* [, *.id*]

Where...	Is...
<i>x, y</i>	The position of the OK button relative to the upper left corner of the dialog box.
<i>dx, dy</i>	The width and height of the button.
<i>.id</i>	An optional identifier for the button.

Remarks

A *dy* value of 14 typically accommodates text in the system font. *.id* is an optional identifier used by the dialog statements that act on this control. Use the **OKButton** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a dropcombo box and the OK and Cancel buttons.

```
Sub main
  Dim cchoices as String
  On Error Resume Next
  cchoices="All"+Chr$(9)+"Nothing"
  Begin Dialog UserDialog 180, 95, "BSL Dialog Box"
    ButtonGroup .ButtonGroup1
    Text 9, 3, 69, 13, "Filename:", .Text1
    DropComboBox 9, 17, 111, 41, cchoices, .ComboBox1
    OKButton 131, 8, 42, 13
    CancelButton 131, 27, 42, 13
  End Dialog
  Dim mydialogbox As UserDialog
  Dialog mydialogbox
  If Err=102 then
    MsgBox "You pressed Cancel."
  Else
    MsgBox "You pressed OK."
  End If
End Sub
```

See Also

Begin...End Dialog, **Button**, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **Dialog**, **DropComboBox**, **GroupBox**, **ListBox**, **OptionButton**, **OptionGroup**, **Picture**, **StaticComboBox**, **Text**, **TextBox**

On...Goto Statement

Branch to a label in the current procedure based on the value of a numeric expression.

Syntax

ON *numeric-expression* **GoTo** *label1* [*,label2* , ...]

Where...

Is...

<i>numeric-expression</i>	Any numeric expression that evaluates to a positive number.
<i>label1</i> , <i>label2</i>	A label in the current procedure to branch to if <i>numeric-expression</i> evaluates to 1, 2, etc.

Remarks

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 "Illegal function call" error is issued.

Example

This example sets the current system time to the user's entry. If the entry cannot be converted to a valid time value, this subroutine sets the variable to Null. It then checks the variable and if it is Null, uses the **On...Goto** statement to ask again.

```
Sub main
    Dim answer as Integer
    answer=InputBox("Enter a choice (1-3) or 0 to quit")
    On answer Goto c1, c2, c3
    MsgBox("You typed 0.")
    Exit Sub
c1: MsgBox("You picked choice 1.")
    Exit Sub
c2: MsgBox("You picked choice 2.")
    Exit Sub
c3: MsgBox("You picked choice 3.")
    Exit Sub
End Sub
```

See Also

Goto, **Select Case**

On Error Statement

Specifies the location of an error-handling routine within the current procedure.

Syntax

ON [Local] **Error** {GoTo label [Resume Next] GoTo 0}

Where...	Is...
<i>label</i>	A string used as a label in the current procedure to identify the lines of code that process errors.

Remarks

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, BSL will terminate the execution of the program.

An **On Error** statement is composed of the following parts:

Part	Definition
Local	Keyword allowed in error-handling routines at the procedure level. Used to 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, BSL generates an error message.
Resume Next	Designates that error-handling code is handled by the statement that immediately follows the statement that caused an error. At this point, use the Err function 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, BSL will search for an error handler in the procedure that called the current procedure (if this fails, BSL will look for a handler belonging to the caller's caller, and so on). 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.

Example

This example prompts the user for a drive and directory name and uses On Error to trap invalid entries.

```
Sub main
    Dim userdrive, userdir, msgtext
in1:  userdrive=InputBox("Enter drive:",,"C:")
    On Error Resume Next
    ChDrive userdrive
    If Err=68 then
        MsgBox "Invalid Drive. Try again."
        Goto in1
    End If
in2:  On Error Goto Errhdlr1
    userdir=InputBox("Enter directory path:")
    ChDir userdrive & userdir
    MsgBox "New default directory is: " & userdrive & userdir
    Exit Sub
Errhdlr1:
    Select Case Err
        Case 75
            msgtext="Path is invalid."
        Case 76
            msgtext="Path not found."
        Case 70
            msgtext="Permission denied."
        Case Else
            msgtext="Error " & Err & ": " & Error$ & "occurred."
    End Select
    MsgBox msgtext & " Try again."
    Resume in2
End Sub
```

See Also

Erl, Err Function, Err Statement, Error Function, Error Statement, Resume

Open Statement

Opens a file or device for input or output.

Syntax

Open filename\$ [**For** mode] [**Access** access] [**lock**] **As** [#] filename% [**Len =** reclen]

Where...	Is...
<i>filename</i> \$	A string or string expression for the name of the file to open.
<i>mode</i>	One of the following keywords: Input – Put data into the file sequentially. Output – Read data from the file sequentially. Append – Add data to the file sequentially. Random – Get data from the file by random access. Binary – Get binary data from the file.
<i>access</i>	One of the following keywords: Read – Read data from the file only. Write – Write data the file only. Read Write – Read or write data to the file.
<i>lock</i>	One of the following keywords to designate access by other processes: Shared – Read or write available on the file. Lock Read – Read data only. Lock Write – Write data only. Lock ReadWrite – No read or write available.
<i>filename</i> %	An integer or expression containing the integer to assign to the open file (between 1 and 255).
<i>reclen</i>	The length of the records (for Random or Binary files only).

Remarks

A file must be opened before any input/output operation can be performed on it.

If *filename*\$ does not exist, it is created when opened in Append, Binary, Output or Random modes.

If *mode* is not specified, it defaults to Random.

If *access* is not specified for Random or Binary modes, *access* is attempted in the following order:
 Read Write, Write, Read.

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.

Use the **FreeFile** function to find the next available value for *filename*%.

Reclen is ignored for Input, Output, and Append *modes*.

Example

This example opens a file for Random access, gets the contents of the file, and closes the file again. The second subprogram, CREATEFILE, creates the file C:\TEMP001 used by the main subprogram.

```
Declare Sub createfile()
Sub main
  Dim acctno as String*3
  Dim recno as Long
  Dim msgtext as String
  Call createfile
  recno=1
  newline=Chr(10)
  Open "C:\TEMP001" For Random As #1 Len=3
  msgtext="The account numbers are:" & newline
  Do Until recno=11
    Get #1,recno,acctno
    msgtext=msgtext & acctno
    recno=recno+1
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub
Sub createfile()
  Rem Put the numbers 1-10 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=1 to 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also

Close, FreeFile

OptionButton Statement

Defines the position and text associated with an option button in a dialog box.

Syntax

OptionButton *x*, *y*, *dx*, *dy*, *text\$* [, *.id*]

Where...	Is...
<i>x</i> , <i>y</i>	The position of the button relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the button.
<i>text\$</i>	A string to display next to the option button. If the width of this string is greater than <i>dx</i> , trailing characters are truncated.
<i>.id</i>	An optional identifier used by the dialog statements that act on this control.

Remarks

You must have at least two **OptionButton** statements in a dialog box. You use these statements in conjunction with the **OptionGroup** statement. A *dy* value of 12 typically accommodates text in the system font. To enable the user to select an option button by typing a character from the keyboard, precede the character in *text\$* with an ampersand (&). Use the **OptionButton** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example creates a dialog box with a group box with two option buttons: "All pages" and "Range of pages."

```
Sub main
  Begin Dialog UserDialog 183, 70, "BSL Dialog Box"
    GroupBox 5, 4, 97, 57, "File Range"
    OptionGroup .OptionGroup2
      OptionButton 16, 12, 46, 12, "All pages", .OptionButton3
      OptionButton 16, 28, 67, 8, "Range of pages", .OptionButton4
    Text 22, 39, 20, 10, "From:", .Text6
    Text 60, 39, 14, 9, "To:", .Text7
    TextBox 76, 39, 13, 12, .TextBox4
    TextBox 44, 39, 12, 11, .TextBox5
    OKButton 125, 6, 54, 14
    CancelButton 125, 26, 54, 14
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin...End Dialog, **Button**, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **Dialog**, **DropComboBox**, **GroupBox**, **ListBox**, **OKButton**, **OptionGroup**, **Picture**, **StaticComboBox**, **Text**, **TextBox**

OptionGroup Statement

Groups a series of option buttons under one heading in a dialog box.

Syntax

OptionGroup *.field*

Where...

Is...

.field A value for the option button selected by the user: 0 for the first option button, 1 for the second button, and so on.

Remarks

The **OptionGroup** statement is used in conjunction with **OptionButton** statements to set up a series of related options. The **OptionGroup** Statement begins the definition of the option buttons and establishes the dialog-record field that will contain the option selection.

Use the **OptionGroup** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example creates a dialog box with a group box with two option buttons: "All pages" and "Range of Pages."

```
Sub main
  Begin Dialog UserDialog 192, 71, "BSL Dialog Box"
    GroupBox 7, 6, 97, 57, "File Range"
      OptionGroup .OptionGroup2
        OptionButton 18, 14, 46, 12, "All pages", .OptionButton3
        OptionButton 18, 30, 67, 8, "Range of pages", .OptionButton4
      Text 24, 41, 20, 10, "From:", .Text6
      Text 62, 41, 14, 9, "To:", .Text7
      TextBox 78, 41, 13, 12, .TextBox4
      TextBox 46, 41, 12, 11, .TextBox5
      OKButton 126, 6, 54, 14
      CancelButton 126, 26, 54, 14
    End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin...End Dialog, **Button**, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **Dialog**, **DropComboBox**, **GroupBox**, **Listbox**, **OKButton**, **OptionButton**, **Picture**, **StaticComboBox**, **Text**, **TextBox**

Option Base Statement

Specifies the default lower bound to use for array subscripts.

Syntax

Option Base *lowerBound*%

Where...	Is...
<i>lowerBound</i>	A number or expression containing a number for the default lower bound: either 0 or 1.

Remarks

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.

Example

This example resizes an array if the user enters more data than can fit in the array. It uses LBound and UBound to determine the existing size of the array and **ReDim** to resize it. **Option Base** sets the default lower bound of the array to 1.

```
Option Base 1
Sub main
    Dim arrayvar() as Integer
    Dim count as Integer
    Dim answer as String
    Dim x, y as Integer
    Dim total
    total=0
    x=1
    count=InputBox("How many test scores do you have?")
    ReDim arrayvar(count)
start:
    Do until x=count+1
        arrayvar(x)=InputBox("Enter test score #" &x & ":")
        x=x+1
    Loop
    answer=InputBox$("Do you have more scores? (Y/N)")
    If answer="Y" or answer="y" then
        count=InputBox("How many more do you have?")
        If count<>0 then
            count=count+(x-1)
            ReDim Preserve arrayvar(count)
            Goto start
        End If
    End If
    x=LBound(arrayvar,1)
    count=UBound(arrayvar,1)
    For y=x to count
        total=total+arrayvar(y)
    Next y
    MsgBox "The average of " & count & " scores is: " & Int(total/count)
End Sub
```

See Also

Dim, Global, LBound, ReDim, Static

Option Compare Statement

Specifies the default method for string comparisons: either case-sensitive or case-insensitive.

Syntax

Option Compare { Binary | Text }

Where...	Means...
Binary	Comparisons are case-sensitive (lowercase and uppercase letters are different).
Text	Comparisons are not case-sensitive.

Remarks

Binary comparisons compare strings based upon the ANSI character set. Text comparisons are based upon the relative order of characters as determined by the settings for your system in the Regional and Language Options of your Windows Control Panel.

Example

This example compares two strings: "JANE SMITH" and "jane smith". When **Option Compare** is Text, the strings are considered the same. If **Option Compare** is Binary, they will not be the same. Binary is the default. To see the difference, run the example once, then run it again, commenting out the **Option Compare** statement.

```
Option Compare Text
Sub main
    Dim strg1 as String
    Dim strg2 as String
    Dim retvalue as Integer
    strg1="JANE SMITH"
    strg2="jane smith"
i:
    retvalue=StrComp(strg1,strg2)
    If retvalue=0 then
        MsgBox "The strings are identical"
    Else
        MsgBox "The strings are not identical"
    Exit Sub
End If
End Sub
```

See Also

Instr, StrComp

Option Explicit Statement

Specifies that all variables in a module *must* be explicitly declared.

Syntax

Option Explicit

Remarks

By default, Basic automatically declares 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.

Example

This example specifies that all variables must be explicitly declared, thus preventing any mistyped variable names.

```
Option Explicit
Sub main
    Dim counter As Integer
    Dim fixedstring As String*25
    Dim varstring As String
    '...(code here)...
End Sub
```

See Also

Const, **Deftype**, **Dim**, **Function...End Function**, **Global**, **ReDim**, **Static**, **Sub...End Sub**

PasswordBox Function

Returns a string entered by the user without echoing it to the screen.

Syntax

PasswordBox[\$](*prompt\$* ,*[title\$]* ,*[default\$]* [,*xpos%* , *ypos%*])

Where...	Is...
<i>prompt\$</i>	A string expression containing the text to show in the dialog box
<i>title\$</i>	The caption for the dialog box's title bar
<i>default\$</i>	The string expression shown in the edit box as the default response.
<i>xpos%</i> , <i>ypos%</i>	The position of the dialog box, relative to the upper left corner of the screen.

Remarks

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 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.

If either *prompt\$* or *default\$* is omitted, nothing is displayed.

Xpos% determines the horizontal distance between the left edge of the screen and the left border of the dialog box, measured in dialog box units. *Ypos%* determines the horizontal distance from the top of the screen to the dialog box's upper edge, also in dialog box units. 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: 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 used.

Once the user presses Enter, or selects the OK button, **PasswordBox** returns the text contained in the password box. If the user selects Cancel, the **PasswordBox** function returns a null string ("").

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).

Example

This example asks the user for a password.

```
Sub main
    Dim retvalue
    Dim a
    retvalue=PasswordBox ("Enter your login password",Password)
    If retvalue<>"" then
        MsgBox "Verifying password"
    ' (continue code here)
    Else
        MsgBox "Login cancelled"
    End If
End Sub
```

See Also

InputBox, **MsgBox** Function, **MsgBox** Statement

Picture Statement

Defines a picture control in a dialog box

Syntax

Picture *x*, *y*, *dx*, *dy*, *filename\$*, *type* [, *.id*]

Where...	Is...
<i>x</i> , <i>y</i>	The position of the picture relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the picture.
<i>filename\$</i>	The name of the bitmap file (a file with .BMP extension) where the picture is located.
<i>type</i>	An integer for the location of the bitmap (0= <i>filename\$</i> , 3=Windows Clipboard).
<i>.id</i>	An optional identifier used by the dialog statements that act on this control.

Remarks

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

Note: The picture will be scaled equally in both directions and centered if the dimensions of the picture are not proportional to *dx* and *dy*.

If *type%* is 3, *filename\$* is ignored.

If the picture is not available (the file *filename\$* does not exist, does not contain a bitmap, or there is no bitmap on the Clipboard), the picture control will display the picture frame and the text "(missing picture)". This behavior can be changed by adding 16 to the value of *type%*. If *type%* is 16 or 19 and the picture is not available, a runtime error occurs.

Example

This example defines a dialog box with a picture, and the OK and Cancel buttons.

```
Sub main
  Begin Dialog UserDialog 148, 73, "BSL Dialog Box"
    Picture 8, 7, 46, 46, "C:\WINDOWS\CIRCLES.BMP", 0
    OKButton 80, 10, 54, 14
    CancelButton 80, 30, 54, 14
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin...End Dialog, **Button**, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **Dialog**, **DropComboBox**, **GroupBox**, **ListBox**, **OKButton**, **OptionButton**, **OptionGroup**, **StaticComboBox**, **Text**, **TextBox**

Pmt Function

Returns a constant periodic payment amount for an annuity or a loan.

Syntax

Pmt (*rate* , *nper* , *pv* , *fv* , *due*)

Where...	Is...
<i>rate</i>	Interest rate per period.
<i>nper</i>	Total number of payment periods.
<i>pv</i>	Present value of the initial lump sum amount paid (as in the case of an annuity) or received (as in the case of a loan).
<i>fv</i>	Future value of the final lump sum amount required (as in the case of a savings plan) or paid (0 as in the case of a loan).
<i>due</i>	An integer value for when the payments are due (0=end of each period, 1= beginning of the period).

Remarks

Rate is assumed to be constant over the life of the loan or annuity. If payments are on a monthly schedule, then *rate* will be 0.0075 if the annual percentage rate on the annuity or loan is 9%.

Example

This example finds the monthly payment on a given loan.

```
Sub main
  Dim aprate, totalpay
  Dim loanpv, loanfv
  Dim due, monthlypay
  Dim yearlypay, msgtext
  loanpv=InputBox("Enter the loan amount: ")
  aprate=InputBox("Enter the loan rate percent: ")
  If aprate >1 then
    aprate=aprate/100
  End If
  totalpay=InputBox("Enter the total number of monthly payments: ")
  loanfv=0
  'Assume payments are made at end of month
  due=0
  monthlypay=Pmt(aprate/12,totalpay,-loanpv,loanfv,due)
  msgtext="The monthly payment is: " & Format(monthlypay, "Currency")
  MsgBox msgtext
End Sub
```

See Also

FV, **IPmt**, **IRR**, **NPV**, **PV**, **PPmt**, **Rate**

PPmt Function

Returns the principal portion of the payment for a given period of an annuity.

Syntax

PPmt (*rate* , *per* , *nper* , *pv* , *fv* , *due*)

Where...	Is...
<i>rate</i>	Interest rate per period.
<i>per</i>	Particular payment period in the range 1 through <i>nper</i> .
<i>nper</i>	Total number of payment periods.
<i>pv</i>	Present value of the initial lump sum amount paid (as in the case of an annuity) or received (as in the case of a loan).
<i>fv</i>	Future value of the final lump sum amount required (as in the case of a savings plan) or paid (0 as in the case of a loan).
<i>due</i>	An integer value for when the payments are due (0=end of each period, 1= of the period).

Remarks

Rate is assumed to be constant over the life of the loan or annuity. If payments are on a monthly schedule, then *rate* will be 0.0075 if the annual percentage rate on the annuity or loan is 9%.

Example

This example finds the principal portion of a loan payment amount for payments made in last month of the first year. The loan is for \$25,000 to be paid back over 5 years at 9.5% interest.

```
Sub main
  Dim aprate, periods
  Dim payperiod
  Dim loanpv, due
  Dim loanfv, principal
  Dim msgtext
  aprate=9.5/100
  payperiod=12
  periods=120
  loanpv=25000
  loanfv=0
  Rem Assume payments are made at end of month
  due=0
  principal=PPmt(aprate/12,payperiod,periods,-loanpv,loanfv,due)
  msgtext="Given a loan of $25,000 @ 9.5% for 10 years," & Chr(10)
  msgtext=msgtext & " the principal paid in month 12 is: "
  MsgBox msgtext & Format(principal, "Currency")
End Sub
```

See Also

FV, IPmt, IRR, NPV, Pmt, PV, Rate

Print Statement

Prints data to an open file or to the screen.

Syntax

Print [*filename%*,] *expressionlist* [{ ; | , }]

Where...

Is...

<i>filename%</i>	An integer expression identifying the open file to use.
<i>expressionlist</i>	A numeric, string, and variant expression containing the list of values to print.

Remarks

The **Print** statement outputs data to the specified *filename%*. *filename%* is the number assigned to the file when it was opened. See the **Open** statement for more information. If this argument is omitted, the **Print** statement outputs data to the screen.

If the *expressionlist* is omitted, a blank line is written to the file.

The values in *expressionlist* 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 the “Input Function” on page 169 for more information on parsing this statement.

Example

This example prints the octal values for the numbers from 1 to 25.

```
Sub main
  Dim x as Integer
  Dim y
  For x=1 to 25
    y=Oct$(x)
    Print x Tab(10) y
  Next x
End Sub
```

See Also

Open, Spc, Tab, Write

PushButton Statement

Defines a custom push button.

Syntax A

PushButton *x, y, dx, dy, text\$ [, .id]*

Syntax B

Button *x, y, dx, dy, text\$ [, .id]*

Where...	Is...
<i>x, y</i>	The position of the button relative to the upper left corner of the dialog box.
<i>dx, dy</i>	The width and height of the button.
<i>text\$</i>	The name for the push button. If the width of this string is greater than <i>dx</i> , trailing characters are truncated.
<i>.id</i>	An optional identifier used by the dialog statements that act on this control.

Remarks

A *dy* value of 14 typically accommodates text in the system font. Use this statement to create buttons other than OK and Cancel. Use this statement in conjunction with the **ButtonGroup** statement. The two forms of the statement (**Button** and **PushButton**) are equivalent. Use the **Button** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a combination list box and three buttons.

```

Sub main
  Dim fchoices as String
  fchoices="File1" & Chr(9) & "File2" & Chr(9) & "File3"
  Begin Dialog UserDialog 185, 94, "BSL Dialog Box"
    Text 9, 5, 69, 10, "Filename:", .Text1
    DropComboBox 9, 17, 88, 71, fchoices, .ComboBox1
    ButtonGroup .ButtonGroup1
    OKButton 113, 14, 54, 13
    CancelButton 113, 33, 54, 13
    PushButton 113, 57, 54, 13, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub

```

See Also

Begin Dialog...End Dialog Statement, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, DropComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Put Statement

Writes a variable to a file opened in Random or Binary mode.

Syntax

Put [#] *filename%*, [*recnumber&*], *varname*

Where...	Is...
<i>filename%</i>	An integer expression identifying the open file to use.
<i>recnumber&</i>	A Long expression containing the record number or the byte offset at which to start writing.
<i>varname</i>	The name of the variable containing the data to write.

Remarks

Filename% is the number assigned to the file when it was opened. See the **Open** statement for more information.

Recnumber& is in the range 1 to 2,147,483,647. If *recnumber&* is omitted, the next record or byte is written.

Note: The commas before and after *recnumber%* are **required**, even if no *recnumber&* is specified.

Varname can be any variable except Object, Application Data Type or Array variables (single array elements can 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 *varname* 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 String 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.

Example

This example opens a file for Random access, puts the values 1-10 in it, prints the contents, and closes the file again.

```
Sub main
' Put the numbers 1-10 into a file
Dim x, y
Open "C:\TEMP001" as #1
For x=1 to 10
    Put #1,x, x
Next x
msgtext="The contents of the file is:" & Chr(10)
For x=1 to 10
    Get #1,x, y
    msgtext=msgtext & y & Chr(10)
Next x
Close #1
MsgBox msgtext
Kill "C:\TEMP001"
End Sub
```

See Also

Close, Get, Open, Write

PV Function

Returns the present value of a constant periodic stream of cash flows as in an annuity or a loan.

Syntax

PV (*rate* , *nper* , *pmt* , *fv* , *due*)

Where...	Is...
<i>rate</i>	Interest rate per period.
<i>nper</i>	Total number of payment periods.
<i>pmt</i>	Constant periodic payment per period.
<i>fv</i>	Future value of the final lump sum amount required (in the case of a savings plan) or paid (0 in the case of a loan).
<i>due</i>	An integer value for when the payments are due (0=end of each period, 1= beginning of the period).

Remarks

Rate is assumed constant over the life of the annuity. If payments are on a monthly schedule, then *rate* will be 0.0075 if the annual percentage rate on the annuity or loan is 9%.

Example

This example finds the present value of a 10-year \$25,000 annuity paying \$1,000 a year at 9.5%.

```
Sub main
  Dim aprate, periods
  Dim payment, annuityfv
  Dim due, presentvalue
  Dim msgtext
  aprate=9.5
  periods=120
  payment=1000
  annuityfv=25000
  Rem Assume payments are made at end of month
  due=0
  presentvalue=PV(aprate/12,periods,-payment, annuityfv,due)
  msgtext= "The present value for a 10-year $25,000 annuity @ 9.5%"
  msgtext=msgtext & " with a periodic payment of $1,000 is: "
  msgtext=msgtext & Format(presentvalue, "Currency")
  MsgBox msgtext
End Sub
```

See Also

FV, IPmt, IRR, NPV, Pmt, PPmt, Rate

Randomize Statement

Seeds the random number generator.

Syntax

Randomize [*number%*]

Where...	Is...
<i>number%</i>	An integer value between -32768 and 32767.

Remarks

If no *number%* argument is given, Basic uses the **Timer** function to initialize the random number generator.

Example

This example generates a random string of characters using the **Randomize** statement and **Rnd** function. The second **For...Next** loop is to slow down processing in the first **For...Next** loop so that **Randomize** can be seeded with a new value each time from the **Timer** function.

```
Sub main
  Dim x as Integer
  Dim y
  Dim str1 as String
  Dim str2 as String
  Dim letter as String
  Dim randomvalue
  Dim upper, lower
  Dim msgtext
  upper=Asc("z")
  lower=Asc("a")
  newline=Chr(10)
  For x=1 to 26
    Randomize timer() + x*255
    randomvalue=Int(((upper - (lower+1)) * Rnd) +lower)
    letter=Chr(randomvalue)
    str1=str1 & letter
    For y = 1 to 1500
      Next y
    Next x
    msgtext=str1
    MsgBox msgtext
  End Sub
```

See Also

Rnd, **Timer**

Rate Function

Returns the interest rate per period for an annuity or a loan.

Syntax

Rate (*nper* , *pmt* , *pv* , *fv* , *due* , *guess*)

Where...	Is...
<i>nper</i>	Total number of payment periods.
<i>pmt</i>	Constant periodic payment per period.
<i>pv</i>	Present value of the initial lump sum amount paid (as in the case of an annuity) or received (as in the case of a loan).
<i>fv</i>	Future value of the final lump sum amount required (in the case of a savings plan) or paid (0 in the case of a loan).
<i>due</i>	An integer value for when the payments are due (0=end of each period, 1= beginning of the period).
<i>guess</i>	A ballpark estimate for the rate returned.

Remarks

In general, a guess of between 0.1 (10 percent) and 0.15 (15 percent) would be a reasonable value for *guess*.

Rate is an iterative function: it improves the given value of *guess* over several iterations until the result is within 0.00001 percent. If it does not converge to a result within 20 iterations, it signals failure.

Example

This example finds the interest rate on a 10-year \$25,000 annuity, that pays \$100 per month.

```
Sub main
  Dim aprate
  Dim periods
  Dim payment, annuitypv
  Dim annuityfv, due
  Dim guess
  Dim msgtext as String
  periods=120
  payment=100
  annuitypv=0
  annuityfv=25000
  guess=.1
  Rem Assume payments are made at end of month
  due=0
  aprate=Rate(periods,-payment,annuitypv,annuityfv, due, guess)
  aprate=(aprate*12)
  msgtext= "The percentage rate for a 10-year $25,000 annuity "
  msgtext=msgtext & "that pays $100/month has "
  msgtext=msgtext & "a rate of: " & Format(aprate, "Percent")
  MsgBox msgtext
End Sub
```

See Also

FV, IPmt, IRR, NPV, Pmt, PPmt, PV

ReDim Statement

Changes the upper and lower bounds of a dynamic array's dimensions.

Syntax

```
ReDim [ Preserve ] variableName ( subscriptRange , ... ) [As [ New ] type ] , ...
```

Where...	Is...
<i>variableName</i>	The variable array name to redimension.
<i>subscriptRange</i>	The new upper and lower bounds for the array.
<i>type</i>	The type for the data elements in the array.

Remarks

ReDim re-allocates memory for the dynamic array to support the specified dimensions, and can optionally re-initialize the array elements. **ReDim** cannot be used at the module level; it must be used inside of a procedure.

The Preserve option is used to change the last dimension in the array while maintaining its contents. If Preserve is not specified, the contents of the array are re-initialized. Numbers will be set to zero (0). Strings and variants will be set to empty ("").

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.

A dynamic array is normally created by using **Dim** to declare an array without a specified *subscriptRange*. The maximum number of dimensions for a dynamic array created in this fashion is 8. If you need more than 8 dimensions, you can use the **ReDim** statement inside of a procedure to declare an array that 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 objects are not supported.

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

The **ReDim** statement cannot be used to change the number of dimensions of a dynamic array once the array has been given dimensions. It can only 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.

Example

This example finds the net present value for a series of cash flows. The array variable that holds the cash flow amounts is initially a dynamic array that is redimensioned after the user enters the number of cash flow periods they have.

```
Sub main
    Dim aprate as Single
    Dim varray() as Double
    Dim cflowper as Integer
    Dim x as Integer
    Dim netpv as Double
    cflowper=InputBox("Enter number of cash flow periods:")
    ReDim varray(cflowper)
    For x= 1 to cflowper
        varray(x)=InputBox("Enter cash flow amount for period #" &x &":")
    Next x
    aprate=InputBox ("Enter discount rate:")
    If aprate>1 then
        aprate=aprate/100
    End If
    netpv=NPV(aprate,varray())
    MsgBox "The Net Present Value is: " & Format(netpv,"Currency")
End Sub
```

See Also

Dim, Global, Option Base, Static

Rem Statement

Identifies a line of code as a comment in a Basic program.

Syntax

Rem *comment*

Where...	Is...
<i>comment</i>	The text of the comment.

Remarks

Everything from **Rem** to the end of the line is ignored.

The single quote (!) can also be used to initiate a comment. Metacommands (for example, **\$CSTRING**) must be preceded by the single quote comment form.

Example

This example defines a dialog box with a combination list box and two buttons. The **Rem** statements describe each block of definition code.

```
Sub main
  Dim fchoices as String
  fchoices="File1" & Chr(9) & "File2" & Chr(9) & "File3"
  Begin Dialog UserDialog 185, 94, "BSL Dialog Box"
Rem The next two lines create the combo box
    Text 9, 5, 69, 10, "Filename:", .Text1
    DropComboBox 9, 17, 88, 71, fchoices, .ComboBox1
Rem The next two lines create the command buttons
    OKButton 113, 14, 54, 13
    CancelButton 113, 33, 54, 13
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

Reset Statement

Closes all open disk files and writes any data in the operating system buffers to disk.

Syntax

Reset

Example

This example creates a file, puts the numbers 1-10 in it, then attempts to Get past the end of the file. The On Error statement traps the error and execution goes to the Debugger code which uses **Reset** to close the file before exiting.

```
Sub main
  ' Put the numbers 1-10 into a file
  Dim x as Integer
  Dim y as Integer
  On Error Goto Debugger
  Open "C:\TEMP001" as #1 Len=2
  For x=1 to 10
    Put #1,x, x
  Next x
  Close #1
  msgtext="The contents of the file is:" & Chr(10)
  Open "C:\TEMP001" as #1 Len=2
  For x=1 to 10
    Get #1,x, y
    msgtext=msgtext & Chr(10) & y
  Next x
  MsgBox msgtext
done:
  Close #1
  Kill "C:\TEMP001"
  Exit Sub
Debugger:
  MsgBox "Error " & Err & " occurred. Closing open file."
  Reset
  Resume done
End Sub
```

See Also

Close

Resume Statement

Halts an error-handling routine.

Syntax A

Resume Next

Syntax B

Resume *label*

Syntax C

Resume [0]

Where...

Is...

label The label that identifies the statement to go to after handling an error.

Remarks

When the **Resume Next** statement is used, control is passed to the statement that immediately follows the statement in which the error occurred. When the **Resume [0]** statement is used, control is passed to the statement in which the error occurred. The location of the error handler that 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.

Example

This example prints an error message if an error occurs during an attempt to open a file. The Resume statement jumps back into the program code at the label, done. From here, the program exits.

```
Sub main
  Dim msgtext, userfile
  On Error GoTo Debugger
  msgtext="Enter the filename to use:"
  userfile=InputBox$(msgtext)
  Open userfile For Input As #1
  MsgBox "File opened for input."
  ' ...etc....
  Close #1
done:
  Exit Sub
Debugger:
  msgtext="Error number " & Err & " occurred at line: " & Erl
  MsgBox msgtext
  Resume done
End Sub
```

See Also

Erl, Err Function, Err Statement, Error, Error Function, On Error, Trappable Errors

Right Function

Returns a string of a specified number of characters copied from the end of another string.

Syntax

Right[\$](*string*\$, *length*%)

Where...	Is...
<i>string</i> \$	A string or expression containing the string to copy.
<i>length</i> %	The number of characters to copy.

Remarks

If *length*% is greater than the length of *string*\$, **Right** returns the whole string.

Right accepts any type of *string*\$, including numeric values, and will convert the input value to a 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 *string*\$ is NULL, a variant of vartype 1 (Null) is returned.

To obtain a string of a specified number of bytes, copied from the end of another string, use the **RightB** function.

Example

This example checks for the extension .BMP in a filename entered by a user and activates the Paintbrush application if the file is found. Note this uses the Option Compare statement to accept either uppercase or lowercase letters for the filename extension.

```
Option Compare Text
Sub main
  Dim filename as String
  Dim x
  filename=InputBox("Enter a .BMP file and path: ")
  extension=Right(filename,3)
  If extension="BMP" then
    Shell "PBrush"
    For I = 1 to 10
      DoEvents
    Next i
    AppActivate "untitled - Paint"
    DoEvents
    Sendkeys "%FO" & filename & "{Enter}", 1
  Else
    MsgBox "File not found or extension not .BMP."
  End If
End Sub
```

See Also

GetField, **Instr**, **Left**, **Len**, **LTrim**, **Mid Function**, **Mid Statement**, **RTrim**, **Trim**

Rmdir Statement

Removes a directory.

Syntax

Rmdir *path*\$

Where... **Is...**

path\$ A string expression identifying the directory to remove.

Remarks

The syntax for *path*\$ 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.

Example

This example makes a new temporary directory in C:\ and then deletes it.

```
Sub main
  Dim path as String
  On Error Resume Next
  path=CurDir(C)
  If path<>"C:\" then
    ChDir "C:\"
  End If
  Mkdir "C:\TEMP01"
  If Err=75 then
    MsgBox "Directory already exists"
  Else
    MsgBox "Directory C:\TEMP01 created"
    MsgBox "Now removing directory"
    Rmdir "C:\TEMP01"
  End If
End Sub
```

See Also

ChDir, ChDrive, CurDir, Dir, Mkdir

Rnd Function

Returns a single precision random number between 0 and 1.

Syntax

Rnd [(*number!*)]

Where...

Is...

<i>number!</i>	A numeric expression to specify how to generate the random numbers. (<0=use the number specified, >0=use the next number in the sequence, 0=use the number most recently generated.)
----------------	--

Remarks

If *number!* is omitted, **Rnd** uses the next number in the sequence to generate a random number. The same sequence of random numbers is generated whenever **Rnd** is run, unless the random number generator is re-initialized by the **Randomize** statement.

Example

This example generates a random string of characters within a range. The **Rnd** function is used to set the range between lowercase "a" and "z". The second **For...Next** loop is to slow down processing in the first **For...Next** loop so that **Randomize** can be seeded with a new value each time from the **Timer** function.

```
Sub main
  Dim x as Integer
  Dim y
  Dim str1 as String
  Dim str2 as String
  Dim letter as String
  Dim randomvalue
  Dim upper, lower
  Dim msgtext
  upper=Asc("z")
  lower=Asc("a")
  newline=Chr(10)
  For x=1 to 26
    Randomize timer() + x*255
    randomvalue=Int(((upper - (lower+1)) * Rnd) +lower)
    letter=Chr(randomvalue)
    str1=str1 & letter
    For y = 1 to 1500
      Next y
    Next x
    msgtext=str1
  MsgBox msgtext
End Sub
```

See Also

Exp, Fix, Int, Log, Randomize, Sgn, Sqr

Rset Statement

Right aligns one string inside another string.

Syntax

Rset *string\$* = *string-expression*

Where...	Is...
<i>string\$</i>	The string to contain the right-aligned characters.
<i>string-expression</i>	The string containing the characters to put into <i>string\$</i> .

Remarks

If *string\$* is longer than *string-expression*, the leftmost 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.

Example

This example uses **Rset** to right-align an amount entered by the user in a field that is 15 characters long. It then pads the extra spaces with asterisks (*) and adds a dollar sign (\$) and decimal places (if necessary).

```
Sub main
  Dim amount as String*15
  Dim x
  Dim msgtext
  Dim replacement
  replacement="*"
  amount=InputBox("Enter an amount:")
  position=InStr(amount, ".")
  If Right(amount,3)<>".00" then
    amount=Rtrim(amount) & ".00"
  End If
  Rset amount="$" & Rtrim(amount)
  length=15-Len(Ltrim(amount))
  For x=1 to length
    Mid(amount,x)=replacement
  Next x
  MsgBox "Formatted amount: " & amount
End Sub
```

See Also

Lset

RTrim Function

Copies a string and removes any trailing spaces.

Syntax

RTrim[\$](*string*\$)

Where...	Is...
<i>string</i> \$	An expression that evaluates to a string.

Remarks

RTrim accepts any type of *string* including numeric values and will convert the input value to a 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 *string* is NULL, a variant of vartype 1 (Null) is returned.

Example

This example asks for an amount and then right-aligns it in a field that is 15 characters long. It uses **Rtrim** to trim any trailing spaces in the amount string, if the number entered by the user is less than 15 digits.

```
Sub main
  Dim amount as String*15
  Dim x
  Dim msgtext
  Dim replacement
  replacement="X"
  amount=InputBox("Enter an amount:")
  position=InStr(amount, ".")
  If position=0 then
    amount=Rtrim(amount) & ".00"
  End If
  Rset amount="$" & Rtrim(amount)
  length=15-Len(Ltrim(amount))
  For x=1 to length
    Mid(amount,x)=replacement
  Next x
  MsgBox "Formatted amount: " & amount
End Sub
```

See Also

GetField, **Left**, **Len**, **LTrim**, **Mid Function**, **Mid Statement**, **Right**, **Trim**

Second Function

Returns the second component (0-59) of a date-time value.

Syntax

Second(*time*)

Where...	Is...
----------	-------

<i>time</i>	An expression containing a date time value.
-------------	---

Remarks

Second accepts any type of *time* including strings and will attempt to convert the input value to a date value.

The return value is a variant of vartype 2 (integer). If the value of *time* is NULL, a variant of vartype 1 (Null) is returned.

Example

This example displays the last saved date and time for a file whose name is entered by the user.

```
Sub main
  Dim filename as String
  Dim ftime
  Dim hr, min
  Dim sec
  Dim msgtext as String
i: msgtext="Enter a filename:"
  filename=InputBox(msgtext)
  If filename="" then
    Exit Sub
  End If
  On Error Resume Next
  ftime=FileDateTime(filename)
  If Err<>0 then
    MsgBox "Error in file name. Try again."
    Goto i:
  End If
  hr=Hour(ftime)
  min=Minute(ftime)
  sec=Second(ftime)
  MsgBox "The file's time is: " & hr & ":" & min & ":" & sec
End Sub
```

See Also

Day, Hour, Minute, Month, Now, Time Function, Time Statement, Weekday, Year

Seek Function

Returns the current file position for an open file.

Syntax

Seek(*filenumber%*)

Where... **Is...**

filenumber% An integer expression identifying an open file to query.

Remarks

Filenumber% is the number assigned to the file when it was opened. See the “Open Statement” on page 225 for more information.

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.

Example

This example reads the contents of a sequential file line by line (to a carriage return) and displays the results. The second subprogram, CREATEFILE, creates the file “C:\TEMP001” used by the main subprogram.

```

Declare Sub createfile
Sub main
  Dim testscore as String
  Dim x
  Dim y
  Dim newline
  Call createfile
  Open "C:\TEMP001" for Input as #1
  x=1
  newline=Chr(10)
  msgtext= "The test scores are: " & newline
  Do Until x=Lof(1)
    Line Input #1, testscore
    x=x+1
    y=Seek(1)
    If y>Lof(1) then
      x=Lof(1)
    Else
      Seek 1,y
    End If
    msgtext=msgtext & newline & testscore
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
End Sub
Sub createfile()
  Rem Put the numbers 10-100 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=10 to 100 step 10
    Write #1, x
  Next x
  Close #1
End Sub

```

See Also

Get, Open, Put, Seek Statement

Seek Statement

Sets the position within an open file for the next read or write operation.

Syntax

Seek [#] *filenumber%*, *position&*

Where...

Is...

<i>filenumber%</i>	An integer expression identifying an open file to query.
<i>position&</i>	A numeric expression for the starting position of the next read or write operation (record number or byte offset).

Remarks

The **Seek** statement. 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 that specifies a negative or zero position.

Filenumber% is an integer expression identifying the open file to **Seek** in. See the "Open Statement" on page 225 for more information.

For files opened in Random mode, *position&* is a record number; for all other modes, *position&* is a byte offset. *Position&* is in the range 1 to 2,147,483,647. The first byte or record in the file is at position 1, the second is at position 2, etc.

Example

This example reads the contents of a sequential file line by line (to a carriage return) and displays the results. The second subprogram, CREATEFILE, creates the file "C:\TEMP001" used by the main subprogram.

```
Declare Sub createfile
Sub main
  Dim testscore as String
  Dim x
  Dim y
  Dim newline
  Call createfile
  Open "C:\TEMP001" for Input as #1
  x=1
  newline=Chr(10)
  msgtext= "The test scores are: " & newline
  Do Until x=Lof(1)
    Line Input #1, testscore
    x=x+1
    y=Seek(1)
    If y>Lof(1) then
      x=Lof(1)
    Else
      Seek 1,y
    End If
    msgtext=msgtext & newline & testscore
  Loop
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
```

```
End Sub
Sub createfile()
  Rem Put the numbers 10-100 into a file
  Dim x as Integer
  Open "C:\TEMP001" for Output as #1
  For x=10 to 100 step 10
    Write #1, x
  Next x
  Close #1
End Sub
```

See Also**Get, Open, Put, Seek Function**

Select Case Statement

Executes a series of statements, depending on the value of an expression.

Syntax

Select Case *testexpression*

[**Case** *expressionlist*
 [*statement_block*]]

[**Case** *expressionlist*
 [*statement_block*]]

.

[**Case Else**
 [*statement_block*]]

End Select

Where...	Is...
<i>testexpression</i>	Any expression containing a variable to test.
<i>expressionlist</i>	One or more expressions that contain a possible value for <i>testexpression</i> .
<i>statement_block</i>	The statements to execute if <i>testexpression</i> equals <i>expressionlist</i> .

Remarks

When there is a match between *testexpression* and one of the values in *expressionlist*, the *statement_block* following the **Case** clause is executed. When the next **Case** clause is reached, execution control goes to the statement following the **End Select** statement.

The *expressionlist(s)* can be a comma-separated list of expressions of the following forms:

expression

expression To expression

Is *comparison_operator expression*

The type of each *expression* must be compatible with the type of *testexpression*.

Note: 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: <, >, =, <=, >=, <>.

Each *statement_block* can contain any number of statements on any number of lines.

Example

This example tests the attributes for a file and, if hidden, changes it to a non-hidden file.

