Reference

Microsoft® VISUAL BASIC™

Programming System for MS-DOS®

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software, which includes information contained in any databases, described in this document is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose, without the express written permission of Microsoft Corporation.

© 1992 Microsoft Corporation. All rights reserved. Printed in the United States of America.

Microsoft MS-DOS, GW-BASIC, and CodeView are registered trademarks and Microsoft QuickBasic, Visual Basic, and Windows are trademarks of Microsoft Corporation.

Btrieve is a registered trademark of SoftCraft, Inc., a Novell Company.

Hercules is registered trademark of InColor and is a trademark of Hercules Computer Technology.

Olivetti is a trademark of Ing. C. Olivetti.

This manual was produced using Microsoft Word for Windows.

Document Number DB27592-0792

Contents

Introduction v

Part 1 Language Summary

Part 2 A-Z Reference

Part 3 Appendixes

Appendix A ASCII and Scan Code Tables 437

Appendix B Reserved Words 441

Appendix C Data Types 445

Appendix D Operators 449

Appendix E Error Messages 453

Introduction

This manual is divided into three parts:

- Part 1, "Language Summary," provides a quick reference for looking up related functions, statements, methods, properties, events, and objects.
- Part 2, "A-Z Reference," is an alphabetic reference for each function, statement, method, property, event, and object.
- Part 3, "Appendixes," contains information on the ASCII character set, reserved words, data types, operators and error messages.

The information in this manual is the best available at the time of publication. In some cases, more up-to-date information may be available in online Help.

Document Conventions

This manual uses the following typographic conventions.

Example of convention	Description
SUB, IF, CHDIR, PRINT, TIME$	In Syntax and tables, words bold and upper-cased indicate language-specific reserved words.
CONSTANT.BI, F1_KEY	Uppercase words indicate filenames, constants, or keywords.
BackColor, Click	Names of properties and events appear with initial letter capitalized.
setup, Form_Click, PRINT	Words you're instructed to type appear in bold.
ctlname, setting	In syntax, italic letters indicate placeholders for information you supply.
[*expressionlist*]	In syntax, items inside square brackets are optional.
{**WHILE**|**UNTIL**}	In syntax, braces and a vertical bar indicate a mandatory choice between two or more items. You must choose one of the items unless all of the items also are enclosed in square brackets.

```
SUB HelloButton_Click ()
  Readout.Text = "Hello, world!"
END SUB
```

This font is used for code and error messages.

```
Readout.Text = "This should all be
  ➥ on one line in the Code window."
```

The line-continuation character (➥) indicates that code continued from one line to the next in the book should be typed all on one line in the Code window.

ENTER

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R. (Key names in online Help have only the initial letter capitalized.) The key names correspond to the names on the IBM Personal Computer keyboard. Other machines may use different names.

ALT+F1

A plus sign (+) between key names indicates a combination of keys. For example, ALT+F1 means to hold down the ALT key while pressing the F1 key.

Multiple Document Interface (MDI)

Acronyms are usually spelled out the first time they are used.

Programming Style in This Manual

The following guidelines are used in writing programs in this manual:

- Keywords and constants appear in all capital letters:

```
' SUB, IF, CHDIR, PRINT, TIME$ are keywords.
PRINT "Title Page"

' TRUE and FALSE are constants
' equal to -1 and 0, respectively.
CONST TRUE = -1
CONST FALSE = 0
```

- Variable names and properties are lowercase with an initial capital letter. Variable names with more than one syllable may contain other capital letters:

```
Gov AS STRING * 30
PerCapIncome AS DOUBLE
```

- An apostrophe (') introduces comments:

```
' This is a comment; these two lines
' are ignored when the program is running.
```

- Control-flow blocks and statements in **SUB** and **FUNCTION** procedures are indented from the enclosing code:

```
SUB Command1_Click
  IF X > 0 THEN
    Text1.Text = "X is greater than 0."
  END IF
END SUB
```

- As noted earlier, lines too long to fit on one line in this manual may be continued on the next line using a line-continuation character ( ).

```
SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE,
  ➥ Y AS SINGLE)
```

Language Summary

Functions, Statements, and Methods

Table 1 lists related functions, statements, and methods grouped by programming task. Items listed in a given category are often used together.

Table 2 lists the methods alphabetically and shows which objects apply to a given method.

Table 1 Functions, Statements, and Methods by Programming Task

Category	Action	Functions/statements/methods
Arrays	Declare and initialize array variables	DIM, REDIM, STATIC
	Reinitialize array variables	ERASE, REDIM, REDIM PRESERVE
	Find the limits of array	LBOUND, UBOUND
	Change default lower limit for arrays	OPTION BASE
Control program flow	Loop	DO...LOOP, FOR...NEXT, WHILE...WEND
	Branch	GOSUB...RETURN, GOTO, ON ERROR, ON...GOSUB, ON...GOTO
	Make decisions	IF...ENDIF, IFTYPEOF,, SELECT CASE
	Exit or pause the program	DOEVENTS, END, STOP, UNLOAD, SYSTEM, CHAIN, RUN
Conversion	ASCII value to string	CHR$
	String to ASCII value	ASC
	Number to string	STR$, FORMAT$
	String to number	VAL
	One numeric data type to another	CCUR, CDBL, CINT, CLNG, CSNG, FIX, INT
	Decimal numbers to other radix strings	HEX$, OCT$
	Date to serial number	DATESERIAL, DATEVALUE
	Serial number to date	DAY, MONTH, WEEKDAY, YEAR, NOW
	Time to serial number	TIMESERIAL, TIMEVALUE
	Serial number to time	HOUR, MINUTE, SECOND, NOW
Copy, Cut, and Paste	Use the CLIPBOARD object	CLEAR, GETTEXT, SETTEXT

Table 1 Continued

Category	Action	Functions/statements/methods
Date/time	Get the current date or time	DATE$, NOW, TIME$
	Set the date or time	DATE$, TIME$
	Time a process	TIMER
Graphics	Work with colors	QBCOLOR, POINT, RGB
	Draw shapes	LINE, CIRCLE, PSET
	Find size of text	TEXTHEIGHT, TEXTWIDTH
	Screen Images	GET, PUT
	Display text	PRINT
	Clear an object, screen, viewport	CLS
Error Trapping	Trap errors while a program is running	ON ERROR, RESUME
	Get error-status data	ERR, ERL
	Get error messages	ERROR$
	Simulate run-time errors	ERROR
File I/O	Access or create a file	OPEN
	Close files	CLOSE, RESET
	Write to a file	PRINT #, PUT, WRITE #
	Control output appearance	SPC, TAB, WIDTH#
	Read from a file	GET, INPUT #, INPUT$, LINE INPUT #
	Get information about a file	EOF, FILEATTR, FREEFILE, LOC, LOF, SEEK
	Set the read/write position in a file	SEEK
	Manage disk drives or directories	CHDRIVE, CURDIR$, CHDIR, MKDIR, RMDIR
	Manage files	DIR$, KILL, LOCK...UNLOCK, NAME

Table 1 Continued

Category	Action	Functions/statements/methods
Manipulating objects	Hide or show forms	SHOW, HIDE
	Load or unload objects	LOAD, UNLOAD
	Move or resize controls	MOVE
	Drag and drop	DRAG
	Direct user input to a control	SETFOCUS
	Update the display	REFRESH
	Print forms	PRINTFORM
	Work with list boxes and combo boxes	ADDITEM, REMOVEITEM
	Display dialog boxes	INPUTBOX$, MSGBOX
Math	General calculations	EXP, LOG, SQR
	Trigonometry	ATN, COS, SIN, TAN
	Numeric conversions	FIX, INT
	Get absolute value	ABS
	Get the sign of an expression	SGN
	Generate random numbers	RANDOMIZE, RND
Printing	Control printer	ENDDOC, NEWPAGE
	Control output appearance	SPC, TAB, TEXTHEIGHT, TEXTWIDTH
	Print	PRINT, PRINTFORM, LPRINT

Table 1 Continued

Category	Action	Functions/statements/methods
Procedures	Define a procedure	FUNCTION, SUB
	Call a SUB procedure	CALL
	Exit from a procedure	EXIT FUNCTION, EXIT SUB END FUNCTION, END SUB
	Declare a reference to an external procedure	DECLARE
Strings	Format strings	FORMAT$
	Create strings of repeating characters	SPACE$, STRING$
	Work with ASCII and ANSI values	ASC, CHR$
	Manipulate strings	INSTR, LEFT$, LTRIM$, MID$, RIGHT$, RTRIM$
	Convert to lowercase or uppercase letters	LCASE$, UCASE$
	Justify a string	LSET, RSET
	Find the length of a string	LEN
Variables and Constants	Declare variables or constants	CONST, DIM, STATIC, SHARED, COMMON
	Assign value	LET
	Set default data type	DEF*type*
	Create a user-defined data type	TYPE
Miscellaneous	Run other programs	SHELL, RUN, CHAIN
	Handle events	DOEVENTS
	Get command-line arguments	COMMAND$
	Find environment variables	ENVIRON$
	Sound a beep	BEEP

Table 2 Methods by Object

Methods / Objects	Check	CLIPBOARD	Combo	Command	Dir	Drive	File	Form	Frame	Label	List	Menu	Option	Picture	PRINTER	SCREEN	Scroll	Text	Timer
ADDITEM			●								●								
CLEAR		●																	
CLS								●						●					
DRAG	●		●	●	●	●	●		●	●	●		●	●			●	●	
ENDDOC															●				
GETTEXT		●																	
HIDE								●								●			
MOVE	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
NEWPAGE															●				
PRINT								●						●	●				
PRINTFORM								●											
REFRESH	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
REMOVEITEM			●								●								
SETFOCUS	●		●	●	●	●	●	●			●		●	●			●	●	
SETTEXT		●																	
SHOW								●								●			
TEXTHEIGHT								●						●					
TEXTWIDTH								●						●					

Properties

Table 3 categorizes properties by function and lists specific programming tasks within each category.