```
Sub main
  Dim filename as String
  Dim attribs, saveattribs as Integer
  Dim answer as Integer
  Dim archno as Integer
  Dim msgtext as String
  archno=32
  On Error Resume Next
  msgtext="Enter name of a file:"
  filename=InputBox(msgtext)
  attribs=GetAttr(filename)
  If Err<>0 then
    MsgBox "Error in filename. Re-run Program."
    Exit Sub
  End If
  saveattribs=attribs
  If attribs>= archno then
    attribs=attribs-archno
  End If
  Select Case attribs
    Case 2,3,6,7
      msgtext=" File: " &filename & " is hidden." & Chr(10)
      msgtext=msgtext & Chr(10) & " Change it?"
      answer=Msgbox(msgtext,308)
      If answer=6 then
        SetAttr filename, saveattribs-2
        MsgBox "File is no longer hidden."
        Exit Sub
      End If
      MsgBox "Hidden file not changed."
    Case Else
      MsgBox "File was not hidden."
  End Select
End Sub
```

See Also

If...Then...Else, On...Goto, Option Compare

SendKeys Statement

Send keystrokes to an active Windows application.

Syntax

SendKeys *string*\$ [, *wait*%]

Where...	Is...
<i>string</i> \$	An expression containing the characters to send.
<i>wait</i> %	A numeric expression to determine whether to wait until all keys are processed before continuing program execution (-1=wait, 0=do not wait).

Remarks

The keystrokes are represented by characters of *string*.

The default value for *wait* is 0 (FALSE).

To specify an ordinary character, enter this character in the *string*. For example, to send character 'a' use "a" as *string*. Several characters can be combined in one string: *string* "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 can be used to specify that the 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 they need to be sent with **SendKeys**. For example *string* "{%}" 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 might 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 'X' characters.

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}

Key	Keyword
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: Special keywords can be used in combination with +, %, and ^. For example: %{} 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 an application before sending keys (unless it is already active).

SendKeys cannot be used to send keys to an application that was not designed to run under Windows.

Example

This example activates the Windows 95 Phone Dialer application, dials the number and then allows the operating system to process events.

```

Sub main
    Dim phonenumber, msgtext
    Dim x
    phonenumber=InputBox("Type telephone number to call:")
    x=Shell("Dialer.exe",1)
    For i = 1 to 5
        DoEvents
    Next i
    AppActivate "Phone Dialer"
    SendKeys phonenumber & "{Enter}",1
    msgtext="Dialing.."
    MsgBox msgtext
    DoEvents
End Sub

```

See Also

AppActivate, DoEvents, Shell

Set Statement

Assigns a variable to an OLE2 object.

Syntax

Set *variableName* = *expression*

Where...	Is...
<i>variableName</i>	An object variable or a variant variable.
<i>expression</i>	An expression that evaluates to an object—typically a function, an object member, or Nothing.

Remarks

The following example shows the syntax for the **Set** statement:

```
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 = GetObject(,"spoly.cpoly")
```

Set differs from **Let** in that **Let** assigns an expression to a Basic variable. For example,

```
Set o1 = o2      Sets the object reference.
Let o1 = o2      Sets the value of the default member.
```

Example

This example displays a list of open files in the software application, Microsoft Visio. It uses the **Set** statement to assign Visio and its document files to object variables. To see how this example works, you need to start Visio and open one or more documents.

```
Sub main
    Dim visio as Object
    Dim doc as Object
    Dim msgtext as String
    Dim i as Integer, doccount as Integer

    'Initialize Visio
    Set visio = GetObject("visio.application") ' find Visio
    If (visio Is Nothing) then
        MsgBox "Couldn't find Visio!"
        Exit Sub
    End If

    'Get # of open Visio files
    doccount = visio.documents.count          'OLE2 call to Visio
    If doccount=0 then
        msgtext="No open Visio documents."
    Else
        msgtext="The open files are: " & Chr$(13)
        For i = 1 to doccount
            Set doc = visio.documents(i) ' access Visio's document method
            msgtext=msgtext & Chr$(13)& doc.name
        Next i
    End If
End Sub
```

```
MsgBox msgtext  
End Sub
```

See Also

CreateObject, Is, Me, New, Nothing, Object Class, Typeof

SetAttr Statement

Sets the attributes for a file.

Syntax

SetAttr *pathname\$*, *attributes%*

Where...	Is...
<i>pathname\$</i>	A string expression containing the filename to modify.
<i>attributes %</i>	An integer containing the new attributes for the file.

Remarks

Wildcards are not allowed in *pathname\$*. If the file is open, you can modify its attributes, but only if it is opened for **Read** access. 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

Example

This example tests the attributes for a file and if it is hidden, changes it to a normal (not hidden) file.

```
Sub main
    Dim filename as String
    Dim attribs, saveattribs as Integer
    Dim answer as Integer
    Dim archno as Integer
    Dim msgtext as String
    archno=32
    On Error Resume Next
    msgtext="Enter name of a file:"
    filename=InputBox(msgtext)
    attribs=GetAttr(filename)
    If Err<>0 then
        MsgBox "Error in filename. Re-run Program."
    Exit Sub
End If
saveattribs=attribs
If attribs>= archno then
    attribs=attribs-archno
End If
Select Case attribs
    Case 2,3,6,7
        msgtext=" File: " &filename & " is hidden." & Chr(10)
        msgtext=msgtext & Chr(10) & " Change it?"
        answer=Msgbox(msgtext,308)
        If answer=6 then
            SetAttr filename, saveattribs-2
```

```
        MsgBox "File is no longer hidden."  
    Exit Sub  
End If  
MsgBox "Hidden file not changed."  
Case Else  
    MsgBox "File was not hidden."  
End Select  
End Sub
```

See Also**FileAttr, GetAttr**

SetField Function

Replaces a field within a string and returns the modified string.

Syntax

SetField[\$](*string*\$, *field_number*%, *field*\$, *separator_chars*\$)

Where...	Is...
<i>string</i> \$	A string consisting of a series of fields, separated by <i>separator_char</i> \$.
<i>field_number</i> %	An integer for the field to replace within <i>string</i> \$.
<i>field</i> \$	An expression containing the new value for the field.
<i>separator_char</i> \$	A string containing the character(s) used to separate the fields in <i>string</i> \$.

Remarks

separator_char\$ can contain multiple separator characters, although the first one will be used as the separator character.

The *field_number*% starts with 1. 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.

It is legal for the new *field*\$ value to be a different size than the old value.

Note: BSL offers a number of extensions that are not included in Visual Basic.

Example

This example extracts the last name from a full name entered by the user.

```
Sub main
  Dim username as String
  Dim position as Integer
  username=InputBox("Enter your full name:")
  Do
    position=InStr(username," ")
    If position=0 then
      Exit Do
    End If
    username=SetField(username,1," "," ")
    username=Ltrim(username)
  Loop
  MsgBox "Your last name is: " & username
End Sub
```

See Also

GetField

Sgn Function

Returns a value indicating the sign of a number.

Syntax

Sgn(*number*)

Where...	Is...
<i>number</i>	An expression for the number to use.

Remarks

The value that the **Sgn** function returns depends on the sign of *number*.

For *numbers* > 0, **Sgn** (*number*) returns 1.

For *numbers* = 0, **Sgn** (*number*) returns 0.

For *numbers* < 0, **Sgn** (*number*) returns -1.

Example

This example tests the value of the variable profit and displays 0 for profit if it is a negative number. The subroutine uses **Sgn** to determine whether profit is positive, negative or zero.

```
Sub main
    Dim profit as Single
    Dim expenses
    Dim sales
    expenses=InputBox("Enter total expenses: ")
    sales=InputBox("Enter total sales: ")
    profit=Val(sales)-Val(expenses)
    If Sgn(profit)=1 then
        MsgBox "Yeah! We turned a profit!"
    ElseIf Sgn(profit)=0 then
        MsgBox "Okay. We broke even."
    Else
        MsgBox "Uh, oh. We lost money."
    End If
End Sub
```

See Also

Exp, Fix, Int, Log, Rnd, Sqr

Shell Function

Starts a Windows application and returns its task ID.

Syntax

Shell(*pathname\$* , [*windowstyle%*])

Where...	Is...
<i>pathname\$</i>	The name of the program to execute
<i>windowstyle%</i>	An integer value for the style of the program's window (1-7).

Remarks

Shell runs an executable program. *Pathname\$* can be the name of any valid .COM, .EXE., .BAT, or .PIF file. Arguments or command line switches can be included. If *pathname\$* is not a valid executable file name, or if **Shell** cannot start the program, an error message occurs.

Windowstyle% is one of the following values:

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).

Shell returns the task ID for the program, a unique number that identifies the running program.

Example

This example activates the Windows 95 Phone Dialer application, dials the number and then allows the operating system to process events.

```
Sub main
  Dim phonenumber, msgtext
  Dim x
  phonenumber=InputBox("Type telephone number to call:")
  x=Shell ("Dialer.exe",1)
  For i = 1 to 5
    DoEvents
  Next i
  AppActivate "Phone Dialer"
  SendKeys phonenumber & "{Enter}",1
  msgtext="Dialing.."
  MsgBox msgtext
  DoEvents
End Sub
```

See Also

AppActivate, **Command**, **SendKeys**

Sin Function

Returns the sine of an angle specified in radians.

Syntax

Sin(*number*)

Where...	Is...
<i>number</i>	An expression containing the angle in radians.

Remarks

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

To convert degrees to radians, multiply by (PI/180). The value of PI is 3.14159.

Example

This example finds the height of the building, given the length of a roof and the roof pitch.

```
Sub main
    Dim height, rooflength
    Dim pitch
    Dim msgtext
    Const PI=3.14159
    Const conversion= PI/180
    pitch=InputBox("Enter the roof pitch in degrees:")
    pitch=pitch*conversion
    rooflength=InputBox("Enter the length of the roof in feet:")
    height=Sin(pitch)*rooflength
    msgtext="The height of the building is "
    msgtext=msgtext & Format(height, "##.##") & " feet."
    MsgBox msgtext
End Sub
```

See Also

Atn, Cos, Tan, Derived Trigonometric Functions

Space Function

Returns a string of spaces.

Syntax

Space[\$](*number*)

Where...

Is...

number A numeric expression for the number of spaces to return.

Remarks

number can be any numeric data type, but will be rounded to an integer. *number* must be between 0 and 32,767.

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).

Example

This example prints the octal numbers from 1 to 15 as a two-column list and uses **Space** to separate the columns.

```
Sub main
    Dim x,y
    Dim msgtext
    Dim nofspace
    msgtext="Octal numbers from 1 to 15:" & Chr(10)
    For x=1 to 15
        nofspace=10
        y=Oct(x)
        If Len(x)=2 then
            nofspace=nofspace-2
        End If
        msgtext=msgtext & Chr(10) & x & Space(nospace) & y
    Next x
    MsgBox msgtext
End Sub
```

See Also

Spc, **String**

Spc Function

Prints a number of spaces.

Syntax

Spc (*n*)

Where... Is...

n An integer for the number of spaces to output.

Remarks

The **Spc** function can be used only inside **Print** statement.

When the **Print** statement is used, the **Spc** function will use the following rules for determining the number of spaces to output:

1. If *n* is less than the total line width, **Spc** outputs *n* spaces.
2. If *n* is greater than the total line width, **Spc** outputs *n Mod width* spaces.
3. If the difference between the current print position and the output line width (call this difference *x*) is less than *n* or *n Mod width*, then **Spc** skips to the next line and outputs *n - x* spaces.

To set the width of a print line, use the **Width** statement.

Example

This example puts five spaces and the string "ABCD" to a file. The five spaces are derived by taking 15 MOD 10, or the remainder of dividing 15 by 10.

```
Sub main
  Dim str1 as String
  Dim x as String*10
  str1="ABCD"
  Open "C:\TEMP001" For Output As #1
  Width #1, 10
  Print #1, Spc(15); str1
  Close #1
  Open "C:\TEMP001" as #1 Len=12
  Get #1, 1,x
  MsgBox "The contents of the file is: " & x
  Close #1
  Kill "C:\TEMP001"
End Sub
```

See Also

Print, Space, Tab, Width

SQLClose Function

Disconnects from an ODBC data source connection that was established by **SQLOpen**.

Syntax

SQLClose (*connection&*)

Where...

Is...

connection& A named argument that must be a long integer, returned by **SQLOpen**.

Remarks

The return is a variant. Success returns 0 and the connection is subsequently invalid. If the connection is not valid, -1 is returned.

Example

This example opens the data source named "BSLTest," gets the names in the ODBC data sources, and closes the connection.

```
Sub main
'   Declarations
'
    Dim outputStr As String
    Dim connection As Long
    Dim prompt As Integer
    Dim datasources(1 To 50) As Variant
    Dim retcode As Variant

    prompt = 5
'   Open the datasource "BSLTest"
    connection = SQLOpen("DSN=BSLTest", outputStr, prompt:=5)

    action1 = 1 ' Get the names of the ODBC datasources
    retcode = SQLGetSchema(connection:=connection,action:=1, qualifier:=qualifier,
ref:=datasources())

'   Close the datasource connection
    retcode = SQLClose(connection)

End Sub
```

See Also

SQLError, **SQLExecQuery**, **SQLGetSchema**, **SQLOpen**, **SQLRequest**, **SQLRetrieve**, **SQLRetrieveToFile**

SQLError Function

Can be used to retrieve more detailed information about errors that might have occurred when making an ODBC function call. Returns errors for the last ODBC function and the last connection.

Syntax

SQLError (*destination*())

Where...	Is...
<i>destination</i>	A two dimensional array in which each row contains one error. A named argument that is required, must be an array of variants.

Remarks

There is no return value. The fields are: 1) character string indicating the ODBC error class/subclass, 2) numeric value indicating the data source native error code, 3) text message describing the error.

If there are no errors from a previous ODBC function call, then a 0 is returned in the caller's array at (1,1). If the array is not two dimensional or does not provide for the return of the three fields above, then an error message is returned in the caller's array at (1,1).

Example

This example forces an error to test SQLError function.

```
sub main
' Declarations
  Dim connection As long
  Dim prompt as integer
  Dim retcode as long
  Dim errors(1 To 3, 1 To 10) as Variant
' Open the datasource
  connection = SQLOpen("DSN=BSLTESTW;UID=DBA;PWD=SQL",outputStr,prompt:=3)
' force an error to test SQLError select a nonexistent table
  retcode = SQLExecQuery(connection:=connection,query:="select * from notable ")
' Retrieve the detailed error message information into the errors array
  SQLError destination:=errors
  retcode = SQLClose(connection)
End Sub
```

See Also

SQLClose, SQLExecQuery, SQLGetSchema, SQLOpen, SQLRequest, SQLRetrieve, SQLRetrieveToFile

SQLExecQuery Function

Executes an SQL statement on a connection established by **SQLOpen**.

Syntax

SQLExecQuery (*connection&* , *query\$*)

Where...	Is...
<i>connection&</i>	A named argument, required. A long integer, returned by SQLOpen .
<i>query\$</i>	A string containing a valid SQL statement. The return is a variant.

Remarks

It returns the number of columns in the result set for SQL **SELECT** statements; for **UPDATE**, **INSERT**, or **DELETE** it returns the number of rows affected by the statement. Any other SQL statement returns 0. If the function is unable to execute the query on the specified data source, or if the connection is invalid, a negative error code is returned.

If **SQLExecQuery** is called and there are any pending results on that connection, the pending results are replaced by the new results.

Example

This example performs a query on the data source.

```
Sub main
'   Declarations
'
'   Dim connection As Long
'   Dim destination(1 To 50, 1 To 125) As variant
'   Dim retcode As long

'   open the connection
connection = SQLOpen("DSN=BSLTest", outputStr, prompt:=3)
'
'   Execute the query
query = "select * from customer"
retcode = SQLExecQuery(connection, query)
'
'   retrieve the first 50 rows with the first 6 columns of each row into
'   the array destination, omit row numbers and put column names in the
'   first row of the array
'
'   retcode = SQLRetrieve(connection:=connection, destination:=destination,
columnNames:=1, rowNumbers:=0, maxRows:=50, maxColumns:=6, fetchFirst:=0)

'   Get the next 50 rows of from the result set
retcode = SQLRetrieve(connection:=connection, destination:=destination,
columnNames:=1, rowNumbers:=0, maxRows:=50, maxColumns:=6)

'   Close the connection
retcode = SQLClose(connection)
End Sub
```

See Also

SQLClose, **SQLError**, **SQLGetSchema**, **SQLOpen**, **SQLRequest**, **SQLRetrieve**, **SQLRetrieveToFile**

SQLGetSchema Function

Returns a variety of information, including information on the data sources available, current user ID, names of tables, names and types of table columns, and other data source/database related information.

Syntax

SQLGetSchema (*connection* , *action%* , *qualifier\$* , *ref()*)

Where...	Is...
<i>connection</i>	A long integer returned by SQLOpen.
<i>action%</i>	Required.
<i>qualifier\$</i>	Required.
<i>ref()</i>	A variant array for the results appropriate to the action requested, must be an array even if only one dimension with one element. The return is a variant.

Remarks

A negative return value indicates an error. A -1 is returned if the requested information cannot be found or if the connection is not valid. The destination array must be properly dimensioned to support the action or an error will be returned. Actions 2 and 3 are not currently supported. Action 4 returns all tables and does not support the use of the *qualifier*. Not all database products and ODBC drivers support all actions.

Action	Meaning
1	List of available datasources (dimension of <i>ref()</i> is one).
2	List of databases on the current connection (not supported).
3	List of owners in a database on the current connection (not supported).
4	List of tables on the specified connection.
5	List of columns in a the table specified by <i>qualifier</i> . (<i>ref()</i> must be two dimensions). Returns column name and SQL data type.
6	The user ID of the current connection user.
7	The name of the current database.
8	The name of the data source for the current connection.
9	The name of the DBMS the data source uses.
10	The server name for the data source.
11	The terminology used by the data source to refer to owners.
12	The terminology used by the data source to refer to a table.
13	The terminology used by the data source to refer to a qualifier.
14	The terminology used by the data source to refer to a procedure.

Example

This example opens the data source named "BSLTest," gets the names in the ODBC data sources, and closes the connection.

```
Sub main
'   Declarations
'
    Dim outputStr As String
    Dim connection As Long
    Dim prompt As Integer
    Dim datasources(1 To 50) As Variant
    Dim retcode As Variant

    prompt = 5
'   Open the datasource "BSLTest"
    connection = SQLOpen("DSN=BSLTest", outputStr, prompt:=5)

    action1 = 1 ' Get the names of the ODBC datasources
    retcode = SQLGetSchema(connection:=connection, action:=1, qualifier:=qualifier,
ref:=datasources())

'   Close the datasource connection
    retcode = SQLClose(connection)

End Sub
```

See Also

SQLClose, SQLError, SQLExecQuery, SQLOpen, SQLRequest, SQLRetrieve, SQLRetrieveToFile

SQLOpen Function

Establishes a connection to an ODBC data source specified in *connectStr* and returns a connection ID in the return, and the completed connection string in *outputStr*. If the connection cannot be established, then a negative number ODBC error is returned.

Syntax

SQLOpen (*connectStr\$* , *outputStr\$* , *prompt%*)

Where...	Is...
<i>connectStr</i>	A named argument, a required parameter.
<i>outputStr</i>	Optional
<i>prompt</i>	Optional.

Remarks

The content of *connectStr* is described in the *Microsoft Programmer's Reference Guide for ODBC*. An example string might be "DSN=datasourcename; UID=myid; PWD=mypassword". The return must be a long.

prompt specifies when the driver dialog box is displayed. When *prompt* is omitted, **SQLOpen** uses 2 as the default.

Value	Meaning
1	Driver dialog is always displayed.
2	Driver dialog is displayed only when the specification is not sufficient to make the connection.
3	The same as 2, except that dialogs that are not required are grayed and cannot be modified.
4	Driver dialog is not displayed. If the connection is not successful, an error is returned.

Example

This example opens the data source named "BSLTest," gets the names in the ODBC data sources, and closes the connection.

```
Sub main
  ' Declarations
  '
  Dim outputStr As String
  Dim connection As Long
  Dim prompt As Integer
  Dim datasources(1 To 50) As Variant
  Dim retcode As Variant

  prompt = 5
  ' Open the datasource "BSLTest"
  connection = SQLOpen("DSN=BSLTest", outputStr, prompt:=5)

  action1 = 1 ' Get the names of the ODBC datasources
  retcode = SQLGetSchema(connection:=connection, action:=1, qualifier:=qualifier,
ref:=datasources())
```

```
' Close the datasource connection
retcode = SQLClose(connection)
```

```
End Sub
```

See Also

SQLClose, SQLError, SQLExecQuery, SQLGetSchema, SQLRequest, SQLRetrieve, SQLRetrieveToFile

SQLRequest Function

Establishes a connection to the data source specified in *connectionStr*, executes the SQL statement contained in *query*, returns the results of the request in the *ref()* array, and closes the connection.

Syntax

SQLRequest(*connectionStr*\$, *query*\$, *outputStr*\$, *prompt*% , *columnNames*% , *ref*())

Where...	Is...
<i>connectionStr</i> \$	A required argument.
<i>query</i> \$	A required argument.
<i>outputStr</i> \$	Contains the completed connection string.
<i>prompt</i> %	An integer that specifies when driver dialog boxes are displayed (see the "SQLOpen Function" on page 277).
<i>columnNames</i> %	An integer with a value of 0 or nonzero. When <i>columnNames</i> is nonzero, column names are returned as the first row of the <i>ref()</i> array. If <i>columnNames</i> is omitted, the default is 0.
<i>ref</i> ()	A required argument that is a two dimensional variant array.

Remarks

In the event that the connection cannot be made, the query is invalid, or other error condition, a negative number error is returned. In the event the request is successful, the positive number of results returned or rows affected is returned. Other SQL statements return 0. The arguments are named arguments. The return is a variant.

Example

This example will open the datasource BSLTESTW and execute the query specified by *query* and return the results in *destination*

```
Sub main
' Declarations
,
Dim destination(1 To 50, 1 To 125) As Variant
Dim prompt As integer

' The following will open the datasource BSLTESTW and execute the query
' specified by query and return the results in destination
,
query = "select * from class"
retcode = SQLRequest("DSN=BSLTESTW;UID=DBA;PWD=SQL", query, outputStr,
prompt, 0, destination())
End Sub
```

See Also

SQLClose, SQLError, SQLExecQuery, SQLGetSchema, SQLOpen, SQLRetrieve, SQLRetrieveToFile

SQLRetrieve Function

Fetches the results of a pending query on the connection specified by *connection* and returns the results in the *destination()* array.

Syntax

SQLRetrieve(*connection*& , *destination()* , *maxColumns*% , *maxRows*% , *columnNames*% , *rowNumbers*% , *fetchFirst*%)

Where...	Is...
<i>connection</i> &	A long.
<i>destination()</i>	A two dimensional variant array.
<i>maxColumns</i> %	An integer and an optional parameter, used to specify the number of columns to be retrieved in the request.
<i>maxRows</i> %	An integer and an optional parameter, used to specify the number of rows to be retrieved in the request.
<i>columnNames</i> %	An integer and an optional parameter, defaults to 0.
<i>rowNumbers</i> %	An integer and an optional parameter, defaults to 0.
<i>fetchFirst</i> %	An integer and an optional parameter, defaults to 0.

Remarks

The return value is the number of rows in the result set or the *maxRows* requested. If the function is unable to retrieve the results on the specified connection, or if there are not results pending, -1 is returned. If no data is found, the function returns 0.

The arguments are named arguments. The return is a variant.

If *maxColumns* or *maxRows* are omitted, the array size is used to determine the maximum number of columns and rows retrieved, and an attempt is made to return the entire result set. Extra rows can be retrieved by using **SQLRetrieve** again and by setting *fetchFirst* to 0. If *maxColumns* specifies fewer columns than are available in the result, **SQLRetrieve** discards the rightmost result columns until the results fit the specified size.

When *columnNames* is nonzero, the first row of the array will be set to the column names as specified by the database schema. When *rowNumbers* is nonzero, row numbers are returned in the first column of *destination()*. **SQLRetrieve** will clear the user's array prior to fetching the results.

When *fetchFirst* is nonzero, it causes the result set to be repositioned to the first row if the database supports the function. If the database does not support repositioning, the result set -1 error will be returned.

If there are more rows in the result set than can be contained in the *destination()* array or than have been requested using *maxRows*, the user can make repeated calls to **SQLRetrieve** until the return value is 0.

Example

This example retrieves information from a data source.

```
Sub main
'   Declarations
'
'   Dim connection As Long
'   Dim destination(1 To 50, 1 To 125) As Variant
'   Dim retcode As long

'   open the connection
'   connection = SQLOpen("DSN=BSLTest",outputStr,prompt:=3)
'
'   Execute the query
'   query = "select * from customer"
'   retcode = SQLExecQuery(connection,query)

'   retrieve the first 50 rows with the first 6 columns of each row into
'   the array destination, omit row numbers and put column names in the
'   first row of the array

'   retcode = SQLRetrieve(connection:=connection,destination:=destination,
columnNames:=1,rowNumbers:=0,maxRows:=50, maxColumns:=6,fetchFirst:=0)

'   Get the next 50 rows of from the result set
'   retcode = SQLRetrieve(connection:=connection,destination:=destination,
columnNames:=1,rowNumbers:=0,maxRows:=50, maxColumns:=6)

'   Close the connection
'   retcode = SQLClose(connection)
End Sub
```

See Also

SQLClose, SQLError, SQLExecQuery, SQLGetSchema, SQLOpen, SQLRequest, SQLRetrieveToFile

SQLRetrieveToFile Function

Fetches the results of a pending query on the connection specified by *connection* and stores them in the file specified by *destination*.

Syntax

SQLRetrieveToFile(*connection&* , *destination\$* , *columnNames%* , *columnDelimiter\$*)

Where...	Is...
<i>connection&</i>	A required argument. A long integer.
<i>destination\$</i>	A required argument. A string containing the file and path to be used for storing the results.
<i>columnNames%</i>	An integer; when nonzero, the first row of the file will be set to the column names as specified by the database schema. If <i>columnNames</i> is omitted, the default is 0.
<i>columnDelimiter\$</i>	Specifies the string to be used to delimit the fields within each row. If <i>columnDelimiter</i> is omitted, a horizontal tab is used to delimit fields.

Remarks

Upon successful completion of the operation, the return value is the number of rows in the result set. If the function is unable to retrieve the results on the specified connection, or if there are not results pending, `-1` is returned.

The arguments are named arguments. The return is a variant.

Example

This example opens a connection to a data source and retrieves information to a file.

```

Sub main
'   Declarations
'
'   Dim connection As Long
'   Dim destination(1 To 50, 1 To 125) As Variant
'   Dim retcode As long

'   open the connection

connection = SQLOpen("DSN=BSLTest",outputStr,prompt:=3)
'
'   Execute the query
'
query = "select * from customer"
retcode = SQLExecQuery(connection,query)

'   Place the results of the previous query in the file named by
'   filename and put the column names in the file as the first row.
'   The field delimiter is %
'
filename = "c:\myfile.txt"
columnDelimiter = "% "
retcode = SQLRetrieveToFile(connection:=connection,destination:=filename,
columnNames:=1,columnDelimiter:=columnDelimiter)

retcode = SQLClose(connection)
End Sub

```

See Also

SQLClose, SQLError, SQLExecQuery, SQLGetSchema, SQLOpen, SQLRequest, SQLRetrieve

Sqr Function

Returns the square root of a number.

Syntax

Sqr(*number*)

Where...**Is...**

number

An expression containing the number to use.

Remarks

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.

Example

This example calculates the square root of 2 as a double-precision floating point value and displays it in scientific notation.

```
Sub main
    Dim value as Double
    Dim msgtext
    value=Cdbl(Sqr(2))
    msgtext= "The square root of 2 is: " & Format(Value,"Scientific")
    MsgBox msgtext
End Sub
```

See Also

Exp, Fix, Int, Log, Rnd, Sgn

Static Statement

Declares variables and allocate storage space.

Syntax

Static *variableName* [As *type*] [,*variableName* [As *type*]] ...

Where...	Is...
<i>variableName</i>	The name of the variable to declare.
<i>type</i>	The data type of the variable.

Remarks

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 a definition of that procedure. See the "Function...End Function Statement" on page 150 or the "Sub...End Sub Statement" on page 289 for more information.

Example

This example puts account numbers to a file using the record variable GRECORD and then prints them again.

```
Type acctrecord
  acctno as Integer
End Type
Sub main
  Static grecord as acctrecord
  Dim x
  Dim total
  x=1
  grecord.acctno=1
  On Error Resume Next
  Open "C:\TEMP001" For Output as #1
  Do While grecord.acctno<>0
i:   grecord.acctno=InputBox("Enter 0 or new account #" & x & ":")
    If Err<>0 then
      MsgBox "Error occurred. Try again."
      Err=0
      Goto i
    End If
    If grecord.acctno<>0 then
      Print #1, grecord.acctno
      x=x+1
    End If
  Loop
  Close #1
  total=x-1
  msgtext="The account numbers are: " & Chr(10)
  Open "C:\TEMP001" For Input as #1
  For x=1 to total
    Input #1, grecord.acctno
    msgtext=msgtext & Chr(10) & grecord.acctno
  Next x
  MsgBox msgtext
  Close #1
  Kill "C:\TEMP001"
```

End Sub

See Also

Dim, Function...End Function, Global, Option Base, ReDim, Sub...End Sub

StaticComboBox Statement

Creates a combination of a list of choices and a text box.

Syntax A

StaticComboBox *x*, *y*, *dx*, *dy*, *text\$*, *.field*

Syntax B

StaticComboBox *x*, *y*, *dx*, *dy*, *stringarray\$()*, *.field*

Where...	Is...
<i>x</i> , <i>y</i>	The upper left corner coordinates of the list box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the combo box in which the user enters or selects text.
<i>text\$</i>	A string containing the selections for the combo box.
<i>stringarray\$</i>	An array of dynamic strings for the selections in the combo box.
<i>.field</i>	The name of the dialog-record field that will hold the text string entered in the text box or chosen from the list box.

Remarks

The **StaticComboBox** statement is equivalent to the **ComboBox** or **DropComboBox** statement, but the list box of **StaticComboBox** always stays visible. All dialog functions and statements that apply to the **ComboBox** apply to the **StaticComboBox** as well.

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 the "Begin Dialog...End Dialog Statement" on page 34 for more information.

The *text\$* argument 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 the following example:

```
dimname = "listchoice"+Chr$(9)+"listchoice"+Chr$(9)+"listchoice"...
```

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. The *field* argument is also used by the dialog statements that act on this control.

Use the **StaticComboBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a static combo box labeled "Installed Drivers" and the OK and Cancel buttons.

```
Sub main
    Dim cchoices as String
    cchoices="MIDI Mapper"+Chr$(9)+"Timer"
    Begin Dialog UserDialog 182, 116, "BSL Dialog Box"
        StaticComboBox 7, 20, 87, 49, cchoices, .StaticComboBox1
        Text 6, 3, 83, 10, "Installed Drivers", .Text1
        OKButton 118, 12, 54, 14
        CancelButton 118, 34, 54, 14
    End Dialog
    Dim mydialogbox As UserDialog
    Dialog mydialogbox
    If Err=102 then
        MsgBox "You pressed Cancel."
    Else
        MsgBox "You pressed OK."
    End If
End Sub
```

See Also

Begin Dialog...End Dialog, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, Dialog, DropComboBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, Text, TextBox

Stop Statement

Halts program execution.

Syntax

Stop

Remarks

Stop statements can be placed anywhere in a program to suspend its execution. Although the **Stop** statement halts program execution, it does not close files or clear variables.

Example

This example stops program execution at the user's request.

```
Sub main
    Dim str1
    str1=InputBox("Stop program execution? (Y/N):")
    If str1="Y" or str1="y" then
        Stop
    End If
    MsgBox "Program complete."
End Sub
```

Str Function

Returns a string representation of a number.

Syntax

Str[\$](*number*)

Where...	Is...
<i>number</i>	The number to represent as a string.

Remarks

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.

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).

Example

This example prompts for two numbers, adds them, then shows them as a concatenated string.

```
Sub main
    Dim x as Integer
    Dim y as Integer
    Dim str1 as String
    Dim value1 as Integer
    x=InputBox("Enter a value for x: ")
    y=InputBox("Enter a value for y: ")
    MsgBox "The sum of these numbers is: " & x+y
    str1=Str(x) & Str(y)
    MsgBox "The concatenated string for these numbers is: " & str1
End Sub
```

See Also

Format, Val

StrComp Function

Compares two strings and returns an integer specifying the result of the comparison.

Syntax

StrComp(*string1\$* , *string2\$* [, *compare%*])

Where...	Is...
<i>string1\$</i>	Any expression containing the first string to compare.
<i>string2\$</i>	The second string to compare.
<i>compare%</i>	An integer for the method of comparison (0=case-sensitive, 1=case-insensitive).

Remarks

StrComp returns one of the following values:

Value	Meaning
-1	<i>string1\$</i> < <i>string2\$</i>
0	<i>string1\$</i> = <i>string2\$</i>
>1	<i>string1\$</i> > <i>string2\$</i>
Null	<i>string1\$</i> = Null or <i>string2\$</i> = Null

If *compare%* is 0, a case sensitive comparison based on the ANSI character set sequence is performed. If *compare%* is 1, a case insensitive comparison is done based upon the relative order of characters as determined by the settings for your system in the Regional and Language Options of your Windows Control Panel. If omitted, the module level default, as specified with **Option Compare** is 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.

Example

This example compares a user-entered string to the string "Smith".

```
Option Compare Text
Sub main
    Dim lastname as String
    Dim smith as String
    Dim x as Integer
    smith="Smith"
    lastname=InputBox("Type your last name")
    x=StrComp(lastname,smith,1)
    If x=0 then
        MsgBox "You typed 'Smith' or 'smith'."
    Else
        MsgBox "You typed: " & lastname & " not 'Smith'."
    End If
End Sub
```

See Also

Instr, Option Compare

String Function

Returns a string consisting of a repeated character.

Syntax A

String[\$](*number* , *Character%*)

Syntax B

String[\$] (*number* , *string-expression*\$)

Where...	Is...
<i>number</i>	The length of the string to be returned.
<i>Character%</i>	A numeric expression that contains an integer for the decimal ANSI of the character to use.
<i>string-expression</i> \$	A string argument, the first character of which becomes the repeated character.

Remarks

number must be between 0 and 32,767.

Character% must evaluate to an integer between 0 and 255.

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

Example

This example places asterisks (*) in front of a string that is printed as a payment amount.

```
Sub main
  Dim str1 as String
  Dim size as Integer
  i: str1=InputBox("Enter an amount up to 999,999.99: ")
  If Instr(str1,".")=0 then
    str1=str1+".00"
  End If
  If Len(str1)>10 then
    MsgBox "Amount too large. Try again."
    Goto i
  End If
  size=10-Len(str1)
  'Print amount in a space on a check allotted for 10 characters
  str1=String(size,Asc("*")) & str1
  MsgBox "The amount is: $" & str1
End Sub
```

See Also

Space, **Str**

Sub...End Sub Statement

Defines a subprogram procedure.

Syntax

```
[ Static ] [ Private ] Sub name [ ( [Optional ] parameter [ As type ] , ... ) ]
```

End Sub

Where...	Is...
<i>name</i>	The name of the subprogram.
<i>parameter</i>	A comma-separated list of parameter names.
<i>type</i>	A data type for <i>parameter</i>

Remarks

A call to a subprogram stands alone as a separate statement. See the “Call Statement” on page 40 for more information. Recursion is supported.

The data type of a parameter can be specified by using a type character or by using the *As* clause. Record parameters are declared by using an *As* clause and a *type* that has previously been defined using the **Type** statement. Array parameters are indicated by using empty parentheses after the *parameter*. The array dimensions are not specified in the **Sub** statement. All references to an array within the body of the subprogram must have a consistent number of dimensions.

If a *parameter* is declared as **Optional**, its value can be omitted when the function is called. Only variant parameters can be declared as optional, and all optional parameters must appear after all required parameters in the **Sub** statement. The function **IsMissing** must be used to check whether an optional parameter was omitted by the user or not. See the **Call** statement for more information on using named parameters.

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 that has a return value.

Example

This example is a subroutine that uses the **Sub...End Sub** function.

```
Sub main
    MsgBox "Hello, World."
End Sub
```

See Also

Call, Dim, Function...End Function, Global, Option Explicit, Static

Tab Function

Moves the current print position to the column specified.

Syntax

Tab (*n*)

Where...	Is...
<i>n</i>	The new print position to use.

Remarks

The **Tab** function can be used only inside **Print** statement. 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:

1. If *n* is less than the total line width, the new print position is *n*.
2. If *n* is greater than the total line width, the new print position is $n \bmod \text{width}$.
3. If the current print position is greater than *n* or $n \bmod \text{width}$, **Tab** skips to the next line and sets the print position to *n* or $n \bmod \text{width}$.

To set the width of a print line, use the **Width** statement.

Example

This example prints the octal values for the numbers from 1 to 25. It uses **Tab** to put five character spaces between the values.

```
Sub main
    Dim x as Integer
    Dim y
    For x=1 to 25
        y=Oct$(x)
        Print x Tab(10) y
    Next x
End Sub
```

See Also

Print, **Space**, **Spc**, **Width**

Tan Function

Returns the tangent of an angle in radians.

Syntax

Tan(*number*)

Where...	Is...
----------	-------

<i>number</i>	An expression containing the angle in radians.
---------------	--

Remarks

number is specified in radians, and can be either positive or negative.

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.

To convert degrees to radians, multiply by $\text{PI}/180$. The value of PI is 3.14159.

Example

This example finds the height of the exterior wall of a building, given its roof pitch and the length of the building.

```
Sub main
    Dim bldglen, wallht
    Dim pitch
    Dim msgtext
    Const PI=3.14159
    Const conversion= PI/180
    On Error Resume Next
    pitch=InputBox("Enter the roof pitch in degrees:")
    pitch=pitch*conversion
    bldglen=InputBox("Enter the length of the building in feet:")
    wallht=Tan(pitch)*(bldglen/2)
    msgtext="The height of the building is: " & Format(wallht, "##.00")
    MsgBox msgtext
End Sub
```

See Also

Atn, Cos, Sin, Derived Trigonometric Functions

Text Statement

Places line(s) of text in a dialog box.

Syntax

Text *x, y, dx, dy, text\$* [, *.id*]

Where...	Is...
<i>x, y</i>	The upper left corner coordinates of the text area, relative to the upper left corner of the dialog box.
<i>dx, dy</i>	The width and height of the text area.
<i>text\$</i>	A string containing the text to appear in the text area defined by <i>x, y</i> .
<i>.id</i>	An optional identifier used by the dialog statements that act on this control.

Remarks

If the width of *text\$* 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 are 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.

Use the **Text** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example defines a dialog box with a combination list and text box and three buttons.

```
Sub main
  Dim ComboBox1() as String
  ReDim ComboBox1(0)
  ComboBox1(0)=Dir("C:\*.*")
  Begin Dialog UserDialog 166, 142, "BSL Dialog Box"
    Text 9, 3, 69, 13, "Filename:", .Text1
    DropComboBox 9, 14, 81, 119, ComboBox1(), .ComboBox1
    OKButton 101, 6, 54, 14
    CancelButton 101, 26, 54, 14
    PushButton 101, 52, 54, 14, "Help", .Push1
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog, Button, ButtonGroup, CancelButton, Caption, CheckBox, ComboBox, Dialog, DropComboBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, StaticComboBox, TextBox

TextBox Statement

Creates a text box in a dialog box.

Syntax

TextBox [**NoEcho**] *x*, *y*, *dx*, *dy*, *.field*

Where... Is...

<i>x</i> , <i>y</i>	The upper left corner coordinates of the text box, relative to the upper left corner of the dialog box.
<i>dx</i> , <i>dy</i>	The width and height of the text box area.
<i>.field</i>	The name of the dialog record field to hold the text string.

Remarks

A *dy* value of 12 will usually accommodate text in the system font. 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 *.field*. The **NoEcho** keyword is often used for passwords; it displays all characters entered as asterisks (*). Use the **TextBox** statement only between a **Begin Dialog** and an **End Dialog** statement.

Example

This example creates a dialog box with a group box, and two buttons.

```
Sub main
  Begin Dialog UserDialog 194, 76, "BSL Dialog Box"
    GroupBox 9, 8, 97, 57, "File Range"
    OptionGroup .OptionGroup2
      OptionButton 19, 16, 46, 12, "All pages", .OptionButton3
      OptionButton 19, 32, 67, 8, "Range of pages", .OptionButton4
    Text 25, 43, 20, 10, "From:", .Text6
    Text 63, 43, 14, 9, "To:", .Text7
    TextBox 79, 43, 13, 12, .TextBox4
    TextBox 47, 43, 12, 11, .TextBox5
    OKButton 135, 6, 54, 14
    CancelButton 135, 26, 54, 14
  End Dialog
  Dim mydialog as UserDialog
  On Error Resume Next
  Dialog mydialog
  If Err=102 then
    MsgBox "Dialog box canceled."
  End If
End Sub
```

See Also

Begin Dialog...End Dialog, **Button**, **ButtonGroup**, **CancelButton**, **Caption**, **CheckBox**, **ComboBox**, **Dialog**, **DropComboBox**, **GroupBox**, **ListBox**, **OKButton**, **OptionButton**, **OptionGroup**, **Picture**, **StaticComboBox**, **Text**

Time Function

Returns a string representing the current time.

Syntax

Time[\$]

Remarks

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.

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).

Example

This example writes data to a file if it has not been saved within the last 2 minutes.

```
Sub main
    Dim tempfile
    Dim filetime, curtime
    Dim msgtext
    Dim acctno(100) as Single
    Dim x, I
    tempfile="C:\TEMP001"
    Open tempfile For Output As #1
    filetime=FileDateTime(tempfile)
    x=1
    I=1
    acctno(x)=0
    Do
        curtime=Time
        acctno(x)=InputBox("Enter an account number (99 to end):")
        If acctno(x)=99 then
            For I=1 to x-1
                Write #1, acctno(I)
            Next I
            Exit Do
        ElseIf (Minute(filetime)+2)<=Minute(curtime) then
            For I=I to x
                Write #1, acctno(I)
            Next I
        End If
        x=x+1
    Loop
    Close #1
    x=1
    msgtext="Contents of C:\TEMP001 is:" & Chr(10)
    Open tempfile for Input as #1
```

```

Do While Eof(1)<>-1
    Input #1, acctno(x)
    msgtext=msgtext & Chr(10) & acctno(x)
    x=x+1
Loop
MsgBox msgtext
Close #1
Kill "C:\TEMP001"
End Sub

```

See Also

Date Function, Date Statement, Time Statement, Timer, TimeSerial, TimeValue

Time Statement

Sets the system time.

Syntax

Time[\$] = *expression*

Where...	Is...
<i>expression</i>	An expression that evaluates to a valid time.

Remarks

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 a 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 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 characters defined in the International section of the Windows Control Panel. **Time** (without the \$) accepts both 12 and 24 hour clocks.

Example

This example changes the time on the system clock.

```

Sub main
    Dim newtime as String
    Dim answer as String
    On Error Resume Next
i: newtime=InputBox("What time is it?")
    answer=InputBox("Is this AM or PM?")
    If answer="PM" or answer="pm" then
        newtime=newtime & "PM"
    End If
    Time=newtime
    If Err<>0 then
        MsgBox "Invalid time. Try again."
        Err=0
        Goto i
    End If
End Sub

```

See Also

Date Function, Date Statement, Time Function, TimeSerial, TimeValue

Timer Function

Returns the number of seconds that have elapsed since midnight.

Syntax

Timer

Remarks

The **Timer** function can be used in conjunction with the **Randomize** statement to seed the random number generator.

Example

This example uses **Timer** to find a Megabucks number.

```
Sub main
  Dim msgtext
  Dim value(9)
  Dim nextvalue
  Dim x
  Dim y
  msgtext="Your Megabucks numbers are: "
  For x=1 to 8
    Do
      value(x)=Timer
      value(x)=value(x)*100
      value(x)=Str(value(x))
      value(x)=Val(Right(value(x),2))
    Loop Until value(x)>1 and value(x)<36
    For y=1 to 1500
      Next y
    Next x
  For y=1 to 8
    For x= 1 to 8
      If y<>x then
        If value(y)=value(x) then
          value(x)=value(x)+1
        End If
      End If
    Next x
  Next y
  For x=1 to 8
    msgtext=msgtext & value(x) & " "
  Next x
  MsgBox msgtext
End Sub
```

See Also

Randomize

TimeSerial Function

Returns a time as a variant of type 7 (date/time) for a specific hour, minute, and second.

Syntax

TimeSerial(*hour%*, *minute%*, *second%*)

Where...	Is...
<i>hour%</i>	A numeric expression for an hour (0-23).
<i>minute%</i>	A numeric expression for a minute (0-59).
<i>second%</i>	A numeric expression for a second (0-59).

Remarks

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.

Example

This example displays the current time using Time Serial.

```
Sub main
  Dim y
  Dim msgtext
  Dim nowhr
  Dim nowmin
  Dim nowsec
  nowhr=Hour (Now)
  nowmin=Minute (Now)
  nowsec=Second (Now)
  y=TimeSerial (nowhr, nowmin, nowsec)
  msgtext="The time is: " & y
  MsgBox msgtext
End Sub
```

See Also

DateSerial, Date Value, Hour, Minute, Now, Second, TimeValue

TimeValue Function

Returns a time value for a specified string.

Syntax

TimeValue(*time\$*)

Where...	Is...
<i>time\$</i>	A string representing a valid date time value.

Remarks

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.

Example

This example writes a variable to a disk file based on a comparison of its last saved time and the current time. Note that all the variables used for the TimeValue function are dimensioned as Double, so that calculations based on their values will work properly.

```
Sub main
    Dim tempfile
    Dim ftime
    Dim filetime as Double
    Dim curtime as Double
    Dim minutes as Double
    Dim acctno(100) as Integer
    Dim x, I
    tempfile="C:\TEMP001"
    Open tempfile For Output As 1
    ftime=FileDateTime(tempfile)
    filetime=TimeValue(ftime)
    minutes= TimeValue("00:02:00")
    x=1
    I=1
    acctno(x)=0
    Do
        curtime= TimeValue(Time)
        acctno(x)=InputBox("Enter an account number (99 to end):")
        If acctno(x)=99 then
            For I=I to x-1
                Write #1, acctno(I)
            Next I
            Exit Do
        ElseIf filetime+minutes<=curtime then
            For I=I to x
                Write #1, acctno(I)
            Next I
        End If
        x=x+1
    Loop
    Close #1
    x=1
    msgtext="You entered:" & Chr(10)
```

```

Open tempfile for Input as #1
Do While Eof(1)<>-1
    Input #1, acctno(x)
    msgtext=msgtext & Chr(10) & acctno(x)
    x=x+1
Loop
MsgBox msgtext
Close #1
Kill "C:\TEMP001"
End Sub

```

See Also

DateSerial, Date Value, Hour, Minute, Now, Second, TimeSerial

Trim Function

Returns a copy of a string after removing all leading and trailing spaces.

Syntax

Trim[\$](*string*)

Where...

string

Is...

An expression containing the string to trim.

Remarks

Trim accepts expressions of type String. **Trim** accepts any type of *string* including numeric values and will convert the input value to a string.

The dollar sign, "\$", in the function name is optional. If specified, the return type is String. If omitted, the function typically returns a variant of vartype 8 (String). If the value of *string* is NULL, a variant of vartype 1 (Null) is returned.

Example

This example removes leading and trailing spaces from a string entered by the user.

```

Sub main
    Dim userstr as String
    userstr=InputBox("Enter a string with leading/trailing spaces")
    MsgBox "The string is: " & Trim(userstr) & " with nothing after it."
End Sub

```

See Also

GetField, Left, Len, LTrim, Mid Function, Mid Statement, Right, RTrim

Type Statement

Declares a user-defined type.

Syntax

```
Type userType
    field1 As type1
    field2 As type2
    ...
End Type
```

Where...	Is...
<i>userType</i>	A user-defined type.
<i>field1</i> , <i>field2</i>	The name of a field in the user-defined type.
<i>type1</i> , <i>type2</i>	A data type: Integer, Long, Single, Double, Currency, String, String* <i>length</i> , variant, or another user-defined type.

Remarks

The user-defined type declared by **Type** 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*.

field cannot 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
```

Example

This example shows a **Type** and **Dim** statement for a record. You must define a record type before you can declare a record variable. The subroutine then references a field within the record.

```
Type Testrecord
    Custno As Integer
    Custname As String
End Type
Sub main
    Dim myrecord As Testrecord
    i: myrecord.custname=InputBox("Enter a customer name:")
    If myrecord.custname="" then
        Exit Sub
    End If
    answer=InputBox("Is the name: " & myrecord.custname & " correct? (Y/N)")
    If answer="Y" or answer="y" then
        MsgBox "Thank you."
    Else
        MsgBox "Try again."
        Goto i
    End If
End Sub
```

See Also

Deftype, **Dim**

Typeof Function

Returns a value indicating whether an object is of a given class (-1=TRUE, 0=FALSE).

Syntax

If Typeof *objectVariable* **Is** *className* **then**. . .

Where...	Is...
<i>objectVariable</i>	The object to test.
<i>className</i>	The class to compare the object to.

Remarks

Typeof can only be used in an **If** statement and cannot be combined with other boolean operators. That is, **Typeof** can 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
```

See Also

CreateObject, GetObject, Is, Me, New, Nothing, Object Class

UBound Function

Returns the upper bound of the subscript range for the specified array.

Syntax

UBound(*arrayname* [, *dimension*])

Where...	Is...
<i>arrayname</i>	The name of the array to use.
<i>dimension</i>	The dimension to use.

Remarks

The dimensions of an array are numbered starting with 1. If the *dimension* is not specified, 1 is used as a default.

LBound can be used with **UBound** to determine the length of an array.

Example

This example resizes an array if the user enters more data than can fit in the array. It uses **LBound** and **UBound** to determine the existing size of the array and **ReDim** to resize it. Option Base sets the default lower bound of the array to 1.

```
Option Base 1
Sub main
    Dim arrayvar() as Integer
    Dim count as Integer
    Dim answer as String
    Dim x, y as Integer
    Dim total
    total=0
    x=1
    count=InputBox("How many test scores do you have?")
    ReDim arrayvar(count)
start:
    Do until x=count+1
        arrayvar(x)=InputBox("Enter test score #" &x & ":")
        x=x+1
    Loop
    answer=InputBox$("Do you have more scores? (Y/N)")
    If answer="Y" or answer="y" then
        count=InputBox("How many more do you have?")
        If count<>0 then
            count=count+(x-1)
            ReDim Preserve arrayvar(count)
            Goto start
        End If
    End If
    x=LBound(arrayvar,1)
    count=UBound(arrayvar,1)
    For y=x to count
        total=total+arrayvar(y)
```

```

Next y
MsgBox "The average of the " & count & " scores is: " & Int(total/count)
End Sub

```

See Also

Dim, Global, LBound, Option Base, ReDim, Static

UCase Function

Returns a copy of a string after converting all lower case letters to upper case.

Syntax

UCase[\$](*string*)

Where...	Is...
<i>string</i>	An expression that evaluates to a string.

Remarks

The translation is based on the settings specified in the Regional and Language Options of your Windows Control Panel.

UCase accepts expressions of type string. **UCase** accepts any type of argument and will convert the input value to a string.

The dollar sign, "\$", in the function name is optional. If specified, the return type is string. If omitted, the function typically returns a variant of vartype 8 (String). If the value of *string* is Null, a variant of vartype 1 (Null) is returned.

Example

This example converts a filename entered by a user to all uppercase letters.

```

Option Base 1
Sub main
    Dim filename as String
    filename=InputBox("Enter a filename: ")
    filename=UCase(filename)
    MsgBox "The filename in uppercase is: " & filename
End Sub

```

See Also

Asc, LCase

Unlock Statement

Controls access to an open file.

Syntax

Unlock [#]filename% [, { record& | [start&] **To** end& }]

Where...	Is...
filename%	An integer expression identifying the open file.
record&	Number of the starting record to unlock.
start&	Number of the first record or byte offset to lock/unlock.
end&	Number of the last record or byte offset to lock/unlock.

Remarks

The *filename%* is the number used in the **Open** statement of the file.

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.

Example

Locks a file that is shared by others on a network, if the file is already in use. The second subprogram, CREATEFILE, creates the file used by the main subprogram.

```
Declare Sub createfile
Sub main
  Dim btgrp, icongrp
  Dim defgrp
  Dim answer
  Dim noaccess as Integer
  Dim msgabort
  Dim msgstop as Integer
  Dim acctname as String
  noaccess=70
  msgstop=16
  Call createfile
  On Error Resume Next
  btgrp=1
  icongrp=64
  defgrp=0
  answer=MsgBox("Open the account file?" & Chr(10), btgrp+icongrp+defgrp)
  If answer=1 then
    Open "C:\TEMP001" for Input as #1
    If Err=noaccess then
      msgabort=MsgBox("File Locked",msgstop,"Aborted")
    Else
      Lock #1
      Line Input #1, acctname
      MsgBox "The first account name is: " & acctname
      Unlock #1
    End If
    Close #1
  End If
  Kill "C:\TEMP001"
End Sub

Sub createfile()
```

```

Rem Put the letters A-J into the file
Dim x as Integer
Open "C:\TEMP001" for Output as #1
For x=1 to 10
    Write #1, Chr(x+64)
Next x
Close #1
End Sub

```

See Also**Lock, Open**

Val Function

Returns the numeric value of the first number found in the specified string.

Syntax

Val(*string\$*)

Where... Is...

Where...	Is...
<i>string\$</i>	A string expression containing a number.

Remarks

Spaces in the source string are ignored. If no number is found, **Val** returns 0.

Example

This example tests the value of the variable profit and displays 0 for profit if it is a negative number. The subroutine uses **Sgn** to determine whether profit is positive, negative or zero.

```

Sub main
    Dim profit as Single
    Dim expenses
    Dim sales
    expenses=InputBox("Enter total expenses: ")
    sales=InputBox("Enter total sales: ")
    profit=Val(sales)-Val(expenses)
    If Sgn(profit)=1 then
        MsgBox "Yeah! We turned a profit!"
    ElseIf Sgn(profit)=0 then
        MsgBox "Okay. We broke even."
    Else
        MsgBox "Uh, oh. We lost money."
    End If
End Sub

```

See Also

CCur, CDbl, CInt, CLng, CSng, CStr, CVar, CVDate, Format, Str

VarType Function

Returns the variant type of the specified variant variable (0-9).

Syntax

VarType(*varname*)

Where...

Is...

varname The variant variable to use.

Remarks

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
9	Object

Example

This example returns the type of a variant.

```
Sub main
  Dim x
  Dim myarray(8)
  Dim retval
  Dim retstr
  myarray(1)=Null
  myarray(2)=0
  myarray(3)=39000
  myarray(4)=CSng(10^20)
  myarray(5)=10^300
  myarray(6)=CCur(10.25)
  myarray(7)=Now
  myarray(8)="Five"
  For x=0 to 8
    retval=VarType(myarray(x))
    Select Case retval
      Case 0
        retstr=" (Empty) "
      Case 1
        retstr=" (Null) "
      Case 2
        retstr=" (Integer) "
      Case 3
        retstr=" (Long) "
      Case 4
        retstr=" (Single) "
```

```
Case 5
    retstr=" (Double) "
Case 6
    retstr=" (Currency) "
Case 7
    retstr=" (Date) "
Case 8
    retstr=" (String) "
End Select
If retval=1 then
    myarray(x)="[null]"
ElseIf retval=0 then
    myarray(x)="[empty]"
End If
MsgBox "The variant type for " &myarray(x) & " is: " &retval &retstr
Next x
End Sub
```

See Also**IsDate, IsEmpty, IsNull, IsNumeric**

Weekday Function

Returns the day of the week for the specified date-time value.

Syntax

Weekday(*date*)

Where...	Is...
<i>date</i>	An expression containing a date time value.

Remarks

The **Weekday** function returns an integer between 1 and 7, inclusive (1=Sunday, 7=Saturday).

Weekday accepts any expression, including strings, and attempts to convert the input value to a date value.

The return value is a variant of vartype 2 (Integer). If the value of *date* is NULL, a variant of vartype 1 (Null) is returned.

Example

This example finds the day of the week on which New Year's Day will fall in the year 2000.

```
Sub main
  Dim newyearsday
  Dim daynumber
  Dim msgtext
  Dim newday as Variant
  Const newyear=2000
  Const newmonth=1
  Let newday=1
  newyearsday=DateSerial(newyear,newmonth,newday)
  daynumber=Weekday(newyearsday)
  msgtext="New Year's day 2000 falls on a " & Format(daynumber, "dddd")
  MsgBox msgtext
End Sub
```

See Also

Date Function, Date Statement, Day, Hour, Minute, Month, Now, Second, Year

While...Wend

Controls a repetitive action.

Syntax

While *condition*

statementblock

Wend

Where...	Is...
<i>condition</i>	An expression that evaluates to TRUE (non-zero) or FALSE (zero).
<i>statementblock</i>	A series of statements to execute if condition is TRUE.

Remarks

The *statementblock* statements are until *condition* becomes 0 (FALSE).

The **While** statement is included in BSL for compatibility with older versions of Basic. The **Do** statement is a more general and powerful flow control statement.

Example

This example opens a series of customer files and checks for the string “*Overdue*” in each file. It uses While...Wend to loop through the C:\TEMP00? files. These files are created by the subroutine CREATEFILES.

```

Declare Sub createfiles
Sub main
  Dim custfile as String
  Dim aline as String
  Dim pattern as String
  Dim count as Integer
  Call createfiles
  Chdir "C:\"
  custfile=Dir$("TEMP00?")
  pattern="*" + "Overdue" + "*"
  While custfile <> ""
    Open custfile for input as #1
    On Error goto atEOF
    Do
      Line Input #1, aline
      If aline Like pattern Then
        count=count+1
      End If
    Loop
  nextfile:
  On Error GoTo 0
  Close #1
  custfile = Dir$
  Wend
  If count<>0 then
    MsgBox "Number of overdue accounts: " & count
  Else
    MsgBox "No accounts overdue"
  End If
  Kill "C:\TEMP001"

```

```
Kill "C:\TEMP002"  
Exit Sub  
atEOF:  
    Resume nxtfile  
End Sub  
  
Sub createfiles()  
    Dim odue as String  
    Dim ontime as String  
    Dim x  
    Open "C:\TEMP001" for OUTPUT as #1  
    odue="*" + "Overdue" + "*"  
    ontime="*" + "On-Time" + "*"  
    For x=1 to 3  
        Write #1, odue  
    Next x  
    For x=4 to 6  
        Write #1, ontime  
    Next x  
    Close #1  
    Open "C:\TEMP002" for Output as #1  
    Write #1, odue  
    Close #1  
End Sub
```

See Also**Do...Loop**

Width Statement

Sets the output line width for an open file.

Syntax

Width [#]*filename%*, *width%*

Where...	Is...
<i>filename%</i>	An integer expression for the open file to use.
<i>width%</i>	An integer expression for the width of the line (0 to 255).

Remarks

Filename% is the number assigned to the file when it is opened. See the **Open** statement for more information.

A value of zero (0) for *width%* indicates there is no line length limit. The default *width%* for a file is zero (0).

Example

This example puts five spaces and the string "ABCD" to a file. The five spaces are derived by taking 15 MOD 10, or the remainder of dividing 15 by 10.

```
Sub main
  Dim str1 as String
  Dim x as String*10
  str1="ABCD"
  Open "C:\TEMP001" For Output As #1
  Width #1, 10
  Print #1, Spc(15); str1
  Close #1
  Open "C:\TEMP001" as #1 Len=12
  Get #1, 1,x
  MsgBox "The contents of the file is: " & x
  Close #1
  Kill "C:\TEMP001"
End Sub
```

See Also

Open, Print

With Statement

Executes a series of statements on a specified variable.

Syntax

With *variable*
 statement_block

End With

Where...

Is...

<i>variable</i>	The variable to be changed by the statements in <i>statement_block</i> .
<i>statement_block</i>	The statements to execute.

Remarks

Variable can be an object or a user-defined type. The **With** statements can be nested.

Example

This example creates a user-defined record type, *custrecord* and uses the **With** statement to fill in values for the record fields, for the record called "John".

```
Type custrecord
  name as String
  ss as String
  salary as Single
  dob as Variant
  street as String
  apt as Variant
  city as String
  state as String
End Type
Sub main
  Dim John as custrecord
  Dim msgtext
  John.name="John"
  With John
    .ss="037-67-2947"
    .salary=60000
    .dob=#10-09-65#
    .street="15 Chester St."
    .apt=28
    .city="Cambridge"
    .state="MA"
  End With
  msgtext=Chr(10) & "Name:" & Space(5) & John.name & Chr(10)
  msgtext=msgtext & "SS#: " & Space(6) & john.ss & chr(10)
  msgtext=msgtext & "D.O.B:" & Space(4) & john.dob
  MsgBox "Done with: " & Chr(10) & msgtext
End Sub
```

See Also

Type

Write Statement

Writes data to an open sequential file.

Syntax

Write #*filename%*, [*expressionlist*]

Where...	Is...
<i>filename%</i>	An integer expression for the open file to use.
<i>expressionlist</i>	One or more values to write to the file.

Remarks

The file must be opened in **Output** or **Append** mode. *Filename%* is the number assigned to the file when it is opened. See the **Open** statement for more information.

If *expressionlist* is omitted, the **Write** statement writes a blank line to the file. (See the "Input Statement" on page 170 for more information.)

Example

This example writes a variable to a disk file based on a comparison of its last saved time and the current time.

```

Sub main
    Dim tempfile
    Dim filetime, curtime
    Dim msgtext
    Dim acctno(100) as Single
    Dim x, I
    tempfile="C:\TEMP001"
    Open tempfile For Output As #1
    filetime=FileDateTime(tempfile)
    x=1
    I=1
    acctno(x)=0
    Do
        curtime=Time
        acctno(x)=InputBox("Enter an account number (99 to end):")
        If acctno(x)=99 then
            If x=1 then Exit Sub
            For I=1 to x-1
                Write #1, acctno(I)
            Next I
            Exit Do
        ElseIf (Minute(filetime)+2)<=Minute(curtime) then
            For I=I to x-1
                Write #1, acctno(I)
            Next I
        End If
        x=x+1
    Loop
    Close #1
    x=1
    msgtext="Contents of C:\TEMP001 is:" & Chr(10)
    Open tempfile for Input as #1

```

```
Do While Eof(1)<>-1
  Input #1, acctno(x)
  msgtext=msgtext & Chr(10) & acctno(x)
  x=x+1
Loop
MsgBox msgtext
Close #1
Kill "C:\TEMP001"
End Sub
```

See Also

Close, Open, Print, Put

Year Function

Returns the year component of a date-time value.

Syntax

Year(*date*)

Where...

Is...

date An expression that can evaluate to a date time value.

Remarks

The **Year** function returns an integer between 100 and 9999, inclusive.

Year accepts any type of *date*, including strings, and will attempt to convert the input value to a date value.

The return value is a variant of vartype 2 (Integer). If the value of *date* is NULL, a variant of vartype 1 (Null) is returned.

Example

This example returns the year for today.

```
Sub main
  Dim nowyear
  nowyear=Year(Now)
  MsgBox "The current year is: " &nowyear
End Sub
```

See Also

Date Function, Date Statement, Day, Hour, Minute, Month, Now, Time Function, Second, Weekday

API Function Calls

Microsoft Dynamics SL Kernel Functions Summary

The following is a list of the Microsoft Dynamics SL kernel functions available in the Basic Script Language and a brief summary of their purpose.

Function/statement	Action
AliasConstantConstant, Alias	Aliases certain constants used specifically by Transaction Import.
AppGetParms Function	Retrieves a command line parameter passed by another Microsoft Dynamics SL application.
AppGetParmValue Function	Retrieves a parameter passed by another application.
AppSetFocus Statement	Sets focus to a designated object.
AppSetParmValue Statement	Adds a parameter to the list of all parameters which will be sent to a calling application.
CallChks Function	Executes Chk event of the specified object.
DateCheck Function	Validates a date.
DateCmp Function	Compares two dates.
DateMinusDate Function	Returns the number of days between two dates.
DatePlusDays Statement	Adds a specified number of days to a date.
DatePlusMonthSetDay Statement	Adds a specified number of months to a date and sets to a valid day.
DateToIntlStr Function	Converts a date into the Windows short date style.
DateToStr Function	Converts a date to a string.
DateToStrSep Function	Converts a date to a string and includes separators.
DBNavFetch Functions	Retrieves a composite record from the database using an SQL statement.
DispFields Statement	Displays the contents of a field structure.
DispForm Statement	Displays a specified form object.
Dparm Function	Convert a date into an SQL parameter string.
Edit_Cancel Statement	Executes the Cancel toolbar button.
Edit_Close Statement	Executes the Close toolbar button.
Edit_Delete Function	Executes the Delete toolbar button.
Edit_Finish Function	Executes the Finish toolbar button.
Edit_First Function	Executes the First toolbar button.
Edit_Last Function	Executes the Last toolbar button.
Edit_New Function	Executes the New toolbar button.
Edit_Next Function	Executes the Next toolbar button.
Edit_Prev Function	Executes the Previous toolbar button.
Edit_Save Statement	Executes the Save toolbar button.
FPAdd Function	Floating point Add function.
FParam Function	Formatting function for a float field passed to an SQL function.
FPDiv Function	Floating point Divide function.
FPMult Function	Floating point Multiply function.
FPRnd Function	Floating point Rounding function.
FPSub Function	Floating point Subtraction function.
GetBufferValue Statement	Obtains buffer value for a specified field.
GetDelGridHandle Function	Returns the resource handle of the memory array used to temporarily hold

Function/statement	Action
	detail lines deleted from the designated SAFGrid control.
GetGridHandle Function	Obtains a grid handle for a spreadsheet object.
GetObjectValue Function	Obtains the field value of a specified object.
GetProp Function	Obtains a property for the specified object.
GetSqlType Function	Determine which type of database server is being used.
GetSysDate Statement	Obtains the current system date.
GetSysTime Statement	Obtains the current system time.
HideForm Statement	Hides specified form object.
IncrStrg Statement	Increments a string value.
IParm Function	Formatting function for an integer field passed to an SQL function.
Launch Function	Launches another executable program.
MClear Statement	Delete all records from the designated memory array.
MClose Statement	Close an existing memory array.
MDelete Function	Delete the current record from the designated memory array.
MDisplay Statement	Display the current contents of the designated memory array in its corresponding spreadsheet control.
Mess Statement	Displays the specified message number.
Messbox Statement	Displays a message box with parameters provided.
Messf Statement	Displays the specified message number with substitution variables.
MessResponse Function	Obtains user response from a message box.
Mextend Function	Extend the grid of an application so that another table's structure can be added to the grid.
mFindControlName	Returns the first and subsequent control names in tab index order.
MFirst Function	Move to the first record in a designated memory array.
MGetLineStatus Function	Returns the line status of the current record in the designated memory array.
MGetRowNum Function	Returns the row/record number of the current record in the designated memory array.
MInsert Statement	Inserts a new record into a designated memory array.
MKey Statement	Defines a key field for a previously opened memory array.
MKeyFind Function	Finds a specific record within a sorted memory array based on designated key field values.
MKeyFld Statement	Defines a key field for a previously opened memory array.
MKeyHctl Statement	Defines a key field for a previously opened memory array.
MKeyOffset Statement	Defines a key field for a previously opened memory array.
MLast Function	Moves to the last record in a designated memory array.
MLoad Statement	Loads a memory array with all records returned from the database by an SQL statement.
MNext Function	Moves to the next record in a designated memory array.
MOpen Functions	Open a new memory array and return a corresponding unique memory array number.
MPrev Function	Moves to the previous record in a designated memory array.
MRowCnt Function	Returns the number of records in a designated memory array.
MSet Statement	Sets a grid column to specified value.
MSetLineStatus Function	Sets the line status of the current record in the designated memory array.
MSetProp Statement	Sets the properties of a grid column at runtime.
MSetRow Statement	Sets the current row / record number of a designated memory array.

Function/statement	Action
MUpdate Statement	Updates the current memory array record of a designated memory array with new data values.
NameAltDisplay Function	Displays the name field with swap character suppressed.
PasteTemplate Function	Pastes information from the designated template into the current application.
PeriodCheck Function	Performs period number validation on current field.
PeriodMinusPeriod Function	Determines the difference between two period numbers.
PeriodPlusPerNum Function	Adds a number of periods to a period.
PVChkFetch Functions	Retrieves a composite record from the database using an SQL statement from the PV property of an SAFMASKEDTEXT control.
SaveTemplate Statement	Saves information from the current application to a designated template.
Sdelete Function	Deletes the current record in view.
SdeleteAll Function	Deletes all records from a table in the view.
SetAddr Statement	Allocates a structure for a specified database table.
SetBufferValue Statement	Sets an underlying Microsoft Dynamics SL application's data buffer field to a specified value.
SetDefaults Function	Displays the default value for the specified object.
SetLevelChg Statement	Sets a certain level number to a different status.
SetObjectValue Function	Sets a specified object field's value.
SetProp Statement	Sets properties of objects at runtime.
SFetch Functions	Fetches the next record into view.
SGroupFetch Functions	Group fetches the next aggregate value into view.
SInsert Statements	Inserts the structure from the current view into a table.
Sparm Function	Formatting function for a string field passed to an SQL function.
Sql Statement	Executes the specified SQL statement.
SqlCursor Statement	Allocates an SQL cursor for a view of a table.
SqlCursorEx	Allocate a new database cursor.
SqlErr Function	Obtains the return value of the specified SQL function.
SqlErrException Statement	Allows an application to trap certain SQL errors.
SqlExec Statement	Executes the specified SQL statement after passing variables.
SqlFetch Functions	Executes the SQL statement and fetches the first record into view.
SqlFree Statement	Frees a cursor.
SqlSubst Statement	Substitutes variables into an SQL statement.
StrToDate Statement	Converts a string to a date field type.
StrToTime Statement	Converts a string to a time field type.
SUpdate Statements	Updates the current view.
TestLevelChg Function	Determines whether or not a specified level has changed.
TimeToStr Function	Converts time to a string.
TranAbort Statement	Aborts the current transaction.
TranBeg Statement	Begins a database transaction.
TranEnd Statement	Ends a database transaction.
TranStatus Function	Returns the current SQL database transaction status.

Note: In the following sections, some example code may exceed one line. In this case, the symbol ↵ indicates a line continuation character. The entire line must be on a single line in the event code window.

AliasConstant Statement

Use this function to alias certain constants used specifically by Transaction Import. This can be helpful if you are using a different language (the English words like *Comment*, *Insert*, and *Change* could be replaced by the other language's alternative words). The possible values are listed below.

Syntax

Call **AliasConstant**(*Constant*, *Alias*)

Remarks

Constants that can be redefined using the AliasConstant function include:

Comment

Insert

Delete

Change

Processed

Separator

Delimiter

LevelN (where *N* is 0 to 9)

Checked

Unchecked

Press

Example

```
Call AliasConstant("Change", "Update")
Call AliasConstant("Delimiter", ";")
Call AliasConstant("Seperator", "!")
Call AliasConstant("Level0", "Batch")
Call AliasConstant("Level1", "Detail")
```

ApplGetParms Function

Retrieve a parameter passed by another Microsoft Dynamics™ SL Software Development Kit (Microsoft SL SDK) application.

Syntax

ParmValue = **ApplGetParms**()

Remarks

The **ApplGetParms** statement can be used to retrieve parameters which were originally passed by another Microsoft SL SDK application using either the **CallApplic** or **CallApplicWait** statement. Multiple parameters can be retrieved by making successive calls to **ApplGetParms**.

If the calling application passed parameters via named parameter sections, using the **ApplSetParmValue** statement in conjunction with either the **CallApplic** or **CallApplicWait** statement, then **ApplGetParms** will only be able to retrieve parameters from the default Microsoft Dynamics SL section. The **ApplGetParmValue** function is the only means by which the called application can retrieve parameters from any named parameter section other than the default section.

The **ApplGetParms** function uses the following argument:

Argument	Type	Description
<i>ParmValue</i>	String	The actual value of the next parameter to be retrieved.

Example

```

Following code for 03.270.00 (the launched application):
Sub Form1_Load
    ' Variable to store the passed Parameter to this screen
    ' Under RDT scenario, this code would be in Form_Load.
    Dim VendorParm$
    VendorParm = ApplGetParms()
    If Trim$(VendorParm) <> "" Then
        ' Screen was called from another application.
        ' Set the value of the ID field to what was
        ' passed in by Launch() function
        serr1 = SetObjectValue("cvendid", VendorParm)
    End If
End Sub

```

See Also

ApplGetParmValue Function, **ApplSetParmValue** Statement

AppGetParmValue Function

Retrieve a parameter passed by another Microsoft SL SDK application.

Syntax

ParmValue = **AppGetParmValue**(*ParmSection*, *ParmName*)

Remarks

Parameters passed to a Microsoft SL SDK application can be retrieved via one of two different methods: **AppGetParms** and **AppGetParmValue**. These functions differ in that **AppGetParms** does not support multiple parameter sections whereas **AppGetParmValue** does provide this more sophisticated functionality. Consequently, **AppGetParmValue** is the only means by which the called application can retrieve parameters from any named parameter section other than the default Microsoft Dynamics SL section. For example, if the calling application sends a parameter specifically designated as a BSL parameter, only the **AppGetParmValue** function can be used to retrieve that particular parameter since the BSL section name can be explicitly queried via the *ParmSection* argument.

Named parameter sections facilitate the elimination of conflicts which can occur in the destination program when the application itself as well as custom BSL code added via the Customization Manager are both attempting to receive different parameters. For example, the Accounts Payable *Document Maintenance* (03.250.00) screen can optionally receive two parameters that facilitate drill-down functionality: Reference Number and Vendor ID. The Form_Load event always calls **AppGetParms** once to determine if any parameters have been passed to the application. If even one parameter exists it is assumed that it is the Reference Number and therefore the application calls **AppGetParms** again expecting the next parameter to be the Vendor ID. If this screen is subsequently customized by adding calls to **AppGetParms** using BSL code, an operational conflict will occur. If such an application were to be called with only one parameter, designed to be received by the custom BSL code, it would instead be received by the call to **AppGetParms** performed by the underlying application. Consequently the call to **AppGetParms** in BSL code would not return any parameter value at all.

AppSetParmValue and **AppGetParmValue** overcome this operational conflict by facilitating the usage of named parameter sections. Using this more sophisticated method, parameters can be passed directly to the application itself and to custom BSL code using the two standard section names declared in *applic.dh* (for example, *PRMSECTION_VBRDT* and *PRMSECTION_BSL*). Parameter sections are not, however, limited to these two standard section names. Thus for example “[XYZ Section]” is a valid section name. The brackets are required since parameter sections themselves are handled similar to section names within an .INI file. In the previously mentioned example, a custom parameter could be sent to the Accounts Payable Document Maintenance screen in the BSL parameter section such that only calls to **AppGetParmValue** specifically requesting BSL parameters would retrieve the parameter.

The **AppGetParmValue** statement uses the following arguments:

Argument	Type	Description
<i>ParmValue</i>	String	The actual value of the parameter being retrieved.
<i>ParmSection</i>	String	Name of the section within the temporary parameter file from which the parameter should be retrieved. Any section name can be used such as “XYZ Section” - provided the calling application utilized a so named parameter section. <i>Applic.DH</i> contains three symbolic constants defining standard section names: <i>PRMSECTION_VBRDT</i> , <i>PRMSECTION_BSL</i> and <i>PRMSECTION_TI</i> . <i>PRMSECTION_TI</i> is reserved for usage in conjunction with Transaction Import. By default, the parameter will be retrieved from the section represented by <i>PRMSECTION_VBRDT</i> if this argument is left blank.
<i>ParmName</i>	String	Logical name of the parameter being retrieved. By default, parameter names are sequentially numbered (for example, <i>PRM01</i> , <i>PRM02</i> <i>PRM99</i>) if they were not explicitly named by the call to AppSetParmValue in the calling application.

Example

The following example illustrates how to pass parameters to a Microsoft SL SDK application and custom BSL code at the same time and avoid conflicts between the two.

Code in the calling application:

```
Call ApplSetParmValue(PRMSECTION_VBRDT, "Batch Nbr", "000001")
Call ApplSetParmValue(PRMSECTION_VBRDT, "Document Nbr", "123456")

Call ApplSetParmValue(PRMSECTION_BSL, "Example Parm", "Example ↵
Parameter To BSL Code")

'Call another Microsoft Dynamics SL application
CallApplicWait( "Microsoft Dynamics SL APP", "")
```

Code in the standard Microsoft SL SDK application (that is, non-customization code) receiving the standard parameters.

```
Dim Parm_BatchNbr           As String
Dim Parm_DocumentNbr      As String

Parm_BatchNbr = ApplGetParmValue(PRMSECTION_VBRDT, "Batch Nbr")
Parm_DocumentNbr = ApplGetParmValue(PRMSECTION_VBRDT, "Document Nbr")
```

Basic Script code, overlaying the standard Microsoft SL SDK application, designed to retrieve custom parameters:

```
Dim Parm_CustomParm        As String

Parm_CustomParm = ApplGetParmValue(PRMSECTION_BSL, "Example Parm")
```

See Also

ApplGetParms Function, **ApplSetParmValue** Statement

ApplGetReturnParms Function

Retrieve a parameter returned from a now terminated secondary application.

Syntax

```
ParmValue = ApplGetReturnParms()
```

Remarks

If a Microsoft SL SDK application needs to pass parameters back to the program from which it was originally called it can do so using one of the parameters to **ScreenExit**. When control subsequently returns to the calling application, it can issue one or more calls to **ApplGetReturnParms** to successively retrieve each individual parameter.

The **ApplGetReturnParms** function has the following arguments:

Argument	Type	Description
<i>ParmValue</i>	String	The actual value of the parameter being retrieved from a now terminated secondary program.

Example

Following is an example of **ApplGetReturnParms** using two projects, MyApp1 and MyApp2:

The following code is in a button Click event in MyApp1:

```
Dim RetParmValue1 As String
Dim RetParmValue2 As String
Dim RetParmValue3 As String

Call ApplSetParmValue("[My Section]", "My Value", "999")
Call CallApplicWait("MyApp2", "")

` Note that the remaining code does not resume execution until MyApp2 has been closed:
RetParmValue1 = ApplGetReturnParms()
RetParmValue2 = ApplGetReturnParms()
RetParmValue3 = ApplGetReturnParms()

If Trim$(RetParmValue1) <> "" Then
Call MsgBox(RetParmValue1, MB_OK, "System Message")
Call MsgBox(RetParmValue2, MB_OK, "System Message")
Call MsgBox(RetParmValue3, MB_OK, "System Message")
End If
```

MyApp2 contains the following code:

```
In Form1_Load:
Dim PassedParm As String
` Retrieve parameter 999 passed by MyApp1 and display it in a message box.
PassedParm = ApplGetParmValue("[My Section]", "My Value")

If Trim$(PassedParm) <> "" Then
Call messagebox("Parameter passed is: " & Str$(PassedParm), MB_OK, "System Message")
End If

In the Form1_Unload:
Call ScreenExit(APPLICRETURNPARMS, "Parm1" + PRMSEP + "Parm2" + PRMSEP + "Parm3")
```

This code passes the three parameters back to MyApp1. The code in MyApp1 above will resume execution and the **MsgBox** functions will then display the strings.

Note: ScreenExit is always required in the Form1_Unload event (whether you are returning parameters or not). APPLICRETURNPARMS is a constant defined in APPLIC.DH that must always be used any time you are passing parameters back to a calling application.

See Also

ScreenExit Statement

App1SetFocus Statement

Set focus to a designated control.

Syntax

Call **App1SetFocus**(*TargetCtrl*)

Remarks

App1SetFocus is the preferred method to explicitly set focus to a target control. Usage of the Visual Basic .SetFocus method will cause a fatal Visual Basic error if the target control is disabled or invisible. Developers must always remember that the design time property setting of the target control cannot be guaranteed to always retain the same value at runtime. For example, the target control may be both visible and enabled in the standard application and therefore the **.SetFocus** will appear to work properly during testing. However, the end user may subsequently apply a customization which, among other things, disables the target control, thereby uncovering a subtle flaw in the underlying application with regards to its usage of the **.SetFocus** method.

The **App1SetFocus** statement uses the following arguments:

Argument	Type	Description
<i>TargetCtrl</i>	Control	Control to which focus should be moved.

Example

```
Call App1SetFocus ("cDiscBal")
```

AppISetParmValue Statement

Add one additional parameter to the list of all parameters which will be sent to the next application called by either the **CallApplic** or **CallApplicWait** statement.

Syntax

Call **AppISetParmValue**(*ParmSection*, *ParmName*, *ParmValue*)

Remarks

There are two different methods for one Microsoft SL SDK application to start another application: **CallApplic** and **CallApplicWait**. Regardless of which statement is used there are also two different methods for the calling program to pass parameters to the called program.

The first method is to pass the parameters to the called program using the argument to **CallApplic** and **CallApplicWait** specifically designed for this purpose. Parameters passed via this method are all grouped together and passed directly to the called application via the physical command line itself. Consequently, under this method the size and/or number of parameters is limited to the maximum command line length less the number of bytes used for internal requirements, which can vary based on the situation.

A more robust method of passing parameters is to use the **AppISetParmValue** statement in conjunction with either the **CallApplic** or **CallApplicWait** statement. The principle advantage of using this method is that it allows the calling application to group parameters into named sections and explicitly label individual parameters using unique parameter names. Grouping parameters into named sections eliminates conflicts that will occur in the called program when the application itself as well as custom BSL code added via the Customization Manager are both attempting to receive different parameters. See the **AppIGetParmValue** function for a more detailed explanation of these potential conflicts.

The first call to **AppISetParmValue** will create a temporary destination parameter file and place the first parameter in that file. By default, this temporary file will be created in the WINDOWS directory. This default can be overridden using the TempDirectory entry in the [Miscellaneous] section of the Solomon.ini file. The following is an example of the C:\Temp directory specified as the parameter file directory in the Solomon.ini file:

```
[Miscellaneous]
TempDirectory=C:\TEMP
```

Subsequent calls to **AppISetParmValue** will write additional parameters to the same temporary destination parameter file. The fully qualified filename of the completed parameter file will then be passed to the called program by the **CallApplic** and **CallApplicWait** statements. Once the called program has successfully loaded, it can call either **AppIGetParms** or **AppIGetParmValue** to retrieve the various parameters passed from the calling program. When the called program terminates execution, the temporary destination parameter file will automatically be deleted. **AppISetParmValue** is only designed to facilitate parameter passing to other applications developed with the Microsoft SL SDK.

The **AppISetParmValue** statement uses the following arguments:

Argument	Type	Description
<i>ParmSection</i>	String	Name of the section within the temporary parameter file to which the new parameter should be added. Any section name can be used such as "XYZ Section". Applic.DH contains three symbolic constants defining standard section names: PRMSECTION_VBRDT , PRMSECTION_BSL and PRMSECTION_TI . PRMSECTION_TI is reserved for usage in conjunction with Transaction Import. By default, the new parameter will be added to the section represented by PRMSECTION_VBRDT if this argument is left blank.
<i>ParmName</i>	String	Name assigned to the new parameter. Any name can be assigned to a parameter such as "Batch Number". By default, the new parameter will be assigned the next sequentially numbered parameter name (for example, PRM01 , PRM02 PRM99) if this argument is left blank.
<i>ParmValue</i>	String	The actual value of the new parameter.

Example

The following example will illustrate two different methods of calling ROI to display the Vendor List report on the screen for all Vendors having a balance greater than zero.