Table 4 lists properties alphabetically and shows which properties apply to a given object.

Table 3 Properties by Programming Task

Category	Action	Property
Appearance	Change color	BackColor, ForeColor
	Change border	BorderStyle
	Change pointer	MousePointer
	Hide or display	Visible
Files	Display filenames	Drive, FileName, Path, Pattern
	Hide or show files with these attributes	Archive, Hidden, Normal, ReadOnly, System

Table 3 Continued

Category	Action	Property
Forms and controls	Set default button	Cancel, Default
	Set timer interval	Interval
	Find parent form	Parent
	Display scroll bar	ScrollBars (text box only)
	Determine type of combo box	Style
	Set tab order	TabIndex, TabStop
	Check state	Value
	Refer to control	CtlName, Index
	Track focus	ActiveControl
	Manipulate list box	List, ListCount, ListIndex, Sorted
	Scrolling	Max, Min, LargeChange, SmallChange
	Control menu items	Enabled, Checked
	Respond to events	Enabled
	Include control menu box on form	ControlBox
	Minimize/maximize form	MaxButton, MinButton, WindowState
	Refer to form	FormName
	Drag and drop	DragMode
Graphics	Control repaint	AutoRedraw
	Resize to fit	AutoSize
	Access image	Picture
	Draw	CurrentX, CurrentY
Printing	Set page to print	NEWPAGE
Size and position	Set position	Left, Top
	Set size	Height, Width
Text	Align	Alignment
	Resize to fit	Autosize
	Display	Caption, Text
	Select	SelLength, SelStart, SelText

Table 4 Properties by Object

Key

- ● Read/write at run time
- ◐ Read-only at run time
- ○ Not available at run time

Properties / Objects	Check	CLIPBOARD	Combo	Command	Dir	Drive	File	Form	Frame	Label	List	Menu	Option	Picture	PRINTER	SCREEN	Scroll	Text	Timer
ActiveControl [1]																◐			
ActiveForm [1]																◐			
Alignment										●									
Archive							●												
Attached																	◐		
AutoRedraw								●						●					
AutoSize										●									
BackColor	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
BorderStyle								◐		●				●				●	
Cancel				●															
Caption	●			●				●	●	●		●	●						
Checked												●							
ControlBox								◐											
ControlPanel [1]																●			
CtlName	○		○	○	○	○	○		○	○	○	○	○	○			○	○	○
CurrentX [1]								●						●					
CurrentY [1]								●						●					
Default				●															
DragMode	●		●	●	●	●	●		●	●	●		●	●			●	●	
Drive [1]						●													
Enabled	●		●	●	●	●	●	●	●	●	●	●	●	●			●	●	●
FileName [1]							●												
FormName								○											
FormType								○											

[1] Not available at design time

Table 4 Continued

Key

- ● Read/write at run time
- ◐ Read-only at run time
- ○ Not available at run time

Properties / Objects	Check	CLIPBOARD	Combo	Command	Dir	Drive	File	Form	Frame	Label	List	Menu	Option	Picture	PRINTER	SCREEN	Scroll	Text	Timer
Height	●		●	●	●	●	●	●	●	●	●		●			◐[1]	●	●	
Hidden							●												
Index	◐		◐	◐	◐	◐	◐		◐	◐	◐	◐	◐	◐			◐	◐	◐
Interval																			●
LargeChange																	●		
Left	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
List [1]			●		◐	◐	◐				●								
ListCount [1]			◐		◐	◐	◐				◐								
ListIndex [1]			●		●	●	●				●								
Max																	●		
MaxButton								◐											
Min																	●		
MinButton								◐											
MousePointer	●		●	●	●	●	●	●	●	●	●		●	●		●	●	●	
MultiLine																		◐	
Normal							●												
Parent [1]	◐		◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐			◐	◐	◐
Path [1]					●		●												
Pattern							●												
ReadOnly							●												
ScaleHeight								●						●					
ScaleWidth								●						●					
ScrollBars																		◐	
SelLength [1]			●															●	
SelStart [1]			●															●	
SelText [1]			●															●	
Separator												●							

[1] Not available at design time

Table 4 Continued

Key

- ● Read/write at run time
- ◐ Read-only at run time
- ○ Not available at run time

Properties / Objects	Check	CLIPBOARD	Combo	Command	Dir	Drive	File	Form	Frame	Label	List	Menu	Option	Picture	PRINTER	SCREEN	Scroll	Text	Timer
SmallChange																	●		
Sorted			◐								◐								
Style			◐																
System							●												
TabIndex	●		●	●	●	●	●		●	●	●		●	●			●	●	
TabStop	●		●	●	●	●	●				●		●	●			●	●	
Tag	●		●	●	●	●	●	●	●	●	●	●	●	●			●	●	●
Text			●								◐[1]							●	
Top	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
Value	●			●									●				●		
Visible	●		●	●	●	●	●	●	●	●	●	●	●	●			●	●	
Width	●		●	●	●	●	●	●	●	●	●		●	●		●[1]	●	●	
WindowState								●											

[1] Not available at design time

Events

Table 5 lists actions that may occur in a Visual Basic application and the events that are related to them.

Table 6 lists all the events alphabetically and shows the objects for which a given event can occur.

Table 5 Events by Action

Action	Event
Change to control	Change; DropDown (Combo box only); PathChange, PatternChange (file list box only)
Drag and drop	DragDrop, DragOver
Keystroke	KeyDown, KeyUp, KeyPress
Mouse operations	Click, DblClick, MouseDown, MouseUp, MouseMove
Shift in focus	GotFocus, LostFocus
Timer interval	Timer
Working with forms and picture boxes	Paint, Resize, Load, Unload

Table 6 Events by Object

Events \ Objects	Check	CLIPBOARD	Combo	Command	Dir	Drive	File	Form	Frame	Label	List	Menu	Option	Picture	PRINTER	SCREEN	Scroll	Text	Timer
Change			●		●	●				●							●	●	
Click	●		●	●	●		●	●		●	●	●	●	●					
DblClick			● [2]				●	●		●	●		●	●					
DragDrop	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
DragOver	●		●	●	●	●	●	●	●	●	●		●	●			●	●	
DropDown			●																
GotFocus	●		●	●	●	●	●	●			●		●	●			●	●	
KeyDown	●		●	●	●	●	●	●			●		●	●			●	●	
KeyPress	●		●	●	●	●	●	●			●		●	●			●	●	
KeyUp	●		●	●	●	●	●	●			●		●	●			●	●	
Load								●											
LostFocus	●		●	●	●	●	●	●			●		●	●			●	●	
MouseDown					●		●	●		●	●			●					
MouseMove					●		●	●		●	●			●					
MouseUp					●		●	●		●	●			●					
Paint								●						●					
PathChange							●												
PatternChange							●												
Resize								●											
Timer																			●
UnLoad								●											

[2] Only if Style = 1 (simple combo box).

Objects

Table 7 lists actions a program or a user may perform and the objects typically used by such an action.

Table 7 Objects by Action

Action	Object
Return the active form or control at run time	SCREEN object
Choose one item from list	List box, combo box, group of option buttons
Copy, cut, or paste text or graphics	CLIPBOARD object
Display character-based drawing	Picture box
Display text the user can't change	Label
Place controls in a group surrounded by a rectangle	Frame
Enter text	Text box
Print text or graphics	PRINTER object
Scroll through items, values, or text	Horizontal and vertical scroll bars
Select a command	Command button, menu
Select a directory	Directory list box
Select a drive	Drive list box
Turn an option on or off	Check box, option button

A-Z Reference

ABS Function

Action Returns the absolute value of a number.

Syntax **ABS**(*numeric-expression*)

Part	Description
numeric-expression	Any valid numeric data type; the absolute value is the unsigned magnitude of its argument; for example, ABS(-2) and ABS(2) are both 2.

See Also SGN Function.

Example This example finds the cube root of an input value. The ABS and SGN functions test whether the current guess is accurate. The DEFDBL statement establishes the default data type for the variables.

```
CLS                                          ' Clear the screen.
DEFDBL A-Z
CONST Precision = .0000001#
CONST LowGuess = -1, HighGuess = 1, ExactGuess = 0
CLS                                          ' Clear the screen.
INPUT "Enter a value: ", Value               ' Prompt for input.
X1 = 0#: X2 = Value                          ' Make the first two guesses.
accuracy$ = "approximate"
DO UNTIL ABS(X1 - X2) < Precision            ' Loop until the difference
    X = (X1 + X2) / 2#                       ' between guesses is less than
    SELECT CASE SGN(X * X * X - Value)       ' the required precision;
    CASE IS = LowGuess                       ' adjust the guesses.
         X1 = X
    CASE IS = HighGuess
         X2 = X
    CASE IS = ExactGuess
         accuracy$ = "Exact"
         X1 = X2
    END SELECT
LOOP
PRINT "The "; accuracy$; " cube root is"; X
```

Absolute Routine

Action Transfers control to a machine-language procedure. Requires VBDOS.QLB (VBDOS.LIB when compiling); include VBDOS.BI.

Syntax **CALL Absolute** ([*argumentlist*,] *offset%*)

Part	Description
argumentlist	Arguments passed to a machine-language procedure as offsets from the current data segment.
offset%	Offset from the current code segment, set by DEF SEG, to the starting location of the procedure.

Remarks The Absolute routine is provided to maintain compatibility with earlier versions of Microsoft QuickBasic. The CALL and DECLARE statements provide a simpler way to use assembly language with Visual Basic.

Although arguments are passed as offsets, the machine-language program is invoked with a far call.

The *offset%* is not passed to the procedure. Your program may need to execute a DEF SEG statement before executing the Absolute routine to set the code segment for the called routine.

Using a noninteger value for *offset%* produces unpredictable results.

See Also CALL Statement (Basic Procedures), DEF SEG Statement, $INCLUDE Metacommand, Interrupt[X] Routine, VARPTR Function, VARSEG Function.