Pass parameters to ROI via one large parameter argument to **CallApplicWait**. This method will not work with a large WHERE clause since the entire contents of ParmStr must fit on the physical command line.

```
Dim ParmStrAs String

ParmStr = "03670/RUN" + PRMSEP
ParmStr = ParmStr + "03670S/FORMAT" + PRMSEP
ParmStr = ParmStr + "Vendor.CurrBal > 0/WHERE" + PRMSEP
ParmStr = ParmStr + "/PSCRN"
Call CallApplicWait("ROI", ParmStr)
```

Pass parameters to ROI using **AppISetParmValue** in conjunction with **CallApplicWait**. Using this method, the report will run properly regardless of the length of the WHERE clause.

```
Call AppISetParmValue(PRMSECTION_VBRDT, "", "03670/RUN")
Call AppISetParmValue(PRMSECTION_VBRDT, "", "03670S/FORMAT")
Call AppISetParmValue(PRMSECTION_VBRDT, "", "Vendor.CurrBal > 0/WHERE")
Call AppISetParmValue(PRMSECTION_VBRDT, "", "/PSCRN")
Call CallApplicWait("ROI", "")
```

See Also

AppIGetParms Function, **AppIGetParmValue** Function

CallChks Function

Perform error checking on a specific object's **Chk** event.

Syntax

IntVar = *CallChks* (*formctl*\$, *ctl*%)

Remarks

This function is useful when you want to manually execute error checking on a specific object because another object's value has been changed. This allows you to "trigger" another **Chk** event under program control.

The **CallChks** function uses the following arguments:

Argument	Type	Description
<i>IntVar</i>	Integer	Any integer variable. (serr, serr1 – serr12 declared in BSL.DH are reserved for this use)
<i>formctl</i>	String	Form object name.
<i>ctl</i>	String	Control object name to execute Chk event.

Example

```
'Object 1 Chk event
Dim Result#, DrAmt#, CrAmt#

DrAmt = GetObjectValue("cdramt")
CrAmt = GetObjectValue("ccramt")
Result = FPAdd(DrAmt, CrAmt, MONEY)
If Result < 0 Then
    Call MsgBox("Negative Entry", MB_OK, "Message")
End If

'Object 2 Chk event (contains ccramt)
serr1 = CallChks("Form1", "object1")
```

See Also

GetObjectValue Function, **SetObjectValue** Function

DateCheck Function

Verify whether or not a date string in MMDDYYYY format represents a valid date.

Syntax

RetVal = **DateCheck**(*DateString*)

Remarks

The **DateCheck** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	0 if the date string represents a valid date. -1 indicates an invalid day. -2 indicates an invalid month.
<i>DateString</i>	String	Date string to be verified. Must be in MMDDYYYY format.

Example

```
datestr = "02291991"
serr = DateCheck(datestr)
If serr = -1 Then
    'Invalid day
Else If serr = -2 Then
    'Invalid month
End If
```

This example performs error checking on a string field that is formatted as a date. The following is its chk event:

```
Sub cuser1_Chk(chkstrg$, retval%)
    serr = DateCheck(chkstrg)
    If serr = -1 Then
        ' Invalid Day
        Call Messbox("Invalid day, please re-enter", MB_OK, "Message")
    End If
    retval = errnomess
    ElseIf serr = -2 Then
        ' Invalid Month
        Call Messbox("Invalid month, please re-enter", MB_OK,
            "Message")
    End If
    retval = errnomess
End Sub
```

See Also

StrToDate Statement

DateCmp Function

Compare two date values.

Syntax

Cmp = **DateCmp**(*Date1*, *Date2*)

Remarks

To determine whether or not a date value is null use **DateCmp**(Date, NULLDATE). NULLDATE is global variable declared in Applic.DH that is properly initialized by the system at the start of all Microsoft SL SDK applications.

The **DateCmp** function uses the following arguments:

Argument	Type	Description
<i>Cmp</i>	Integer	<0 if <i>Date1</i> < <i>Date2</i> 0 if the two dates are equal. >0 if <i>Date1</i> > <i>Date2</i>
<i>Date1</i>	SDate user-defined datatype (declared in Applic.DH)	First date value
<i>Date2</i>	SDate user-defined datatype (declared in Applic.DH)	Second date value

Example

```
Dim TestDate1 As Sdate
Dim TestDate2 As Sdate
TestDate1.Val = GetObjectValue("cinvcdate")
TestDate2.Val = GetObjectValue("cdocdate")
serr1 = DateCmp(TestDate1, TestDate2)
If serr1 = 0 Then
    Call MsgBox("Dates are equal", MB_OK, "Message")
ElseIf serr1 > 0 Then
    Call MsgBox("invdate greater", MB_OK, "Message")
ElseIf serr1 < 0 Then
    Call MsgBox("docdate greater", MB_OK, "Message")
End If
```

DateMinusDate Function

Return the number of days between two dates.

Syntax

NbrDays = **DateMinusDate**(*Date1*, *Date2*)

Remarks

The **DateMinusDate** function uses the following arguments:

Argument	Type	Description
<i>NbrDays</i>	Long	Number of days between <i>Date1</i> and <i>Date2</i> including the ending date. If <i>Date1</i> > <i>Date2</i> then the number of days between the two dates will be a negative value.
<i>Date1</i>	SDate user-defined datatype (declared in <i>Applic.DH</i>)	Beginning date
<i>Date2</i>	SDate user-defined datatype (declared in <i>Applic.DH</i>)	Ending date

Example

```
Dim Date1 As Sdate
Dim Date2 As Sdate

Date1.Val = GetObjectValue("cdate1")
Date2.Val = GetObjectValue("cdate2")

serr1 = DateMinusDate(Date1, Date2)
Call MsgBox("Number of Days is " + Str$(serr1), MB_OK, "Message")
```

See Also

DatePlusDays Statement, **DatePlusMonthSetDay Statement**

DatePlusDays Statement

Add a designated number of days to an existing date.

Syntax

Call **DatePlusDays**(*CurrDate*, *NbrDaysToAdd*, *ResultingDate*)

Remarks

The **DatePlusDays** statement uses the following arguments:

Argument	Type	Description
<i>CurrDate</i>	SDate user-defined datatype (declared in <i>Applic.DH</i>)	Starting date value.
<i>NbrDaysToAdd</i>	Integer	Number of days to add to <i>CurrDate</i> . Negative values are supported.
<i>ResultingDate</i>	SDate user-defined datatype (declared in <i>Applic.DH</i>)	Result of <i>CurrDate</i> + <i>NbrDaysToAdd</i> .

Example

```
Dim OldDate As Sdate
Dim NewDate As Sdate
Dim DaysToAdd%

OldDate.Val = GetObjectValue("cdocdate")
DaysToAdd = 30

Call DatePlusDays(OldDate, DaysToAdd, NewDate)
Call MsgBox("New Date is " + DateToStrSep(NewDate), MB_OK, "Message")
```

See Also

DateMinusDate Function, **DatePlusMonthSetDay** Statement

DatePlusMonthSetDay Statement

Add a designated number of months to an existing date and set the day portion of the resulting date to a specific day of the month.

Syntax

Call **DatePlusMonthSetDay**(*CurrDate*, *NbrMthsToAdd*, *SetSpecificDay*, *ResultingDate*)

Remarks

The **DatePlusMonthSetDay** statement uses the following arguments:

Argument	Type	Description
<i>CurrDate</i>	SDate user-defined datatype (declared in Applic.DH)	Starting date value.
<i>NbrMthsToAdd</i>	Integer	Number of months to add to <i>CurrDate</i> . Negative values are supported.
<i>SetSpecificDay</i>	Integer	Desired value for the day portion of the resulting date. In cases where <i>SetSpecificDay</i> is beyond the last valid day for the relevant month, the system will automatically set the actual day value equal to the last valid day of that month.
<i>ResultingDate</i>	SDate user-defined datatype (declared in Applic.DH)	Resulting date.

Example

```
Dim OldDate As Sdate
Dim NewDate As Sdate
Dim DayOfMonth%
Dim MonthsToAdd%

DayOfMonth = 30
MonthsToAdd = 3
Call StrToDate("11211992", OldDate)

Call DatePlusMonthSetDay (OldDate, MonthsToAdd, DayOfMonth, NewDate)

'NewDate will be 02/28/1993 even though the DayOfMonth is 30
Call MsgBox("New Date is " + DateToStrSep(NewDate), MB_OK, "Message")
```

See Also

DateMinusDate Function, **DatePlusDays** Statement

DateToStr Function

Convert a date value from an SQL date format into a string in MMDDYYYY format.

Syntax

DateString = **DateToStr**(*DateToConvert*)

Remarks

The **DateToStr** function uses the following arguments:

Argument	Type	Description
<i>DateString</i>	String	<i>DateToConvert</i> converted to a string in MMDDYYYY format.
<i>DateToConvert</i>	SDate user-defined datatype (declared in Applic.DH)	Date value to be converted.

Example

```
'Example sets a field to current system date
Dim NewDate as Sdate
Dim NewStrDate$
Call GetSysDate(NewDate)
NewStrDate = DateToStr(NewDate)
serr1 = SetObjectValue("cpaydate", NewStrDate)
```

See Also

DateToStrSep Function, **StrToDate** Statement

DateToStrSep Function

Convert a date value from an SQL date format into a string in MM/DD/YYYY format.

Syntax

DateString = **DateToStrSep**(*DateToConvert*)

Remarks

The **DateToStrSep** and **DateToStr** functions differ only in the fact that **DateToStrSep** inserts a single character separator between the month, day and year portions of the string.

The **DateToStrSep** function uses the following arguments:

Argument	Type	Description
<i>DateString</i>	String	<i>DateToConvert</i> converted to a string in MM/DD/YYYY format.
<i>DateToConvert</i>	SDate user-defined datatype (declared in Applic.DH)	Date value to be converted.

Example

```
Dim TodaysDate As Sdate
Call GetSysDate(TodaysDate)
Call MsgBox("Current Date is" + DateToStrSep(TodaysDate), MB_OK, "Message")
```

See Also

DateToStr Function, **StrToDate** Statement

DispFields Statement

Display the value of the underlying data field(s) corresponding to the designated control(s).

Syntax

Call **DispFields**(*Form*, *FirstControl*, *LastControl*)

Remarks

Each data control is associated with an underlying Visual Basic variable via a combination of its **FieldName** property and an associated **SetAddr** call. The system will automatically redisplay the new value of relevant controls anytime the system is the entity modifying the value of the underlying Visual Basic variable, such as when a new record is loaded. However, when the application directly modifies the value of a Visual Basic variable underlying a data control, then it may also need to call the **DispFields** statement to display the new value in the relevant control.

DispFields can be used to display a range of controls based on their **TabIndex** property order.

The **DispFields** statement uses the following arguments:

Argument	Type	Description
<i>Form</i>	Control	Form containing all controls between <i>FirstControl</i> and <i>LastControl</i> inclusive. PNULL can be passed to include all loaded forms.
<i>FirstControl</i>	Control	First control whose underlying data value is to be displayed. PNULL can be passed to include all controls on the designated <i>Form</i> .
<i>LastControl</i>	Control	Last control whose underlying data value is to be displayed. PNULL can be passed to include all controls on the designated <i>Form</i> .

Example

The following code snippet from the Payroll Earnings Type Maintenance screen illustrates how the **DispFields** statement should be used after the Visual Basic variable underlying a particular data control has been modified programmatically. The code is from the **Chk** event of the Earnings Type combo box control.

```
'Displaying all fields on a form
If Level = 0 Then
    'Set defaults for all objects on NewInfo subform
    'then re-display the results
    serr1 = SetDefaults("NewInfo","")
    Call DispFields ("NewInfo","")
End If

'Displaying one field on a form
Dim NewDate as Sdate
Dim NewStrDate$
Call GetSysDate(NewDate)
NewStrDate = DateToStr(NewDate)
serr1 = SetObjectValue("cpaydate", NewStrDate)
Call DispFields ("Form1","cpaydate")
```

See Also

MDisplay Statement, **SetAddr Statement**

DispForm Statement

Display a designated subform.

Syntax

Call **DispForm**(*SubFormName*, *CenterIt*)

Remarks

DispForm will cause the designated subform to be displayed modally, meaning that no other form from the same application can receive focus until the subform is hidden via a call to **HideForm**.

Form1 is always displayed automatically by the system. Consequently, this call is only necessary for subforms. However the subform must have previously been loaded in the Form1_Load event using the **LoadForm** statement.

The **DispForm** statement uses the following arguments:

Argument	Type	Description
<i>SubFormName</i>	Control	Form to be displayed modally.
<i>CenterIt</i>	Integer	TRUE is the subform is to be centered on the screen. FALSE if the form should be displayed at its design time coordinates.

Example

```
'Display new subform  
Call DispForm("NewForm", True)
```

See Also

[HideForm Statement](#)

DParm Function

Convert a date into an SQL parameter string.

Syntax

```
SQLParmStr = DParm(DateToConvert)
```

Remarks

The **DParm** function uses the following arguments:

Argument	Type	Description
<i>SQLParmStr</i>	String	<i>DateToConvert</i> converted into an SQL parameter string.
<i>DateToConvert</i>	SDate user-defined datatype (declared in <i>Applic.DH</i>)	Date value to convert.

Example

```
'Example retrieves the last voucher date for  
'current vendor and selects a count of all documents  
'less than the last voucher date  
  
Dim SqlStr$  
Dim CountDoc As Long  
Dim DateComp As Sdate  
  
DateComp.Val = GetObjectValue("clastvodate")  
  
SqlStr = "Select Count(*) from APDoc Where DocDate < "  
+ DParm(DateComp)  
  
Call Sql(c1, SqlStr)  
  
serr1 = sgroupfetch1(c1, CountDoc, Len(CountDoc))  
  
Call MsgBox("Number of Documents: " + str$(CountDoc), MB_OK, "Message")
```

See Also

FParm Function, IParm Function, SParm Function

Edit_Cancel Statement

Executes the Cancel toolbar button.

Syntax

Call Edit_Cancel

Remarks

This function corresponds to the Cancel item on the Edit menu and to the **Cancel** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Cancel function from a BSL program.

Example

Example used on push button to perform Cancel function

```
Sub Cancel_Click()  
    Call Edit_Cancel  
End Sub
```

See Also

Edit_Close Statement, Edit_Finish Function

Edit_Close Statement

Executes the Close toolbar button.

Syntax

Call Edit_Close

Remarks

This function corresponds to the Close item on the Edit menu and to the **Close** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Close function from a BSL program.

Example

Example used on push button to perform close function

```
Sub Close_Click()  
    Call Edit_Close  
End Sub
```

See Also

Edit_Next Function, Edit_Last Function

Edit_Delete Function

Executes the Delete toolbar button.

Syntax

RetVal = **Edit_Delete** (*LevelNumber*)

Remarks

This function corresponds to the Delete item on the Edit menu and to the **Delete** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Delete function on the specified Level% from a BSL program.

The **Edit_Delete** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example navigates to 4th row in Journal entry screen, then deletes the row.

```
Dim LvlStr$, Lvl%, I%
serr1 = GetProp("cacct", PROP_LEVEL, LvlStr)
Lvl = Val(LvlStr)
serr1 = Edit_First(Lvl)

'Already on first row, go down 3 more
For I = 1 To 3
    serr1 = Edit_Next(Lvl)
Next I
'Delete existing row
serr1 = Edit_Delete(Lvl)
```

See Also

Edit_New Function, **Edit_Save Statement**

Edit_Finish Function

Executes the Finish toolbar button.

Syntax

`RetVal=Edit_Finish(LevelNumber)`

Remarks

This function corresponds to the Finish item on the Edit menu and to the **Finish** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Finish function from a BSL program.

The **Edit_Finish** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example performs Finish in a Push Button

```
serr1=Edit_Finish
```

See Also

[Edit_Cancel Statement](#), [Edit_Save Statement](#)

Edit_First Function

Executes the First toolbar button.

Syntax

RetVal = **Edit_First** (LevelNumber)

Remarks

This function corresponds to the First item on the Edit menu and to the **First** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's First database navigation function on the specified Level% from a BSL program.

The **Edit_First** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example navigates through detail lines of an existing journal entry. It goes to the first line first, then goes to the next line until done.

```
Sub NavToEndOfList_Click()  
    Dim LvlStr$, Lvl%  
    serr1 = GetProp("cacct", PROP_LEVEL, LvlStr)  
    Lvl = Val(LvlStr)  
    serr1 = Edit_First(Lvl)  
    While serr1 <> NotFound  
        serr1 = Edit_Next(Lvl)  
    Wend  
End Sub
```

See Also

Edit_Next Function, **Edit_Last** Function

Edit_Last Function

Executes the Last toolbar button.

Syntax

RetVal = **Edit_Last** (LevelNumber)

Remarks

This function corresponds to the Last item on the Edit menu and to the **Last** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Last database navigation function on the specified Level% from a BSL program.

The **Edit_Last** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example navigates through detail lines of an existing journal entry. It goes to the last line first, then goes to the previous line until done.

```
Sub NavToTopOfList_Click()
    Dim LvlStr$, Lvl%
    serr1 = GetProp("cacct", PROP_LEVEL, LvlStr)
    Lvl = Val(LvlStr)
    serr = Edit_Last(Lvl)
    While serr = 0
        serr = Edit_Prev(Lvl)
    Wend
End Sub
```

See Also

Edit_Next Function, **Edit_First** Function

Edit_New Function

Executes the New toolbar button.

Syntax

RetVal = **Edit_New** (LevelNumber)

Remarks

This function corresponds to the New item on the Edit menu and to the **New** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's New function on the specified Level% from a BSL program.

The **Edit_New** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example inserts a new document under program control:

```
Dim LvlStr$, Lvl%  
serr1 = GetProp("crefnbr", PROP_LEVEL, LvlStr)  
Lvl = Val(LvlStr)  
serr1 = Edit_New(Lvl)
```

See Also

Edit_Save Statement, Edit_Delete Function

Edit_Next Function

Executes the Next toolbar button.

Syntax

RetVal = **Edit_Next** (LevelNumber)

Remarks

This function corresponds to the Next item on the Edit menu and to the **Next** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Next database navigation function on the specified Level% from a BSL program.

The **Edit_Next** function uses the following arguments:

Argument	Type	Description
RetVal	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
LevelNumber	Integer	Level number to perform operation.

Example

Example navigates through detail lines of an existing journal entry. It goes to the first line first, then goes to the next line until done.

```
Sub NavToEndOfList_Click()
    Dim LvlStr$, Lvl%
    serr1 = GetProp("cacct", PROP_LEVEL, LvlStr)
    Lvl = Val(LvlStr)
    serr1 = Edit_First(Lvl)
    While serr1 <> NotFound
        serr1 = Edit_Next(Lvl)
    Wend
End Sub
```

Example inserts a new row into row 6 of an existing GL Journal entry

```
Sub InsertAtRowSix_Click()
    Dim Cntr As Integer
    Dim Dval As Double
    Dim Ivis As String
    Dim Lvl As Integer

    serr = GetProp("cacct", PROP_VISIBLE, Ivis)
    serr = GetProp("cacct", PROP_LEVEL, Lvl)

    Cntr = 1
    serr = Edit_First(Lvl)

    While serr = 0 And Cntr < 8
        If cntr <> 6 Then
            serr = Edit_Next(Lvl)
        ElseIf Cntr = 6 Then
            serr = Edit_New(Lvl)
            If serr = 0 Then
                serr = SetObjectValue("cacct", "3080")
                serr = SetObjectValue("csub", "03000AA00001")
                serr = SetObjectValue("ctrandate", "09071994")
                serr = SetObjectValue("ctrandesc", Ivis)
                serr = SetObjectValue("cdramt", "100")
                serr = SetObjectValue("ccramt", ".00000000")
            End If
        End If
        Cntr = Cntr + 1
    Wend
End Sub
```

See Also

Edit_Prev Function, **Edit_Last** Function

Edit_Prev Function

Executes the Previous toolbar button.

Syntax

RetVal = **Edit_Prev** (LevelNumber)

Remarks

This function corresponds to the Previous item on the Edit menu and to the **Previous** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Previous database navigation function on the specified Level% from a BSL program.

The **Edit_Prev** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr1 - serr12 are reserved for this use)
<i>LevelNumber</i>	Integer	Level number to perform operation.

Example

Example navigates through detail lines of an existing journal entry. It goes to the last line first, then goes to previous line until done.

```
Sub NavToTopOfList_Click()  
    Dim LvlStr$, Lvl%  
    serr1 = GetProp("cacct", PROP_LEVEL, LvlStr)  
    Lvl = Val(LvlStr)  
    serr1 = Edit_Last(Lvl)  
    While serr1 = 0  
        serr1 = Edit_Prev(Lvl)  
    Wend  
End Sub
```

See Also

Edit_Next Function, **Edit_First** Function

Edit_Save Statement

Executes the Save toolbar button.

Syntax

Call Edit_Save

Remarks

This function corresponds to the Save item on the Edit menu and to the **Save** button on the toolbar. Use of this function allows you to execute Microsoft Dynamics SL's Save database function from a BSL program.

Example

```
Dim TimeOfDay As Stime
Call GetSysTime (TimeOfDay)
serr1 = SetObjectValue("cuser1", TimeOfDay)
Call Edit_Save
```

See Also

Edit_New Function, Edit_Delete Function

ExportCustom Function

Description

Export one or more customizations to an ASCII text file.

Syntax

```
RetVal = ExportCustom(KeySegScreenNbr, KeySegUserID, KeySegBegLevel, KeySegEndLevel,
KeySegLanguageld, KeySegCpnyName, OutputFile, AppendToFile)
```

Remarks

Customizations can be copied to other databases by first exporting them from the source database to an ASCII text file using the **ExportCustom** function. The resulting export file can subsequently be imported into a destination database using the **ImportCustom** function. The Export Customizations (91.500) and Import Customizations (91.510) application screens utilize these two functions to perform their export and import work.

Each call to **ExportCustom** will export all customizations whose unique key corresponds to the various *KeySeg* parameters.

The following are valid values for *KeySegBegLevel* and *KeySegEndLevel*:

100 - Language

200 - Supplemental Product

300 - All Users

400 - One User

500 - Self

The **ExportCustom** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	A non-zero return value indicates an error occurred during the export.
<i>KeySegScreenNbr</i>	String	Screen ID of the RDT application to which the customization being exported applies. For example, if the customization applies to the Sales Order and Memo screen (05.260.00), then 05260 should be passed. SQL wildcard characters are supported.
<i>KeySegUserID</i>	String	If the level of the source customization is Self then User ID should contain the ID of the user who created the customization. Correspondingly if the level of the source customization is One User then User ID should contain the ID of the user to which the customization applies. SQL wildcard characters are supported.
<i>KeySegBegLevel</i>	Integer	Microsoft Dynamics SL allows various levels of customization to exist for any particular screen. For example, two customizations could exist for the Sales Order and Memo screen (05.260.00). One customization could be for All Users and the other could be for One User named THOMAS. Both customizations are for the same screen. The only difference is the Customization Level. <i>KeySegBegLevel</i> should contain the beginning (lowest) level number of all the customizations which are to be exported for the specified <i>KeySegScreenNbr</i> .
<i>KeySegEndLevel</i>	Integer	Ending (highest) level number of all the customizations which are to be exported for the specified <i>KeySegScreenNbr</i> . If only a single customization is to be exported then <i>KeySegBegLevel</i> and <i>KeySegEndLevel</i> should both contain the precise level number of the particular customization which is to be exported.
<i>KeySegLanguageld</i>	String	Language Id of the source customization. SQL wildcard characters are supported.
<i>KeySegCpnyName</i>	String	Multiple customizations can exist within the Supplemental Product customization level, differentiated only by their Company Name. <i>KeySegCpnyName</i> facilitates the export of a particular customization at the Supplemental Product level by allowing a unique Company Name to be specified. SQL wildcard characters are supported.

Argument	Type	Description
<i>OutputFile</i>	String	Fully qualified name of the file to which the customization(s) are to be exported. The standard file extension for customization export files is CST.
<i>AppendToFile</i>	Integer	True if the customization should be appended to the <i>OutputFile</i> . False if the contents of <i>OutputFile</i> should be overwritten.

Example

The following example illustrates a simple process that will export all of the Custom records contained in the Mem_Custom memory array.

Note: The initialization of the Mem_Custom array, such as opening and loading the memory array, is purposely not included so as to focus on the ExportCustom function call.

```

Dim Mem_Fetch                As Integer
Dim Mem_MaintFlg            As Integer
Dim Nbr_Selected_Recs_Processed As Integer
Dim File_Append_Flag        As Integer
Dim Error_Encountered       As Integer

Call status(StartProcess, False, "", DISP_ONLY)
Mem_Fetch = mfirst(Mem_Custom, Mem_MaintFlg)
While ((Mem_Fetch = 0) And (Error_Encountered = 0))
  'In the event the output file already exists, the first
  'customization should replace its entire contents. All
  'customizations subsequently exported should be appended
  'to the output file.
  If (Nbr_Selected_Recs_Processed = 0) Then
    File_Append_Flag = False
  Else
    File_Append_Flag = True
  End If

  'Export the current customization
  Error_Encountered = ExportCustom(bCustom.ScreenId, ↓
    bCustom.UserId, bCustom.Sequence, bCustom.Sequence, ↓
    bCustom.LanguageId, bCustom.Company_Name, "CUSTOM.CST", ↓
    File_Append_Flag)

  'Update counter controlling file append
  Nbr_Selected_Recs_Processed = Nbr_Selected_Recs_Processed + 1

  Mem_Fetch = mnext(Mem_Custom, Mem_MaintFlg)
Wend

Call status(EndProcess, False, "", DISP_ONLY)

```

See Also

ImportCustom Function

FPAdd Function

Add two double-precision floating-point values together with a designated rounding precision.

Syntax

Result = **FPAdd**(*Dbl1*, *Dbl2*, *Precision*)

Remarks

Error conditions occurring during the addition operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPAdd** function uses the following arguments:

Argument	Type	Description
<i>Result</i>	Double	Return value
<i>Dbl1</i>	Double	First double value
<i>Dbl2</i>	Double	Second double value
<i>Precision</i>	Integer	Rounding precision.

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in `Applic.DH`:

- **MONEY** – Monetary value
- **INV_UNIT_QTY** – Inventory unit quantity
- **UNITS** – Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** – Inventory unit price
- **PERCENT** – Percentage value

Example

```
'Add the current to future balance for a total balance

Dim CurrentBalance#
Dim FutureBalance#
Dim TotalBal#

CurrentBalance = GetObjectValue("ccurrbal")
FutureBalance = GetObjectValue("cfuturebal")
TotalBal = FPAdd(CurrentBalance, FutureBalance, MONEY)
Call MsgBox(Str$(TotalBal), MB_OK, "Message")
```

See Also

FPDiv Function, **FPMult** Function, **FPRnd** Function, **FPSub** Function

FParam Function

Convert a double-precision floating-point value into an SQL parameter string.

Syntax

SQLParmStr = **FParam**(*DbItoConvert*)

Remarks

The **FParam** function uses the following arguments:

Argument	Type	Description
<i>SQLParmStr</i>	String	<i>DbItoConvert</i> converted into an SQL parameter string.
<i>DbItoConvert</i>	Double	Double-precision floating-point value to convert.

Example

```
Dim MaxAmount#
Dim SqlStr$
Dim CountDoc As Long

'Obtain documents with balance over MaxAmount
MaxAmount = 1000
SqlStr = "Select Count(*) From ARDoc Where DocBal > "
SqlStr = SqlStr + FParam(MaxAmount)
Call Sql(c1, SqlStr)
serr1 = sgroupfetch1(c1, CountDoc, Len(CountDoc))
Call MsgBox ("Documents over Max: " + Str$(CountDoc), MB_OK, "Message")
```

See Also

DParam Function, **IParam** Function, **SParam** Function

FPDiv Function

Divide one double-precision floating-point value by another with a designated rounding precision.

Syntax

Result = **FPDiv**(*Db11*, *Db12*, *Precision*)

Remarks

This function will divide the value of *Db11* by *Db12* and return the result.

Error conditions occurring during the division operation, such as division by zero, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPDiv** function uses the following arguments:

Argument	Type	Description
<i>Result</i>	Double	Return value
<i>Db11</i>	Double	First double value
<i>Db12</i>	Double	Second double value
<i>Precision</i>	Integer	Rounding precision.

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in `Applic.DH`:

- **MONEY** -Monetary value
- **INV_UNIT_QTY** – Inventory unit quantity
- **UNITS** – Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** – Inventory unit price
- **PERCENT** – Percentage value

Example

```
Dim Var1#, Var2#, Result#

Var1 = 100
Var2 = 10
Result = FPDiv(Var1, Var2, MONEY)

'Result is 10
Call MsgBox(Str$(Result), MB_OK, "Message")
```

See Also

FPAdd Function, **FPMult** Function, **FPRnd** Function, **FPSub** Function

FPMult Function

Multiply two double-precision floating-point values together to a designated rounding precision.

Syntax

Result = **FPMult**(*Db1*, *Db2*, *Precision*)

Remarks

Error conditions occurring during the multiplication operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPMult** function uses the following arguments:

Argument	Type	Description
<i>Result</i>	Double	Return value
<i>Db1</i>	Double	First double value
<i>Db2</i>	Double	Second double value
<i>Precision</i>	Integer	Rounding precision.

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in `Applic.DH`:

- **MONEY** -Monetary value
- **INV_UNIT_QTY** – Inventory unit quantity
- **UNITS** – Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** – Inventory unit price
- **PERCENT** – Percentage value

Example

```
Dim Var1#, Var2#, Result#

Var1 = 100
Var2 = 10
Result = FPMult(Var1, Var2, MONEY)

'Result is 1000
Call MsgBox(Str$(Result), MB_OK, "Message")
```

See Also

FPAAdd Function, FPDIV Function, FPRnd Function, FPSub Function

FPRnd Function

Round a double-precision floating-point value to a designated rounding precision.

Syntax

Result = **FPRnd**(*DbItoRound*, *Precision*)

Remarks

Error conditions occurring during the rounding operation, such as an overflow error, are handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPRnd** function uses the following arguments:

Argument	Type	Description
<i>Result</i>	Double	Return value
<i>DbItoRound</i>	Double	Value to be rounded
<i>Precision</i>	Integer	Rounding precision.

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in `Applic.DH`:

- **MONEY** -Monetary value
- **INV_UNIT_QTY** – Inventory unit quantity
- **UNITS** – Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** – Inventory unit price
- **PERCENT** – Percentage value

Example

Example issues an SQL statement to retrieve a Sum of total documents. Displays the results in a message box and formats the number.

```
Dim SqlStr$
Dim Result#

SqlStr = "Select Sum(OrigDocAmt) From ARDoc"
Call Sql(C1, SqlStr)
serr1 = sgroupfetch1(C1, Result, Len(Result))

Call MsgBox(Format$(FPRnd(Result, MONEY) , "$###,###,###.00"), MB_OK, "Message")
```

See Also

FPAAdd Function, **FPAdiv** Function, **FPMult** Function, **FPSub** Function

FPSub Function

Subtract one double-precision floating-point value from another with a designated rounding precision.

Syntax

Result = **FPSub**(*Db11*, *Db12*, *Precision*)

Remarks

This function will subtract the value of *Db12* from *Db11* and return the result.

Error conditions occurring during the subtraction operation, such as an overflow error, will be handled automatically by the system. These types of errors will cause the appropriate error message to be either displayed on the screen or written to the process status log depending on the context in which the error occurred. After an error condition has been properly reported, the application will be terminated.

The **FPSub** function uses the following arguments:

Argument	Type	Description
<i>Result</i>	Double	Return value
<i>Db11</i>	Double	First double value
<i>Db12</i>	Double	Second double value
<i>Precision</i>	Integer	Rounding precision.

Note: The precision parameter can be an explicit precision value as well as one of the following symbolic constants defined in `Applic.DH`:

- **MONEY** -Monetary value
- **INV_UNIT_QTY** – Inventory unit quantity
- **UNITS** – Work units such as hours worked in Payroll
- **INV_UNIT_PRICE** – Inventory unit price
- **PERCENT** – Percentage value

Example

```
Dim Var1#, Var2#, Result#

Var1 = 1000
Var2 = 300

Result = FPSub(Var1, Var2, MONEY)

'Result will be 700.00
Call MsgBox(Format$(Result, "####.00"), MB_OK, "Message")
```

See Also

FPAAdd Function, FPDiv Function, FPMult Function, FPRnd Function

GetBufferValue Statement

Obtains an underlying Microsoft Dynamics SL application's data buffer field value.

Syntax

Call **GetBufferValue**(*bTable.FieldName*, *Str*)

Remarks

If a BSL application issues its own **SetAddr** calls it can then reference any of these structures from within code. However, if the underlying application's structures need to be referenced and these fields are not represented as objects on the form, this statement allows the BSL application to obtain these values. If the fields were objects on the form, the BSL application can simply issue **GetObjectValue** instead.

The **GetBufferValue** statement uses the following arguments:

Argument	Type	Description
<i>bTable.FieldName</i>	String	SQL Table.FieldName that you wish to retrieve.
<i>Str</i>	String	String variable in which to store the contents of the buffer's field value.

Example

```
' get account number from gltran record  
Dim AccountValue As String * 10  
Call GetBufferValue("bgltran.acct",AccountValue)
```

See Also

SetBufferValue Statement

GetDelGridHandle Function

Returns the resource handle of the memory array used to temporarily hold detail lines deleted from the designated **SAFGrid** control.

Syntax

`DelMemHandle = GetDelGridHandle(SpreadsheetObj)`

Remarks

Each **SAFGrid** control is associated with two underlying memory arrays that the system opens automatically during Form Load. The primary memory array holds the records that are actually visible in the grid. The resource handle to this primary memory array is actually returned by the **GetGridHandle** function. However another array is also created to temporarily hold user-deleted records until a save operation is performed and the deletions are actually committed to the database. The resource handle to this memory array can be retrieved using the **GetDelGridHandle** function.

Once the resource handle for the deleted record memory array has been retrieved, the application can use it to loop through user-deleted records using calls such as **MFirst** and **MNext**.

The **GetDelGridHandle** function uses the following arguments:

Argument	Type	Description
<i>DelMemHandle</i>	Integer	Resource handle for the memory array holding records deleted from the designated Spreadsheet.
<i>SpreadSheetObj</i>	String	SpreadSheet object name.

Example

```
serr1 = GetDelGridHandle("Spread1")
```

See Also

GetGridHandle Function

GetGridHandle Function

Obtains the grid handle for a Spreadsheet object.

Syntax

`IntVar = GetGridHandle (SpreadControlName$)`

Remarks

Returns the memory array handle for the specified spreadsheet object. Useful if the application wants to use memory array functions to navigate through a spreadsheet object. Should be used with extreme caution since the underlying application may override memory array assumptions performed in the BSL program.

The **GetGridHandle** function uses the following arguments:

Argument	Type	Description
<i>IntVar</i>	Integer	Any integer variable. (serr, serr1 - serr12 declared in BSL.DH are reserved for this use)
<i>SpreadControlName</i>	String	Object name of the spreadsheet object.

Example

```
Dim MemHandle As Integer
MemHandle = GetGridHandle ("Spread1")
```

See Also

MSet Statement

GetObjectValue Function

Obtains the value of a specified object on the form.

Syntax

```
RetVal = GetObjectValue("ControlName")
```

Remarks

Whenever a BSL application needs to obtain data values from any of the controls on the form, this function is used.

The **GetObjectValue** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	User-Defined	Any variable type. Must match database field type of the ControlName.
<i>ControlName</i>	String	Name of the control whose field value you want to obtain.

Example

```
Dim TestDate1 As Sdate
TestDate1.Val = GetObjectValue ("cinvcdate")

'Double field example
Dim CrTotal#
CrTotal = Val(GetObjectValue ("ccrtot"))

'Logical field example
'Note that an integer variable type is declared
Dim Var1%
Var1 = GetObjectValue ("cmultichk")

If Var1 = 0 Then
    Call MsgBox("Value is UnChecked or False", MB_OK, "Message")
ElseIf Var1 = 1 Then
    Call MsgBox("Value is Checked or True", MB_OK, "Message")
End If

'Integer field example
Dim CycleCount%
CycleCount = GetObjectValue ("ccycle")

'String field example
Dim DispCountry As String
DispCountry = GetObjectValue ("ccountry")
```

See Also

[SetObjectValue Function](#)

GetProp Function

Obtains a property value for a specified object.

Syntax

RetVal = **GetProp**(ObjectName, PropName, PropValue)

Remarks

This function allows the BSL application to obtain a property value of any object on the screen. In addition to the property value, the application can also obtain the level associated with the particular object. This function is useful in the **Edit_** functions as well as the **SetProp** functions because you can check the value of the property before setting it.

The **GetProp** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 are reserved for this use)
<i>ObjectName</i>	String	Object name.
<i>PropName</i>	String	Property name you want the value of. Note: Any native property name that is available for the object may be specified.
<i>PropValue</i>	User Defined	Variable to retrieve the property value into.

The following values for the **PropertyName** argument are defined as symbolic constants in BSL.DH:

Symbolic constant	Valid data type	Valid data values
PROP_BLANKERR (required)	Integer	TRUE / FALSE
PROP_CAPTION	String	
PROP_CUSTLIST	String	
PROP_ENABLED	Integer	TRUE / FALSE
PROP_HEADING	String	
PROP_LEVEL	String	
PROP_MASK	String	
PROP_MIN	String	
PROP_MAX	String	
PROP_TABSTOP	Integer	TRUE / FALSE
PROP_VISIBLE	Integer	TRUE / FALSE

Example

Example navigates through batches in Journal Entry screen:

```
Sub NavThruBatches_Click()
    Dim Lvl As Integer
    Dim LvlStr As String * 1

    ' Since cbatnbrh is a key field, level will be
    ' returned as "0,k" not "0"
    ' So declare lvlstr as a one-character string, so
    ' we get back only "0", and then manually convert it
    ' for use in Edit calls
    serr = GetProp("cbatnbrh",PROP_LEVEL, LvlStr)
    Lvl = Val(LvlStr)

    'Perform navigation to first batch
    serr = Edit_First(Lvl)
    While serr <> NotFound
        DoEvents
        serr = Edit_Next(Lvl)
    Wend
End Sub
```

See Also

SetProp Statement, Edit_Next Function

GetSysDate Statement

Retrieve the current system date.

Syntax

Call **GetSysDate**(Date)

Remarks

The **GetSysDate** statement uses the following arguments:

Argument	Type	Description
<i>Date</i>	SDate user-defined datatype (declared in Applic.DH)	Date variable to be initialized to the current system date.

Example

This sample routine sets an existing date field to the system date.

```
Sub setdate_Click()  
    Dim NewDate as Sdate  
    Dim NewStrDate$  
  
    Call GetSysDate(NewDate)  
    NewStrDate = DateToStr(NewDate)  
  
    serr1 = SetObjectValue("cpaydate", NewStrDate)  
    Call DispFields("Form1", "cpaydate")  
End Sub
```

See Also

GetSysTime Statement

GetSysTime Statement

Retrieve the current system time.

Syntax

Call **GetSysTime**(*Time*)

Remarks

The **GetSysTime** statement uses the following arguments:

Argument	Type	Description
<i>Time</i>	STime user-defined datatype (declared in Applic.DH)	Time variable to be initialized to the current system time.

Example

```
'Obtain current system time
Dim SystemTime As STime
Call GetSysTime(SystemTime)
Call MsgBox("Current Time is: " + TimeToStr(SystemTime), MB_OK, "Message")
```

See Also

[GetSysDate Statement](#)

HideForm Statement

Hide a designated subform previously displayed via a call to **DispForm**.

Syntax

Call **HideForm**(*SubFormName*)

Remarks

This function is typically used in the click event of the OK or Cancel button of the designated subform.

The **HideForm** statement uses the following arguments:

Argument	Type	Description
<i>SubFormName</i>	Control	Form to be hidden.

Example

```
Sub OkButton_Click()
    Call HideForm("NewForm")
End Sub
```

See Also

[DispForm Statement](#)

ImportCustom Function

Import customizations from an ASCII customization export file.

Syntax

RetVal = **ImportCustom**(*ImportFile*, *ConflictResolution*, *ErrorHandling*)

Remarks

Customizations can be imported directly into a database from a customization export file which previously by either the Export Customizations (91.500) screen or the **ExportCustom** function.

Each call to **ImportCustom** will import all customizations contained within the *ImportFile*. In the event an error occurs during the import operation a description of the error will be written to the process status log.

The **ImportCustom** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Zero if no errors occurred. Non-zero if one or more errors occurred (regardless of which error handling option was used).
<i>ImportFile</i>	String	Fully qualified name of the file from which the customization(s) are to be imported.
<i>ConflictResolution</i>	Integer	Option specifying how conflicts with existing customizations should be resolved. The following are valid values along with a corresponding description: 0 – Overwrite Existing Customization 1 – Reject New Customization 2 – Merge Both Customizations
<i>ErrorHandling</i>	Integer	Option specifying how errors detected in the <i>ImportFile</i> should be handled. The following are valid values along with a corresponding description: 0 – Ignore Syntax Errors 1 – Reject Entire File

Conflict Resolution Notes

While processing any particular Import File the system may detect that a customization already exists in the position where a new customization is destined for storage. For example, suppose that a customization with the following keys already exists in the database: Screen ID: 05260, Customization Level: One User, User ID: THOMAS and a blank Company Name. If the Import File contains a customization with these same key field values then an import conflict will occur. In this case there are several different methods of Conflict Resolution:

- Using the Overwrite option, the new customization will overwrite the current customization.
- However, if you are certain that you never want to overwrite any existing customizations, then you can specify the Reject Customization option. In the event of a conflict, this option will cause the new customization to be rejected.
- The most sophisticated option allows you to Merge new customizations together with existing customizations. This is an extremely powerful feature which allows two customizations to be merged into one new customization down to the property level. For example, assume the position of ControlA has been customized so that its on-screen position varies from the standard screen. Now suppose that we want to import a customization in which a reference will be made to ControlA. In particular the new customization is going to disable ControlA. If the sophisticated Merge option is utilized, then the result will be that ControlA will have both its screen position and enabled properties customized! The only case in which a Merge cannot be successfully carried out by the system is in the case where a conflict occurs at the property or BSL code procedure level. In our example, if the new customization being imported also customizes the screen position of ControlA, then a conflict at the property level will result. In such a case the new customization will take precedence over the existing customization.

Error Handling Notes

These options control how the system should respond to errors which may occur during the import operation. An Import Error occurs anytime a syntax error is found to exist in the Import File currently being processed. For example, each customization in the Import File must begin with a line that starts with 'Begin Customization' and goes on to specify other key values. If the ASCII text in the Import File is 'Beginnn Customization', then an Import Error will result since the keyword Begin is misspelled.

There are two different methods of Error Handling

- The Reject Entire File option will cause the entire Import File to be rejected if any Import Errors occur while importing any of the customizations contained therein. Since an Import File can contain many customizations this option facilitates an "all or nothing" type of import operation.
- An alternate method of handling Import Errors is to Ignore them. This option should be used with caution. The usefulness of this option is primarily during development of sophisticated customizations containing advanced Basic Script code. The ability to Ignore errors essentially provides the ability to compile the entire customization, including all Basic Script code, and receive a list of all errors in the entire customization in a single attempt. You can subsequently fix all of the errors and import the customization again this time using the Reject Entire File method of handling Import Errors. If an error actually occurs during the import process and you have chosen to Ignore errors the resulting customization may not operate properly. Suffice it to say if you receive any error during the import process you should make appropriate corrections to the Import File and import the customization again before attempting to actually use the screen being customized.

In addition to importing new customizations, the **ImportCustom** function can also be used to automatically delete existing customizations. The Import File simply needs to contain a line beginning with " Delete" followed by all the key field values necessary to identify a unique customization. For example: ' Delete Screen: 05260 Sequence: 500 UserId:"SYSADMIN" CompanyName:"'

You will notice a field called Sequence in the preceding example. Sequence is the technical term for customization level. The following are valid values for Sequence:

- 100** – Supplemental Product
- 200** – Language
- 300** – All Users
- 400** – One User
- 500** – Self

Within the Import File a Delete line can occur anywhere a Begin Customization line can occur. Thus a Delete line cannot occur within a Begin Customization and End Customization block of text. When you examine the contents of an Import File you will notice that its composition is very similar to a Visual Basic ASCII form file. The following diagram illustrates the fundamental structure of an Import File.

Example

Begin Customization with various key fields identifying a unique customization.

```
Begin ControlType ControlName
    Customized Properties
End

Begin ControlType ControlName
    Customized Properties
End

Begin Macro Text
Sub ProcedureName( )
End Sub

End Customization

Begin Customization ...Various key fields identifying a unique customization...
End Customization
.
.
.
Delete ...Various key fields identifying a unique customization...
```

The following code fragment illustrates how to import all customizations contained in a file called CUSTOM.CST. Conflicts with existing customizations will be resolved by specifying that the new customization should overwrite the existing customization. All customizations contained within the CUSTOM.CST import file will be rejected if any syntax errors are encountered.

```
Dim Record_Count As Integer

Call Status(StartProcess, False, "", DISP_ONLY)

Record_Count = ImportCustom("CUSTOM.CST", 0, 1)

Call Status(EndProcess, False, "", DISP_ONLY)
```

See Also

ExportCustom Function

ImportField Function

This function returns one field of information from the data file to the calling function and/or subroutine running within Transaction Import.

Syntax

ImportField(*IntIndex*)

Remarks

You can also use it to retrieve the entire data line by specifying an index of -1.

The **ImportField** function returns the following:

Argument	Type	Description
<i>IntIndex</i>	Integer	Specifies the offset number in the Transaction Import data file.

Example

Following examples retrieve fields 1, 2, and 3 from the Transaction Import data file and are passed to the SetObjectValue function:

```
serr=SetObjectValue("cvendid",ImportField(1))
serr=SetObjectValue("cterms",ImportField(2))
serr=SetObjectValue("cdocamt",ImportField(3))
```

See Also

SetObjectValue Function

IncrStrg Statement

Increment a string representation of a whole number value.

Syntax

Call **IncrStrg**(*StringNbr*, *Length*, *Increment*)

Remarks

The **IncrStrg** statement uses the following arguments:

Argument	Type	Description
<i>StringNbr</i>	String	String whose current whole number is to be incremented.
<i>Length</i>	Integer	Size of <i>StringNbr</i> . It is not required that this value equal the full size of <i>StringNbr</i> . For example if the string can actually hold 10 bytes but currently the developer only desires to use 6 byte values then a value of 6 can be passed.
<i>Increment</i>	Integer	Amount by which <i>StringNbr</i> is to be incremented.

Example

```
Dim BatNbrLength As Integer

BatNbrLength = LenB(Trim$(bGLSetup.LastBatNbr))

'Increment last batch number to the next sequential value (within the
'size of batch numbers actually being used - i.e., BatNbrLength).
Call IncrStrg(bGLSetup.LastBatNbr, BatNbrLength, 1)
```

IParm Function

Convert an integer into an SQL parameter string.

Syntax

SQLParmStr = **IParm**(*IntToConvert*)

Remarks

The **IParm** function uses the following arguments:

Argument	Type	Description
<i>SQLParmStr</i>	String	<i>IntToConvert</i> converted into an SQL parameter string.
<i>IntToConvert</i>	Integer	Integer value to convert. Note: INTMIN and INTMAX are global constants available that may optionally be used. These represent -32768 and 32767 respectively, which are the minimum and maximum small integer range values.

Example

These examples assume the following SQL statement was used to create a stored procedure called `GLTran_Module_BatNbr_LineNbr`

```
Select * from GLTran
      where Module = @parm1
      and BatNbr = @parm2
      and LineNbr between @parm3beg and @parm3end
      order by Module, BatNbr, LineNbr;
```

This code snippet illustrates how the previously defined stored procedure can be used to fetch a single transaction having a `LineNbr` of 84 in GL Batch #000123.

```
SqlStr = " GLTran_Module_BatNbr_LineNbr" + SParm("GL") + ␣
        SParm("000123") + IParm(84) + IParm(84)
GLTran_Fetch = SqlFetch1(CSR_GLTran, SqlStr, bGLTran, LenB(bGLTran))
```

This code snippet illustrates the previously defined stored procedure can be used to fetch all transactions in GL Batch #000123.

```
SqlStr = " GLTran_Module_BatNbr_LineNbr" + SParm("GL") + ␣
        SParm("000123") + IParm(INTMIN) + IParm(INTMAX)
GLTran_Fetch = SqlFetch1(CSR_GLTran, SqlStr, bGLTran, LenB(bGLTran))

While ( GLTran_Fetch = 0)
    GLTran_Fetch = SFetch1( CSR_GLTran, bGLTran, LenB(bGLTran))
Wend
```

See Also

DParm Function, **FParm** Function, **SParm** Function

Is_TI Function

Returns whether or not the application is in Transaction Import mode.

Syntax

`IntVar = Is_TI`

Remarks

This function is useful when you want to suppress user-responses and/or dialogs when the application is running in Transaction Import mode.

The **Is_TI** function returns the following:

True — Application is running in Transaction Import mode.

False — Application is not running in Transaction Import mode.

Example

Following example determines whether the application is in Transaction Import mode:

```
If Is_TI = False Then
  'Perform logic or routine only desired in standard mode. (not Transaction Import mode)
End If
```

See Also

CallChks Function, GetGridHandle Function

Launch Function

Launches another executable program.

Syntax

RetVal = **Launch** (CommandLine, SAFApp, WaitFlag, WindowFlag)

Remarks

Allows you to launch any Microsoft Dynamics SL screen (or any other Windows application) from any other Microsoft Dynamics SL screen. You can also pass parameters to the screens and use the **ApplGetParms** function in the Launched application to retrieve the parameters.

The **Launch** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Any integer variable. (serr, serr1 - serr12 declared in BSL.DH are reserved for this use)
<i>CommandLine</i>	String	The command line required to execute the application. (parameters included separated by PRMSEP constant declared in BSL.DH)
<i>SAFApp</i>	Integer	Whether or not the application is a Microsoft Dynamics SL application. (True or False)
<i>WaitFlag</i>	Integer	Whether or not the launched application is modal. When the launched application is modal, no other form is able to be clicked until the application is closed. When this parm is True and the application being launched is another application, then the launched application shares the same SQL session with the application that launched it. Therefore, an extra session is not used. This exhibits the same behavior as Quick Maintenance.
<i>WindowFlag</i>	Integer	State of the launched application. BSL.DH has two constants declared for this purpose: LaunchMaximize or LaunchMinimize. Zero launches the application in its normal state.

To launch a report, you must launch the ROI application with the appropriate parameters. The following are the available parameters (and their descriptions) which you can pass to ROI:

Parameter	Description
/RUN	Follows the report number you wish to execute.
/WHERE	Follows the SQL where clause restrictions.
/DEBUG	Creates a text file called RSWRUN.? in the Microsoft Dynamics SL program directory. This TXT file contains all of the parameters passed to the report. (Only useful for debugging purposes)
/FORMAT	Follows the format name or number of the particular report.
/PSCRN	Prints the report to screen if specified. (Default is current printer).
/TEMPLATE	Follows the Report Template name that you wish to load for the report.

Example

Following code for 03.270.00 (the launched application):

```
Sub Form1_Load
    ' Variable to store the passed Parameter to this screen
    Dim VendorParm$

    VendorParm = ApplGetParms()

    If Trim$(VendorParm) <> "" Then
        ' Screen was called from another application.
        ' Set the value of the ID field to what was
        ' passed in by Launch() function
        serr1 = SetObjectValue("cvendid", VendorParm)
    End If
End Sub
```

Following code for 03.010.00 (the launching application). Even though this example uses a push button, it could be placed in the **Chk** event of a particular control. The syntax would be very similar except that chkstrg is evaluated and passed to the **Launch**() function:

```
Sub VendorScrn_Click()
    ' Obtain the Current ID in this screen and pass
    ' as a parameter to other EXE

    Dim CmdLine$ 'Stores the command line string for Launch()
    Dim VendParm$ 'Stores the current ID from screen

    VendParm = GetObjectValue("cvendid")

    If Trim$(VendParm) <> "" Then
        ' Note that VendParm is passed twice with a PRMSEP between
        ' Also note the space following 0327000. This is also
        ' required.
        CmdLine = "0327000 " + VendParm + PRMSEP + VendParm

        ' The third parm indicates Wait = True. This will allow
        ' the Launched application to share the same SQL session,
        ' therefore, an extra session is not used. This exhibits the
        ' same behavior from a shared session perspective as Quick
        ' Maintenance.
        serr1 = Launch(CmdLine, True, True, 0)
    End If
End Sub
```

Launching the ROI

The following example will print the Vendor Trial Balance for the current vendor in screen 03.270.00. This code is placed in the click event of a push button. Note that you must have one space following ROI.EXE in the first parameter:

```
Dim VendorId$, ParmStr$
VendorId = GetObjectValue("cvendid")
ParmStr = "ROI.EXE " + PRMSEP + "03650/RUN" + PRMSEP + "03650C/FORMAT"
+ PRMSEP + "Vendor.VendId =" + sparm(VendorID) + "/WHERE"
+ PRMSEP + "/PSCRN"
```

```
serr1 = Launch(ParmStr, True, True, 0)
```

See Also

ApplGetParms Function, **ApplGetParmValue** Function, **ApplSetParmValue** Statement

MCallChks Function

Perform error checking on a specific grid object's column.

Syntax

IntVar = (*gridhandle%*, *ctlbeg\$*, *ctlend\$*)

Remarks

This function is useful when you want to manually execute error checking on a grid object column because another object's value has been changed. This allows you to "trigger" the **Chk** event under program control.

The **MCallChks** function uses the following arguments:

Argument	Type	Description
<i>IntVar</i>	Integer	Any integer variable. (<i>serr</i> , <i>serr1</i> – <i>serr12</i> declared in BSL.DH are reserved for this use)
<i>gridhandle</i>	Integer	Grid object handle.
<i>ctlbeg</i>	String	Beginning control object name to execute Chk event.
<i>ctlend</i>	String	Ending control object name to execute Chk event.

See Also

CallChks Function, **GetGridHandle** Function

MClear Statement

Delete all records from the designated memory array.

Syntax

Call **MClear**(*MemHandle*)

Remarks

The **MClear** statement can be used to clear an existing memory array of its contents. The array will stay allocated and can be subsequently re-used.

The **MClear** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle.

See Also

MOpen Function

MClose Statement

Close an existing memory array.

Syntax

Call **MClose**(*MemHandle*)

Remarks

MClose can be used to close a memory array previously opened using one of the **MOpen** functions. Memory arrays created automatically by the **DetailSetup** functions should not be closed by the application.

The **MClose** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle.

Example

Illustrates how to close a memory array no longer needed by the application.

```
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer
'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), "", 0, "", 0, ; "", 0)
'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)
'Initialize cursor with a SQL statement and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1( CSR_Account, SqlStr, bAccount, Len(bAccount))
'Read through all subsequent Account records, inserting each one into the 'memory
array.
While( Account_Fetch = 0)
    'Insert current Account record into the memory array
    Call MInsert( Mem_Account)
    'Fetch the next Account record
    Account_Fetch = SFetch1( CSR_Account, bAccount, Len(bAccount))
Wend

'Close the memory array
Call MClose( Mem_Account)
```

See Also

MOpen Function

MDelete Function

Delete the current record from the designated memory array.

Syntax

RecFetch = **MDelete**(*MemHandle*, *RecMaintFlg*)

Remarks

The current record in a memory array can be deleted by using the **MDelete** function. After the record is deleted, the system will automatically navigate to the next memory array record since there must always be a current record in memory arrays, assuming of course that one or more records exist. Consequently, the return value and corresponding record status apply to the next record after the deleted record.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **SAFGrid** appearance with the memory array.

The **MDelete** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if the next record is successfully fetched. NOTFOUND is returned if no subsequent records exist in the specified memory array (for example, when the last record itself is deleted). This does not mean that no additional records exist in the memory array. Rather it simply means that no records exist after the record just deleted.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	Status of the memory array record (assuming it was successfully fetched). Applic.DH contains the following symbolic constants defining possible memory array record status values: INSERTED , UPDATED and NOTCHANGED

See Also

DetailSetup Functions, **MDisplay Statement**, **MInsert Statement**, **MUpdate Statement**

MDisplay Statement

Display the current contents of the designated memory array in its corresponding **SAFGrid** control.

Syntax

Call **MDisplay**(*MemHandle*)

Remarks

Each **SAFGrid** control is associated with an underlying memory array that is opened automatically by the system during the **DetailSetup** call. Anytime data within this memory array is modified directly by the application, as opposed to by the user, the **SAFGrid** control must be subsequently redisplayed.

When **MDisplay** is called, the current memory array record will be displayed at the top of the **SAFGrid** control. Thus, for example, if the application wants the first memory array record to display at the top of the **SAFGrid** control it should initially call **MFirst** to move to the first record and then call **MDisplay**.

The **MDisplay** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle. This memory array must be associated with an SAFGrid control.

Example

```
Sub cBegProcessing_Click ()
    Dim RecFound As Integer
    Dim MemMaintFlg As Integer
    Dim Nbr_Of_Batches_Processed As Integer

    'Explicitly initialize processing counter to zero BEFORE calling
    'ProcValidBatch() for the FIRST time.
    Nbr_Of_Batches_Processed = 0

    RecFound = MFirst(MemHandle, MemMaintFlg)

    While (RecFound = 0)
        If (bCurrBatchSelected = True) Then
            'Process the selected batch
            Call ProcValidBatch(Nbr_Of_Batches_Processed)

            'Delete current and get next memory array batch record
            RecFound = MDelete(MemHandle, MemMaintFlg)
        Else
            'Current batch is not selected so get the next batch from the
            'memory array.
            RecFound = MNext(MemHandle, MemMaintFlg)
        End If
    Wend

    'Redisplay the grid with the modified contents of the memory array.
    RecFound = MFirst(MemHandle, MemMaintFlg)
    Call MDisplay(MemHandle)
End Sub
```

See Also

DetailSetup Functions

Mess Statement

Displays a message from the message file and waits for the user to choose a button.

Syntax

Call **Mess**(*MsgNumber*)

Remarks

When Microsoft Dynamics SL is installed, an ASCII text file called Messages.csv is copied to the Microsoft Dynamics SL program directory. This file contains all messages relating to the product, including all independently developed applications created with the Microsoft SL SDK. Each message has, among other things, a message number. A particular message can be displayed to the screen by merely passing its associated message number to the **Mess** statement.

The **Messf** statement should be used if the actual text of the message contains replaceable parameters.

The **MessResponse** function can be used to determine which button was chosen by the user to close the message box.

The standard Visual Basic MsgBox statement should not be used in applications developed with the Microsoft SL SDK in order to avoid conflicts with other Utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic MsgBox statement. The **MessBox** statement has been provided to facilitate similar functionality to the standard Visual Basic MsgBox statement with the exception that **MessBox** does not conflict with other Microsoft Dynamics SL utilities.

The **Mess** statement uses the following arguments:

Argument	Type	Description
<i>MsgNumber</i>	Integer	Number of the message from the message file that is to be displayed.

Each record (message) contained within the Messages.csv file contains the following fields separated by a comma:

Field name	Comments
Message Number	Message number which is required by most of the message API's such as Mess .
Category	0 for all messages. All independently developed applications must also use 0.
Language	0 for English.
Type	For future use. Currently set the field to 0.
Box Type	See detailed explanation of Box Type values below.
Record Type	S for messages created and maintained by Microsoft Dynamics SL developers. Independent developers should not use S for their new messages since they could be deleted or modified by Microsoft Dynamics SL developers.
Unattended Default Button	This value is used by Transaction Import to respond to messages displayed by the underlying application during the import operation.
Message Text	Actual text of the message. Replaceable parameters appear as "%s". For example: My first name is %s and my last name is %s.

The Box Type field within the Messages.csv file can have the following values:

Box type value	Icon	Buttons	Default button
0	None	OK	OK
16	Stop	OK	OK
48	Exclamation	OK	OK
64	Information	OK	OK
36	Question	Yes / No	Yes
292	Question	Yes / No	No
33	Question	OK / Cancel	OK

Example

The Payroll Employee Maintenance screen allows the user to enter the number of personal exemptions claimed by any particular employee on the Miscellaneous Information subscreen. Anytime this value is changed the user is prompted, via a message, as to whether or not the new total number of personal exemptions is to be utilized by each individual deduction for calculation purposes.

Message number 739 is the actual message displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: "Do you want to update the employee's deductions with the new exemption value?" This particular message also displays two buttons in the message box - a Yes button and a No button. The MessResponse function is subsequently called to determine which of these two buttons the user actually selected to close the message box.

```
Sub cDfltPersExmpt_Chk (chkstrg As String, retval As Integer)
    Dim MemArray_NbrRecs As Integer

    MemArray_NbrRecs = MArrayCnt(MemArray_EmpDeduction)

    'If the memory array has any records in it then prompt the user
    'whether or not he/she wants to update the number of PERSONAL
    'EXEMPTIONS on ALL existing employee deductions.

    If (MemArray_NbrRecs > 0) Then
        Call Mess( 739)
        If (MessResponse() = IDYES) Then
            Call MSet(F0225004.cNbrPersExmpt, chkstrg)
            Call MDisplay(MemArray_EmpDeduction)
        End If
    End If

End Sub
```

Example calls an existing message in the Messages table.

The message type is an OK Button.

```
' Warning - Qty in warehouse location will go negative
Call mess(578)
```

See Also

MessBox Statement, Messf Statement, MessResponse Function

MessBox Statement

Displays a message and waits for the user to choose a button.

Syntax

Call **MessBox**(*Msg, Type, Title*)

Remarks

The standard Visual Basic MsgBox statement should not be used in applications developed with the Microsoft SL SDK in order to avoid conflicts with other utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic MsgBox statement. The **MessBox** statement has been provided to facilitate similar functionality to the standard Visual Basic MsgBox statement with the exception that **MessBox** does not conflict with other Microsoft Dynamics SL utilities.

The **MessResponse** function can be used to determine which button was chosen by the user to close the message box.

The **MessBox** statement uses the following arguments:

Argument	Type	Description
<i>Msg</i>	String	Message text to be displayed.
<i>Type</i>	Integer	Numeric value controlling the icon style, buttons to be displayed as well as which button is the default button.
<i>Title</i>	String	Text to display in the title bar of the message dialog box.

Example

```
Dim FactAmt#
FactAmt = GetObjectValue("cannmemo2")

If FactAmt <> 0 Then
    Call MessBox("Display Excel During Execution?", MB_YESNO, " Message")
End If
```

See Also

Mess Statement, Messf Statement, MessResponse Function

Messf Statement

Formats a message from the message file with replaceable parameters and then displays it and waits for the user to choose a button.

Syntax

Call **Messf**(*MsgNumber*, *Parm1Str*, *Parm2Str*, *Parm3Str*, *Parm4Str*, *Parm5Str*, *Parm6Str*)

Remarks

When Microsoft Dynamics SL is installed, an ASCII text file called Messages.csv is copied to the Microsoft Dynamics SL program directory. This file contains all messages relating to the product, including all independently developed applications created with the Microsoft SL SDK. Each message has, among other things, a message number. The message can also contain up to six replaceable parameters by placing a %s at the appropriate point(s) within the actual message text. A particular message can be subsequently displayed to the screen by merely passing its associated message number to the **Messf** statement along with data values for each replaceable parameter.

The **Mess** statement should be used if the actual text of the message does not contain replaceable parameters.

The **MessResponse** function can be used to determine which button was chosen by the user to close the message box.

The standard Visual Basic MsgBox statement should not be used in applications developed with the Microsoft SL SDK in order to avoid conflicts with other Utilities such as Cut/Copy/Paste and Transaction Import. These utilities have built-in sophistication to respond to messages from the underlying application during the particular operation being performed such as paste or import. However, this automated functionality does not apply to messages displayed using the standard Visual Basic MsgBox statement. The **MessBox** statement has been provided to facilitate similar functionality to the standard Visual Basic MsgBox statement with the exception that **MessBox** does not conflict with other Microsoft Dynamics SL utilities.

The **Messf** statement uses the following arguments:

Argument	Type	Description
<i>MsgNumber</i>	Integer	Number of the message from the message file that is to be displayed.
<i>Parm1Str</i>	String	Data value for the first replaceable parameter.
<i>Parm2Str</i>	String	Data value for the second replaceable parameter. Blank if the message text contains only one replaceable parameter.
<i>Parm3Str</i>	String	Data value for the third replaceable parameter. Blank if the message text contains less than three replaceable parameters.
<i>Parm4Str</i>	String	Data value for the fourth replaceable parameter. Blank if the message text contains less than four replaceable parameters.
<i>Parm5Str</i>	String	Data value for the fifth replaceable parameter. Blank if the message text contains less than five replaceable parameters.
<i>Parm6Str</i>	String	Data value for the sixth replaceable parameter. Blank if the message text contains less than six replaceable parameters.

Each record (message) contained within the Messages.csv file contains the following fields separated by a comma:

Field Name	Comments
Message Number	Message number which is required by most of the message API's such as Messf .
Category	0 for all messages. All independently developed applications must also use 0.
Language	0 for English.
Type	For future use. Currently set the field to 0.
Box Type	See detailed explanation of Box Type values below.
Record Type	S for messages created and maintained by Microsoft Dynamics SL developers. Independent developers should not use S for their new messages since they could be deleted or modified by Microsoft Dynamics SL developers.
Unattended Default Button	This value is used by Transaction Import to respond to messages displayed by the underlying application during the import operation.
Message Text	Actual text of the message. Replaceable parameters appear as "%s". For example: My first name is %s and my last name is %s.

The Box Type field within the Messages.csv file can have the following values:

Box Type Value	Icon	Buttons	Default Button
0	None	OK	OK
16	Stop	OK	OK
48	Exclamation	OK	OK
64	Information	OK	OK
36	Question	Yes / No	Yes
292	Question	Yes / No	No
33	Question	OK / Cancel	OK

Example

The Payroll Manual Check screen uses the following code to warn the user of the fact that a Batch is out of balance.

Message number 818 is the actual message displayed and its associated text in the Microsoft Dynamics SL message file reads as follows: "Batch is out of balance by %s. Do you want to edit?" This particular message also displays two buttons in the message box – a Yes button and a No button. The MessResponse function is subsequently called to determine which of these two buttons the user actually selected to close the message box.

```
'Make sure that the batch itself is in balance with the documents.
  Batch_Out_Of_Bal_Amt = FPSub(bBatch.CtrlTot, bBatch.DrTot, MONEY)

  If (Batch_Out_Of_Bal_Amt <> 0#) Then

      Call Messf( 818, Str$(Batch_Out_Of_Bal_Amt), "", "", "", "", "")

      If (MessResponse() = IDYES) Then
          'User decided to edit the batch - so abort the Finish
          retval = ErrNoMess
          'Set focus on the Batch Control Total field
          Call ApplSetFocus(cCtrlTot)
      End If

  End If

End If
```

See Also

Mess Statement, MessageBox Statement, MessResponse Function

MessResponse Function

Returns the button chosen by the user to close the last message box displayed using the **Mess**, **Messf** or **MessageBox** statements.

Syntax

ButtonId = **MessResponse**()

Remarks

The **MessResponse** function returns one of the following symbolic constants declared in `Applic.DH`:

Return Value	Description
IDOK	OK button selected.
IDYES	Yes button selected.
IDNO	No button selected.
IDCANCEL	Cancel button selected.
IDABORT	Abort button selected.
IDRETRY	Retry button selected.
IDIGNORE	Ignore button selected.

Example

```

Dim Response%
Dim WSheet As Object

Call MessageBox("Display Excel During Execution?", MB_YESNO, " Message")
Response = MessResponse()

DoEvents

'Start Microsoft Excel and create the object.
Set WSheet = CreateObject("Excel.Sheet")
'Make Excel Visible for Demo purposes

If Response = IDYES Then
    WSheet.Application.Visible = True
End If

```

See Also

Mess Statement, Messf Statement

MExtend Function

Extend the grid of an application so that another table's structure can be added to the grid.

Syntax

```
RetVal = MExtend(MemHandle, bTable1, bTable1Length)
```

Remarks

Although this function is provided to extend the number of tables which are accessed by the grid control, the BSL application is responsible for all database I/O to the new table. This includes using memory array functions to cycle through the grid at save time to determine if the contents of the structure of the underlying table has changed and perform the appropriate action.

The **MExtend** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Non-zero if the operation is successful. Otherwise it is zero.
<i>MemHandle</i>	Integer	Unique handle to a previously opened memory array.
<i>bTable1</i>	User-defined datatype	Supplementary table structure to be added to the existing memory array.
<i>bTable1Length</i>	Integer	Size of supplementary table structure. For example, LenB(bTable1).

Example

Following example extends the grid for the chart of accounts screen. It adds the definition of a new table, Xaccount, to the account grid.

```
Global GridHandle As Integer

'Calls in Form1_Load event:
  Call SetAddr("bXAccount", bXAccount, nXAccount, LenB(bXAccount))
  Call SQLCursor(c1, NOLEVEL)

'Calls in Form1_Display
GridHandle = GetGridHandle("Spread1")
serr1 = MExtend(GridHandle, bXAccount, LenB(bXAccount))
```

See Also

GetGridHandle Function, **SetAddr** Statement

mFindControlName

Returns the first and subsequent control names in tab index order.

Syntax

CtlName = mFindControlName(*FirstFlag*)

The **mFindControlName** function uses the following arguments:

Argument	Type	Description
ControlName	String	The name property of the control.
FirstFlag	Integer	1 = Indicates finding the first control (the control whose Tab Index property is zero) 0 = Indicates finding the next control in Tab Index order.

Remarks

This function is useful when you want to write code that is “generic” but that needs to rely on specific control names. By writing generic versus screen-specific code, you can re-use it more often. This function would most often be used in cases where you wish to programmatically cycle through all controls within a screen and set specific properties based on certain characteristics (such as background color or tool tip text).

Note: On controls that are added by Customization Manager, the Tab Index property is set to 900. This value may optionally be changed and would be desirable to ensure expected tab order results. If, however, there are also inserted controls that have identical tab index orders. The **mFindControlName** function finds these controls in the order in which they were created.

Examples - mFindControlName

This example cycles through all controls to determine which ones have a PV property. If a control has a PV property, its **ToolTipText** property is set to indicate that F3 lookup is available:

```
Dim CtlName As String
Dim SResult As String
' Find the first control on the screen
CtlName = mFindControlName(1)
While (Trim$(CtlName) <> "")
    ' Get the PV property to determine F3 (inquiry)
    serr1 = GetProp(CtlName, "PV", SResult)
    If serr1 = 0 And Trim(SResult) <> "" Then
        ' A PV property exists
        serr1 = GetProp(CtlName, "ToolTipText", SResult)
        If serr1 = 0 And Trim(SResult) = "" Then
            ' Set the tooltip text property if it is not in use
            SResult = "Press F3 for list of Possible Values."
            Call SetProp(CtlName, "ToolTipText", SResult)
        End If
    End If
    ' Find the next control
    CtlName = mFindControlName(0)
Wend
```

The next example cycles through all controls to determine which ones have a **blankerr** property of True; these would be required fields. If a control has a **blankerr** property of True, its background color is set to blue to provide a visual indication that it is required:

```
Dim CtlName As String
Dim SResult As String
' Find the first control on the screen
CtlName = mFindControlName(1)
While (Trim$(CtlName) <> "")
    ' Get the 'blankerr' property to determine required controls.
    serr1 = GetProp(CtlName, "Blankerr", iresult)
    If serr1 = 0 And iresult = True Then
        ' Blankerr property found and control is required.
        serr1 = GetProp(CtlName, "BackColor", SResult)
        If serr1 = 0 Then
            lresult = &HFFFF80
            Call SetProp(CtlName, "BackColor", SResult)
        End If
    End If
    ' Find the next control
    CtlName = mFindControlName(0)
Wend
```

MFirst Function

Move to the first record in a designated memory array.

Syntax

RecFetch = **MFirst**(*MemHandle*, *RecMaintFlg*)

Remarks

MFirst moves to the first record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **SAFGrid** appearance with the memory array.

The **MFirst** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if no records exist in the specified memory array.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	Status of the memory array record (assuming it was successfully fetched). <i>Applic.DH</i> contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED

See Also

MLast Function, **MNext** Function, **MPrev** Function

MGetLineStatus Function

Returns the line status of the current record in the designated memory array.

Syntax

RecMaintFlg = **MGetLineStatus**(*MemHandle*)

Remarks

The **MGetLineStatus** function allows the application to retrieve the status of the current memory array record at any time.

The **MGetLineStatus** function uses the following arguments:

Argument	Type	Description
<i>RecMaintFlg</i>	Integer	Status of the current memory array record. <i>Applic.DH</i> contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED
<i>MemHandle</i>	Integer	Memory array resource handle.

See Also

MSetLineStatus Function

MGetRowNum Function

Returns the row / record number of the current record in the designated memory array.

Syntax

CurrRecNbr = **MGetRowNum**(*MemHandle*)

Remarks

The **MGetRowNum** function uses the following arguments:

Argument	Type	Description
<i>CurrRecNbr</i>	Integer	Row / record number of the current record.
<i>MemHandle</i>	Integer	Memory array resource handle.

See Also

MSetRowNum Statement

MInsert Statement

Insert a new record into a designated memory array.

Syntax

Call **MInsert**(*MemHandle*)

Remarks

MInsert is used to add a new record to a memory array. This is accomplished by copying the contents of all data structures previously associated with the designated memory array into a new memory array record. Data structures are associated with a memory array in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array. The new memory array record will have a line status of INSERTED.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **SAFGrid** appearance with the memory array.

The **MInsert** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle.

Example

This example illustrates how records can be inserted into a memory array under program control.

```
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), "", 0, "", 0, "", 0)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1( CSR_Account, SqlStr, bAccount, Len(bAccount))

'Read through all subsequent Account records, inserting each one into
'the memory array.
While( Account_Fetch = 0)

    'Insert current Account record into the memory array
    Call MInsert( Mem_Account)

    'Fetch the next Account record
    Account_Fetch = SFetch1( CSR_Account, bAccount, Len(bAccount))

Wend
```

See Also

MDelete Function, **MOpen** Functions, **MSetLineStatus** Function, **MUpdate** Statement

MKey Statement

Define a key field for a previously opened memory array.

Syntax

Call **MKey**(*MemHandle*, *KeySegmentNbr*, *TableDotFieldName*, *Ascending*)

Remarks

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an **SAFGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (notated by a “k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The **MKey** statement is the simplest and most common of all these methods to define a memory array key field. **MKey** is so simple because the system will automatically determine the requisite key field information for the *TableDotFieldName* based both on the **SetAddr** call for the relevant table and its corresponding data dictionary information in the database. The two restrictions of the **MKey** method are that it can only be used for fields whose table exists in the database (as opposed to a memory variable existing only within Visual Basic code) and a **SetAddr** call must have already been issued for the relevant table.

Multi-segment keys can be defined by successive calls to **MKey** with different *KeySegmentNbr* argument values.

The **MKey** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Unique handle to a previously opened memory array.
<i>KeySegmentNbr</i>	Control	Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.
<i>TableDotFieldName</i>	String	Name of the designated key field in a Table.FieldName format such as "Account.Acct".
<i>Ascending</i>	Integer	True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.

Example

This example illustrates how to open a memory array and define multiple key fields.

```
Dim Mem_ValEarnDed      As Integer

Call SetAddr(NOLEVEL, "bValEarnDed", bValEarnDed, nValEarnDed, Len(bValEarnDed))

Mem_ValEarnDed = MOpen(True, bValEarnDed, Len(bValEarnDed), "", 0, "", 0, 0,
    "", 0)

'Set up use of MKeyFind() for memory array
Call MKey(Mem_ValEarnDed, 0, "bValEarnDed.EarnTypeId", True)
Call MKey(Mem_ValEarnDed, 1, "bValEarnDed.DedId", True)
```

See Also

MKeyFind Function, **MKeyFld** Statement, **MKeyHctl** Statement, **MKeyOffset** Statement, **MOpen** Functions

MKeyFind Function

Find a specific record in a sorted memory array based on designated key field values.

Syntax

RecFetch = **MKeyFind**(*MemHandle*, *KeySeg1Val*, *KeySeg2Val*, *KeySeg3Val*, *KeySeg4Val*, *KeySeg5Val*)

Remarks

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an **SAFGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (i.e., notated by a "k" in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

If a record whose key fields exactly match the *KeySeg?Val* arguments does not exist, the system positions to the closest match. It will however still return a NOTFOUND to the application.

The **MKeyFind** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if an exact match cannot be located.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>KeySeg1Val</i>	Integer, Double or String	Desired value for the first key segment.
<i>KeySeg2Val</i>	Integer, Double or String	Desired value for the second key segment. PNULL if the memory array only has one key segment.
<i>KeySeg3Val</i>	Integer, Double or String	Desired value for the third key segment. PNULL if the memory array has less than three key segments.
<i>KeySeg4Val</i>	Integer, Double or String	Desired value for the fourth key segment. PNULL if the memory array has less than four key segments.
<i>KeySeg5Val</i>	Integer, Double or String	Desired value for the fifth key segment. PNULL if the memory array has less than five key segments.

Example

This example illustrates how to open a memory array and load the entire chart of accounts into the newly created array and then find a specific account record.

```
Dim Mem_Account      As Integer
Dim CSR_Account      As Integer
Dim SqlStr           As String
Dim Account_Fetch   As Integer

'Open memory array to hold Chart of Accounts
  Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), "", 0, "", 0,
    "", 0)

'Set up use of MKeyFind() for memory array
  Call MKey(Mem_Account, 0, "bAccount.Acct", True)

'Allocate cursor
  Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
  SqlStr = "Select * from Account order by Acct"
  Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount, Len(bAccount))

'Read through all subsequent Account records, inserting each one into
'the memory array.
  While( Account_Fetch = 0)

      'Insert current Account record into the memory array
        Call MInsert( Mem_Account)

      'Fetch the next Account record
        Account_Fetch = SFetch1(CSR_Account, bAccount, Len(bAccount))

  Wend

'Find the memory array record for a specific account
  Account_Fetch = MKeyFind( Mem_Account, "2020", "", "", "", "")
```

See Also

MKey Statement, MKeyFld Statement, MKeyHctl Statement, MKeyOffset Statement, MOpen Functions, MSetRowNum Statement

MKeyFld Statement

Define a key field for a previously opened memory array.

Syntax

Call **MKeyFld**(*MemHandle*, *KeySegmentNbr*, *TableDotFieldName*, *bTable*, *Ascending*)

Remarks

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an **SAFGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (notated by a "k" in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The **MKeyFld** method is similar to the **MKey** method except that it does not require a **SetAddr** call for the relevant table.

Multi-segment keys can be defined by successive calls to **MKeyFld** with different *KeySegmentNbr* argument values.

The **MKeyFld** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Unique handle to a previously opened memory array.
<i>KeySegmentNbr</i>	Integer	Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.
<i>TableDotFieldName</i>	String	Name of the designated key field in a Table.FieldName format such as "Account.Acct".
<i>bTable</i>	User-defined datatype	Memory array table structure containing the designated key field. This table structure must also have been previously passed to the MOpen call.
<i>Ascending</i>	Integer	True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.

Example

This example illustrates how to open a memory array and define multiple key fields.

```
Dim Mem_ValEarnDed As Integer

Mem_ValEarnDed = MOpen(True, bValEarnDed, Len(bValEarnDed), "", 0, "", 0, ↓
"", 0)

'Set up use of MKeyFind() for memory array
Call MKeyFld(Mem_ValEarnDed, 0, "bValEarnDed.EarnTypeId", bValEarnDed, ↓
True)
Call MKeyFld(Mem_ValEarnDed, 1, "bValEarnDed.DedId", bValEarnDed, True)
```

See Also

MKey Statement, **MKeyFind** Function, **MKeyHctl** Statement, **MKeyOffset** Statement, **MOpen** Functions,

MKeyHctl Statement

Define a key field for a previously opened memory array.

Syntax

Call **MKeyHctl**(*MemHandle*, *KeySegmentNbr*, *KeyFieldControl*, *Ascending*)

Remarks

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an **SAFGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (notated by a “k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The **MKeyHctl** method acquires information on the designated key field from a control having the key field itself in its **FieldName** property.

Multi-segment keys can be defined by successive calls to **MKeyHctl** with different *KeySegmentNbr* and *KeyFieldControl* argument values.

The **MKeyHctl** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Unique handle to a previously opened memory array.
<i>KeySegmentNbr</i>	Integer	Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.
<i>KeyFieldControl</i>	Control	Name of the control whose FieldName property refers to the designated key field in a Table.FieldName format such as “Account.Acct”.
<i>Ascending</i>	Integer	True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.

See Also

MKey Statement, **MKeyFind Function**, **MKeyFld Statement**, **MKeyOffset Statement**, **MOpen Functions**

MKeyOffset Statement

Define a key field for a previously opened memory array.

Syntax

Call **MKeyOffset**(*MemHandle*, *KeySegmentNbr*, *bTable*, *KeyFldByteOffset*, *KeyFldDataType*, *KeyFldDataLength*, *Ascending*)

Remarks

Occasionally a program will need the ability to easily locate a particular record within a memory array based on one or more key field values. The **MKeyFind** function can be used to accomplish this goal assuming the sort order for the memory array has been previously defined. Memory arrays associated with an **SAFGrid** control automatically have their sort order initialized by the **DetailSetup** function based on the key field control(s) contained within the grid (notated by a “k” in the levels property of the controls). All other memory arrays must have their sort order explicitly defined via one of several different methods. Each of the methods to define a key field, such as **MKey**, **MKeyFld**, **MKeyHctl** and **MKeyOffset**, vary primarily in the way they acquire detailed information on a key field such as datatype, size and byte offset within a user-defined datatype.

The **MKeyOffset** method is the most flexible method of defining memory array key fields but it is also the most detailed to code. It is designed to facilitate the definition of a key field that does not exist in the database and therefore has no correlated data dictionary information in the database. This situation can occur if one of the user-defined datatypes in a memory array is only declared within Visual Basic and does not exist within the database. In such a case, the system has no way of determining the byte offset from the beginning of the structure for any particular field, the field datatype nor the length of the field. The **MKeyOffset** statement allows the developer to explicitly pass all of this detailed information relating to the designated key field since it does not exist in the SQL data dictionary.

Multi-segment keys can be defined by successive calls to **MKeyOffset** with different *KeySegmentNbr* argument values.

The **MKeyOffset** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Unique handle to a previously opened memory array.
<i>KeySegmentNbr</i>	Integer	Memory array key segment whose key field is being defined. The first key segment number is always zero. Multi-segment keys must have contiguous key segment values such as 0 and 1 as opposed to 0 and 3. The maximum allowable number of key segments is five.
<i>bTable</i>	User-defined datatype	Memory array table structure containing the designated key field. This table structure must also have been previously passed to the MOpen call.
<i>KeyFldByteOffset</i>	Integer	This argument is designed to help the system locate the first byte of the designated key field. The system will already know the memory location of the first byte of the entire user-defined datatype via the <i>bTable</i> argument. The byte offset tells the system how far the first byte of the designated key field is offset from the first byte of the entire user-defined datatype. If the designated key field is the first field in the user-defined datatype then a value of zero should be passed.
<i>KeyFldDataType</i>	Integer	Specifies the datatype of the designated key field. The following datatype constants are declared in <i>Applic.DH</i> : String Float Integer Date Time

Argument	Type	Description
<i>KeyFldDataLength</i>	Integer	Size of the designated key field. For example, <code>LenB(bTable.KeyFld)</code> . Note: It is critical to use <code>LenB()</code> instead of <code>Len()</code> for <i>all</i> non-null "table length" parameters.
<i>Ascending</i>	Integer	True if the key segment should be sorted ascending. False to implement a descending sort sequence for the key segment currently being defined.

Example

The following example illustrates a memory array containing only selected fields from the Employee table that is nevertheless sorted by Employee ID. By only storing selected Employee fields in the memory array, much less memory will be consumed for each record within the memory array.

Since not all fields in the Employee database table are contained within the `Employee_SelFld` user-defined datatype, the data dictionary information in the SQL database corresponding to the standard Employee table is not usable by the system for purposes of determining the required key field information. Consequently, **MKeyOffset** must be utilized to implement sorting on the Employee ID key field.