Example This example uses the Absolute routine to execute a machine-language program stored in an array. The program prints a message indicating whether or not a math coprocessor is installed.

Note The Absolute routine requires the Quick library VBDOS.QLB. To load VBDOS.QLB, start Visual Basic with the /L switch.

```
CONST nASMBYTES = 14        ' AsmBytes is a label.
DEFINT A-Z                  ' nASMBYTES is a symbolic constant.
DIM AsmProg(1 TO (nASMBYTES / 2))

CLS                         ' Clear the screen.
.' The machine-language program stored as data to read into the array
AsmBytes:
DATA &H55                :  ' PUSH BP         Save base pointer.
DATA &H8B, &HEC          :  ' MOV  BP,SP      Get our own.
DATA &HCD, &H11          :  ' INT  11H        Make the ROM-BIOS call.
DATA &H8B, &H5E, &H06    :  ' MOV  BX,[BP+6] Get argument address.
```

```
DATA &H89, &H07          :  ' MOV  [BX],AX   Save list in argument.
DATA &H5D                :  ' POP  BP        Restore base pointer.
DATA &HCA, &H02, &H00    :  ' RET  2         Pop argument off stack
                            '                and make far return.

' Poke the machine-language program into the array
P = VARPTR(AsmProg(1))              ' Get the starting offset of the array.
DEF SEG = VARSEG(AsmProg(1))        ' Change the segment.
FOR i = 0 TO nASMBYTES - 1
    READ J
    POKE (P + i), J
NEXT i

' Execute the program. The program expects a single integer argument.
CALL ABSOLUTE(X%, VARPTR(AsmProg(1)))
DEF SEG                             ' Restore the segment.
                                    ' X% now contains bit-encoded
                                    ' equipment list returned by MS-DOS.
CoProcessor = X% AND &H2            ' Mask off all but the coprocessor bit.
IF CoProcessor = 2 THEN             ' Print the appropriate message.
    PRINT "Math coprocessor present."
ELSE
    PRINT "No math coprocessor."
END IF
END
```

ACCESSKEY_FORECOLOR Constant

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the foreground color of all access keys.

Syntax **SCREEN.ControlPanel**({**ACCESSKEY_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	ACCESSKEY_FORECOLOR has a value of 0.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
15	(Default) High-intensity white.

Example This example illustrates the effect of changing the value of the ACCESSKEY_FORECOLOR constant. To try this example, create a form containing a command button. Set the command button Caption property to "&DoIt."

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.
SUB Command1_Click ()
    SCREEN.ControlPanel(ACCESSKEY_FORECOLOR) = 13
END SUB
```

ACTIVE_BORDER_BACKCOLOR, ACTIVE_BORDER_FORECOLOR Constants

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the background or foreground border color of the active window.

Syntax **SCREEN.ControlPanel({ACTIVE_BORDER_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({ACTIVE_BORDER_FORECOLOR | *subscript%*})[= *color%*]

Part	Description
subscript%	ACTIVE_BORDER_BACKCOLOR has a value of 1; ACTIVE_BORDER_FORECOLOR has a value of 2.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
7	(Default - ACTIVE_BORDER_BACKCOLOR) Black.
0	(Default - ACTIVE_BORDER_FORECOLOR) White.

Example This example illustrates the effect of changing the value of the ACTIVE_BORDER_BACKCOLOR and ACTIVE_BORDER_FORECOLOR constants. To try this example, create a form containing a command button. Set the command button Caption property to "&DoIt."

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

.SUB Command1_Click ()
    SCREEN.ControlPanel(ACTIVE_BORDER_BACKCOLOR) = 3
    SCREEN.ControlPanel(ACTIVE_BORDER_FORECOLOR) = 13
END SUB
```

ACTIVE_WINDOW_SHADOW Constant

Applies To ControlPanel property, SCREEN object.

Action Determines if the active form is displayed with a shadow.

Syntax **SCREEN.ControlPanel({ACTIVE_WINDOW_SHADOW** | *subscript%*})[= *boolean%*]

Part	Description
subscript%	ACTIVE_WINDOW_SHADOW has a value of 3.
boolean%	Valid settings are:

Value	Description
True (-1)	(Default) Shadow is displayed.
False (0)	No shadow is displayed.

Example This example illustrates the effect of changing the value of the ACTIVE_WINDOW_SHADOW constant. To try this example, create a form containing a check box. Set the check box Caption property to "Shadow Toggle."

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Check1_Click ()
    IF SCREEN.ControlPanel(ACTIVE_WINDOW_SHADOW) = -1 THEN
        SCREEN.ControlPanel(ACTIVE_WINDOW_SHADOW) = 0
    ELSE
        SCREEN.ControlPanel(ACTIVE_WINDOW_SHADOW) = -1
    END IF
END SUB
```

ActiveControl, ActiveForm Properties

Applies To SCREEN object.

Action Returns a handle to the control that has the focus (ActiveControl); invalid if all controls on the form are invisible or disabled. Returns a handle to the form that contains the active control (ActiveForm); invalid if no forms are loaded. Read-only at run time; not available at design time.

Syntax **SCREEN.ActiveControl**
SCREEN.ActiveForm

Remarks Use the ActiveControl and ActiveForm properties to access the form or control that has the focus at run time.

ActiveControl and ActiveForm can be used to refer to a control or a form together with one of its properties or methods. For example:

```
SCREEN.ActiveControl.Tag = "OkButton"
```

If the form has no controls, ActiveForm returns the form with the focus.

See Also GotFocus Event, LostFocus Event.

Example This example displays the text of the active control when the form is clicked. Click either the text box or command button to set the focus, then click the form to display the active control caption within the label. To try this example, create a form containing a text box, a label, and a command button.

```
SUB Form_Click ()
    IF TYPEOF SCREEN.ActiveControl IS TextBox THEN
        Label1.Caption = SCREEN.ActiveControl.Text
    ELSE
        Label1.Caption = "Button: " + SCREEN.ActiveControl.Caption
    END IF
END SUB
```

ADDITEM Method

Applies To Combo box, list box.

Action Adds an item to a list box or combo box at run time.

Syntax [*form.*]{*listbox* | *combobox*}.**ADDITEM** *item$* [, *index%*]

Part	Description
item$	String expression to add to the list.
index%	Integer specifying the position at which *item$* is inserted in the list; an *index%* of 0 represents the first position.

Remarks If *index%* is omitted, *item$* is added to the proper sorted position if the Sorted property is True (-1), or to the end of the list if the Sorted property is set to False (0).

Since the Sorted property does not distinguish between uppercase and lowercase, you may need to write your own sort routine for a case-sensitive ADDITEM implementation. Use caution when specifying *index%* with ADDITEM when the Sorted property is set to True (-1). If *index%* adds an element to a specific location that violates the sort order, subsequent additions may not be properly sorted.

Use the List, ListCount, and ListIndex properties to access items displayed in the list box at run time.

ListIndex is set to -1 (no item selected) immediately after an ADDITEM operation.

Use the List property rather than the ADDITEM method to replace an item in a list box or combo box:

```
Combo1.List(1) = "Students"
```

See Also REMOVEITEM Method.

Example The example uses the ADDITEM method to add 100 items to a list box. the REMOVEITEM method is used to remove items from the list box control. To try this example, create a form containing a list box.

```
DECLARE SUB AddRemItemDemo ()
DIM SHARED X%

SUB AddRemItemDemo ()
   SELECT CASE X%
    CASE 1
         Msg$ = "Choose OK to add 100 items to your list box."
         MSGBOX Msg$                        ' Display message.
         FOR I% = 1 TO 100                  ' Count from 1 to 100.
              Entry$ = "Entry " + STR$(I%)' Create entry.
              List1.ADDITEM Entry$          ' Add the entry.
         NEXT I%
    CASE 2
         Msg$ = "Choose OK to remove every other entry."
         MSGBOX Msg$                        ' Display message.
         FOR I% = 1 TO 50                   ' Determine how to
              List1.REMOVEITEM I%           ' remove every other
         NEXT I%                            ' item.
    CASE 3
         Msg$ = "Choose OK to remove all items from the list box."
         MSGBOX Msg$                        ' Display message.
         DO WHILE List1.ListCount           ' As long as there are
              List1.REMOVEITEM 0            ' still items in the
         LOOP                               ' list box, remove them.
    END SELECT
END SUB

SUB Form_Click ()
   X% = X% + 1
   CALL AddRemItemDemo
   IF X% = 3 THEN X% = 0
END SUB
```

Alignment Property

Applies To Label.

Action Sets or returns the alignment of text in a label control.

Syntax [*form.*]*label*.**Alignment**[= *setting%*]

Part	Description
setting%	Valid settings are:

Value	Justification
0	(Default) Left.
1	Right.
2	Center.

See Also AutoSize Property.

Example This example changes the label alignment when you click the command button. To try this example, create a form containing a label and a command button.

```
SUB Command1_Click ()
    SELECT CASE Label1.Alignment
    CASE 0
       Label1.Alignment = 1
       Command1.Caption = "Right"
    CASE 1
       Label1.Alignment = 2
       Command1.Caption = "Center"
    CASE ELSE
       Label1.Alignment = 0
       Command1.Caption = "Left"
   END SELECT
END SUB
```

Archive, Hidden, Normal, ReadOnly, System Properties

Applies To File list box.

Action Determines if a file list box displays files with Archive, Hidden, Normal, ReadOnly, or System attributes.

Syntax [*form.*]*filelistbox.property*[= *boolean%*]

Part	Description
property	Any valid file attribute property (Archive, Hidden, Normal, ReadOnly, or System).
boolean%	Valid settings are:

Value	Description
True (-1)	(Default - Archive, Normal, and ReadOnly) File list box displays files with the specified file attribute.
False (0)	(Default - Hidden and System) File list box does not display files with the specified file attribute.

Remarks Use these properties to control which kinds of files are displayed in a file list box, based on standard file attributes used in the operating environment.

See Also Path Property, Pattern Property, SYSTEM Statement.

Example This example switches the Archive property on and off. To see the effects of the setting, you must have already set the archive bit of at least one file to zero (0). To try this example, create a form containing a file list box and a check box.