Code to declare the user-defined datatype containing only selected fields from the Employee table. Notice that the Name field is being deliberately declared before the `EmpId` field so as to further illustrate the complete flexibility of **MKeyOffset**.

```
Type Employee_SelFld
    Name           As String * 30
    EmpId          As String * 10
End Type

Global bEmployee_SelFld As Employee_SelFld
```

Code to open a memory array for the `bEmployee_SelFld` user-defined datatype and define Employee ID as the key field.

```
Dim Mem_Employee_SelFld As Integer

Mem_Employee_SelFld = MOpen( TRUE, bEmployee_SelFld, ↵
    Len(bEmployee_SelFld), PNULL, 0, PNULL, 0, PNULL, 0)

Call MKeyOffset(Mem_Employee_SelFld, 0, bEmployee_SelFld, 30, ↵
    DATA_TYPE_STRING, LenB(bEmployee_SelFld.EmpId), True)
```

Code to load the memory array with relevant selected fields for all employees in the database. Notice that the order of the fields in the SQL Select statement correspond to the order of the fields in the `Employee_SelFld` user-defined datatype.

```
Dim CSR_Employee_SelFld As Integer
Dim SqlStr As String
Dim Employee_SelFld_Fetch As Integer

'Allocate a cursor
Call SqlCursor(CSR_Employee_SelFld, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch
'the first record
SqlStr = "Select Name, EmpId from Employee Order By EmpId"
Employee_SelFld_Fetch = SqlFetch1(CSR_Employee_SelFld, SqlStr, ↵
    bEmployee_SelFld, LenB(bEmployee_SelFld))

'Read through all subsequent Employee records, inserting each one into
```

```
'the memory array.
While(Employee_SelFld_Fetch = 0)
    'Insert current Employee record into the memory array
    Call MInsert( Mem_Employee_SelFld)
    'Fetch the next Employee record
    Employee_SelFld_Fetch = SFetch1(CSR_Employee_SelFld, ↓
    bEmployee_SelFld, LenB(bEmployee_SelFld))
Wend
```

See Also

MKey Statement, MKeyFind Function, MKeyFld Statement, MKeyHctl Statement, MOpen Functions

MLast Function

Move to the last record in a designated memory array.

Syntax

RecFetch = **MLast**(*MemHandle*, *RecMaintFlg*)

Remarks

MLast moves to the last record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be needed to properly synchronize the grid appearance with the memory array.

The **MLast** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if no records exist in the specified memory array.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	Status of the memory array record (assuming it was successfully fetched). Applic.DH contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED

See Also

MFirst Function, **MNext Function**, **MPrev Function**

MLoad Statement

Load a memory array with all records returned from the database by an SQL statement.

Syntax

Call **MLoad**(*MemHandle*, *Cursor*)

Remarks

There are two ways to load data directly from the database into a memory array. The most obvious method is to insert one record at a time into the memory array until no additional records are returned from the database. A simpler method is to load the entire array via a single call to the **MLoad** statement. The only requirement is that the *Cursor* passed as a parameter to the **MLoad** statement must already be initialized with an SQL Select statement or stored procedure so it is ready to begin returning data. The cursor can be initialized by passing the SQL Select statement or stored procedure, along with any necessary parameters, to the **Sql** statement.

The **MLoad** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>Cursor</i>	Integer	SQL database cursor. This cursor must have already been initialized with an SQL Select statement or stored procedure.

Example

The following example illustrates a memory array containing only selected fields from the Employee table that is nevertheless sorted by Employee ID. By only storing selected Employee fields in the memory array, much less memory is consumed for each record within the memory array.

Since not all fields in the Employee database table are contained within the Employee_SelFld user-defined datatype, the data dictionary information in the SQL database corresponding to the standard Employee table is not usable by the system for purposes of determining the required key field information. Consequently, **MKeyOffset** must be utilized to implement sorting on the Employee ID key field.

Code to declare the user-defined datatype containing only selected fields from the Employee table. Notice that the Name field is being deliberately declared before the EmpId field so as to further illustrate the complete flexibility of **MKeyOffset**.

```
Type Employee_SelFld
    Name           As String * 30
    EmpId  As String * 10
End Type

Global bEmployee_SelFld           As Employee_SelFld
```

Code to open a memory array for the bEmployee_SelFld user-defined datatype and define Employee ID as the key field.

```
Dim Mem_Employee_SelFld           As Integer

Mem_Employee_SelFld = MOpen( TRUE, bEmployee_SelFld, Len(bEmployee_SelFld), ↓
    "", 0, "", 0, "", 0)

Call MKeyOffset(Mem_Employee_SelFld, 0, bEmployee_SelFld, 30, ↓
    DATA_TYPE_STRING, Len(bEmployee_SelFld.EmpId), True)
```

Code to load the memory array with relevant selected fields for all employees in the database. Notice that the order of the fields in the SQL Select statement correspond to the order of the fields in the Employee_SelFld user-defined datatype.

```
Dim CSR_Employee_SelFld           As Integer
Dim SqlStr                       As String
Dim Employee_SelFld_Fetch As Integer

'Allocate a cursor
  Call SqlCursor(CSR_Employee_SelFld, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch
'the first record
  SqlStr = "Select Name, EmpId from Employee Order By EmpId"
  Employee_SelFld_Fetch = SqlFetch1(CSR_Employee_SelFld, SqlStr,
    bEmployee_SelFld, Len(bEmployee_SelFld))

'Read through all subsequent Employee records, inserting each one into the
'memory array.
  While(Employee_SelFld_Fetch = 0)

    'Insert current Employee record into the memory array
      Call MInsert( Mem_Employee_SelFld)

    'Fetch the next Employee record
      Employee_SelFld_Fetch = SFetch1(CSR_Employee_SelFld,
        bEmployee_SelFld, Len(bEmployee_SelFld))

  Wend
```

See Also**Sql Statement**

MNext Function

Move to the next record in a designated memory array.

Syntax

RecFetch = **MNext**(*MemHandle*, *RecMaintFlg*)

Remarks

MNext moves to the next record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **SAFGrid** appearance with the memory array.

The **MNext** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if the current record is already the last record in the specified memory array as well as when no records exist.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	Status of the memory array record (assuming it was successfully fetched). <i>Applic.DH</i> contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED

See Also

MFirst Function, **MLast** Function, **MPrev** Function

MOpen Functions

Open a new memory array and return a corresponding unique memory array number.

Syntax

```
MemHandle = MOpen(DelRetToSystem, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length)
```

```
MemHandle = MOpen8(DelRetToSystem, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length, bTable5, bTable5Length, bTable6, bTable6Length, bTable7,
bTable7Length, bTable8, bTable8Length)
```

Remarks

A memory array must be opened before insert, update, delete or memory array navigation operations can be performed on it. The memory array handle returned by **MOpen** and **MOpen8** is used when performing these types of operations on memory arrays.

MOpen allocates memory for up to four table structures whereas **MOpen8** can handle up to eight different table structures for each memory array record.

MOpen is only used to open memory arrays which are not associated with an **SAFGrid** control. Memory arrays which correspond to grids are opened automatically by either the **DetailSetup** or **DetailSetup8** function.

The **MOpen** function uses the following arguments (**MOpen8** has eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0)

Argument	Type	Description
<i>MemHandle</i>	Integer	Unique handle to the newly created memory array. If a new memory array was successfully opened this value will be >= 0.
<i>DelRetToSystem</i>	Control	Controls the handling of system memory with respect to deleted memory array lines. True causes memory consumed by deleted memory array records to be returned to system memory. False causes the memory to be retained. Normally True should be used.
<i>bTable1</i>	User-defined datatype	First table structure of memory array.
<i>bTable1Length</i>	Integer	Size of first table structure. For example, LenB(<i>bTable1</i>). Note: It is critical to use LenB() instead of Len() for <i>all</i> non-null "table length" parameters.
<i>bTable2</i>	User-defined datatype	Second table structure of memory array. PNULL if the memory array only contains one table structure.
<i>bTable2Length</i>	Integer	Size of second table structure. Zero if the memory array only contains one table structure.
<i>bTable3</i>	User-defined datatype	Third table structure of memory array. PNULL if the memory array contains less than three table structures.
<i>bTable3Length</i>	Integer	Size of third table structure. Zero if the memory array contains less than three table structures.
<i>bTable4</i>	User-defined datatype	Fourth table structure of memory array. PNULL if the memory array contains less than four table structures.
<i>bTable4Length</i>	Integer	Size of fourth table structure. Zero if the memory array contains less than four table structures.

The table structures passed to **MOpen** do not have to be actual database tables. They can be either a simple data item, such as a double, or any user-defined data type created with the Visual Basic Type statement.

Example

This example illustrates how to open a memory array and load the entire chart of accounts into the newly created array.

```

Dim Mem_Account    As Integer
Dim CSR_Account    As Integer
Dim SqlStr         As String
Dim Account_Fetch As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), PNULL, 0, PNULL,
0, PNULL, 0)

'Allocate cursor
    Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL stored procedure and immediately fetch
'first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1(CSR_Account, SqlStr, bAccount, LenB(bAccount))

'Read through all subsequent Account records, inserting each one
'into the memory array.
    While( Account_Fetch = 0)

        'Insert current Account record into the memory array
            Call MInsert( Mem_Account)

        'Fetch the next Account record
            Account_Fetch = SFetch1(CSR_Account, bAccount, LenB(bAccount))

    Wend

```

See Also

MClear Statement, MClose Statement, MDelete Function, MDisplay Statement, MFirst Function, MInsert Statement, MKey Statement, MKeyFind Function, MLast Function, MNext Function, MPrev Function, MUpdate Statement

MPrev Function

Move to the previous record in a designated memory array.

Syntax

RecFetch = **MPrev**(*MemHandle*, *RecMaintFlg*)

Remarks

MPrev moves to the previous record of a specified memory array and copies the contents of the array record into the data structure(s) previously specified in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array.

When this call is used on a memory array associated with an **SAFGrid** control (memory arrays opened automatically via the **DetailSetup** function), an **MDisplay** call will be necessitated to properly synchronize the **SAFGrid** appearance with the memory array.

The **MPrev** function uses the following arguments:

Argument	Type	Description
<i>RecFetch</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if the current record is already the first record in the specified memory array as well as when no records exist.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	Status of the memory array record (assuming it was successfully fetched). <i>Applic.DH</i> contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED

See Also

MFirst Function, **MLast Function**, **MNext Function**

MRowCnt Function

Returns the number of records in a designated memory array.

Syntax

NumRecs = **MRowCnt**(MemHandle)

Remarks

The **MRowCnt** function uses the following arguments:

Argument	Type	Description
<i>NumRecs</i>	Integer	Number of records in the designated memory array.
<i>MemHandle</i>	Integer	Memory array resource handle.

Example

```
Global ArrayHandle%, Rows%
ArrayHandle = GetGridHandle("Spread1")
Rows = MRowCnt(ArrayHandle)
```

See Also

MOpen Functions

MSet Statement

Explicitly set the value of a particular control for every record in its corresponding **SAFGrid** control.

Syntax

Call **MSet**(*Control*, *NewDataValue*)

Remarks

The **MSet** statement uses the following arguments:

Argument	Type	Description
<i>Control</i>	Control	Control which is bound to the Record.FieldName whose value is to be changed for every record in the relevant SAFGrid . Note: This statement is only for use with controls associated with an SAFGrid control.
<i>NewDataValue</i>	String	New data value for the Record.FieldName associated with the designated <i>Control</i> . The data value must be in a string format.

See Also

MSetProp Statement

MSetLineStatus Function

Set the line status of the current record in the designated memory array.

Syntax

```
RetVal = MSetLineStatus(MemHandle, NewLineStatus)
```

Remarks

Each record within a memory array has its own line status such as INSERTED, UPDATED or NOTCHANGED. The system automatically modifies the line status based on the type of activity last performed on any particular memory array record. For example if a record is inserted into a memory array using **MInsert**, then the status of that new memory array record will automatically be set to INSERTED. Occasionally, however, the application may need to assign a specific line status to a particular memory array record. In such cases the **MSetLineStatus** function can be used to carry out this task.

One common usage of this function is when a memory array associated with an **SAFGrid** control is being loaded from the database under application control. In these cases records are being inserted into the memory array via successive **MInsert** calls. However since the data is in no way modified between the time it is fetched and the time it is inserted into the memory array the line status of the resultant memory array record is forced to change from INSERTED to NOTCHANGED.

The **MSetLineStatus** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	-1 if an invalid memory array handle is passed.
<i>MemHandle</i>	Integer	Memory array resource handle.
<i>RecMaintFlg</i>	Integer	New status of the current memory array record. Applic.DH contains the following symbolic constants defining possible memory array record status values: INSERTED, UPDATED and NOTCHANGED

Example

This example illustrates how to load a memory array from the database and at the same time force the line status of all newly inserted memory array records to be NOTCHANGED.

```
Dim Mem_Account As Integer
Dim CSR_Account As Integer
Dim SqlStr As String
Dim Account_Fetch As Integer
Dim Retval As Integer

'Open memory array to hold Chart of Accounts
Mem_Account = MOpen( TRUE, bAccount, Len(bAccount), "", 0, "", 0, "", 0)

'Allocate cursor
Call SqlCursor( CSR_Account, NOLEVEL)

'Initialize cursor with a SQL statement and immediately fetch first record
SqlStr = "Select * from Account order by Acct"
Account_Fetch = SqlFetch1( CSR_Account, SqlStr, bAccount, Len(bAccount))

'Read through all subsequent Account records, inserting each one into
'the memory array.
While( Account_Fetch = 0)

    'Insert current Account record into the memory array
    Call MInsert( Mem_Account)

    'Since the record ALREADY exists in the database, reset the line
    'status of the current memory array record so that the application
```

```
'will be able to detect whether or not any calls to MUpdate were  
'subsequently made for the current record.  
    Retval = MSetLineStatus(Mem_Account, NOTCHANGED)  
  
'Fetch the next Account record  
    Account_Fetch = SFetch1(CSR_Account, bAccount, Len(bAccount))  
  
Wend
```

See Also**MGetLineStatus Function, MInsert Statement, MUpdate Statement**

MSetProp Statement

Set the value of a particular property for both the designated form-view control as well as its associated **SAFGrid** control.

Syntax

Call **MSetProp**(*Control*, *PropertyName*, *NewPropValue*)

Remarks

At runtime, each column of an **SAFGrid** control is associated with an underlying (form view) Microsoft SL SDK custom control. The **MSetProp** statement can be used to modify a property setting for both the underlying control as well as the relevant column in the associated grid. For example, an entire column can be disabled using the **MSetProp** statement.

If the application wants to modify property values on a line by line basis, as opposed to an entire column, then it will need to manage the property settings using calls to the **SetProps** statement from within the **LineGotFocus** event of each individual detail line.

The **MSetProp** statement uses the following arguments:

Argument	Type	Description
<i>Control</i>	Control	Control whose designated property value is to be modified both in form-view as well as grid-view.
<i>PropertyName</i>	String	Name of the property whose value is to be modified.
<i>NewPropValue</i>	String or Integer	New property value. The actual datatype varies based on the property being referenced.

Example

```
'Disable the ExtRefNbr Column
Call MsetProp("cextrefnbr", PROP_ENABLED, False)

'Enable the ExtRefNbr Column
Call MsetProp("cextrefnbr", PROP_ENABLED, True)
```

See Also

SetProp Statement

MSetRowNum Statement

Set the current row / record number of a designated memory array.

Syntax

Call **MSetRowNum**(MemHandle, NewCurrRecNbr)

Remarks

An application can jump to a specific memory array record via either the **MKeyFind** function or the **MSetRowNum** statement. **MSetRow** jumps to a specific record number whereas **MKeyFind** locates the record with designated key field values.

The **MSetRowNum** statement uses the following arguments:

Argument	Type	Description
MemHandle	Integer	Memory array resource handle.
NewCurrRecNbr	Integer	Row / record number of the desired memory array record.

See Also

MGetRowNum Function, **MKeyFind** Function

MUpdate Statement

Update the current memory array record of a designated memory array with new data values.

Syntax

Call **MUpdate**(*MemHandle*)

Remarks

MUpdate is used to update an existing record of a specified memory array. This is accomplished by copying the new contents of all data structures previously associated with the designated memory array over the existing memory array record. Data structures are associated with a memory array in either the **MOpen** or **DetailSetup** function call used to originally open the relevant memory array. If the memory array record had a line status of **INSERTED** prior to the **MUpdate** call then the line status will remain **INSERTED**. In all other cases, the memory array record will have a line status of **UPDATED**.

The **MUpdate** statement uses the following arguments:

Argument	Type	Description
<i>MemHandle</i>	Integer	Memory array resource handle.

See Also

MDelete Function, **MInsert** Statement, **MOpen** Functions, **MSetLineStatus** Function

NameAltDisplay Function

Displays a string field with swap character suppressed.

Syntax

```
StrVar = NameAltDisplay ( FieldName)
```

Remarks

To accommodate alternate sorting by name, Microsoft Dynamics SL stores most address and name fields based on where the "@" is placed in the name. For example, in order for "The Jones Company" to sort with the Js, this name must be entered as The @Jones Company. Microsoft Dynamics SL then stores this entry as Jones Company~The. If you wish to subsequently retrieve and/or manipulate this field in BSL code, this function "flips" the name and suppress the ~ and @. In this example, Jones Company~The as stored in the database is displayed as The Jones Company.

The **NameAltDisplay** function uses the following arguments:

Argument	Type	Description
<i>StrVar</i>	String	Any string variable in which to store the swapped version of the field.
<i>FieldName</i>	String	Object or Field name.

Example

```
Dim SwapName$, OrigName$

OrigName = GetObjectValue("cname")
Call MsgBox(OrigName, MB_OK, "Orig Name")

SwapName = NameAltDisplay(OrigName)
Call MsgBox(SwapName, MB_OK, "Swap Name")
```

See Also

GetObjectValue Function, SetObjectValue Function

PasteTemplate Function

Paste information from the designated template into the current application.

Syntax

RetVal = **PasteTemplate**(*TemplateID*)

Remarks

The Template feature makes it possible to store data from the current screen and subsequently paste that data into the same screen at a later time. Templates can be saved to the database programmatically using the **SaveTemplate** statement as well as via the Template menu item on the Edit menu. Once a template has been created it can be pasted into its source application under program control using the **PasteTemplate** function as well as via the Template menu item on the Edit menu.

The **PasteTemplate** statement uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Zero if no errors occurred. The NOTFOUND symbolic constant, declared in <i>Applic.DH</i> , is returned if the <i>TemplateID</i> does not exist.
<i>TemplateID</i>	String	ID of the template whose information is to be pasted into the current Microsoft SL SDK application screen.

See Also

SaveTemplate Statement

PeriodCheck Function

Verify whether or not a period string in YYYYPP format represents a valid fiscal period.

Syntax

RetVal = **PeriodCheck**(*PeriodString*)

Remarks

The **PeriodCheck** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	Value of the period number portion of <i>PeriodString</i> if the string represents a valid fiscal period. Otherwise a value of -1 will be returned if the string is invalid. A period string is invalid if the period portion is less than one or greater than the number of valid fiscal periods defined in the GLSetup record.
<i>PeriodString</i>	String	Period string to be verified. Must be in YYYYPP format.

Example

```
'Example placed in Chk event of a String user field
'The mask property on this field is Custom mask Type, 99-9999

Dim Year$, Period$

Year = Right$(chkstrg, 4)
Period = Left$(chkstrg, 2)
serr = PeriodCheck(Year + Period)
```

PeriodMinusPeriod Function

Return the number of periods between two fiscal periods.

Syntax

NbrPeriods = **PeriodMinusPeriod**(*PerNbr1*, *PerNbr2*)

Remarks

The **PeriodMinusPeriod** function uses the following arguments:

Argument	Type	Description
<i>NbrPeriods</i>	Integer	Number of periods between <i>PerNbr1</i> and <i>PerNbr2</i> . If <i>PerNbr1</i> > <i>PerNbr2</i> then the number of fiscal periods between the two periods will be a negative value.
<i>PerNbr1</i>	String	Beginning fiscal period. Must be in YYYYPP format.
<i>PerNbr2</i>	String	Ending fiscal period. Must be in YYYYPP format.

Note: This function will use the number of periods in a fiscal year, as specified in the GLSetup record, in order to derive an accurate result.

Example

```
Dim Period1$, Period2$
Dim NumPers%

Period1 = GetObjectValue("cperpost")
Period2 = "199312"
NumPers = PeriodMinusPeriod(Period1, Period2)

If NumPers < 0 Then
    Call MsgBox("Cannot post back further than 12-93", MB_OK, _
"Microsoft Dynamics SL Message")
End If
```

See Also

PeriodPlusPerNum Function

PeriodPlusPerNum Function

Add a designated number of periods to an existing fiscal period.

Syntax

ResultingPerNbr = **PeriodPlusPerNum**(*CurrPerNbr*, *NbrPeriodsToAdd*)

Remarks

The **PeriodPlusPerNum** function uses the following arguments:

Argument	Type	Description
<i>ResultingPerNbr</i>	String	Result of <i>CurrPerNbr</i> + <i>NbrPeriodsToAdd</i> .
<i>CurrPerNbr</i>	String	Starting fiscal period. Must be in YYYYPP format.
<i>NbrPeriodsToAdd</i>	Integer	Number of fiscal periods to add to <i>CurrPerNbr</i> . Negative values are supported.

Note: This function will use the number of periods in a fiscal year, as specified in the GLSetup record, in order to derive an accurate result.

Example

```
'Increment PerPost to the next fiscal period
Dim NewPeriod$, PeriodToPost$, NumberOfPeriods%

NumberOfPeriods = 1
PeriodToPost = GetObjectValue("cperpost")

NewPeriod = PeriodPlusPerNum(PeriodToPost, NumberOfPeriods)
```

See Also

PeriodMinusPeriod Function

SaveTemplate Statement

Save information from the current application to a designated template.

Syntax

Call **SaveTemplate**(*TemplateID*, *Description*, *AppliesToUserID*, *IncludeLowerLevels*, *StartingLevelNbr*)

Remarks

The Template feature makes it possible to store data from the current screen and subsequently paste that data into the same screen at a later time. These timesaving templates can be saved to the database programmatically using the **SaveTemplate** statement as well as via the Template menu item on the Edit menu. Each template can contain complete transactions and entities or individual fields selected by the user. Relative date and period features allow a template to paste data relative to the current date and fiscal period. Templates can be designated as private to a specific user or marked public for availability to all users. Templates are stored in the system database and therefore they are independent of any particular application database.

Unless otherwise specified, all date and period values pasted from a template will be equal to the business date, located on the File menu, and the current period for the module. To override this default action, the user entering data for the template must specify a new relative date or period value for each desired field. This is done immediately before saving a template. Specifying a relative date or period value for a field contained in grid detail lines will change the template value of that field for all detail lines. Relative values can be defined by selecting the pertinent date or period field and pressing F2 to start the Relative Date or Relative Period screen — whichever is appropriate.

The **SaveTemplate** statement uses the following arguments:

Argument	Type	Description
<i>TemplateID</i>	String	ID of the template being created or updated. If a template with the designated <i>TemplateID</i> already exists, it will be overwritten. The <i>TemplateID</i> can be up to 30 characters.
<i>Description</i>	String	Description of the template.
<i>AppliesToUserID</i>	String	Microsoft Dynamics SL user ID to which the template applies. By default, the template will be Public if <i>AppliesToUserID</i> is left blank.
<i>IncludeLowerLevels</i>	Integer	False if only data for the <i>StartingLevelNbr</i> is to be saved to the template. Otherwise, data for lower levels, in addition to <i>StartingLevelNbr</i> , will also be saved to the template.
<i>StartingLevelNbr</i>	Integer	Number of the first application level for which data is to be saved to the template. For example on Batch / Document / Detail screens, a value of zero could be passed to start the save with the Batch level. <i>Applic.DH</i> contains two symbolic constants which can also be passed: <i>CcpSelectedFields</i> — Only those fields currently highlighted by the user will be saved to the template. <i>CcpAllLevels</i> — Data from all application levels will be saved to the template regardless of the <i>IncludeLowerLevels</i> argument value.

See Also

PasteTemplate Function

SDelete Statement

Delete the current record from a designated table within an existing SQL view.

Syntax

Call **SDelete**(*Cursor*, *TablesDeletingFrom*)

Remarks

A value of “*.*” can be passed as a table name, meaning that all current records in the existing view will be deleted. Please note that this call requires that the current record already be fetched via such functions as **SqlFetch1** or **SFetch1** so that the designated cursor actually has a current record.

The **SDelete** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor.
<i>TablesDeletingFrom</i>	String	Name of each table, in the specified cursor's view, from which the current record is to be deleted. Multiple table names should be separated by commas.

Example

```
'Example is deleting in the Delete event

'Note that the following fetch been has issued in
'a preceding event prior to the delete call
serr1 = SqlFetch1(c1, "XCustInfo_All" +
sparm(chkstrg), bXCustAddlInfo, Len(bXCustAddlInfo))

Sub Delete(level%, retval%)
  If Level = 0 Then
    serr1 = sdelete(c1, "XCustAddlInfo")
  End If
End Sub
```

See Also

SDeleteAll Function, **SFetch** Functions, **SqlFetch** Functions

SDeleteAll Function

Delete all records from the designated table(s) within a predefined view.

Syntax

RetVal = **SDeleteAll**(*Cursor*, *TablesDeletingFrom*)

Remarks

Deletes records from some or all of the tables in a view based on the current restrictions. A view must have already been initialized for the specified cursor via functions such as **Sql**, **SFetch1** or **SqlFetch4**. If no restriction is specified when the cursor is initialized, then this call will remove all records from the designated table(s) in the view.

The **SDeleteAll** function uses the following arguments:

Argument	Type	Description
<i>RetVal</i>	Integer	
<i>Cursor</i>	Integer	SQL database cursor.
<i>TablesDeletingFrom</i>	String	Name of each table, in the specified cursor's view, from which all records within the current view are to be deleted. Multiple table names should be separated by commas. A value of "*" can be used to delete records from all tables within the view.

Example

Delete all vendors having a zero balance.

```
Dim SqlStr As String

'Initialize the cursor with a SQL statement
SqlStr = "Select * from Vendor where CurrBal = 0 and FutureBal = 0"
Call Sql(CSR_Vendor_Del, SqlStr)

'Delete all records matching the restriction clause of the SQL statement
'used to initialize the CSR_Vendor_Del cursor.
Call SDeleteAll(CSR_Vendor_Del, "*.*")
```

See Also

SDelete Statement, SFetch Functions, Sql Statement, SqlFetch Functions

SetAddr Statement

Associate a table name together with a Visual Basic variable and an optional screen level number.

Syntax

Call **SetAddr**(*LevelNbr*, *TableNameStr*, *bTableName*, *nTableName*, *bTableNameLength*)

Remarks

The **SetAddr** statement facilitates proper runtime binding between a Visual Basic data variable and relevant data entry controls - including controls created using the Customization Manager.

Although SWIM has access to extensive information about any particular data entry control, one vital piece of information cannot be determined merely by looking at the control itself. In particular, SWIM needs to know where the data for the control is stored in memory. Since the data is actually stored in an underlying Visual Basic variable, SWIM has no means of directly determining where that Visual Basic variable is stored in memory. The **SetAddr** statement is used by the application to resolve this problem.

To facilitate the explanation of the linkage between the **SetAddr** statement and corresponding data entry controls, consider the following user-defined datatype and a corresponding **SetAddr** call for the Account database table:

```
Type Account

    Acct                As String * 10
    Active              As Integer
    ConsolAcct         As String * 10
    CuryId              As String * 4
    Descr               As String * 30
    NoteID              As Long
    RatioGrp           As String * 2
    SummPost           As String * 1
    Type                As String * 2
    User1               As String * 30
    User2               As String * 30
    User3               As Double
    User4               As Double

End Type

Global bVBVarAccount As Account, nVBVarAccount As Account

Call SetAddr( "bAccount", bVBVarAccount, nVBVarAccount, ,
Len(bVBVarAccount))
```

The **SetAddr** call itself associates the value of the *TableNameStr* argument together with the memory location of the first byte of the Visual Basic variable passed via the *bTableName* argument. If the relevant table is a database table, SWIM can access detailed information relating to each individual field contained therein using the SQL data dictionary. For example, SWIM can determine the name, data type, maximum size as well as the relative placement (byte offset) of each individual field within the table. After the **SetAddr** call in the above example, SWIM would know that the first byte of *bVBVarAccount* is at a particular location in memory, hereafter referred to as location M, and furthermore that the first byte of *bVBVarAccount.Acct* is offset zero bytes from location M as well as the fact that *Acct* is ten bytes long. Similarly, it would also know that the first byte of *bVBVarAccount.Active* is offset ten bytes from location M, and is two bytes long since it is an integer. Since the value of "bAccount" is passed as the *TableNameStr* argument it is the string literal associated with memory location M. Anytime SWIM encounters "bAccount.<SomeFieldName>" in a *FieldName* property it will have all of this detailed information readily available so it can access the

corresponding data at the appropriate memory location. The same concept applies when SWIM encounters “bAccount.<SomeFieldName>” in any of the **DBNav**, **Default**, **PV** or **Trigger** properties.

As previously mentioned, the detailed information acquired by SWIM as a result of the **SetAddr** call can be directly linked to the **FieldName** property of data entry controls. The **FieldName** property contains a **Struct.FieldName** value along with other more detailed information such as field offset value, declare type and length. Once they have been fully initialized, these values facilitate the linkage between the control and the associated memory location where the data value of the control is stored. In the vast majority of cases the detailed field information is initialized automatically by SWIM in using information acquired via a corresponding **SetAddr** call and the SQL data dictionary. Usage of an unbound control is the only case where the developer must fill in the detailed field information manually since it will not exist in the SQL data dictionary.

The **Struct.FieldName** portion of the **FieldName** property must always be populated with a string value in the “bTableName.FieldName” format. Using the above example, a control for the Acct field would have a value of “bAccount.Acct” for the **Struct.FieldName** portion of its **FieldName** property. Similarly the Active control would have a value of “bAccount.Active” for the **Struct.FieldName** portion of its **FieldName** property.

The first portion of this **Struct.FieldName** string, “bAccount” in our example, is used to link the control with a particular **SetAddr** call that had the same value for its **TableNameStr** argument. Once that initial linkage is made the exact memory location can be determined automatically by correlating the last portion of the **Struct.FieldName** string, “Acct” or “Active” in our example, with the detailed field information acquired as a result of the relevant **SetAddr** call.

In general, the **SetAddr** call facilitates the linkage of a “bTableName.FieldName” type of string literal with a precise location in memory for the corresponding data value.

The **SetAddr** statement uses the following arguments:

Argument	Type	Description
<i>TableNameStr</i>	String	Table name string literal which can subsequently be used by SWIM to link the table name portion of a “bTableName.FieldName” type of string literal with a precise location in memory. By convention this value should begin with a “b” such as “bAccount” for the Account table. NOTE: This value does not have to correlate to a database table. In these cases, the system will assume it refers to an unbound data buffer. All references to unbound “table names” in DBNav , Default , FieldName and PV properties must be accompanied with manually entered detailed field information since the system will be unable to access this information from the SQL data dictionary.
<i>bTableName</i>	User-defined datatype	Visual Basic variable whose datatype corresponds to the table referred to in the <i>TableNameStr</i> argument. For example if “bAccount” is passed then the Visual Basic variable passed via this argument must be a user-defined datatype whose structure precisely corresponds to the Account table in the database.
<i>nTableName</i>	User-defined datatype	Visual Basic variable whose datatype corresponds to the table referred to in the <i>TableNameStr</i> argument. If <i>TableNameStr</i> does not correlate to a database table then PNULL must be passed as the value for this argument. This value will be properly initialized with null values as long as the relevant table is a database table. Any structure of the relevant datatype can then be easily blanked out using the <i>bTableName</i> = <i>nTableName</i> methodology.
<i>bTableNameLength</i>	Integer	Length (size) of the entire <i>bTableName</i> user-defined type such as <code>Len(bTableName)</code> . Note: In Customization code, it is critical to use <code>Len()</code> instead of <code>LenB()</code> .

Example

External .dh file

In the following example, we create an external file named Country.dh. A structure that matches the Country table is declared. Note that the exact size of the fields must be declared. The buffer and NULL buffer structures are declared globally.

```
Type Country
  CountryId      As String * 3
  Descr          As String * 30
End Type
Global bCountry As Country, nCountry As Country
```

The Country.dh file is then used in Customization events:

```
(general) declarations
'$include: "cu\country.dh"

Sub Form1_Load
  Call Setaddr("bcountry", bcountry, ncountry, Len(bcountry))
  Call Sqlcursor(c1, NOLEVEL)
End Sub
```

SetBufferValue Statement

Sets an underlying application's data buffer field to a specified value.

Syntax

Call **SetBufferValue**(*bTable.FieldName*, *Str*)

Remarks

If a BSL application issues its own **SetAddr** calls it can then reference any of these structures from within code. However, if the underlying application's structures need to be referenced and these fields are not represented as objects on the form, this statement allows the BSL application to obtain these values. If the fields were objects on the form, the BSL application can simply issue **SetObjectValue** instead.

Be aware that usage of this call at times could compromise the underlying application functionality. Especially if a dependent field is changed. Likewise, the underlying application may change your value if you have set it too early.

The **SetBufferValue** statement uses the following arguments:

Argument	Type	Description
<i>bTable.FieldName</i>	String	SQL Table.FieldName that you wish to set.
<i>Str</i>	String	String or string variable that contains the value that your changing the specific bTable.FieldName to.

Example

```
Call SetBufferValue("bGLTran.RefNbr","000099")
```

See Also

[GetBufferValue Statement](#)

SetDefaults Statement

Set one or more controls to their default value using either their Default Property or Default Event code.

Syntax

```
RetVal = SetDefaults (FormObjectName, FieldObjectName)
```

Remarks

Each data object has both a Default Property as well as a Default Event. Any particular data object can use one of these two methods to define a default data value for itself. The system uses these methods any time a particular data object is to be initialized to its default value. An exhaustive discussion of all the times when this occurs is beyond the scope of the **SetDefaults** statement. However, one such time an object is set to its default value is when the application explicitly directs the system to do so via usage of the **SetDefaults** statement in reference to the relevant object.

The **SetDefaults** statement can be used to default all objects in a subform. The `Level_SetDefaults` statement is functionally equivalent except it can be used to explicitly default all objects having a particular level number.

Since **SetDefaults** implies a change in the data value of the designated object, the system “marks” the object as requiring error checking. The system does not, however, immediately perform the requisite error checking (for example, it does not immediately fire the Chk event). The error checking is nevertheless guaranteed to occur prior to any updates to the database.

Note that when an application needs to null out a particular field, perhaps because the field is no longer applicable, it should explicitly do so programmatically and then redisplay the relevant object using the `DispFields` statement. After the object has been redisplayed, it can then be disabled using the `SetProps` statement. The **SetDefaults** statement should not be used in these cases even if the relevant control has no Default Property value and no code within Default Event. A developer may well wonder why this caution would be expressed since the field is, in fact, nulled out when no Default Property or Event code exists and, therefore, the application appears to work properly during testing. The following code conceptually illustrates how to properly null out and disable a control which is no longer applicable:

```
Record.Field = NULL (0 for numeric datatypes , "" for string datatype)
Call DispFields("Form1", "cField")
Call SetProps("cField", PROP_ENABLED, False)
```

The **SetDefaults** function uses the following arguments:

Argument	Type	Description
RetVal	Integer	Any integer variable. (serr, serr1 - serr12 declared in BSL.DH are reserved for this use)
FormObjectName	String	Form name. Can be "" to include all forms in application.
FieldObjectName	String	Field name. Can be "" to include all objects in a specific form.

Example

Example fetches from custom table, evaluates the return value and sets up custom form for entry/edit.

```
serr1 = sqlfetch1(c1, "XCustAddlInfo_CustId" +  
  sparm(chkstrg), bXCustAddlInfo, Len(bXCustAddlInfo))  
  
If serr1 = NOTFOUND Then  
  serr1 = SetDefaults("NewInfo", "")  
  bXCustAddlInfo.CustId = chkstrg  
  Found_Cust = "N"  
Else  
  Found_Cust = "Y"  
End If  
  
Call DispFields("NewInfo", "")
```

See Also

DispField Statements, SetProp Statement

SetLevelChg Statement

Set the update status of a specific level.

Syntax

Call **SetLevelChg**(*LevelNbr*, *Status*)

Remarks

Each update level, as defined by the Levels property of the **SAFUpdate** control, has a corresponding level status flag that is automatically maintained by the system. The purpose of the level status flag is to facilitate the optimization of database updates performed in response to Microsoft Dynamics SL toolbar buttons. In general, these flags allow the system to only perform database updates for update levels which have in fact changed. If no information has changed then no information needs to be saved.

As previously mentioned, these update flags are automatically maintained by the system. When an existing record is loaded the flag is set to NOTCHANGED. If any non-key field is subsequently modified then the level status flag for the corresponding level is set to UPDATED. When a new record is being entered, the level status flag is set to INSERTED.

The **SetLevelChg** statement allows the application to override the current value of the status flag for a particular level. This can be useful if a data value is modified programmatically and therefore the system needs to be notified that something has changed so the corresponding information will actually be saved when the user presses the Save toolbar button.

The **SetLevelChg** statement uses the following arguments:

Argument	Type	Description
<i>LevelNbr</i>	Integer	Level whose status flag is to be explicitly set.
<i>Status</i>	Integer	Level status flag. The following valid values are defined as symbolic constants in <code>Applic.DH</code> : INSERTED, UPDATED, NOTCHANGED.

Example

The Payroll Earnings Type Maintenance screen contains a button to automatically populate the grid with all Deductions. This amounts to inserting records into the grid (into its underlying memory array) under program control. Since the data is not entered via the user interface by the user, the system needs to be notified that information at the grid level (LEVEL1 in this case) has been programmatically updated and therefore needs to be saved. However, such notification only needs to occur if the system is not already aware that data has changed.

```
'If any records were inserted into the memory array then we need to make
'sure that the level status for the detail level is something other than
'NOTCHANGED so the system will know that something needs to be saved.
  If (AnyRecsInserted = True) Then

      If (TestLevelChg(LEVEL1) = NOTCHANGED) Then
          Call SetLevelChg(LEVEL1, UPDATED)
      End If

  End If
```

See Also

TestLevelChg Function

SetObjectValue Function

Sets a specified object field's value.

Syntax

IntVar = **SetObjectValue**(ObjectName, Value)

Remarks

This function allows you to set the value of any bound object on the screen. The object must be on the form in order to set it. If you wish to set a value of a field that is not on the form, you can use the **SetBufferValue** function instead.

Note that you can use this function to set disabled and invisible objects that you make invisible or hidden. You cannot set disabled or invisible standard Microsoft Dynamics SL objects.

The **Chk** event of the object you set is also executed so that all error checking takes place.

The **SetObjectValue** function uses the following arguments:

Argument	Type	Description
IntVar	Integer	Any integer variable. (serr, serr1 - serr12 declared in BSL.DH are reserved for this use)
ObjectName	String	Name of the object whose value you wish to set.
Value	String	Value you wish to set ObjectName to.

Example

```
Dim CommissionAmount As Double
Dim ExtendedAmount As Double

ExtendedAmount = Val (chkstrg$)
If ExtendedAmount <= 1000 Then
    CommissionAmount = 1
ElseIf ExtendedAmount <= 2000 then
    CommissionAmount = 2
ElseIf ExtendedAmount <= 3000 then
    CommissionAmount = 3
Else
    CommissionAmount = 4
End If

serr1 = SetObjectValue ("ccmnpct", Str$(CommissionAmount))
```

See Also

GetObjectValue Function, **SetBufferValue** Function

SetProp Statement

Sets the properties of objects at runtime.

Syntax

Call **SetProp** (ObjectName, PropertyName, PropertyValue)

Remarks

Allows the application to set property values at runtime. This function should not be used to set default properties.

The **SetProp** statement uses the following arguments:

Argument	Type	Description
<i>ObjectName</i>	String	Name of object whose property you wish to change.
<i>PropertyName</i>	String	Property you wish to change.
<i>PropertyValue</i>	Integer	Value to change property to.

The following valid values for the PropertyName argument are defined as symbolic constants in BSL.DH:

Symbolic Constant	Valid Datatype	Valid Data Values
PROP_BLANKERR (required)	Integer	TRUE / FALSE
PROP_CAPTION	String	
PROP_CUSTLIST	String	
PROP_ENABLED	Integer	TRUE / FALSE
PROP_HEADING	String	
PROP_MASK	String	
PROP_MIN	String	
PROP_MAX	String	
PROP_TABSTOP	Integer	TRUE / FALSE
PROP_VISIBLE	Integer	TRUE / FALSE

Example

```
'Enable push button
Call SetProp("OpenButton", PROP_ENABLED, True)

'Disable push button
Call SetProp("OpenButton", PROP_ENABLED, False)

'Make an object invisible
Call SetProp("cname", PROP_VISIBLE, False)

'Example uses a password dialog for setting a field invisible at runtime
Dim PassWord$

PassWord = PasswordBox$("Enter the Password Override","Message")
' Get password (case sensitive)
If Trim$(PassWord) = "SYSADMIN" Then
    MsgBox(PassWord)
    Call SetProp("clastchkdate", PROP_VISIBLE, True)
End If
```

See Also

SetDefaults Statement, MsetProp Statement

SFetch Functions

Used to retrieve a composite record from the database based on some pre-defined SQL statement or stored procedure.

Syntax

```
RetVal = SFetch1(Cursor, bTable1, bTable1Length)
```

```
RetVal = SFetch4(Cursor, bTable1, bTable1Length, bTable2, bTable2Length, bTable3, bTable3Length, bTable4, bTable4Length)
```

```
RetVal = SFetch8(Cursor, bTable1, bTable1Length, bTable2, bTable2Length, bTable3, bTable3Length, bTable4, bTable4Length, bTable5, bTable5Length, bTable6, bTable6Length, bTable7, bTable7Length, bTable8, bTable8Length)
```

Remarks

In order to fetch information from the server it must first know what tables, records and fields are being queried from a particular cursor. Consequently the cursor must first be initialized with either a SQL statement or stored procedure via the use of the **Sql** statement or the **SqlFetch1**, **SqlFetch4** or **SqlFetch8** functions. Once the database view has been established these functions will retrieve the next sequential record in the view consistent with the Order By clause of the SQL statement used to initialize the view. After the last record in the view has been returned all subsequent calls to **SFetch1**, **SFetch4** and **SFetch8** will return NOTFOUND.

SFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either **SFetch4** or **SFetch8** can be used.

The **SFetch1** function uses the following arguments (**SFetch4** and **SFetch8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0)

Argument	Type	Description
<i>RetVal</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if no additional records exist in the current view.
<i>Cursor</i>	Integer	SQL database cursor.
<i>bTable1</i>	User-defined datatype	Table structure corresponding to the primary table in the SQL statement.
<i>bTable1Length</i>	Integer	Size of first table structure. For example, LenB(bTable1) . Note: It is critical to use LenB() instead of Len() for all non-null "table length" parameters.