```
SUB Check1_Click ()
    File1.Archive = Check1.Value
END SUB
```

ASC, CHR$ Functions

Action Returns the ASCII code for the first character in a string expression (ASC), or returns the character corresponding to a specified ASCII code (CHR$).

Syntax **ASC**(*stringexpression$*)
CHR$(*code%*)

Part	Description
stringexpression$	Any string expression.
code%	Valid ASCII code (0 - 255).

Remarks ASC and CHR$ are approximately inverses of each other. CHR$ takes a number and converts it to a one-character string; ASC takes the string converts it back to the number again.

If *stringexpression$* is null, Visual Basic generates the run-time error message, "Illegal function call."

CHR$ is often used to send a special character to the screen or to the printer. For example, you can send a formfeed (CHR$(12)) to clear the screen and return the cursor to the Home position.

CHR$ can be used to include a double quotation mark (") in a string. For example:

```
Msg$ = CHR$(34) + "Quoted string" + CHR$(34)
```

This string, when printed, appears on the screen as:

```
"Quoted string"
```

Example

This example uses ASC to display the the ASCII code of any key entered by the user. CHR$ is used to define the Escape key as the exit key.

```
CLS                                 ' Clear the screen.
PRINT TAB(20); "Press 'Escape Key' to END"
PRINT : PRINT " Extended", "ASCII", "Description"
PRINT " Code", "Code"
VIEW PRINT 6 TO 23
DO
    KeyPress$ = INKEY$
    SELECT CASE LEN(KeyPress$)
        CASE 0:                     ' Do nothing, since no key was pressed.
        CASE 1:                     ' Standard ASCII key.
            ASCII% = ASC(KeyPress$)
            PRINT , ASCII%, "Standard ASCII Key"
        CASE 2:                     ' Extended key (Scan + ASCII codes).
            SCAN% = ASC(LEFT$(KeyPress$, 1))
            ASCII% = ASC(RIGHT$(KeyPress$, 1))
            PRINT SCAN%; "+"; ASCII%, , "Extended Key"
    END SELECT
LOOP UNTIL KeyPress$ = CHR$(27)
```

ATN Function

Action

Returns the arctangent of a specified numeric expression.

Syntax

ATN(*numeric-expression*)

Part	Description
numeric-expression	Ratio between the sides of a right triangle.

Remarks ATN (arctangent) is the inverse of tangent (TAN). ATN evaluates the ratio of two sides of a right triangle (*numeric-expression*) and returns the corresponding angle in radians. The expression evaluated is the ratio of the lengths of the side opposite the angle and the side adjacent to the angle. The arctangent is given in radians and is in the range -pi/2 to pi/2 radians, where pi = 3.141593 and pi/2 radians = 90 degrees.

ATN is calculated in single precision if *numeric-expression* or angle is an integer or single-precision value. If you use any other numeric data type, they are calculated in double precision.

See Also COS Function, SIN Function, TAN Function.

Example This example finds the tangent of PI/4 and then uses the ATN function to find the arctangent of the value. Additional examples show how to find the cosine (COS), sine (SIN), and tangent (TAN) of a value.

```
CLS                                  ' Clear the screen.
CONST PI = 3.141592653#
PRINT ATN(TAN(PI / 4!)), PI / 4!     ' Output is:  .7853981635  .7853981635.
PRINT (COS(180 * (PI / 180)))        ' Output is:  -1.
PRINT (SIN(90 * (PI / 180)))         ' Output is:  1.
PRINT (TAN(45 * (PI / 180)))         ' Output is:  .9999999997050103.
LOCATE 24, 1: PRINT "Press any key to continue...";
DO
LOOP WHILE INKEY$ = ""
```

Attached Property

Applies To Horizontal scroll bar, vertical scroll bar.

Action Determines if a scroll bar is attached to the form; read-only at run time.

Syntax [*form.*]*scrollbar*.**Attached**

Valid settings are:

Value	Description
True (-1)	Scroll bar attached to bottom or right border of the form (horizontal or vertical scroll bar).
False (0)	(Default) Scroll bar not attached to the form.

Remarks

Setting Left, Height, Width, or Top at run time when Attached = True (-1) will generate a read-only error.

Only one horizontal and one vertical scroll bar at a time may be attached.

When a scroll bar is attached to a border, the border itself becomes a scroll bar.

The SCROLLBAR_BACKCOLOR and SCROLLBAR_FORECOLOR settings of the ControlPanel property cannot be used to change the color of scroll bars that are attached to an object. Instead, attached scroll bars derive their color from:

- The control's BackColor or ForeColor properties.
- The forms's ACTIVE_BORDER_BACKCOLOR and ACTIVE_BORDER_FORECOLOR ControlPanel property settings.

When a scroll bar is attached, changes to the scroll bar's Left, Top, Width, and Height properties are not allowed. The scroll bar is automatically sized when the form is sized. Unattached scroll bars do not automatically change size or position when the form is resized.

See Also

Scrollbars Property.

Example

This example displays the value of a horizontal scroll bar's Attached property in a text box when the command button is clicked. To try this example, create a form containing a horizontal scroll bar, a text box, and a command button. Set property values:

- For horizontal scroll bar, Attached = True.
- For text box, MultiLine = True.

```
SUB Command1_Click ()
    Text1.Text = STR$(HScroll1.Attached)
END SUB

SUB Form_Load ()
    Command1.Caption = "Click Here"
END SUB
```

AutoRedraw Property

Applies To

Form, picture box.

Action

Determines whether or not the background image of a form or picture box is "persistent" (automatically repainted when necessary).

Syntax

[*form.*][*picturebox.*]**AutoRedraw**[= *boolean%*]

Part	Description

boolean% Valid settings are:

Value	Description
True (-1)	Enables automatic repainting of a form or picture box. Print output is written to the screen and to an image stored in memory. The object does not receive Paint events; it is repainted when necessary, using the memory image.
False (0)	(Default) Disables automatic repainting of an object and writes print output only to the screen. The Paint event is invoked when necessary to repaint the object.

Remarks

AutoRedraw automatically redraws the output from the PRINT method on a form or in a picture box. For example, the object is resized or redisplayed after being hidden by another form.

When AutoRedraw is set to True (-1), no Paint events are generated.

Regardless of the state of AutoRedraw, if you set the BackColor property, all graphics and print output, including the background image, is erased. In general, all print output should be redisplayed in a Paint event unless AutoRedraw is set to True (-1).

When AutoRedraw is set to False (0), if you clear the drawing area with the CLS method, background characters are not deleted. To clear the background graphics, set AutoRedraw to True (-1) and then use the CLS method.

You can save a substantial amount of memory by setting AutoRedraw to False (0) on forms that do not need to be repainted.

For sizable forms, the background image is the size of the screen. For nonsizable forms, the background image is the size of the form.

If the form size is changed at run time, a new background image is created with the new size. Any newly exposed areas are filled in with the background color or pattern.

See Also

AutoSize Property, Paint Event, REFRESH Method.

Example

This example loads a list into a combo box control and displays the list count on the form. When you choose the command button, the list is cleared. The combo box list is automatically updated by setting the form's AutoRedraw property to True. To try this example, create a form containing a combo box and a command button. Set object properties:

- Form, AutoRedraw = True
- Combo box, Style = 2 (Dropdown List)

```
SUB Command1_Click ()
    IF Combo1.ListCount <> 0 THEN
       Combo1.REMOVEITEM 0
    END IF
    CLS                                    ' Clear the screen.
```

```
    PRINT "Number of Items in List:"; Combo1.ListCount
END SUB

SUB Form_Load ()
    Combo1.ADDITEM "Breakfast"              ' Add each item to list.
    Combo1.ADDITEM "Lunch"
    Combo1.ADDITEM "Dinner"
    PRINT "Number of Items in List:"; Combo1.ListCount
END SUB
```

AutoSize Property

Applies To Label.

Action Determines if a label is automatically resized to fit its contents.

Syntax [*form.*]*label*.**AutoSize**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	Automatically resizes the label to fit its contents.
False (0)	(Default) Keeps size of the label constant; control contents are clipped when they exceed the area of the control.

See Also Alignment Property, AutoRedraw Property.

Example This example shows how the AutoSize property can be used with a label control. To try this example, create a form containing two label controls. Set the form's BackColor property to 1 (Blue).

```
SUB Form_Load ()
    Label1.Caption = "This is an autosize test"
    Label1.AutoSize = 1
    Label2.Caption = "This is an autosize test"
    Label2.AutoSize = 0
END SUB
```

BackColor, ForeColor Properties

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action Sets or returns the background (BackColor) or foreground (ForeColor) color of an object.

Syntax [*form.*][*ctlname.*]**BackColor**[= *color%*]
[*form.*][*ctlname.*]**ForeColor**[= *color%*]

Part	Description
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
7	(Default - BackColor) White.
0	(Default - ForeColor) Black.

Remarks Changing the background color of a form causes a repaint of the form background, which clears all text on the background even if AutoRedraw is set to True (-1).

When THREE_D is set to False(0), control borders are drawn in the foreground color of the control.

Example This example resets background colors randomly twice each second for a form and a picture control. To try this example, create a form containing a picture box and a timer. Set the timer control's Interval property to 500.

```
SUB Timer1_Timer ()
    BackColor = (INT(RND * 15))
    Picture1.BackColor = (INT(RND * 15))
END SUB
```

BEEP Statement

Action Generates a beep sound from your computer's speaker; this is the same sound produced with the PRINT CHR$(7) statement.

Syntax **BEEP**

See Also PLAY Statement (Music), PRINT Statement, SOUND Statement.

Example This example uses the BEEP statement to indicate an error in the response.

```
CLS                                        ' Clear the screen.
DO
   INPUT "Do you hear a beep (Y or N)"; Response$
   R$ = UCASE$(MID$(Response$, 1, 1))
   IF R$ <> "Y" OR R$ = "N" THEN EXIT DO
   BEEP
LOOP
```

BLOAD, BSAVE Statements

Action Loads a file created by BSAVE into memory (BLOAD); copies the contents of an area of memory to a file (BSAVE).

Syntax **BLOAD** *filespec$*[, *offset%*]
BSAVE *filespec$*, *offset%*, *length%*

Part	Description
filespec$	For BLOAD, a memory-image file created by BSAVE; for BSAVE, a file or device to which an area of memory is copied.
offset%	For BLOAD, offset of the address where loading starts; for BSAVE, offset of the starting address of the area of memory being saved.
length%	Number of bytes to copy in the range 0 to 65,535, inclusive.

Remarks You can use BSAVE and BLOAD to save a screen image to a file and later restore that image from disk to memory. However, this could interfere with any forms that are present on the screen. To use BSAVE and BLOAD with forms, you should first unload or hide any visible forms.

The BSAVE statement allows data or programs to be saved as memory-image files on disk. A memory-image file is a byte-for-byte copy of what is in memory as well as control information used by BLOAD to load the file.

The BLOAD statement allows a program or data saved as a memory-image file to be loaded anywhere in memory.

The starting address for loading is determined by the specified offset and the most recent DEF SEG statement. If *offset%* is omitted, the segment address and offset contained in the file (the address used in the BSAVE statement) are used. Thus, the file is loaded at the address used when saving the file.

If you supply an offset, the segment address used is the segment set by the most recently executed DEF SEG statement. If there has been no DEF SEG statement, the Visual Basic data segment (DGROUP) is used as the default.

If the offset is a long integer, or a single- or double-precision number, it is converted to an integer. If the offset is a negative number between -1 and -32,768, inclusive, it is treated as an unsigned 2-byte offset.

Do not use BLOAD to load a file into an expanded memory array. Do not use BSAVE to transfer an expanded memory array to an output file.

If you start Visual Basic (VBDOS.EXE) using the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays < 16K in size.
- Fixed-length string arrays < 16K in size.
- User-defined-type arrays < 16K in size.

If you want to use BLOAD to load a file into an array (< 16K), or if you want to use BSAVE to transfer an array (< 16K) to an output file, start Visual Basic (VBDOS.EXE) without the /Ea switch. Without the /Ea switch, no arrays are stored in expanded memory.

Programs written in earlier versions of Microsoft Basic no longer work if they use VARPTR to access numeric arrays.

Because BLOAD does not perform an address-range check, it is possible to load a file anywhere in memory. Be careful not to write over Visual Basic or the operating system.

Because different screen modes use memory differently, do not load graphic images in a screen mode other than the one used when they were created.

See Also DEF SEG Statement.

Example This example draws a magenta cube inside a white box and then uses the BSAVE statement to store the drawing in the file MAGCUBE.GRH. The BLOAD statement is used to retrieve and display the drawing.

```
DIM Cube(1 TO 675)                    ' Declaration.
CLS                                   ' Clear the screen.
SCREEN 1
LINE (140, 25)-(140 + 100, 125), 3, B ' Draw a white box.

' Draw the outline of a magenta cube inside the box.
DRAW "C2 BM140,50 M+50,-25 M+50,25 M-50,25"
DRAW "M-50,-25 M+0,50 M+50,25 M+50,-25 M+0,-50 BM190,75 M+0,50"
GET (140, 25)-(240, 125), Cube        ' Save the drawing in the array Cube.

' Set segment to the segment of the array named Cube and store the drawing
' in the file MAGCUBE.GRH. Note: 2700 is the number of bytes in Cube
' (4 bytes per array element * 675).
```

```
DEF SEG = VARSEG(Cube(1))
BSAVE "MAGCUBE.GRH", VARPTR(Cube(1)), 2700
LOCATE 20, 10
PRINT "This image saved in file MAGCUBE.GRH"
PRINT "Press any key."
DO WHILE INKEY$ = ""
LOOP
CLS
LOCATE 20, 10                          ' Display the saved image.
BLOAD "MAGCUBE.GRH", VARPTR(Cube(1))
PRINT "This image loaded from MAGCUBE.GRH."
DEF SEG                                ' Restore default Basic segment.
PUT (80, 10), Cube                     ' Put the drawing on the screen.
```

BorderStyle Property

Applies To Form, label, picture box, text box.

Action Specifies the border character scheme for a form or control; read-only at run time.

Syntax [*form.*][*label.* | *picturebox.* | *textbox.*]**BorderStyle**[= *setting%*]

Remarks

Part	Description
setting%	Valid settings are:

Form Values	Meaning	Description
0	None	Not sizable, no caption desplayed, no Minimize or Maximize buttons; no Control-menu box, no title bar.
1	Fixed Single	Sizable only with Maximize or Minimize buttons; optional Control-menu box, title bar, Maximize and Minimize buttons.
2	Sizable Single	(Default) Border can be resized with any optional border elements listed above.
3	Fixed Double	Not sizable; optional Control-menu box, optional title bar; no Minimize or Maximize buttons.

Part	Description

Form Values	Meaning	Description
4	Sizable Double	
5	Fixed Solid	
6	Sizable Solid	

Control Values	Meaning	Description
0	None	(Default - label).
1	Single Line	(Default - text box and picture box).
2	Double Line	

With a form, the BorderStyle property determines key characteristics that visually identify a form as either a window or a dialog box. Setting 3 (Fixed Double) is used for standard dialog boxes.

The ControlPanel property constants ACTIVE_BORDER_BACKCOLOR and ACTIVE_BORDER_FORECOLOR are used to determine the colors associated with an object's border.

If a title bar is not present on a form, the form cannot be moved at run time.

Mouse events are detected within border areas (of controls that have borders). CurrentX and CurrentY values can be used to indicate when the mouse is not in the client area of the control.

When THREE_D is set to False (0), control borders are drawn in the foreground color of the control.

See Also Caption Property, ControlBox Property, MinButton Property, MaxButton Property.

BSAVE Statement

See BLOAD Statement.

CALL Statement (Basic Procedures)

Action Transfers control to a SUB procedure.

Syntax

Syntax 1:

CALL *name* [([**BYVAL**] *argument*, [**BYVAL**] *argument...*)]

Syntax 2:

name [[**BYVAL**] *argument*, [**BYVAL**] *argument...*]

Part	Description
name	Name of SUB procedure to call; limited to 40 characters.
BYVAL	Passes argument by value rather than by reference; cannot be used if argument is an array.
argument	Variable, array, or expression passed to a procedure; use this syntax if argument is an array: `CALL ShellSort(IntArray())`

Remarks The CALL keyword is optional when you use the CALL statement. When you omit CALL, you must declare the procedure in a DECLARE statement. If you dont't, when you save the program, Visual Basic automatically generates a DECLARE statement for you. When you omit CALL, you must omit the parentheses around the argument list.

If you omit the CALL keyword, you cannot follow the procedure name with a colon (:), the Visual Basic statement-separation character. If you do, Visual Basic treats the procedure name as a label instead of a call to the procedure.

The CALL statement passes arguments by reference or by value, depending on how the argument is defined in the DECLARE or SUB statement.

Within the CALL statement, the BYVAL keyword is informational. An argument must be defined as being passed by value in the procedure, or in the DECLARE statement for the procedure to pass that argument by value. For example:

```
DECLARE SUB Mult(BYVAL X!, Y!)
.
.
.
CALL Mult (BYVAL X!, Y!)
```

BYVAL in the CALL statement is optional; deleting it has no effect. Adding BYVAL to Y! in the CALL statement without changing the DECLARE statement generates an error.

See Also Absolute Routine, CALL[S] Statement (Non-Basic Procedures), DECLARE Statement (Basic Procedures), SUB Statement.

Example

This example uses a recursive FUNCTION procedure (a function that calls itself) to find and return the length of a string. The CALL statement is then used to call a SUB procedure that prints a message on the 25th line of the display.

```
DECLARE FUNCTION StrgLng% (X$)
DECLARE SUB PrintMessage (message$)
CLS                              ' Clear the screen.
INPUT "Enter a string: ", InString$
length% = StrgLng%(InString$)
message$ = "The length of the string is" + STR$(length%)
CALL PrintMessage(message$)

SUB PrintMessage (message$)
    LOCATE 24, 1
    PRINT message$;
END SUB

FUNCTION StrgLng% (X$)
    IF X$ = "" THEN
         ' The length of a null string is zero
         StrLng% = 0
    ELSE
         ' Non-null string - make a recursive call.
         ' The length of a non-null string is 1 plus
         ' the length of the rest of the string.
         StrgLng% = 1 + StrgLng%(MID$(X$, 2))
    END IF
END FUNCTION
```

CALL[S] Statement (Non-Basic Procedures)

Action

Transfers control to a procedure written in a language other than Basic.; Syntax varies depending on the specified argument type.

Syntax

Syntax 1: For non-array arguments

CALL *name* [([[{**BYVAL**|**SEG**}] *argument*] [,[{**BYVAL**|**SEG**}] *argument*])]...
CALLS *name* [*argument*] [, *argument*]...

Syntax 2: For array arguments

CALL *name* [([*argument*[()]] [, *argument*[()]])]...
name [*argument*[()]] [, *argument*[()]]...

Part	Description
name	Name of non-Basic procedure to call.

Part	Description
BYVAL	Passes *argument* by value rather than by reference; cannot be used if *argument* is an array.
SEG	Passes *argument* as a segmented (far) address; cannot be used if *argument* is an array.
argument	Variable, array, or expression passed to a procedure (whole array arguments cannot be passed by CALLS); use this syntax if *argument* is an array: `CALL ShellSort(IntArray())`

Remarks

The CALL keyword is optional when you use the CALL statement. When you omit CALL, you must declare the procedure in a DECLARE statement. If you dont't, when you save the program, Visual Basic automatically generates a DECLARE statement for you. When you omit CALL, you must omit the parentheses around the *argument* list.

CALLS is the same as using CALL with a SEG keyword before each argument; every argument in a CALLS statement is passed as a segmented address.

If *name* refers to an assembly-language procedure, it must be a name declared with PUBLIC (symbol).

The result of the BYVAL keyword differs from Visual Basic's pass by value:

```
CALL Difference (BYVAL A,(B))
```

For the first *argument*, only the value of A is passed to Difference. In contrast, (B) is evaluated, a temporary location is created for the value, and the address of the temporary location is passed to Difference.

You can use Visual Basic's pass by value for an *argument*, but the procedure in the other language must accept an address, because the address of the temporary location is passed.

Be careful using the SEG keyword to pass elements of arrays because Visual Basic may move arrays in memory before the called routine starts to execute. Anything in an argument list that causes memory movement may create problems. You can safely pass variables using SEG if the argument list for the CALL statement contains only simple variables, arithmetic expressions, or arrays indexed without the use of intrinsic or defined functions.

See Also

CALL Statement (Basic Procedures), DECLARE Statement (Non-Basic Procedures).

Example

This example uses the CALL and CALLS statements to pass a Basic array to a C function.

Note: To run this program, you must: Compile the C function separately with the medium-model (/AM) switch. Place the object module in a Quick library or link it to the Basic main program

```
DEFINT A-Z
DECLARE SUB AddArr CDECL (SEG Array, SEG Num)
DIM A(1 TO 100) AS INTEGER
```

```
CLS                     ' Clear the screen.
N = 15                  ' Fill the array with the numbers 1 to N.
FOR I = 1 TO N
    A(I) = I
NEXT I

CALLS AddArr(A(1), N)   ' Call the C function using CALLS the first time
                        ' and increment all array elements by 1.
CALL AddArr(A(1), N)    ' Call the same C function again using CALL and
                        ' increment all array elements by 1.

FOR I = 1 TO N          ' Print the modified array; all elements will be
    PRINT A(I),         ' incremented by a total of two - for example,
NEXT I                  ' what was 1 will now be 3.
END

C Function addarr.
/* Add one to the first num elements of array arr.*/
void far addarr(arr,num)
int far *arr;
int far *num;
{
   int i;
   for(i=0;i<*num;i++) arr[i]++;
}
```

Cancel Property

Applies To Command button.

Action Determines if a command button is the default Cancel button on a form.

Syntax [*form.*]*commandbutton*.**Cancel**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	Enables the command button as the Cancel button for the form.
False (0)	(Default) Disables this functionality.

Remarks Regardless of which control has the focus, the Cancel button's Click event is invoked if the Esc key is pressed and the control with the focus does not receive a keyboard event (KeyDown, KeyPress, or KeyUp).

A commmand button with its Cancel property set to True (-1) is referred to as the form's Cancel button.

Only one button on a form can be the Cancel button.

When Cancel is set to True (-1) for one button, it is automatically set to False (0) for all other buttons.

Use the Cancel property to give users the option of canceling uncommitted changes and returning the form to its previous state.

Code for the Cancel property should be placed in the command button's Click event.

For a form that supports an irreversible action such as a delete operation, make the Cancel button the default button. To do this, set the button's Default property to True (-1).

See Also Click Event, Default Property, KeyDown Event, KeyPress Event, KeyUp Event.

Example This example shows how to use the Cancel property. When the first command button is selected or the Esc key is pressed, the Click event for the command button is executed. To try this example, create a form containing two command buttons. Set the first command button's Cancel property to True.

```
SUB Command1_Click ()
    Command1.caption = "Cancel"
END SUB

SUB Command2_Click ()
    Command2.Caption = "Not Cancel"
END SUB
```

Caption Property

Applies To Check box, command button, form, frame, label, menu, option button.

Action Sets or returns the text display in a control or the title bar of a form; text cannot be changed by the user at run time.

Syntax [*form.*][*ctlname.*]**Caption**[= *stringexpression$*]

Part	Description
stringexpression$	Text string to be displayed.

Remarks

When you create a new object, the default Caption property is the same as the default FormName property (for a form) or the default CtlName property (for a control — for example, Form1, Label1, and Command1). However, captions can differ from an object's FormName or CtlName property; you can label forms and controls more descriptively.

A form's title bar display depends on the BorderStyle property setting. A form cannot display a caption when its BorderStyle property is set to 0.

A caption that is too long for the form or control's title bar is clipped on the right.

To assign an access key to a control, include an ampersand (&) in the control's caption immediately preceding the character that you want to appear highlighted at run time. To insert an ampersand in the caption, precede it by another ampersand (for example, "&&"). The color of the highlighted character can be set by defining the ACCESSKEY_FORECOLOR constant.

To automatically resize a label control to fit its caption, set AutoSize to True (-1).

See Also

Text Property.

Example

This example changes the caption on the command button each time the user clicks the button. To try this example, create a form containing a command button.

```
SUB Command1_Click ()
    ' Check caption, then change it.
    IF Command1.Caption = "Clicked" THEN
        Command1.Caption = "OK"
    ELSE
        Command1.Caption = "Clicked"
    END IF
END SUB
```

CHAIN Statement

Action

Transfers control from the current program to another Visual Basic program.

Syntax

CHAIN *filespec$*

Part	Description
filespec$	String expression representing the file name to execute (may also contain a path specification); with MS-DOS version 2.1 or earlier, *filespec$* must include a complete path, including drive.

Remarks

Programs running in the programming environment assume a .BAS extension if no extension is given and cannot chain to executable files (files with a .COM or .EXE extension).

Programs running outside the programming environment assume an .EXE extension and cannot chain to Basic (.BAS) source files.

You can pass variables between programs using the COMMON Statement to set up a blank COMMON block.

When you compile a program outside the Visual Basic environment, the VBDRT10.LIB library does not support COMMON. To use COMMON with chained programs outside the environment, you must either:

- Select the EXE Requiring Run-Time Module option from the Make EXE File dialog when you compile. This is the default option (uses the file VBDRT10.EXE).
- Compile from the command line (BC.EXE) without the /O option.

When programs use VBDRT10.EXE, files are left open during chaining unless they are explicitly closed with the CLOSE statement.

CHAIN is similar to RUN; the main differences are that RUN closes all open files and does not support data blocks declared with COMMON.

CHAIN is equivalent to RUN when the /O compiler option is used.

If you compile a program (without /O) that does not use ISAM and then attempt to CHAIN to another non-/O program that does use ISAM, Visual Basic generates the error message, "Error loading run-time module (VBDRT10.EXE) - ISAM support removed." Likewise, if you compile a program (without /O) that does not use forms and then attempt to CHAIN to another non-/O program that does use forms, Visual Basic generates the error message, "Error loading run-time module (VBDRT10.EXE) - form support removed."

See Also CALL Statement (Basic Procedures), COMMON Statement, RUN Statement, SHELL Statement.

Example This example uses the COMMON statement to pass variables between two chained programs. It uses the CHAIN statement in the first program (PROG1) to load and run a second program (PROG2), which finds and prints the average of the values.

Note: After running this example from the programming environment, you will be left in PROG2.BAS.

```
DIM values(1 TO 50)                    ' Set up the variables that are
COMMON values(), NumValues%            ' passed from PROG1 to PROG2.

' The following lines of code create the second program on disk so that it
' can load and run with the CHAIN statement. It is, in fact, a series of
' PRINT # statements that puts the proper Basic statements in a file.

CLS                                    ' Clear the screen.
OPEN "PROG2.BAS" FOR OUTPUT AS #1
PRINT #1, "' This is PROG2 for the CHAIN and COMMON statement examples."
PRINT #1, " DIM X(1 TO 50)"
PRINT #1, " COMMON X(), N%"
```

```
PRINT #1, " PRINT"
PRINT #1, " IF N% > 0 THEN"
PRINT #1, "     Sum% = 0"
PRINT #1, "     FOR I% = 1 TO N%"
PRINT #1, "         Sum% = Sum% + X(I%)"
PRINT #1, "     NEXT I%"
PRINT #1, "     PRINT "; CHR$(34); "The average of the values is";
PRINT #1, CHR$(34); "; Sum% / N%"
PRINT #1, " END IF"
PRINT #1, " END"
CLOSE #1                                ' End of PROG2 creation section.

CLS                                     ' Clear the screen.
PRINT "Enter values one per line. Type 'END' to quit."
NumValues% = 0
DO
    INPUT "-> ", N$
    IF I% >= 50 OR UCASE$(N$) = "END" THEN EXIT DO
    NumValues% = NumValues% + 1
    values(NumValues%) = VAL(N$)
LOOP
PRINT "Press any key to continue... "
DO WHILE INKEY$ = ""
LOOP
CHAIN "PROG2.BAS"
END
```

Change Event

Applies To

Combo box, directory list box, drive list box, horizontal scroll bar, label, text box, vertical scroll bar.

Action

Occurs when the contents of a control have changed.

Syntax

SUB *ctlname*_**Change ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks

A Change event procedure synchronizes or coordinates the data displayed between controls. For instance, you can write a Change event procedure for a scroll bar so that it updates a label that displays the scroll bar's Value property. Or, you might use a Change event procedure to display data and formulas in a work area and results in another area.

Change event procedures are also useful for updating properties in file-system controls. For example, you can update the Path property for a directory list box to reflect a change in a drive list box's Drive property.

How and when a Change event occurs varies with the control:

Control	Description
Combo box	Occurs when the user enters data or changes the contents of a Dropdown Combo (Style = 0), or a Simple Combo (Style = 1).
Text box	Occurs when the user enters data or when the Text property is changed in code.
Directory list box	Occurs when the user selects a new directory, or when the Path property is changed in code.
Drive list box	Occurs when the user selects a new drive or when the Drive property is changed in code.
Scroll bars	Occurs when the user scrolls or when the Value property is changed in code.
Label	Occurs when the Caption property is changed in code.

A Change event procedure can sometimes cause a cascading event. This event occurs when the control's Change event alters the control's contents — for example, by setting a property in code that determines the control's value, such as the Text property setting for a text box. To prevent a cascading event:

- If possible, avoid writing a Change event procedure for a control that alters that control's contents. Otherwise, set a flag variable that blocks out further changes while the current change is in progress.
- Avoid creating two or more controls whose Change event procedures mutually affect each other — for example, two text boxes that update each other.

See Also KeyDown Event, KeyPress Event, KeyUp Event, PathChange Event, PatternChange Event.

Example In this example, each time the scroll bar is clicked the numeric setting of the scroll bar's Value property is displayed in a text box. To try this example, create a form containing a text box and a horizontal scroll bar. Set scroll bar properties:

- Max = 1000.
- LargeChange = 100.