Note: **SGroupFetch1**, **SGroupFetch4** or **SGroupFetch8** must be used if the SQL statement used to initialize the cursor contained one or more of the following components:

- Group aggregate functions (such as Count and Sum)
- DISTINCT keyword
- GROUP BY clause
- HAVING clause
- Subqueries

Example

This example links the Inventory table to an existing screen for lookup and navigation only.

```
(general) declarations:
'$include: "inventor.dh"

Form1_Load
Call SetAddr(c1, "bInventory", bInventory, Len(bInventory))
Call SqlCursor(c1, NOLEVEL)
```

```
Dim SqlStmt$
SqlStmt = "Select * from Inventory Where ClassID = 'BOX'Order By ClassID"
Call Sql(c1, SqlStmt)
serr1 = Sfetch1(c1, bInventory, Len(bInventory))
While serr1 <> NOTFOUND
    'Check LastCost
    If bInventory.LastCost > 0 Then
        'Last Cost is positive
    End If
    serr1 = Sfetch1(c1, bInventory, Len(bInventory))
Wend
```

See Also**Sql Statement, SqlFetch Functions, SGroupFetch Functions**

SGroupFetch Functions

Used to retrieve a composite record from the database based on some pre-defined SQL statement or stored procedure containing one or more group aggregate functions and/or clauses.

Syntax

```
RetVal = SGroupFetch1(Cursor, bTable1, bTable1Length)
```

```
RetVal = SGroupFetch4(Cursor, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length)
```

```
RetVal = SGroupFetch8(Cursor, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length, bTable5, bTable5Length, bTable6, bTable6Length, bTable7,
bTable7Length, bTable8, bTable8Length)
```

Remarks

In order to fetch information from the server it must first know what tables, records and fields are being queried from a particular cursor. Consequently the cursor must first be initialized with either an SQL statement or stored procedure via the use of the **Sql** statement. **SGroupFetch1**, **SGroupFetch4** or **SGroupFetch8** are only designed for cases where the SQL statement used to initialize the cursor contains one or more of the following:

- Group aggregate functions (such as Count and Sum)
- DISTINCT keyword
- GROUP BY clause
- HAVING clause
- Subqueries

The logically equivalent **SFetch1**, **SFetch4** and **SFetch8** functions should be used if the SQL statement does not contain any of the above referenced items.

Once the database view has been established these functions will retrieve the next sequential record or data value in the view consistent with the Order By or Group By clause of the SQL statement used to initialize the view. After the last record or data value in the view has been returned all subsequent calls to **SGroupFetch1**, **SGroupFetch4** and **SGroupFetch8** will return NOTFOUND.

SGroupFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either **SGroupFetch4** or **SGroupFetch8** can be used.

The **SGroupFetch1** function uses the following arguments (**SGroupFetch4** and **SGroupFetch8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0).

Argument	Type	Description
<i>RetVal</i>	Integer	0 if a record or data value is successfully fetched. NOTFOUND is returned if no additional records exist in the current view.
<i>Cursor</i>	Integer	SQL database cursor.
<i>bTable1</i>	User-defined datatype	Table structure corresponding to the primary table or data value in the SQL statement.
<i>bTable1Length</i>	Integer	Size of first table structure or data value. For example, LenB(bTable1) or Len(DoubleVariable) . Note: It is critical to use LenB() instead of Len() for all non-null "table length" parameters.

Note: The type and size of the data returned can vary when the SQL statement contains one or more group aggregates. For the COUNT group aggregate, the data will always be returned as a 4-byte integer (Long Visual Basic datatype). The MIN and MAX group aggregates always return the same data type and length as the field on which the aggregate is based. The SUM and AVG group aggregates always return 8-byte floating point values (Double Visual Basic datatype).

Example

This example retrieves the last voucher date for the current vendor and selects a count of all documents less than the last voucher date.

```
Dim SqlStr$
Dim CountDoc As Long
Dim DateComp As Sdate

DateComp.Val = GetObjectValue("clastvodate")

SqlStr = "Select Count(*) from APDoc Where DocDate < " + Dparm(DateComp)

Call Sql(c1, SqlStr)
serr1 = sgroupfetch1(c1, CountDoc, Len(CountDoc))

Print "Number of Documents: " + str$(CountDoc)
```

Second Example

```
Dim QtyOnHand#, TotCost#, SqlStmt$, ItemID$, MessStr$

SqlStmt = "Select Sum(QtyOnHand), Sum(TotCost) from ItemSite Where ItemSite.InvtId =
@parm1 Order By ItemSite.InvtId"

ItemId = GetObjectValue("cinvtid")
Call Sql(c1, SqlStmt)
Call SqlSubst(c1, "parm1", ItemID)
Call SqlExec(c1)
serr1 = sgroupfetch4(c1, QtyOnHand, Len(QtyOnHand), TotCost, Len(TotCost), "", 0, "",
0)
MessStr = "Qty On Hand " + Str$(QtyOnHand) + "Total Cost "+ Str$(TotCost)

Call MsgBox( MessStr, MB_OK, "Message")
```

See Also

Sql Statement, SFetch Functions

Sinsert Statements

Insert one record into each specified table within an existing database view.

Syntax

Call **Sinsert1**(*Cursor*, *TablesInsertingInto*, *bTable1*, *bTable1Length*)

Call **Sinsert4**(*Cursor*, *TablesInsertingInto*, *bTable1*, *bTable1Length*, *bTable2*, *bTable2Length*, *bTable3*, *bTable3Length*, *bTable4*, *bTable4Length*)

Call **Sinsert8**(*Cursor*, *TablesInsertingInto*, *bTable1*, *bTable1Length*, *bTable2*, *bTable2Length*, *bTable3*, *bTable3Length*, *bTable4*, *bTable4Length*, *bTable5*, *bTable5Length*, *bTable6*, *bTable6Length*, *bTable7*, *bTable7Length*, *bTable8*, *bTable8Length*)

Remarks

New records can be programmatically inserted directly into an existing database table via the use of the **Sinsert1**, **Sinsert4** and **Sinsert8** statements. In order to insert information into the database, the server must first know what tables are being referenced by a particular cursor. Consequently the cursor must first be initialized with either an SQL statement or stored procedure via the use of the **Sql** statement or the **SqlFetch1**, **SqlFetch4** or **SqlFetch8** functions. Once the database view has been established these functions will insert one new record into each table referenced in the *TablesInsertingInto* argument using data from corresponding table structure arguments.

Sinsert1 is designed for SQL statements referencing a single table. In this case the *TablesInsertingInto* is always the name of the single table actually referenced. For more advanced SQL statements having one or more table joins either **Sinsert4** or **Sinsert8** can be used. The referencing of more than one table does not automatically force the insertion of a record into every table in the view anytime **Sinsert4** or **Sinsert8** is used on the corresponding cursor. A single record will only be inserted into each table explicitly specified in the *TablesInsertingInto* argument so long as each table name so specified is also referenced in the SQL statement which was used to initialize the current view. Thus, for example, if TableA and TableB are the only two tables referenced in the SQL statement used to initialize the current view then a value of TableXYZ would be invalid for the *TablesInsertingInto* argument.

The **Sinsert1** function uses the following arguments (**Sinsert4** and **Sinsert8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0)

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor.
<i>TablesInsertingInto</i>	String	Name of each table, in the specified cursor's view, into which a new record is to be inserted. Multiple table names should be separated by commas.
<i>bTable1</i>	User-defined datatype	Table structure corresponding to the primary table in the SQL statement. Data in this structure will be inserted into its corresponding database table if the name of the said database table is explicitly specified in the <i>TablesInsertingInto</i> argument.
<i>bTable1Length</i>	Integer	Size of first table structure. For example, LenB (<i>bTable1</i>). Note: It is critical to use LenB () instead of Len () for <i>all</i> non-null "table length" parameters.

Example

Example attached to Update event of the Microsoft Dynamics SL screen:

```
If Level = 0 Then
  If Found_Cust = "Y" then
    Call supdate1(c1,"XCustAddlInfo",
      bXCustAddlInfo,Len(bXCustAddlInfo))
  Else
    Call sinsert1(c1,"XCustAddlInfo",
      bXCustAddlInfo,Len(bXCustAddlInfo))
  End If
End If
```

Example 2

```
Insert new entry into Customer and ARHist tables
Call Sinsert4(c1, "*.*", bCustomer, Len(bCustomer),
  bARHist, Len(bARHist), "", 0, "", 0)
```

See Also

Sql Statement, SqlFetch Functions

SParm Function

Convert a string into an SQL parameter string.

Syntax

SQLParmStr = **SParm**(*StrToConvert*)

Remarks

The **SParm** function uses the following arguments:

Argument	Type	Description
<i>SQLParmStr</i>	String	<i>StrToConvert</i> converted into an SQL parameter string.
<i>StrToConvert</i>	String	String value to convert.

Example

These examples assume the following SQL statement was used to create a stored procedure called `Employee_EmpId`

```
Select * from Employee where EmpId LIKE @parm1 order by EmpId
```

This code snippet illustrates how to use the `Employee_EmpId` stored procedure to fetch the employee record for employee #000581.

```
SqlStr = "Employee_EmpId" + SParm("000581")
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee, LenB(bEmployee))
```

This code snippet illustrates how to use the `Employee_EmpId` stored procedure to fetch all employee records by using a wildcard value for the `EmpId` parameter.

```
SqlStr = "Employee_EmpId" + SParm(SQLWILDSTRING)
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee, LenB(bEmployee))

While ( Employee_Fetch = 0)
    Employee_Fetch = SFetch1( CSR_Employee, bEmployee, LenB(bEmployee))
Wend
```

This code snippet illustrates how to use the `Employee_EmpId` stored procedure to fetch all employee records beginning and ending with zero by using a combination of string literals and single character wildcard values for the `EmpId` parameter.

```
SQLParmStr = "0" + SQLWILDCHAR + SQLWILDCHAR + SQLWILDCHAR + ␣
SQLWILDCHAR + "0"
SqlStr = "Employee_EmpId" + SParm(SQLParmStr)
Employee_Fetch = SqlFetch1(CSR_Employee, SqlStr, bEmployee,␣
LenB(bEmployee))

While ( Employee_Fetch = 0)
    Employee_Fetch = SFetch1( CSR_Employee, bEmployee, LenB(bEmployee))
Wend
```

See Also

DParam Function, FParam Function, IParam Function

Sql Statement

Initialize a new database view.

Syntax

Call **Sql**(*Cursor*, *SqlStr*)

Remarks

Takes the specified SQL text, compiles it, and then executes it. If fetch operations are required then one of the **SFetch** functions must be called. If the SQL statement is performing an Update Set, Delete From or Insert Into operation, no subsequent fetch operations are required. If the SQL statement references one or more parameters, the **SqlSubst** and **SqlExec** functions must also be called.

The **Sql** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor.
<i>SqlStr</i>	String	SQL statement or stored procedure to be used in initializing a new database view. If a stored procedure is used then all parameters must be sequentially appended in order of occurrence within the original Create Procedure statement using calls to SqlSubst .

Example

```
Dim SqlStr As String

'Set AP Documents which are marked as current to non-current
SqlStr = "Update APDoc Set Current = 'False' where APDoc.DocBal = 0 ↓
        and PerEnt <=" + sparm(NextPerNbr) + " and Current = 'True'"
Call Sql(C_APClose, Trim$(SqlStr))

'Process posted AP batches
SqlStrg = "Select * from Batch Where Module = 'AP' and Status = 'P' ↓
        Order By BatNbr"
Call Sql(CSR_Batch, SqlStrg)

serr1 = SFetch1(c1, bBatch, LenB(bBatch))
While serr1 = 0
    'Process the batch
    ....
    'Get the next batch
        serr1 = SFetch1(CSR_Batch, bBatch, LenB(bBatch))
Wend
```

See Also

SFetch Functions, **SqlSubst** Statement, **SqlExec** Statement, **SqlFetch** Functions

SqlCursor Statement

Allocate a new database cursor.

Syntax

Call **SqlCursor**(*Cursor, Flags*)

Remarks

All read/write communication between an application and the database must occur through a database cursor. A cursor is basically a database resource used to track low level information required to implement SQL database read/write operations. For example, a cursor tracks the SQL statement used to initialize the current view, what individual fields were selected, the current record within the view as well as other more detailed information.

Each level within a screen must have a corresponding cursor allocated within the Form_Load event of Form1 to facilitate database read/write activity at that level. Additionally, many of the SQL API calls within the Microsoft SL SDK , such as **Sql**, **SFetch1**, **SqlFetch1** and **SUpdate1** require a database cursor as one of their arguments.

Each application can allocate a maximum of 36 cursors. Cursors no longer needed by the application can optionally be freed via the use of the **SqlFree** statement. All cursors are automatically released by the system when the application terminates execution (when **ScreenExit** is called).

The **SqlCursor** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	Variable to be initialized with a resource handle for an SQL database cursor.
<i>Flags</i>	Integer	One or more special flags notating what and/or how the cursor will be used. At a minimum the <i>Flags</i> parameter must contain one of the LEVEL0 through LEVEL9 or NOLEVEL symbolic constants defined in Applic.DH. Cursors not explicitly associated with a particular level number should be allocated using the NOLEVEL flag.

The following symbolic constants can optionally be passed as *Flags* (by adding them to the required LEVEL0 through LEVEL9 or NOLEVEL symbolic constants):

- **SqlList** - Read / Write operations performed on the cursor will automatically be buffered to improve performance. If an application updates even one record on a buffered cursor, it must update all records read with that cursor within the same database transaction. Failure to comply with this requirement will result in a sparse update error.
- **SqlSystemDb** - All database operations will be directed to the system database as opposed to the application database.

Example

```
'Example declares a dynamic cursor in Form1_Load
'because it will be used elsewhere in the application

'NoLevel indicates no updating on this cursor,
'only lookups will be performed

Call SqlCursor(c1, NoLevel)

'NoLevel + SQLList indicates a buffered cursor

Call SqlCursor(c2, NoLevel + SqlList)

'Example declares a static cursor in Chk event. Note
'that a previous SqlCursor call was not made for c3:

Dim SqlStmt$
Dim CountResult As Long
SqlStmt = "Select Count(*) from Customer"
Call Sql(c3, SqlStmt)
serr1 = SgroupFetch1(c3, CountResult, Len(CountResult))
Call MsgBox("Count is: " + Str$(CountResult), MB_OK, "Message")
```

See Also

SFetch Functions, SInsert Statements, Sql Statement, SqlFetch Functions, SqlFree Statement, SUpdate Statements

SqlCursorEx

Allocate a new database cursor.

Syntax

Call **SqlCursorEx**(*Cursor*, *Flags*, *CursorName*, *ReferencedTableNames*, *UpdateTableNames*)

Remarks

All read/write communication between an application and the database must occur through a database cursor. A cursor is basically a database resource used to track low level information required to implement SQL database read/write operations. For example, a cursor tracks the SQL statement used to initialize the current view, what individual fields were selected, the current record within the view as well as other more detailed information.

Each level within a screen must have a corresponding cursor allocated within the Form_Load event of Form1 to facilitate database read/write activity at that level. Additionally, many of the SQL API calls within the Microsoft SL SDK, such as **Sql**, **SFetch1**, **SqlFetch1** and **SUpdate1** require a database cursor as one of their arguments. If the cursor handle passed to one of these SQL API calls has not been previously allocated then it will automatically be allocated during the call. However it is important to be aware of the fact that all cursors not explicitly allocated, via a **SqlCursorEx** call, are automatically allocated as read-only cursors.

Each application can allocate a maximum of 36 cursors. Cursors no longer needed by the application can optionally be freed via the use of the **SqlFree** statement. All cursors are automatically released by the system when the application terminates execution (when **ScreenExit** is called).

The **SqlCursorEx** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	Variable to be initialized with a resource handle for an SQL database cursor.
<i>Flags</i>	Integer	One or more optimization flags notating what and/or how the cursor will be used. At a minimum the Flags parameter must contain one of the LEVEL0 through LEVEL9 or NOLEVEL symbolic constants defined in Applic.DH. Cursors not explicitly associated with a particular level number should be allocated using the NOLEVEL flag.
<i>CursorName</i>	String	Alias name associated with the cursor. This value is used solely to enhance the readability of diagnostic messages.
<i>ReferencedTableNames</i>	String	Comma delimited list of table names that will be referenced by the cursor. This list should be thought of as being applicable so long as the cursor remains allocated.
<i>UpdateTableNames</i>	String	Comma delimited list of all tables which may be updated by the cursor. The principle usage of this list is to facilitate performance optimizations related to tables in the cursor's view that will never be updated. If no table names are specified then by default it will be assumed that all of the referenced tables will be updated at some point. Any table name appearing in UpdateTableNames must also be specified in ReferencedTableNames. This list of table names should not be confused with the table names passed to any of the SInsert , SUpdate or SDelete statements. The names passed to those statements identify the actual tables updated in a particular database operation -- which may be a subset of the <i>UpdateTableNames</i> argument.

Note: The following optimization flags are implemented as symbolic constants in Applic.DH. They can optionally be passed via the Flags argument by adding them to the required LEVEL0 through LEVEL9 or NOLEVEL symbolic constant:

- **SqlFastReadOnly** – Similar to SqlReadOnly, the cursor will be used exclusively for read operations. On the Microsoft® SQL Server™ platform, all database operations occurring on a SqlFastReadOnly cursor will be serviced directly from SQL Server's faster low-level API. Please refer to the note below for more information regarding the practical implications of utilizing SQL Server's low level

API. Cursors with this flag should not be used to access tables which have received insert, update or delete operations from a different cursor in the same database transaction.

- **SqlList** — Read / Write operations performed on the cursor will automatically be buffered to improve performance. If an application updates even one record on a buffered cursor, it must update all records read with that cursor within the same database transaction. Failure to comply with this requirement will result in a sparse update error. In the SQL Server environment, all database operations are buffered where possible - regardless of whether the SqlList flag has been specified. Refer to the SqlNoList flag for more information.
- **SqlLock** — Every composite record retrieved with this type of cursor will be locked. This flag effectively avoids the overhead of explicitly locking every record that is fetched from the database within a transaction. SqlLock is primarily designed for usage in process-like scenarios where it is known in advance that every record accessed by the cursor needs to be updated. If the application will ever be waiting for user-input while the cursor still has a current record then the cursor should probably not be allocated with the SqlLock flag. This is due to the simple fact that the current record would be locked while waiting for the user to respond - thus potentially causing contention across the system. Records retrieved on SqlLock cursors will remain locked until the next record is retrieved. For this reason, the application should always continue to fetch records until a NOTFOUND is returned - thereby unlocking the last record which was actually retrieved. It is strongly recommended that applications only implement the usage of SqlLock to facilitate optimum performance - as opposed to using it to implement their own multi-user contention management scheme (“process XYZ must be running because such and such a record is locked”). This flag is only supported on the SQL Server platform.
- **SqlNoList** — This flag forces the buffering of read/write operations to be suppressed. In the SQL Server environment all database operations are buffered by default, where possible, regardless of whether or not the SqlList flag has been specified. Consequently on the SQL Server platform, buffering must be explicitly suppressed if for some reason the application does *not* want a particular cursor to be buffered. In some cases this default buffering is automatically suppressed due to low level restrictions.
- **SqlNoSelect** — This flag indicates that the cursor will be used with any SQL statement except Select statements. The most common usage of this type of cursor will be to process Update and Delete SQL statements. On the SQL Server platform, all database operations occurring on a SqlNoSelect cursor will be serviced directly from SQL Server's faster low-level API. Please refer to the note below for more information regarding the practical implications of utilizing SQL Server's low level API.
- **SqlReadOnly** — The cursor will be used exclusively for read operations. The cursor will not be used for record inserts, updates or deletions.
- **SqlSingleRow** — This flag is to be used with cursors which will never process more than one composite record after each Select statement. This flag is primarily designed to facilitate optimization on the SQL Server platform. On the SQL Server platform, all database operations occurring on a SqlSingleRow cursor will be serviced directly from SQL Server's faster low-level API. Please refer to the note below for more information regarding the practical implications of utilizing SQL Server's low level API. **SFetch** calls on cursors of this type should not be separated from the associated **Sql** call by any other database operations. The simplest method to satisfy this requirement is via the usage of the **SqlFetch** function.
- **SqlSystemDb** — All database operations will be directed to the system database as opposed to the application database.

Note: Regarding the optimization flags which invoke usage of SQL Server's low-level API (SqlFastReadOnly, SqlNoSelect and SqlSingleRow): On the SQL Server platform, all database operations occurring on a cursor optimized with one of these flags will bypass SQL Server's standard cursor API. Instead, database operations will be serviced directly from SQL Server's low level API. The advantage here is that the low level API is faster. However when using the low level API the cursor can encounter contention with other cursors in the same application (which is not the case when all operations are serviced by SQL Server's cursor API). SQL statements processed via this type of cursor must include all of the fields for every table referenced in the statement. For example, partial record Select statements are not supported on this type of cursor.

SqlErr Function

Obtain the return value for the SQL operation last performed.

Syntax

RetVal = **SqlErr**()

Remarks

This function can be used after any SQL call that is declared as a statement (as opposed to a function), in order to obtain the return value. For example, the call **SInsert1** is declared as a subroutine and does not return a value to the application. In most cases, the application does not need to check the return from this call, since by default SWIM traps all return codes except 0 and the NOTFOUND symbolic constant. However in special cases where the **SqlErrException** statement is used to give the application more control over error handling, the application will need to obtain the return code and this function is used for that purpose.

The **SqlErr** function returns one of the following integer global constants declared in *Applic.DH*:

Return Value	Description
DUPLICATE	The record last updated or inserted caused a duplicate error as defined by one or more unique indexes on the relevant table.

Example

This example illustrates how to insert a uniquely numbered Batch record into the database. The example assumes that a database transaction is already active when the illustrated procedure is called. **SqlErrException** and **SqlErr** are used to detect duplicate batch number error without causing Swim to abort the transaction. The sample procedure receives two parameters:

- **BatchStruct** - A Batch record which is to be saved to the database, having all relevant fields ALREADY initialized EXCEPT the batch number itself.
- **AutoNbr_SqlStr** - The name of an "auto number" stored procedure which will fetch *AutoBat* and *LastBatNbr* fields (in that order) from one of the setup records.

```
Sub BATCH_AUTONBR_INSERT (BatchStruct As Batch, ByVal AutoNbr_SqlStr As String)
    Dim AutoNbrFetch As Integer

    'Allocate cursor resources
        Call SqlCursor(CSR_AutoNbr, NOLEVEL)
        Call SqlCursor(CSR_Batch_AutoNbr_Insert, NOLEVEL)

    'Setup cursor with stored procedure so it will be able to
    'execute an SInsert1()
    Call Sql(CSR_Batch_AutoNbr_Insert, "Batch_Module_BatNbr" + ␣
    sparm("") + sparm(""))

    'Fetch the necessary fields for auto batch numbering from the Setup
    'record specified by AutoNbr_SqlStr
        AutoNbrFetch = SqlFetch1(CSR_AutoNbr, AutoNbr_SqlStr, AutoNbr, ␣
        LenB(AutoNbr))

    'Turn ON exception error checking for DUPLICATE error condition so
    'Swim will not go into abort mode if a duplicate batch number happens
    'to already exist.
        Call SqlErrException(EXCEPTION_ON, DUPLICATE)

    Do
        'Increment AutoNbr.LastNbrUsed to next sequential value
```

```
'(within the size of batch numbers actually being used).
  Call incrstrg(AutoNbr.LastNbrUsed, 6, 1)

BatchStruct.BatNbr = AutoNbr.LastNbrUsed

'Attempt to insert batch record with new batch number
  Call SInsert1(CSR_Batch_AutoNbr_Insert, "Batch", BatchStruct, ↵
    LenB(BatchStruct))

Loop While (SqlErr() = DUPLICATE)

'Write changes to Setup record back to database
  Call SUpdate1(CSR_AutoNbr, ".*", AutoNbr, LenB(AutoNbr))

'Turn OFF exception error checking for DUPLICATE.
  Call SqlErrException(EXCEPTION_OFF, DUPLICATE)

'Free up cursor resources
  Call SqlFree(CSR_AutoNbr)
  Call SqlFree(CSR_Batch_AutoNbr_Insert)

End Sub
```

See Also

SqlErrException Statement

SqlErrException Statement

Toggle automatic error handling logic for one or more database error codes.

Syntax

Call **SqlErrException** (*ToggleFlag*, *ErrorToExcept*)

Remarks

By default, all error codes except 0 and NOTFOUND are trapped within SWIM and are not returned to the application. To alter this behavior an application can use the **SqlErrException** function to tell SWIM not to trap certain errors and instead return them to the application. This statement is used in conjunction with the **SqlErr** function.

Note that the application is responsible for toggling the exception back off once it is no longer needed.

The **SqlErrException** statement uses the following arguments:

Argument	Type	Description
<i>ToggleFlag</i>	Integer	Used to tell SWIM to turn the error exception on or off. The EXCEPTION_ON and EXCEPTION_OFF symbolic constants are defined in Applic.DH as the only two valid values.
<i>ErrorToExcept</i>	Integer	Error code to be returned to the application rather than handled automatically by the system. If only duplicate record errors should be excepted from automatic error handling logic then the DUPLICATE symbolic constant defined in Applic.DH should be passed. The RETURN_ALL_ERRVALS symbolic constant defined in Applic.DH should be passed if all errors are to be returned to the application.

Example

```
Dim DupFlag%
DupFlag = DUPLICATE

'Tell the Kernel to return DUPLICATE status messages
Call SqlErrException(EXCEPTION_ON, DUPLICATE)

'Loop until a non duplicate number is assigned
Do Until DupFlag <> DUPLICATE

    'Increment last reference number
    Call IncrStrg(bARSetup.LastRefNbr, Len(bARSetup.LastRefNbr, 1)

    'Attempt to insert the new refnbr
    bARDoc.RefNbr = bARSetup.LastRefNbr
    bRefNbr.RefNbr = bARDoc.RefNbr
    Call sinser1(c8, "RefNbr", bRefNbr, Len(bRefNbr))

    'Check the return value for duplicates
    DupFlag = SqlErr() 'DUPLICATE or 0

Loop 'Until exited with a good Refnbr
```

See Also

[SqlErr Function](#)

SqlExec Statement

Execute an inline SQL statement.

Syntax

Call **SqlExec**(*Cursor*)

Remarks

Execute a dynamic SQL statement on a cursor previously initialized with calls to the **Sql** and **SqlSubst** statements (in that order).

The **SqlExec** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor

Example

```
Dim VendorCount&, SqlStmt$

SqlStmt = "Select Count(*) from Vendor Where VendId Like @parm1"

Call Sql(c1, SqlStmt)
Call SqlSubst(c1, "parm1", "V001%")
Call SqlExec(c1)
serr1 = sgroupfetch1(c1, VendorCount, Len(VendorCount))

Call MsgBox("Count is " + Str$(VendorCount), MB_OK, "Message")
```

See Also

Sql Statement, SqlSubst Statement

SqlFetch Functions

Used to initialize a new database view and immediately retrieve a composite record.

Syntax

```
RetVal = SqlFetch1(Cursor, SqlStr, bTable1, bTable1Length)
```

```
RetVal = SqlFetch4(Cursor, SqlStr, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length)
```

```
RetVal = SqlFetch8(Cursor, SqlStr, bTable1, bTable1Length, bTable2, bTable2Length, bTable3,
bTable3Length, bTable4, bTable4Length, bTable5, bTable5Length, bTable6, bTable6Length, bTable7,
bTable7Length, bTable8, bTable8Length)
```

Remarks

In order to fetch information from the database a database view must first be initialized specifying what tables, fields and restriction criteria are to be utilized. Secondly an actual request for data from an existing view must be sent to the server. Each of the **SqlFetch1**, **SqlFetch4** and **SqlFetch8** functions effectively perform both of these operations in a single call which would otherwise require a combination of two calls (for example, **Sql** and **SqlFetch1**). In looping situations where a program needs to sequentially read through multiple records in a view, these functions are convenient for initializing the view and immediately fetching the first record. However they should not be used to fetch subsequent records since the view is being re-established each time **SqlFetch1**, **SqlFetch4** or **SqlFetch8** is called and therefore they will always only fetch the first record in the view. In these cases, **SFetch1**, **SFetch4** or **SFetch8** can be used to fetch subsequent records.

SqlFetch1 is designed for SQL statements returning data from a single table. For more advanced SQL statements having one or more table joins either **SqlFetch4** or **SqlFetch8** can be used.

The **SqlFetch1** function uses the following arguments (**SqlFetch4** and **SqlFetch8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0)

Argument	Type	Description
<i>RetVal</i>	Integer	0 if a record is successfully fetched. NOTFOUND is returned if no records matching the restriction criteria exist in the newly established view.
<i>Cursor</i>	Integer	SQL database cursor.
<i>SqlStr</i>	String	SQL statement or stored procedure to be used in initializing a new database view. If a stored procedure is used then all parameters must be sequentially appended in order of occurrence within the original Create Procedure statement. These parameter values must also be properly converted to SQL parameters via usage of the SParm , IParm , FParm and DParm functions - whichever is appropriate for the datatype of each individual parameter.
<i>bTable1</i>	User-defined datatype	Table structure corresponding to the primary table in the SQL statement.
<i>bTable1Length</i>	Integer	Size of first table structure. For example, LenB (<i>bTable1</i>). Note: It is critical to use LenB () instead of Len () for all of the non-null "table length" parameters.

SGroupFetch1, **SGroupFetch4** or **SGroupFetch8** must be used if the SQL statement used to initialize the cursor contains one or more of the following components:

- Group aggregate functions (such as Count and Sum)
- DISTINCT keyword
- GROUP BY clause
- HAVING clause
- Subqueries

Example

Example from CustId chk event which fetches new table info.

```

SUB ccustid_Chk(chkstrg$, retval%)
    serr1 = sqlfetch1(c1, "XCustAddlInfo_CustId" + sparm(chkstrg), ↓
bXCustAddlInfo, Len(bXCustAddlInfo))
    If serr1 = NOTFOUND Then
        serr1 = SetDefaults("NewInfo", "")
        bXCustAddlInfo.CustId = chkstrg
        Found_Cust = "N"
    Else
        Found_Cust = "Y"
    End If
    Call DispFields("NewInfo", "")
END SUB

```

Second Example

```

'Select Unreleased APDocs and Join Vendor Table
Dim SqlStmt$
SqlStmt$ = "Select * from APDoc, Vendor Where APDoc.VendId = ↓
Vendor.VendId(+) And APDoc.Rlsed = 'False'
Order by APDoc.RefNbr

serr1 = sqlfetch4(c1, SqlStmt, bAPDoc, Len(bAPDoc), bVendor, Len(bVendor), ↓
"", 0, "", 0)

```

See Also

Sql Statement, SqlFetch Functions, SParm Function, IParm Function, FParm Function, DParm Function, SGroupFetch Functions

SqlFree Statement

Free a database cursor no longer needed by the application.

Syntax

Call **SqlFree**(*Cursor*)

Remarks

The **SqlFree** statement is used to deallocate cursors previously allocated with the **SqlCursor** statement. The usage of this statement is entirely optional. It is normally used when a cursor is no longer needed by an application whose total number of cursors is very close to the maximum limit. All cursors are automatically released by the system when the application terminates execution (when **ScreenExit** is called).

The **SqlFree** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	Resource handle for the SQL database cursor to be deallocated.

Example

Example uses **SqlFree** in a general subroutine.:

```
Get APSetup Information set global variable NextRefNbr

Sub GetSetupInfo()
    Call SqlCursor(c1)
    Call SqlFetch1(c1, "APSetup_All", bAPSetup, Len(bAPSetup))
    NextRefNbr = bAPSetup.LastRefNbr
    Call SqlFree(c1)
End Sub
```

See Also

SqlCursor Statement

SqlSubst Statement

Specify a data value for a substitution parameter in an inline SQL statement previously executed via the **Sql** function.

Syntax

Call **SqlSubst**(*Cursor*, *ParmName*, *ParmValue*)

Remarks

SqlSubst allows you to specify values for one or more substitution variables in an inline SQL statement that is executed via the **Sql** statement. It is not to be used for a stored procedure.

This statement must be called after the **Sql** statement and before the **SqlExec** statement. It can be called any number of times before **SqlExec** is called, depending on how many parameter values there are to substitute into the SQL statement.

The **SqlSubst** statement uses the following arguments:

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor previously initialized with an inline SQL statement containing one or more parameters.
<i>ParmName</i>	String	Name of the SQL statement parameter (not including the '@' character).
<i>ParmValue</i>	String	Data value to be used for the designated parameter.

Example

```
Dim VendorCount&, SqlStmt$

SqlStmt = "Select Count(*) from Vendor Where VendId Like @parm1"

Call Sql(c1, SqlStmt)
Call SqlSubst(c1, "parm1", "V001%")
Call SqlExec(c1)
serr1 = sgroupfetch1(c1, VendorCount, Len(VendorCount))

Call MsgBox ("Count is " + Str$(VendorCount), MB_OK, "Message")
```

See Also

Sql Statement, **SqlExec Statement**

StrToDate Statement

Convert a date value from a string in MMDDYYYY format into an SQL date format.

Syntax

Call **StrToDate**(*DateStringToConvert*, *SQLDate*)

The **StrToDate** statement uses the following arguments:

Argument	Type	Description
<i>DateStringToConvert</i>	String	String in MMDDYYYY format.
<i>SQLDate</i>	SDate user-defined datatype (declared in Applic.DH)	Converted date value.

Example

```
Dim NewDate As Sdate
Call StrToDate("10311994", NewDate)
```

See Also

DateToStr Function, **DateToStrSep** Function

StrToTime Statement

Convert a time value from a string in HHMMSShh format into an SQL time format.

Syntax

Call **StrToTime**(*TimeStringToConvert*, *SQLTime*)

The **StrToTime** statement uses the following arguments:

Argument	Type	Description
<i>TimeStringToConvert</i>	String	String in HHMMSShh format.
<i>SQLTime</i>	STime user-defined datatype (declared in Applic.DH)	Converted time value.

Example

```
Dim TimeVal As Stime
Dim TimeStr As String
TimeStr = "12010000"
Call StrToTime(TimeStr, TimeVal)
```

See Also

TimeToStr Function

Update Statements

Update one record of each specified table within an existing database view.

Syntax

Call **SUpdate1**(Cursor, TablesUpdating, bTable1, bTable1Length)

Call **SUpdate4**(Cursor, TablesUpdating, bTable1, bTable1Length, bTable2, bTable2Length, bTable3, bTable3Length, bTable4, bTable4Length)

Call **SUpdate8**(Cursor, TablesUpdating, bTable1, bTable1Length, bTable2, bTable2Length, bTable3, bTable3Length, bTable4, bTable4Length, bTable5, bTable5Length, bTable6, bTable6Length, bTable7, bTable7Length, bTable8, bTable8Length)

Remarks

Existing records can be programmatically updated directly via the use of the **SUpdate1**, **SUpdate4** and **SUpdate8** statements. However, before a record can be updated it must first be fetched using one of the **SFetch1**, **SFetch4** and **SFetch8** or **SqlFetch1**, **SqlFetch4** and **SqlFetch8** functions. The fetch operation which precedes the update must be made using the same database view / cursor on which the update will occur. For example, if the fetch occurs on Cursor A then the update must also occur on Cursor A as opposed to some unrelated Cursor XYZ. Nevertheless, once the database view has been established these functions will update the current record in the view for each table referenced in the *TablesUpdating* argument using data from corresponding table structure arguments.

SUpdate1 is designed for SQL statements referencing a single table. In this case the *TablesUpdating* is always the name of the single table actually referenced. For more advanced SQL statements having one or more table joins either **SUpdate4** or **SUpdate8** can be used. The referencing of more than one table does not automatically force the current record of every table within the view to be updated anytime **SUpdate4** or **SUpdate8** is used on the corresponding cursor. The current record of a particular table in the view will only be updated if its corresponding table name is explicitly specified in the *TablesUpdating* argument so long as each table name so specified is also referenced in the SQL statement which was used to initialize the current view. Thus, for example, if TableA and TableB are the only two tables referenced in the SQL statement used to initialize the current view then a value of TableXYZ would be invalid for the *TablesUpdating* argument.

The **SUpdate1** function uses the following arguments (**SUpdate4** and **SUpdate8** respectively have four and eight table structures and corresponding lengths. PNULL should be passed for unused table structure parameters as well as a corresponding length of zero such as PNULL, 0).

Argument	Type	Description
<i>Cursor</i>	Integer	SQL database cursor.
<i>TablesUpdating</i>	String	Name of each table, in the specified cursor's view, whose current record is to be updated. Multiple table names should be separated by commas.
<i>bTable1</i>	User-defined datatype	Table structure corresponding to the primary table in the SQL statement. Data from this structure will be used to overwrite existing data in the corresponding current record if the name of the said database table is explicitly specified in the <i>TablesUpdating</i> argument.
<i>bTable1Length</i>	Integer	Size of first table structure. For example, LenB (<i>bTable1</i>). Note: It is critical to use LenB () instead of Len () for all non-null "table length" parameters.

Note: Database updates occurring on cursors allocated by **SqlCursorEx** using the **SqlList** flag (buffered cursors) have two unique requirements. First, the application must update all records it reads, using a buffered cursor, if it updates even one record. Failure to comply with this requirement will result in a sparse update error. Secondly, the application should not modify the view on the cursor after updates have occurred until after the transaction has ended. For example if the application is reading and updating Table A records on buffered Cursor A then Cursor A should not be used for any other purpose until after the database transaction has ended. If no updates are made using Cursor A then this requirement does not apply.

Example

This simple example updates all GLTran records in General Ledger Batch 000001 as being released. Within Microsoft Dynamics SL, the release of GLTran records actually entails additional application logic which is not directly relevant to the illustration of the SUpdate4 statement and therefore it has been removed from this example.

This example assumes the GLTran_Module_BatNbr_LineNbr stored procedure was originally created with the following SQL statement:

```
Select * from GLTran
    where Module = @parm1
    and BatNbr = @parm2
    and LineNbr between @parm3beg and @parm3end
    order by Module, BatNbr, LineNbr
```

Since the above SQL statement only retrieves data from a single table (for example, the GLTran table) SUpdate1 would be adequate. However in this example, SUpdate4 is actually used to illustrate how to pass "", 0 for unused table structure arguments.

```
Dim CSR_GLTran      As Integer
Dim SqlStr          As String
Dim GLTranFetch    As Integer

'Allocate a database cursor. A buffered cursor (i.e., SqlList) can be
'used to speed up performance since ALL GLTran records read within the
'database transaction will also be updated.
    Call SqlCursor( CSR_GLTran, NOLEVEL + SqlList)

'Begin a database transaction since all updates to the database must occur
'within a transaction.
    Call TranBeg(True)

'Initialize SqlStr with a stored procedure and associated parameters which
'can be used to fetch all GLTran records in GL Batch 000001.
    SqlStr = "GLTran_Module_BatNbr_LineNbr" + sparm("GL") + sparm("000001")
    + iparm(INTMIN) + iparm(INTMAX)

'Initialize cursor with a SQL stored procedure and immediately fetch
'first record
    GLTranFetch = SqlFetch4(CSR_GLTran, SqlStr, bGLTran, Len(bGLTran),
    "", 0, "", 0, "", 0)

While (GLTranFetch = 0)

    'Release current transaction
        bGLTran.Posted      = "U"
        bGLTran.Rlsed      = LTRUE

    'Update the record last fetched on CSR_GLTran (i.e., the current
    'GLTran record) with the modified contents of bGLTran
        Call SUpdate4(CSR_GLTran, "GLTran", bGLTran, Len(bGLTran),
        "", 0, "", 0, "", 0)
```

```
'Load next transaction record
    GLTranFetch = SFetch4(CSR_GLTran, bGLTran, Len(bGLTran),,
        "", 0,"", 0, "", 0)

Wend

'End the database transaction to commit all updates to the database.
Call TranEnd
```

See Also**SFetch Functions, SqlFetch Functions, SqlCursor Statement**

TestLevelChg Function

Return the current update status flag for a specific level.

Syntax

Status = **TestLevelChg**(*LevelNbr*)

Remarks

Each update level, as defined by the Levels property of the **SAFUpdate** control, has a corresponding level status flag that is automatically maintained by the system. The purpose of the level status flag is to facilitate the optimization of database updates performed in response to Microsoft Dynamics SL toolbar buttons. In general, these flags allow the system to only perform database updates for update levels which have in fact changed. If no information has changed then no information needs to be saved.

As previously mentioned, these update flags are automatically maintained by the system. When an existing record is loaded the flag is set to NOTCHANGED. If any non-key field is subsequently modified then the level status flag for the corresponding level is set to UPDATED. When a new record is being entered, the level status flag is set to INSERTED.

The **TestLevelChg** function allows the application to access the current value of this flag for a specific level. The current level status flag can be overridden by the application using the **SetLevelChg** statement.

The **TestLevelChg** function uses the following arguments:

Argument	Type	Description
<i>Status</i>	Integer	Current value of the level status flag for the designated <i>LevelNbr</i> . The following possible values are defined as symbolic constants in <i>Applic.DH</i> : INSERTED, UPDATED, NOTCHANGED.
<i>LevelNbr</i>	Integer	Level whose status flag is to be returned.

Example

The Payroll Earnings Type Maintenance screen contains a button to automatically populate the grid with all Deductions. This amounts to inserting records into the grid (into its underlying memory array) under program control. Since the data is not entered via the user interface by the user, the system needs to be notified that information at the grid level (LEVEL1 in this case) has been programmatically updated and therefore needs to be saved. However, such notification only needs to occur if the system is not already aware that data has changed.

```
'If any records were inserted into the memory array then we need to make
'sure that the level status for the detail level is something other than
'NOTCHANGED so the system will know that something needs to be saved.
  If (AnyRecsInserted = True) Then

      If (TestLevelChg(LEVEL1) = NOTCHANGED) Then
          Call SetLevelChg(LEVEL1, UPDATED)
      End If

  End If
```

See Also

SetLevelChg Statement

TimeToStr Function

Convert a time value from SQL time format into a string in HHMMSShh format.

Syntax

TimeString = **TimeToStr**(*TimeToConvert*)

Remarks

The **TimeToStr** function uses the following arguments:

Argument	Type	Description
<i>TimeString</i>	String	<i>TimeToConvert</i> converted to a string in HHMMSShh format.
<i>TimeToConvert</i>	STime user-defined datatype (declared in Applic.DH)	Time value to be converted.