```
SUB HScroll1_Change ()
    Text1.Text = STR$(HScroll1.Value)
END SUB
```

CHDIR, MKDIR, RMDIR, FILES Statements

Action

Changes a drive's default directory (CHDIR); creates a subdirectory (MKDIR); removes a subdirectory (RMDIR); displays the contents of the current or specified directory (FILES). FILES is not available when forms are showing.

Syntax

CHDIR *pathname$*
MKDIR *pathname$*
RMDIR *pathname$*
FILES [*filespec$*]

Part	Description
pathname$	Path of the new default directory or subdirectory to create, or subdirectory to remove; may contain up to 64 characters in the following format: `[drive:][\]directory[\directory]`
filespec$	Path or file name; may include drive and MS-DOS wildcard characters (? or *); if omitted, FILES displays contents of the current directory.

Remarks

The CHDIR, MKDIR, and RMDIR statements work like their MS-DOS counterparts; however, they cannot be shortened to the abbreviated versions CD, MD, and RD.

CHDIR changes the default directory but not the default drive. For example, if the default drive is C, the following statement changes the default directory on drive D, but C remains the default drive:

```
CHDIR "D:\TMP"
```

Use CURDIR$ to return the current directory and CHDRIVE to change the default drive.

You can use MKDIR to create a MS-DOS directory with a name that contains an embedded space. However, although you can access that directory through MS-DOS, you can remove it only with the Visual Basic RMDIR statement.

The directory to be removed must be empty except for the working directory ('.') and the parent directory ('..'). Otherwise, Visual Basic generates one of the following errors:

- "Path not found"
- "Path/File access error"

You can use the MS-DOS wildcard characters, the question mark (?) or asterisk (*). A question mark matches any single character in the file name or extension. An asterisk matches one or more characters starting at that position.

Output from the FILES statement cannot be directed to a form; this statement always prints to the background screen. Any reference to FILES when forms are showing causes

Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this statement.

Example

This example uses the MKDIR, CHDIR, RMDIR, and FILES statements to create a subdirectory, display its files, move to the root directory, and finally remove the new directory.

```
CLS                     ' Clear the screen.
MKDIR "\SECRET"       ' Makes directory called \SECRET on the current drive.
CHDIR "\SECRET"       ' Changes to directory \SECRET on the current drive.

OPEN "MESSAGE.DAT" FOR OUTPUT AS #1 ' Creates file in the current directory.
CLOSE #1

' Show a directory listing with MESSAGE.DAT appearing in the directory.
CLS
PRINT "List of the .DAT files in the current directory:"
PRINT
FILES "*.DAT"

KILL "MESSAGE.DAT"      ' Delete the file MESSAGE.DAT from the disk drive.
CHDIR "\"               ' Change the current directory to the root of the
                        ' current drive.
RMDIR "\SECRET"         ' Remove the directory \SECRET from the current drive.
END
```

CHDRIVE Statement

Action

Changes the current drive.

Syntax

CHDRIVE *drive$*

Part	Description
drive$	String expression that specifies a drive; if more than one character, CHDRIVE uses only the first letter.

Remarks

The drive specified must correspond to an existing drive and must be in the range A to lastdrive, where lastdrive is the maximum drive letter you may have set in your CONFIG.SYS file.

If you omit the argument *drive$*, Visual Basic generates an error.

If you supply a null argument ("") for *drive$*, the current drive does not change.

CHDRIVE is not case sensitive ("C" is the same as "c").

Since only the first letter of the *drive$* argument is used, the colon (:) is always optional.

Example

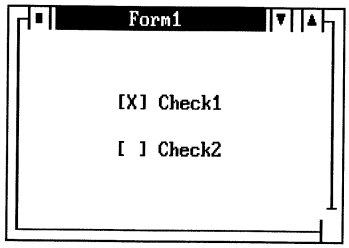
This example uses the CURDIR$ function and the CHDRIVE statement within the ChangeDataDrive SUB procedure. CURDIR$ is used to determine the current drive from a given file specification. CHDRIVE changes to a new default drive if the one specified is different from the current drive.

```
CLS                                          ' Clear the screen.
DECLARE SUB ChangeDataDrive (filespec$)
ON ERROR RESUME NEXT
filespec$ = "A:"
ChangeDataDrive filespec$
filespec$ = "C:\DOS"
ChangeDataDrive filespec$

SUB ChangeDataDrive (filespec$)
    curdrive$ = LEFT$(CURDIR$, 2)            ' Get the current drive.
    ' Determine if there is a drive name on the filespec.
    IF INSTR(1, filespec$, ":") = 2 THEN
         newdrive$ = LEFT$(filespec$, 2)
    ELSE
         newdrive$ = curdrive$
    END IF

    ' Ensure that the new drive is different from the
    ' currently logged drive.
    IF newdrive$ <> curdrive$ THEN
         CHDRIVE newdrive$                   ' It isn't, so change it.
         PRINT "Logged drive changed from "; curdrive$; " to "; newdrive$
    ELSE                                     ' It is, so don't change it.
         PRINT "New drive same as logged drive. No change occurred."
    END IF
END SUB
```

Check Box Control

Action

Displays an option that can be toggled (on/off, true/false, checked/unchecked); can also be grayed. An X in a check box indicates that the option is selected.



Remarks

Events

Click	DragDrop	DragOver	GotFocus
KeyDown	KeyPress	KeyUp	LostFocus

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	Caption	CtlName	DragMode
Enabled	ForeColor	Height	Index
Left	MousePointer	Parent	TabIndex
TabStop	Tag	Top	Value
Visible	Width		

The user can change the check box setting either by:

- Clicking the control.
- Pressing the Spacebar when the control has the focus.

To determine the state of the check box (unchecked, checked, or grayed) use the Value property.

Check boxes can be used in groups to display multiple choices to the user. You can group together controls of the same type by creating a control array.

While check boxes and option buttons appear to function similarly, there is an important difference: When a user selects an option button, other option buttons in the same group are automatically deselected. By contrast, any number of check boxes in a group can be selected.

The minimum size of a check box is 1 x 3; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum values are ignored.

See Also

Option Button Control.

Example

This example uses the Parent property and a check box control to display the state of a form's control. To try this example, create a form containing a command button and a check box. Set the form's AutoRedraw property to True.

```
DECLARE SUB CheckCheck (Source AS CONTROL) ' Declare sub procedure.

SUB CheckCheck (Source AS CONTROL)
    IF Source.Value = 0 THEN
        Source.Parent.CLS                              ' Clear parent form.
        Source.Parent.PRINT "CheckBox is Unchecked"    ' Print on parent form.
        ELSE
```

```
        Source.Parent.CLS                          ' Clear parent form.
        Source.Parent.PRINT "CheckBox is Checked"  ' Print on parent form.
    END IF
END SUB

SUB Command1_Click ()
    CheckCheck Form1.Check1            ' Call procedure in other module.
END SUB
```

Checked Property

Applies To

Menu.

Action

Determines if a check mark is displayed next to a menu command.

Syntax

[*form.*]*menuitem*.**Checked**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	Places a check mark next to the menu command, indicating that the item is selected.
False (0)	(Default) Removes a check mark from a menu command.

Remarks

A check mark cannot be displayed next to a top-level menu item.

You can set the value of this property at run time by writing code in any of the following:

- A startup procedure.
- A Form_Load event procedure.
- A Click event procedure attached to a menu control.

See Also

Enabled Property, Separator Property, Visible Property.

Example

This example displays a single menu on the menu bar. When you open the menu and choose the menu item, the checked symbol (•) is placed next to the menu item. To try this example from within the Form Designer, choose Menu Design Window from the WIndow menu. Set the first menu control properties:

- CtlName = MnuName
- Caption = MenuName

Choose Next and select the right (►) button. Set the second menu control properties:

- CtlName = MnuItem
- Caption = MenuItem

Choose Done to close the Menu Design window.