Example

```
'If any records were inserted into the memory array then we need to make
'sure that the level status for the detail level is something other than
'NOTCHANGED so the system will know that something needs to be saved.
```

```
  If (AnyRecsInserted = True) Then

      If (TestLevelChg(LEVEL1) = NOTCHANGED) Then
          Call SetLevelChg(LEVEL1, UPDATED)
      End If

  End If
```

See Also

StrToTime Statement

TranAbort Statement

Abort the current database transaction.

Syntax

Call **TranAbort**

Remarks

The **TranAbort** statement allows the application to abort a database transaction which was initiated using the **TranBeg** statement.

Calling **TranAbort** directly is not, however, the recommended method to abort transactions. If the transaction to be aborted is an update operation for an application screen then the recommended method is for the application to set RetVal in the Update event of the **SAFUpdate** control to either an error message number or the ErrNoMess symbolic constant defined in Applic.DH. On the other hand, if the abort is to occur during a process then a fatal error message should be written to the Event Log using the **Status** statement. This will also have the effect of aborting the current database transaction.

Example

```
Call TranBeg(True)
....
'Perform Processing
.....
'Determine any error condition that would cause the transaction to abort
If ErrorCondition Then
    Call TranAbort
End If

'Perform the Insert Or Update
Sinsert...
OR
Supdate...
OR
Sdelete...

'End Transaction/Commit Work to Database
Call TranEnd
```

See Also

TranBeg Statement, TranEnd Statement

TranBeg Statement

Begin a database transaction.

Syntax

Call **TranBeg**(*IsAbortable*)

Remarks

All updates to the database must occur within a database transaction. All updates to the database will not actually be committed to the database until the transaction is ended via the **TranEnd** statement.

If any errors occur during the database transaction then the system will automatically abort (roll back) all updates occurring during the transaction as opposed to committing them to the database. If the transaction is an update operation for an application screen then it will be aborted when the application sets RetVal in the Update event of the **SAFUpdate** control to either an error message number or the ErrNoMess symbolic constant defined in Applic.DH. On the other hand, if the transaction occurs during a process then it will be aborted when a fatal error message is reported using the **Status** statement.

The **TranBeg** statement uses the following argument:

Argument	Type	Description
<i>IsAbortable</i>	Integer	True if the transaction is abortable (which is virtually always the case). Otherwise, False. Errors occurring during non-abortable transactions will cause the application to immediately terminate.

Example

```

Call TranBeg(True)
'Perform Processing
'Determine any error condition that would cause the transaction to abort
If ErrorCondition Then
    Call TranAbort
End If

'Perform the Insert Or Update
Sinsert...
OR
Supdate...
OR
Sdelete

'End Transaction/Commit Work to Database
Call TranEnd

```

See Also

TranEnd Statement, TranStatus Function

TranEnd Statement

End the current database transaction and commit all updates to the database.

Syntax

Call **TranEnd**

Remarks

If any errors occurred during the database transaction then the system will automatically abort (roll back) all updates which occurred during the transaction as opposed to committing them to the database. If the transaction is an update operation for an application screen then it will be aborted when the application sets RetVal in the Update event of the **SAFUpdate** control to either an error message number or the ErrNoMess symbolic constant defined in Applic.DH. On the other hand, if the transaction occurs during a process then it will be aborted when a fatal error message is reported using the **Status** statement.

Example

```
Call TranBeg(True)
'Perform Processing
'Determine any error condition that would cause the transaction to abort
If ErrorCondition Then
    Call TranAbort
End If

'Perform the Insert Or Update
Sinsert...
OR
Supdate...
OR
Sdelete...

'End Transaction/Commit Work to Database
Call TranEnd
```

See Also

TranBeg Statement, TranStatus Function

TranStatus Function

Returns the status of the current or last database transaction.

Syntax

```
IntegerErrVal = TranStatus()
```

Remarks

If a database transaction is currently not open then the status of the last performed database transaction is returned.

A return value of zero indicates that the transaction was successful.

A non-zero return value indicates a fatal error occurred during the transaction and therefore it will be, or was, aborted. In this case, the actual return value itself is the number of the error message describing the nature of the problem.

Example

```
Call TranBeg(True)
'Perform Processing
'Determine any error condition that would cause the transaction to abort
If ErrorCondition Then
    Call TranAbort
End If

'Perform the Insert Or Update
Sinsert...
OR
Supdate...
OR
Sdelete...

'Test for Database condition
If TranStatus() <> 0 Then
    Call TranAbort
End If

'End Transaction/Commit Work to Database
Call TranEnd
```

See Also

TranBeg Statement, TranEnd Statement

BSL Coding Tips and Techniques

Declaring Variables

Do not be redundant in your variable declarations. Consider declaring these as GLOBAL rather than DIM in each subroutine. This way, you will only need to declare them once. Place Option Explicit in General_Declarations as well as at the very beginning of any external files. This is a Visual Basic function that assures explicit variable declaration. Visual Basic and BSL both support the usage of \$, %, and # as String, Integer, and Double variable types respectively.

You do not need to explicitly set custom fields to "" (NULL) under program control. By declaring two global variables as shown in the type structure examples, you simply need to set bRecord = nRecord to null out the record or bRecord.FieldName = nRecord.FieldName to null out a particular field.

Including External Files

When creating any customization that requires BSL code, Customization Manager automatically adds the following code to the General_Declarations event:

```
'$include "BSL.DH"
```

BSL.dh is a .txt format file in the Microsoft Dynamics SL Application Program directory that contains global declarations of structures and variables that can be used in a BSL program. These are provided so that return values for functions, for example, need not be declared explicitly by the BSL program (serr=, serr1=, etc.). The BSL.DH file also contains declarations of the functions and subroutines in the Microsoft Dynamics SL kernel (SWIMAPI.DLL) that can be used in a BSL program. All standard Visual Basic functions and routines (Format\$, Trim\$, etc.) which are documented in the Basic Script Language Reference of this manual and those functions explicitly declared in BSL.DH can be used in a BSL program. Note that the declarations of the functions and subroutines in BSL.DH provide syntax on the function or the subroutine's usage. Questions on the usage syntax of any kernel function and the specific constants defined is answered by referencing BSL.DH.

An external file can be created that contains application-specific functions, constants, and subroutines (in text format) and included in the same fashion as BSL.DH. You may also create your own functions in a DLL and declare them in the same manner as kernel functions in BSL.DH. These external files can be maintained in your favorite editor. If you know the Microsoft Dynamics SL object name, you need not add any code in the event window for the object. For example, if you have created a routine that you want to have called by ccustid_Chk, you can name this routine ccustid_Chk. When you include this file, this routine is automatically called. The only two exceptions to this are that you cannot have Update nor Delete as a Sub name. These are Visual Basic reserved words. In these two cases, you would need to have code in the screen call your external routines.

Any time you modify code in your external file, you must "force" Customization Manager to recompile your code. A simple technique to "force" this compilation is to simply add an extra comment or carriage return in the General_Declarations Event code window, click on OK, and save the customization. Note that any customization that is imported which contains BSL code is automatically compiled at import time.

Levels and Events

Any time a new field is inserted onto a form, you must specify the Level associated with that field. A level identifies the logical association of the records and fields that are maintained by a particular application. For example, the Journal Entry screen (01.010.00) has two distinct levels: batch and detail. The batch level contains the fields associated with the batch table while the detail level contains fields associated with the GLTran (General Ledger Transactions) table. These levels are numbered starting at zero (0). Therefore, Batch is Level 0 and Transaction is Level 1.

These level numbers are very important any time the database is updated. The Microsoft Dynamics SL kernel executes the appropriate Update Event (Save, Finish, Insert, Delete) for as many times as there are levels for the particular application. In the example noted above (Journal Entry) the appropriate Update event executes twice, once for the Batch (Level 0) and once for the GLTran (Level 1) if fields in both levels require updating/inserting.

When attaching BSL code to the update event, you must determine which level number you wish to associate your code. Typically, this is performed by placing a Select Case Level at the beginning of the appropriate event. For example, the following code in 01.010 executes upon the appropriate Level in the Update event:

```
Select Case Level
  Case 0 'Batch
    Call MsgBox("Batch Updated", MB_OK, "Message")
  Case 1 'Transaction
    Call MsgBox("Grid Updated", MB_OK, "Message")
End Select
```

Message Functions

There are several message functions that a BSL program can reference. Use of these functions eliminates the necessity of declaring variables for holding message strings. For example, in standard Visual Basic, you must declare 3 variables for holding the message string, type, and title. You can create a message, assign it a Msg_ID and Msg_Type of 1, and add it to the message table. This way, you can eliminate the need to declare these extra variables and reference the message number in the mess statement call. This is also very handy when you need to use the same message in multiple programs or routines. Note that you can also use any of the existing messages with type Msg_Type = 1 as long as you provide the appropriate message number and applicable substitution strings.

Make sure that if you do create a new message, you assign a very high number to it so that Microsoft Dynamics SL developers do not accidentally overwrite it in the future. The Msg_Id field in this table is an integer so you may start messages in the 30000 range and probably be safe. You can also pass additional substitution parameters to messages.

Sample Date Functions

This sample routine sets an existing date field to the system date:

```
Sub setdate_Click()
    Dim NewDate as Sdate
    Dim NewStrDate$
    Call GetSysDate(NewDate)
    NewStrDate = DateToStr(NewDate)
    serr1 = SetObjectValue("cpaydate", NewStrDate)
    Call DispFields("Form1", "cpaydate")
End Sub
```

This example compares two dates. Note that you must include the .Val in the variable names where appropriate:

```
Sub Test_Click()
    Dim TestDate1 As Sdate
    Dim TestDate2 As Sdate
    TestDate1.Val = GetObjectValue("cinvdate")
    TestDate2.Val = GetObjectValue("cdocdate")
    serr1 = DateCmp(TestDate1, TestDate2)
    If serr1 = 0 Then
        Call MsgBox("Dates are equal", MB_OK, "Message")
    ElseIf serr1 > 0 Then
        Call MsgBox("invdate greater", MB_OK, "Message")
    ElseIf serr1 < 0 Then
        Call MsgBox("docdate greater", MB_OK, "Message")
    End If
End Sub
```

This sample uses the BSL WeekDay function:

```
Dim TestDate1 As Sdate
TestDate1.Val = GetObjectValue("cinvdate")
serr1 = WeekDay(TestDate1.Val)
Select Case serr1
    Case 1
        Call MsgBox("Sunday", MB_OK, "Message")
    Case 2
        Call MsgBox("Monday", MB_OK, "Message")
    Case 3
        Call MsgBox("Tuesday", MB_OK, "Message")
    Case 4
        Call MsgBox("Wednesday", MB_OK, "Message")
    Case 5
        Call MsgBox("Thursday", MB_OK, "Message")
    Case 6
        Call MsgBox("Friday", MB_OK, "Message")
    Case 7
        Call MsgBox("Saturday", MB_OK, "Message")
End Select
```

Structures

You cannot simply declare a variable, issue an SQL statement, and fetch the result of the SQL statement into this variable. You must declare a structure for ANY results of a SQL statement (including aggregate functions). If you create special structures for aggregate functions or special select statements (select fld1, fld2 from record) (not recommended) you must include all fields in the select statement AND any fields in the restriction not included in the select list.

For example, the following select statement:

```
Select CustId, Name, Zip From Customer Where  
City = 'Findlay' and Zip = '45840' and CurrBal > 0;
```

would require the following structure definition:

```
Type Result  
  CustId As String * 10  
  NameAs String * 30  
  ZipAs String * 10  
  CityAs String * 30  
  CurrBal As Double  
End Type  
Global bCustResult As Result, nCustResult As Result
```

bCustResult is used in all sfetch calls.

The following aggregate function:

```
Select CustId, Sum(OrigDocAmt) From ARDoc  
Group By CustId;
```

would require the following structure definition:

```
Type SumResult  
  CustId As String * 10  
  Amount As Double  
End Type  
Global bSumResult As SumResult
```

When using aggregate SQL functions as used in the second example above, you must use the sgroupfetch SQL functions.

Transactional Flow of User Fields

User fields in Microsoft Dynamics SL transaction detail tables automatically carry into GL if you post in detail. For example if you place ARTran.User1 in the detail area of Hand Prepared Invoices (08.010.00), the corresponding GLTran.User1 field is set to this same value at release time. This way, custom fields can be reported on in GL Transaction reports.

Treating Character Fields Like Date Fields

Even though string and float fields are the only user field types, you can still make a character field behave like a date field. When you insert either the User1 or User2 field onto a form, size it accordingly, then set the Mask property to 99/99/99. To handle error checking the following code must then be inserted:

To default the field to the current system date, the following default event must exist:

```
Sub cuser1_Default(newvalue$, retval%)
    Dim GetDate As Sdate
    Call GetSysDate(GetDate)
    newvalue = Mid$(datetostr(getdate), 1, 4) +
    Mid$(datetostr(getdate), 7, 2)
End Sub
```

This default event must be explicitly forced to execute by the following (Note that this would typically be placed in a preceding chk event):

```
serr = setdefaults("Form1", "cuser1")
```

To perform error checking on this field, place the following in its chk event:

```
Sub cuser1_Chk(chkstrg$, retval%)
    serr = DateCheck(chkstrg)
    If serr = -1 Then
        ' Invalid Day
        Call MsgBox("Invalid day, please re-enter", MB_OK, "Message")
        retval = errnomess
    ElseIf serr = -2 Then
        'Invalid Month
        Call MsgBox("Invalid month, please re-enter", MB_OK, "Message")
        retval = errnomess
    End If
End Sub
```

Using SQL Statements

Using BSL requires knowledge of the SQL syntax and its usage in the database and Microsoft Dynamics SL kernel. For SQL Statement creation and syntax, refer to the appropriate SQL database documentation that is provided with your SQL software.

There are two methods for retrieving records using the BSL `sqlfetch()` functions. You may declare a string variable that stores the SQL statement and use this variable in the `sqlfetch` call, or you can create an SQL stored procedure and reference the stored procedure name in the `sqlfetch` call. Using a stored procedure is much simpler because it minimizes the amount of BSL code and makes maintaining the SQL statement much easier if you want to use the same SQL statement in several places. To do this you only need to change the stored procedure and not the Visual Basic code. For example, the following two uses of `sqlfetch1` accomplish identical tasks but the latter uses much less code:

Without stored procedure:

```
Dim SqlStr$
SqlStr = "Select * from Customer where CustId =" + sparm(chkstrg) + "Order By CustId"
serr1 = sqlfetch1(c1, SqlStrg, bCustomer, Len(bCustomer))
```

With stored procedure:

```
Create Procedure GetCustomer @parml AS
Select * from Customer where CustId = @parml Order By CustId;
```

In Code:

```
serr1 = sqlfetch1(c1, "GetCustomer" + sparm(chkstrg), bCustomer, Len(bCustomer))
```

You may also use stored procedures for executing INSERT, DELETE, and UPDATE statements with parameters.

Working with Grids

You can add additional functionality to the grid object. The Spread1 object has the LineGotFocus and LineChk events exposed. You can evaluate under program control what happens when the row receives focus (LineGotFocus) and when the user leaves the row (LineChk). Following are some BSL examples:

This customization was designed for 01.010 but should work with any grid screen for which you want to default a value in the grid to the value of the same field of the previous line.

```
(general) Declarations
Global PriorValue$

Sub Insert(level%, retval%)
  If Level = 0 Then
    PriorValue = ""
  End If
End Sub

Sub cuser1_Chk(chkstrg$, retval%)
  PriorValue = chkstrg
End Sub

Sub cuser1_Default(newvalue$, retval%)
  If Trim$(PriorValue) <> "" Then
    newvalue = PriorValue
  End If
End Sub

Sub spread1_LineGotFocus(maintflg%, retval%)
  Dim CheckedValue$
  If maintflg = INSERTED Then
    CheckedValue = GetObjectValue("cuser1")
    If Trim$(CheckedValue) = "" Then
      serr1 = setdefaults("Form1","cuser1")
    End If
  End If
End Sub
```

This example uses 01.010 to automatically create an auto-balancing Journal entry.

Place Batch.User3 on bottom of Form. This is used to store the plug amount.

```
General Declarations
Global Diff#

Sub CreatePlug()
  CrTotal = Val(GetObjectValue ("ccrtot"))
  DrTotal = Val(GetObjectValue ("cdrtot"))
  Diff = FPSub(DrTotal, CrTotal, 2)
  serr1 = SetObjectValue ("cuser3", Str$(Diff))
End Sub

Sub cdramt_Chk(chkstrg$, retval%)
```

```
    Call CreatePlug
End Sub

Sub ccramt_Chk(chkstrg$, retval%)
    Call CreatePlug
End Sub

Sub spread1_LineChk(action%, maintflg%, retval%)
    Call CreatePlug
End Sub

Sub cacct_Chk(chkstrg$, retval%)
    Dim PlugAmount As Double
    If Trim$(chkstrg) = "0000" Then
        ' Plug the Offset
        PlugAmount = GetObjectValue ("cuser3")
        If PlugAmount > 0 Then
            Serr1 = SetObjectValue ("ccramt", Str$(PlugAmount))
        ElseIf PlugAmount < 0 Then
            Serr1 = SetObjectValue ("cdramt",
                Str$(-1 * PlugAmount))
        End If
    End If
End Sub
```

This example accumulates the total transaction amount in Voucher Entry (03.010.00) and defaults the transaction amount for each line to the amount required to "Balance" the document to the details.

Place following 3 lines in General Declarations

```
Dim Original#
Dim Sum#
Dim LineAmount#

Sub spread1_linegotfocus(maintflg%, retval%)
    Original = Val (GetObjectValue ("cOrigDocAmt"))
    If maintflg = INSERTED and Original <> 0 Then
        Sum = Val (GetObjectValue ("cDocBal"))
        LineAmount = FPSub (Original, Sum, 2)
        Serr1 = SetObjectValue ("cTranAmt", Str$(LineAmount))
    End If
End Sub
```

Parameter Passing Methods

The Microsoft Dynamics SL kernel provides the ability to send and receive parameters between screens and reports, customizations that are created with the Basic Script Language (BSL), and custom applications developed with the Microsoft SL SDK. Parameter passing allows a developer to “call” a custom application or a Microsoft Dynamics SL screen or report from either another Microsoft Dynamics SL screen or a custom screen created with Microsoft SL SDK.

Overview

Because each Microsoft Dynamics SL or Microsoft SL SDK-developed application is a separate executable program (.exe), command parameters can be easily passed between one application and another. This allows the developer to create a single solution that is comprised of two separate applications. It also allows the developer to implement custom “Quick Print” or “Drill Down” functionality by calling the appropriate application EXE or the Report Option Interpreter (ROI) with the specified parameters.

The architecture to support parameter passing between applications is a “layered” design so that parameters that are passed between applications using different methods remain isolated. This prevents parameters that are in use by two Microsoft SL SDK-developed applications from being compromised if a customization using BSL uses different parameters for the same applications. Transaction Import starts any target application screen via command line parameters and therefore must keep its own parameters isolated so that any parameters used by BSL or the underlying Microsoft SL SDK application are also not compromised.

The following Microsoft Dynamics SL kernel parameter passing functions are available in BSL and Microsoft SL SDK:

Kernel function	Where available	When used
CallApplic	Microsoft SL SDK	To launch a second application, where the launched application is <i>not</i> modal
CallApplicWait	Microsoft SL SDK	To launch a second application, where the launched application <i>is</i> modal
AppIGetParms	Microsoft SL SDK/BSL	To retrieve parameters passed by another application via the command line
Launch	BSL	To launch another application
AppIGetReturnParms	Microsoft SL SDK	To retrieve parameters from a terminated secondary Microsoft SL SDK application
AppIGetParmValue	Microsoft SL SDK/BSL	To retrieve parameters from the parameter file that were passed from another application via the AppISetParmValue function
AppISetParmValue	Microsoft SL SDK/BSL	To add parameters that will be sent to an application via CallApplic, CallApplicWait, or Launch.

These functions allow parameters to be processed separately for Microsoft SL SDK, BSL, and Transaction Import (TI) and allow a program to access any parameter type from any other application (Microsoft SL SDK, BSL, TI).

For a detailed description of each of the Microsoft SL SDK functions, including the appropriate syntax, see “API Function Calls

” on page 315.

Temporary Parameter File

A temporary file, known as the parameter file (.prm file name extension), is automatically created by the Microsoft Dynamics SL kernel and contains any values that are to be passed to the program just loaded. This file is similar in format to a Windows .ini file and contains sections, entries, and values. Upon the loading of a Microsoft SL SDK application this uniquely named file is created and is deleted after the called application has completed. This file is created in either the Microsoft Dynamics SL program directory or the directory specified by the TempDirectory entry in the [Miscellaneous] section of the Solomon.ini file.

The following entry in the Solomon.ini file illustrates specifying the C:\TEMP directory as the location where parameter files are created:

```
[Miscellaneous]
TempDirectory=C:\TEMP
```

Traditional parameter passing uses a single command line string. The CallApplic, CallApplicWait, and Launch functions all support the ability to specify a single EXE name with parameters and ApplGetParms can retrieve any parameter passed via the command line. However, due to an operating system limitation of 128 characters allowed in a single command line, certain scenarios may exceed this limit. Therefore, AppISetParmValue, ApplGetParmValue and the temporary parameter file are provided so that you can write, read, and pass parameters of any length.

Building and Retrieving Parameters

Two statements are used to build and retrieve parameters: AppISetParmValue and ApplGetParmValue.

AppISetParmValue

This statement sets the destination parameter values to be passed to **CallApplicWait**, **CallApplic**, or **Launch**. It enables you to specify the exact section, entry, and value in the parameter file.

The first execution of this statement creates the temporary destination parameter file and places the first set of values in that file. Subsequent executions of this statement within the same program are added to this destination parameter file.

When **CallApplicWait**, **CallApplic**, or **Launch** is executed, the completed destination parameter file name is passed in the command line of the called application.

ApplGetParmValue

This statement obtains the parameter values for the current application from the parameter file. This statement enables you to specify the exact section, entry, and value in the parameter file that you set with **AppISetParmValue**. This statement is called in the Load event of the target application.

Parameter Passing Example - Printing Reports

When using `Launch` or `CallApplicWait` to execute ROI, the parameter file is automatically created and used by ROI to interpret the appropriate options. You need not write any code to retrieve parameters that are sent to ROI.

The following example prints the Vendor List for a specific classification of vendors whose balance is greater than zero. The result of the execution of any of these examples is the creation of a parameter file that contains separate entries for each parameter:

```

` The Report Name
Call ApplSetParmValue("", "", "03670/RUN")
` The Format Name
Call ApplSetParmValue("", "", "03670S/FORMAT")
` The WHERE clause restriction
Call ApplSetParmValue("", "", "vr_03670s.currbal > 0 And vr_03670s.vendid Like
'_____1'/WHERE")
` Print the report to the screen
Call ApplSetParmValue("", "", "/PSCRN")
` Launch ROI with appropriate parameters
serr1 = Launch("ROI ", True, True, 0)

```

BSL Report Example: Simple WHERE Clause

```

Call ApplSetParmValue("", "", "03670/RUN")
Call ApplSetParmValue("", "", "03670S/FORMAT")
Call ApplSetParmValue("", "", " vr_03670s.currbal > 0 And vr_03670s.vendid Like
'_____1'/WHERE")
Call ApplSetParmValue("", "", "/PSCRN")
serr1 = Launch("ROI ", True, True, 0)

```

BSL Report Example: Complex WHERE Clause

```
' Note that you are still limited to WHERE clause size of 255 (an operating system
limit)
Call ApplSetParmValue("", "", "03670/RUN")
Call ApplSetParmValue("", "", "03670S/FORMAT")
Call ApplSetParmValue("", "", "vr_03670s.vendid = 'V00100' or vr_03670s.vendid =
'V00101' or vr_03670s.vendid = 'V00102' or vr_03670s.vendid = 'V00201'/WHERE")
Call ApplSetParmValue("", "", "/PSCRN")
serr1 = Launch("ROI ", True, True, 0)dd
```

While this report is displayed to screen, the parameter file can be examined for its contents. For this example, it would contain the following:

```
[VBRDT]
REMOVEFILE=Y
SETPARMV=6
PRM01=4
PRM02=28584
PRM03=03670/RUN
PRM04=03670S/FORMAT
PRM05=Vendor.VendId = 'V00100' or Vendor.VendId = 'V00101' or Vendor.VendId = 'V00102'
or Vendor.VendId = 'V00201'/WHERE
PRM06=/PSCRN
[BSL]
SETPARMV=2
```

Parameter-Passing Example – Sending Parameters

The following examples create a new section, entry, and value in a parameter file before executing the Customer Maintenance screen:

```
Call ApplSetParmValue("[MyScreen]", "CustomerId", "C299")
serr1 = Launch("0826000 ", True, True, 0)
```

After this code has executed, the following entry is written to the parameter file:

```
[MyScreen]
CustomerId=C299
```

Parameter-Passing Example – Receiving Parameters

The following BSL code in the 08.260 Customer screen's Form1_Display event intercepts the parameters from the appropriate sections and entries of the parameter file:

```
Dim PassedParm As String
PassedParm = ApplGetParmValue("[MyScreen]", "CustomerId")

If Trim$(PassedParm) <> "" Then
    ' only set field value if parameter is not blank
    serr1 = SetObjectValue("ccustid", PassedParm)
End If
```

Note: You need not be concerned about creating or deleting the parameter file, since the Microsoft Dynamics SL kernel does that.

Using Parameter Files with ApplGetParmValue

The ApplGetParmValue statement retrieves parameters from the parameter file. This file is created when the ApplSetParmValue statement is called. The kernel automatically determines the name to

make this parameter file (the extension is always PRM). There may be cases when the developer desires the ability to assign a name to this parameter file and create it with a non-Microsoft Dynamics SL application. The developer may determine whether the parameter file remains on the disk or is deleted. This section explains how to create a parameter file and pass this file name to a Microsoft SL SDK application.

The following BSL code can be placed in the Form1_Display event of *Vendor* (03.270.00):

```
Dim PassedParm As String
PassedParm = ApplGetParmValue("[MySection]", "Id")

If Trim$(PassedParm) <> "" Then
    serr1 = SetObjectValue("cvendid", PassedParm)
End If
```

For this code to retrieve the value of the Id entry from MySection of the parameter file, a parameter file must first be constructed. The ApplSetParmValue function, when called from either a BSL or Microsoft SL SDK application, will create a temporary parameter file. However, an application that launches the Vendor (03.270.00) screen may be neither a Microsoft SL SDK nor a BSL application.

Use a text editor, such as Microsoft Notepad, to create a file called MYFILE.TXT in the Microsoft Dynamics SL program directory. In this file, place the following:

```
[VBRDT]
REMOVEFILE=N

[MYSECTION]
ID=V00100
```

Note: The VBRDT section contains an entry that causes this parameter file to not be deleted when the Vendor (03.270.00) screen is closed. Under normal circumstances, you would want this file removed then. Setting REMOVEFILE=Y will accomplish this.

The following techniques describe the various methods to launch the Vendor (03.270.00) screen with the appropriate parameter file so that vendor V00100 is automatically loaded:

Create a shortcut on your desktop and specify "<Microsoft Dynamics SL Program directory>\AP\0327000 \t @MYFILE.TXT" as the command line and <Microsoft Dynamics SL Program directory> as the Start In directory. The \t is required because a tab character must separate parameters, as is evidenced by the PRMSEP constant being set to \t in BSL.DH.

Note: If you will be launching a Microsoft Dynamics SL application using this technique, you need to ensure from within your application that the working directory is the Microsoft Dynamics SL Program directory, otherwise, you receive the "Unable to execute MSDynamicsSL.exe file not found" error.

From an existing Microsoft Dynamics SL application, place the following event code in the Click event of a pushbutton object:

```
serr1 = Launch("0327000 " + PRMSEP + "@MYFILE.TXT", True, True, 0)
```

From a Microsoft SL SDK application, place the following in the Click event of a pushbutton object:

```
Call CallApplicWait("0327000" + PRMSEP + "@MYFILE.TXT")
```

Note: When the kernel creates a parameter file, a unique parameter file name is always created to facilitate multi-user use. When a defined parameter file is used, as in this example, the developer is responsible for any concurrency issues.

How BSL Compares to Other Versions of Basic

Differences Between BSL and Earlier Versions of Basic

If you are familiar with older versions of Basic (those that predate Windows), you will notice that BSL includes many new features and changes from the language you have learned. BSL more closely resembles other higher level languages popular today, such as C and Pascal.

The topics below describe some of the differences you will notice between the older Basics and BSL.

Line Numbers and Labels

Older versions of Basic require numbers at the beginning of every line. More recent versions do not support these line numbers; in fact, they will generate error messages.

If you want to reference a line of code, you can use a label. A label can be any combination of text and numbers. Usually, it is a single word followed by a colon, which is placed at the beginning of a line of code. These labels are used by the **Goto** statement.

Subroutines and Modularity of the Language

BSL is a modular language; code is divided into subroutines and functions. The subroutines and functions you write use the BSL statements and functions to perform actions.

Global Variables

The placement of variable declarations determines their scope:

Scope	Definition
Local	Dimensioned inside a subroutine or function. The variable is accessible only to the subroutine or function that dimensioned it.
Module	Dimensioned outside any subroutine or function. The variable is accessible to any subroutine or function in the same file.
Global	Dimensioned outside any subroutine or function using the Global statement. The variable is accessible to any subroutine or function in any module (file).

Data Types

Modern Basic is now a typed language. In addition to the standard data types—numeric, string, array, and record—BSL includes variants and objects.

Variables that are defined as variants can store any type of data. For example, the same variable can hold integers one time, and then, later in a procedure, it can hold strings.

Objects give you the ability to manipulate complex data supplied by an application, such as windows, forms or OLE2 objects.

Dialog Box Handling

BSL contains extensive dialog box support to give you great flexibility in creating and running your own custom dialog boxes. You define a dialog box with dialog control statements between the **Begin Dialog...End Dialog** statements, and then display it using the **Dialog** statement (or function).

BSL stores information about the selections the user makes in the dialog box. When the dialog box is closed, your program can access this information.

BSL also includes statements and functions to display other types of boxes.

- **Message boxes** — Notify the user of an event
- **Password boxes** — Do not echo the user's keystrokes on the screen
- **Input boxes** — Prompt for a single line of input

Financial Functions

BSL includes a list of financial functions, for calculating such things as loan payments, internal rates of return, or future values based on a company's cash flows.

Date and Time Functions

The date and time functions have been expanded to make it easier to compare a file's date to today's date, set the current date and time, time events, and perform scheduling-type functions (such as finding the date for next Tuesday).

Object Handling

Windows includes OLE2 Object Handling, the ability to link and embed objects from one application into another. An object is the end product of a software application, such as a document from a word processing application. An offshoot of that ability is the **Object** data type that permits your BSL code to access another software application through its objects and change those objects.

Environment Control

BSL includes the ability to call another software application (**AppActivate**), and send the application keystrokes (**SendKeys**). Other environment control features include the ability to run an executable program (**Shell**), temporarily suspend processing to allow the operating system to process messages (**DoEvents**), and return values in the operating system environment table (**Environ\$**).

Differences Between BSL and Visual Basic

BSL is very similar to Microsoft's Visual Basic; however, there are some differences.

Functions and Statements Unique to BSL

BSL offers a few statements and functions not found in Visual Basic:

- \$CStrings
- \$Include
- \$NoCStrings
- Assert
- GetField\$
- SetField\$

Control-Based Objects

BSL does not predefine or include any Visual Basic object, such as a Button Control. As a result, a Visual Basic property such as "BorderStyle" is not an intrinsic part of BSL. This does not mean that as an integrator, you cannot define a BSL object that has BorderStyle as a property. You will probably define many objects that are intrinsic to your application in the process of integration.

Dialog Box Capabilities and VBA

Visual Basic does not have a syntax to create or run dialog boxes. In contrast, BSL has a set of functions and statements to enable the use of dialog boxes (they are similar to those in Word).

Microsoft offers a modified version of Visual Basic in some of its products, such as Excel. Called Visual Basic for Applications (VBA), this version does provide dialog box handling statements and functions.

Differences Between BSL and Word Basic

Word Basic is a precursor to Visual Basic that is included in Microsoft Word. Word Basic supports dialog boxes, but it does not support objects.

Dialog Box Capabilities

The dialog box capabilities in BSL and Word are very similar. Word does offer some statements and functions that BSL does not, such as DigFilePreview.

In response to the need for certain types of dialog box support, BSL offered some dialog box options before Word Basic did. Later, Word Basic came out with their own syntax for these options. As a result, there are minor differences in the way the two languages handle dialog boxes.

Button vs. PushButton

Button is the original BSL syntax; PushButton is the Word Basic syntax. The two are interchangeable, and BSL supports both.

PushButton is preferred, and is used throughout the Examples.

Dialog Box Units

The measurement units used in the two dialog box syntaxes are different. BSL supports both, and you can choose to use either.

Since many of our clients have built scripts based on the original BSL units, those are the ones used in the Examples. As a result, if you use Word units, some of the dialog boxes created in the Examples might look odd.

User Input Mechanisms

There are slight differences in some of the mechanisms for user input:

BSL	Word Basic
StaticComboBox or ComboBox (interchangeable in BSL)	ComboBox (Word Basic supports only this syntax)
DropComboBox	N/A

Appendix 1: Trappable Errors

Error Codes

The following table lists the runtime errors that BSL 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
70	Permission denied
71	Disk not ready
74	Can't rename with different drive
75	Path/File access error
76	Path not found
91	Object variable set to Nothing
93	Invalid pattern
94	Illegal use of NULL
102	Command failed
429	Object creation failed
438	No such property or method
439	Argument type mismatch
440	Object error
901	Input buffer would be larger than 64K

Error code	Error text
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

Appendix 2: Derived Trigonometric Functions

Derived Trigonometric Functions

A number of trigonometric functions can be written in Basic using the built-in functions. The following table lists several of these functions:

Function	Computed by...
Secant	$\text{Sec}(x) = 1/\text{Cos}(x)$
CoSecant	$\text{CoSec}(x) = 1/\text{Sin}(x)$
CoTangent	$\text{CoTan}(x) = 1/\text{Tan}(x)$
ArcSine	$\text{ArcSin}(x) = \text{Atn}(x/\text{Sqr}(-x*x+1))$
ArcCosine	$\text{ArcCos}(x) = \text{Atn}(-x/\text{Sqr}(-x*x+1))+1.5708$
ArcSecant	$\text{ArcSec}(x) = \text{Atn}(x/\text{Sqr}(x*x-1))+\text{Sgn}(x-1)*1.5708$
ArcCoSecant	$\text{ArcCoSec}(x) = \text{Atn}(x/\text{Sqr}(x*x-1))+(\text{Sgn}(x)-1)*1.5708$
ArcCoTangent	$\text{ArcTan}(x) = \text{Atn}(x)+1.5708$
Hyperbolic Sine	$\text{HSin}(x) = (\text{Exp}(x)-\text{Exp}(-x))/2$
Hyperbolic Cosine	$\text{HCos}(x) = (\text{Exp}(x)+\text{Exp}(-x))/2$
Hyperbolic Tangent	$\text{HTan}(x) = (\text{Exp}(x)-\text{Exp}(-x))/(\text{Exp}(x)+\text{Exp}(-x))$
Hyperbolic Secant	$\text{HSec}(x) = 2/(\text{Exp}(x)+\text{Exp}(-x))$
Hyperbolic CoSecant	$\text{HCoSec}(x) = 2/(\text{Exp}(x)-\text{Exp}(-x))$
Hyperbolic Cotangent	$\text{HCotan}(x) = (\text{Exp}(x)+\text{Exp}(-x))/(\text{Exp}(x)-\text{Exp}(-x))$
Hyperbolic ArcSine	$\text{HArcSin}(x) = \text{Log}(x+\text{Sqr}(x*x+1))$
Hyperbolic ArcCosine	$\text{HArcCos}(x) = \text{Log}(x+\text{Sqr}(x*x-1))$
Hyperbolic ArcTangent	$\text{HArcTan}(x) = \text{Log}((1+x)/(1-x))/2$
Hyperbolic ArcSecant	$\text{HArcSec}(x) = \text{Log}((\text{Sqr}(-x*x+1)+1)/x)$
Hyperbolic ArcCoSecant	$\text{HArcCoSec}(x) = \text{Log}((\text{Sgn}(x)*\text{Sqr}(x*x+1)+1)/x)$
Hyperbolic ArcCoTangent	$\text{HArcCoTan}(x) = \text{Log}((x+1)/(x-1))/2$

Glossary

BSL

The acronym for the Basic Script Language (BSL).

BSL.DH

A text format file in the Microsoft Dynamics SL program directory that contains global and function declarations for all of the kernel functions that can be used in Basic Script Language.

call by reference

Arguments passed by reference to a procedure can 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 argument 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 can receive their arguments by value.

comment

A comment is text that documents a 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 is interpreted as a metacommand. Lines beginning with the keyword **Rem** are also interpreted as comments.

control ID

This can be either a text string, in which case it is the name of the control, or it can be a numeric ID. Note that control IDs are case-sensitive and do not include the dot that appears before the ID. Numeric IDs depend on the order in which dialog controls are defined. You can find the numeric ID using the **DlgControlID** function.

dialog control

An item in a dialog box, such as a list box, combo box, or command button.

function

A procedure that 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.

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.

metacommand

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

name

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 more than 40 characters in length. Type characters are not considered part of a name.

precedence order

The system BSL uses to determine which operators in an expression to evaluate first, second, and so on. Operators with a higher precedence are evaluated before those with lower precedence. Operators with equal precedence are evaluated from left to right. The default precedence order (from high to low) is: numeric, string, comparison, logical.

procedure

A series of BSL statements and functions executed as a unit. Both subprograms (**Sub**) and functions (**Function**) are called procedures.

subprogram

A procedure that does not return a value.

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 precision floating point
- # Double precision floating point
- @ Currency exact fixed point

vartype

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

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

Index

\$

\$CStrings 63
 \$Include 168
 \$NoCStrings 214

A

Abs 29
 AppActivate 30
 Asc 31
 Assert 31
 Atn 32

B

Beep Statement 33
 Begin Dialog...End Dialog Statement 34
 Button 38
 ButtonGroup 39

C

Call 40
 CancelButton 42
 Caption 43
 CCur 44
 CDbl 45
 ChDir 46
 ChDrive 47
 CheckBox 48
 Chr 50
 Clnt 51
 Clipboard 52
 CLng 53
 Close 54
 ComboBox 55
 Command 57
 Const 58
 Cos 59
 CreateObject 60
 CSng 61
 CStr 62
 CurDir 64
 CVar 65
 CVDate 66

D

Date 67
 Date Statement 68
 DateSerial 69
 DateValue 70
 Day 71
 DDEAppReturnCode 71
 DDEExecute 72

DDEInitiate 74
 DDEPoke 76
 DDERequest 78
 DDETerminate 80
 Declare 81
 Deftype 83
 Derived Trigonometric Functions 477
 Dialog Boxes 84, 85
 Dialog Function 84
 Dialog Statement 85
 Dim 86
 Dir 89
 DlgControlID Function 91
 DlgEnable Function 94
 DlgEnable Statement 96
 DlgEnd Statement 98
 DlgFocus Function 100
 DlgFocus Statement 101
 DlgListBoxArray Function 102
 DlgListBoxArray Statement 104
 DlgSetPicture Statement 106
 DlgText Function 108
 DlgText Statement 110
 DlgValue Function 112
 DlgValue Statement 114
 DlgVisible Function 116
 DlgVisible Statement 117
 Do...Loop 119
 DoEvents 120
 DropComboBox 121
 DropListBox 123

E

Environ 125
 Eof 126
 Erase 127
 Erl 129
 Err Function 130
 Err Statement 131
 Error Function 132
 Error Statement 133
 Exit 134
 Exp 135

F

FileAttr 136
 FileCopy 137
 FileDateTime 138
 FileLen 139
 Files
 parameter 466
 Fix 140
 For...Next 141
 Format 143
 FreeFile 149
 Function...End Function 150
 Functions
 kernel 465
 FV 152

G

Get 153
GetAttr 155
GetField 157
GetObject 158
Global 160
GoTo 163
GroupBox 164

H

Hex 165
Hour 166

I

If...Then...Else 167
Input Function 169
Input Statement 170
InputBox 172
InStr 173
Int 175
IPmt 176
IRR 177
Is 178
Is Operator 178
Is_TI 368
IsDate 179
IsEmpty 180
IsMissing 181
IsNull 182
IsNumeric 183

K

Kernel functions 465
Kill 184

L

LBound 186
LCase 187
Left 188
Len 189
Let 190
Like 191
Line Input 192
ListBox 193
Loc 195
Lock 196
Lof 198
Log 199
Lset 200
LTrim 201

M

Me 202
Methods

parameter passing 465
Mid Function 203
Mid Statement 204
Minute 205
Mkdir 206
Month 207
Msgbox Function 208
Msgbox Statement 210

N

Name 212
New 213
New Operator 213
Nothing 215
Now 216
NPV 217
Null 218

O

Object Class 219
Oct 220
OKButton 221
On Error 223
On Goto 222
Open 225
Option Base 229
Option Compare 230
Option Explicit 231
OptionButton 227
OptionGroup 228

P

Parameter file 466
Parameters
 passing 465
 passing methods 465
 sending/receiving 465
PasswordBox 232
Picture 233
Pmt 234
PPmt 235
Print 236
PRM file 466
PushButton 237
Put 238
PV 240

R

Randomize 241
Rate 242
ReDim 244
Rem 246
Reset 247
Resume 248
Right 249
Rmdir 250
Rnd 251
Rset 252

RTrim 253

S

Second 254
Seek Function 255
Seek Statement 256
Select Case 258
SendKeys 260
Set 262
SetAttr 264
SetField 266
Sgn 267
Shell 268
Sin 269
Space 270
Spc 271
SQLClose 272
SQLError 273
SQLExecQuery 274
SQLGetSchema 275
SQLOpen 277
SQLRequest 278
SQLRetrieve 279
SQLRetrieveToFile 281
Sqr 282
Static 283
StaticComboBox 284
Stop 285
Str 286
StrComp 287
String 288
Sub...End Sub 289

T

Tab 290
Tan 291
Temporary parameter file 466
Text 292
TextBox 293
Time Function 294
Time Statement 295
Timer 296
TimeSerial 297
TimeValue 298
Trappable Errors 475
Trim 299
Type 300
Typeof 301

U

UBound 302
UCase 303
Unlock 304

V

Val 305
VarType 306

W

Weekday 308
While...Wend 309
Width 311
With 312
Write 313

Y

Year 314