```
SUB MenuItem_Click ()                    ' Toggle checked symbol on menu item.
    IF MenuItem.Checked = -1 THEN
         MenuItem.Checked = 0
    ELSE
         MenuItem.Checked = -1
    END IF
END SUB
```

CHR$ Function

See ASC Function.

CIRCLE Statement

Action Draws a circle or an ellipse on the screen; not available when forms are showing.

Syntax **CIRCLE [STEP]** (*x!*,*y!*), *radius!*[,[*color%*] [,[*start!*] [,[*end!*] [, *aspect!*]]]]

Part	Description
STEP	Specifies that coordinates are relative to the current graphics cursor position.
(*x!*,*y!*)	Coordinates for the center of the circle or ellipse.
radius!	Radius of the circle or ellipse in the units of the current coordinate system, determined by the most recent SCREEN, VIEW, and WINDOW statements.
color%	Color attribute that sets the circle's color.
start!	Starting angle for the arc, in radians.
end!	Ending angle for the arc, in radians.
aspect!	Ratio of the length of the y axis to the length of the x axis, used to draw ellipses; default is the value required to draw a round circle in the current screen mode.

Remarks You must be in a graphics screen mode (for example, screen mode 1) to use the CIRCLE statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

The arguments *start!* and *end!* can range from -2pi radians to 2pi radians, where pi ÷ 3.141593. The default value for *start!* is 0 radians; the default value for *end!* is 2pi radians.

To convert values from degrees to radians, multiply the angle (in degrees) by pi/180 (approximately equal to 0.0174532925199433).

If *start!* or *end!* is negative, CIRCLE draws a radius to *start!* and treats the angle as positive. If *end!* is negative, CIRCLE draws a radius to *end!* and treats the angle as positive. If both *start!* and *end!* are negative, CIRCLE draws a radius to both *start!* and *end!* and treats the angle as positive.

The start angle can be less than the end angle. If you specify *end!* but not *start!*, the arc is drawn from 0 to end; if you specify *start!* but not *end!*, the statement draws an arc from *start!* to 2pi.

The aspect ratio is calculated as:

```
aspect = 4 * (ypixels/xpixels)/3
```

where xpixels by ypixels is the screen resolution.

If the aspect ratio is less than one, *radius!* is the x radius. If *aspect!* is greater than one, *radius!* is equal to the y radius.

To draw a radius to angle 0 (a horizontal line segment to the right), do not give the angle as -0; use a small nonzero value instead. For example:

```
SCREEN 2                                ' Draws a pie-shaped one-
CIRCLE (200,100),60,,-.0001,-1.57       ' quarter wedge of a circle.
```

You can omit an argument in the middle of the statement, but you must include the argument's commas. If you omit a trailing argument, do not include its corresponding commas. In the following statement, *color%* has been omitted:

```
CIRCLE STEP (150,200),94,,0.0,6.28
```

The CIRCLE statement updates the graphics cursor position to the center of the ellipse or circle.

You can use coordinates that are outside the screen or viewport.

You can show coordinates as absolutes, or you can use the STEP keyword to show the position of the center point in relation to the previous point of reference. For example, if the previous point of reference were (10,10), then:

```
CIRCLE STEP (10,5), 75
```

would draw a circle with radius 75, and center offset 10 from the current x coordinate and 5 from the current y coordinate. The circle's center center would be (20,15).

Example This example uses the CIRCLE statement to:

- Draw a circle with the upper-left quarter missing.
- Position a second circle, using relative coordinates, within the missing quarter.
- Draw a small ellipse inside the small circle, using a different aspect ratio.

```
CLS                                 ' Clear the screen.
CONST pi = 3.141593                 ' Define constant pi.
SCREEN 2                            ' Set screen mode to 2.

                                                ' Draw a circle with the upper-left
CIRCLE (320, 100), 200, , -pi, -pi / 2          ' quarter missing; use negative
                                                ' numbers so radii are drawn.

                                    ' Use relative coordinates to draw a circle
CIRCLE STEP(-100, -42), 100         ' within the missing quarter.

                                    ' Draw a small ellipse inside the circle.
CIRCLE STEP(0, 0), 100, , , , 5 / 25

                                    ' Display the drawing until a key is pressed.
LOCATE 25, 1: PRINT "Press any key to end.";
DO
LOOP WHILE INKEY$ = ""
SCREEN 0
```

CLEAR Method

Applies To CLIPBOARD object.

Action Clears the contents of the CLIPBOARD object.

Syntax **CLIPBOARD.CLEAR**

Remarks Use the CLIPBOARD object and applicable methods to provide cut, copy, and paste commands in your applications.

If the CLIPBOARD keyword is omitted, Visual Basic defaults to CLEAR statement functionality.

See Also CLEAR Statement.

Example This example illustrates the use of the CLEAR, GETTEXT, and SETTEXT methods and the CLIPBOARD object. Click the form to copy a text string from the Clipboard to a string variable. To try this example, create a form containing a text box control.

```
DECLARE SUB ClipboardTextDemo ()

SUB ClipboardTextDemo ()
    ON ERROR RESUME NEXT                ' Setup error handling.
    Msg$ = "Type anything you like into the text box below."
    Text1.text = INPUTBOX$(Msg$)        ' Get text from user.
    Msg$ = "Choose OK to copy the contents of the text box "
    Msg$ = Msg$ + "to the Clipboard."
    MSGBOX Msg$                         ' Display message.
    CLIPBOARD.CLEAR                     ' Clear Clipboard.
    CLIPBOARD.SETTEXT Text1.text        ' Put text on Clipboard.
    IF CLIPBOARD.GETTEXT() <> "" THEN
        Text1.text = ""                 ' Clear the text box.
        Msg$ = "The text is now on the Clipboard."
    ELSE
        Msg$ = "There is no text on the Clipboard."
        MSGBOX Msg$                     ' Display error message.
    END IF
    Msg$ = Msg$ + " Choose OK to copy the text from the "
    Msg$ = Msg$ + CHR$(13) + CHR$(10) + "Clipboard back to the text box"
    Msg$ = Msg$ + CHR$(13) + CHR$(10) + "in reverse order."
    MSGBOX Msg$                         ' Display message.
    Temp$ = CLIPBOARD.GETTEXT()         ' Get Clipboard text.
    FOR I% = LEN(Temp$) TO 1 STEP -1    ' Reverse the text.
        Text1.text = Text1.text + MID$(Temp$, I%, 1)
    NEXT I%
END SUB
SUB Form_Click ()
    CALL ClipboardTextDemo          ' Call Clipboard example.
END SUB
```

CLEAR Statement

Action

By itself, CLEAR performs the following actions:

- Closes all files.
- Releases all the buffers.
- Clears all common variables.
- Resets numeric variables and arrays.
- Initializes the stack.
- Sets string variables to null.
- CLEAR also changes the size of the stack if *stacksize&* is specified.

Syntax **CLEAR** [,,*stacksize&*]

Part	Description
stacksize&	Sets the size, in bytes, of the stack.

Remarks If your program has deeply nested subroutines or procedures, or if you use recursive procedures, you may want to use a CLEAR statement to increase the stack size. You may also want to increase the stack size if your procedures have a large number of arguments.

Clearing the stack destroys the return addresses placed on the stack during a GOSUB operation. This makes it impossible to execute a RETURN statement correctly, and Visual Basic generates a "RETURN without GOSUB" run-time error message. If you use a CLEAR statement in a SUB or FUNCTION statement, Visual Basic generates the run-time error message, "Illegal function call."

See Also CLEAR Method, STACK Function, STACK Statement.

Example This example uses the CLEAR statement to clear all program variables and to set aside 2,000 bytes on the stack for the program.

```
CLEAR , , 2000
```

Click Event

Applies To Check box, combo box, command button, directory list box, file list box, form, label, list box, menu, option button, picture box.

Action Occurs when the user selects a form or control using the mouse or the keyboard.

Syntax **SUB Form_Click ()**
SUB *ctlname*_**Click ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks For a form, the Click event occurs when the user clicks a disabled control or a blank area of a form using the left-mouse button.

For a control, the Click event occurs when the user:

- Clicks a control with the left mouse button.
- Selects an item in a combo box or list box, either by pressing the arrow keys or by clicking the left mouse button.
- Presses the Spacebar when a command button or check box has the focus.

Typically, you attach a Click event procedure to a command button, menu, or picture box control to carry out commands.

To distinguish between left and right mouse buttons, use the MouseDown and MouseUp events.

You can trigger the Click event in code by:

- Changing the ListIndex property to a new non-negative value.
- Setting a command button's Value property to True (-1).
- Setting an option button's Value property to True (-1).
- Changing a check box's Value property.

You can use a control's Value property to test for the state of the control in code.

A Click event generated for a form, file list box, list box, label, or picture box also includes this event ordering: MouseDown, MouseUp, Click.

When a menu title is selected, the associated menu is displayed. Any code written for the menu title's Click event is executed. It is not advisable to write menu title Click event code that will cause a Paint or any other events. A menu title's Click event should only be used to change property values of lower level menu items.

See Also DblClick Event, Default Property.

Example In this example, each time a picture box is clicked, it moves diagonally across a form. To try this example, create a form containing a picture box positioned in the lower-left corner of the form.

```
SUB Picture1_Click ()
    Picture1.Move Picture1.Left + 5, Picture1.Top - 2
END SUB
```

CLIPBOARD Object

Action The CLIPBOARD object is accessed by specifying the CLIPBOARD keyword. This object provides access to a temporary storage area used to cut, copy, or paste text between edit fields (in combo box and text box controls) in the current application.

Syntax **CLIPBOARD.***method*

Part	Description
method	Any valid CLIPBOARD method.

Remarks

Methods

CLEAR Method	GETTEXT Method	SETTEXT Method

The CLIPBOARD object is accessed by including the keyword CLIPBOARD when referencing applicable methods. For example:

```
CLIPBOARD.CLEAR
```

The CLIPBOARD object can contain only one set of data. This data is lost when another set of data is placed on the Clipboard.

The ability to cut, copy, and paste text in a text box is automatically provided to the user. The CLIPBOARD object allows you to use this feature with other controls by allowing you programmatic access.

The CLIPBOARD object can be used to facilitate common programming tasks. For example, the CLIPBOARD can be used in a simple search-and-replace routine.

Example

See CLEAR Method example.

CLOSE Statement

Action

Concludes I/O to a file, device, or ISAM table; with no arguments, closes all open files, devices, or ISAM tables (ISAM available in Professional Edition only).

Syntax

CLOSE [[#]*filenumber%* [,[#]*filenumber%*]...]

Part	Description
filenumber%	Number specifying an open file, device or ISAM table.

Remarks

The association of a file, device, or ISAM table with the argument *filenumber%* ends when a CLOSE statement is executed. You then can reopen the file, device, or table using the same or a different file number.

CLOSE with no arguments closes all open files or ISAM tables and commits any pending ISAM transactions.

A CLOSE statement for a file or device that was opened for sequential output writes the final buffer of output to that file or device.

CLOSE releases all buffer space associated with the closed file, device, or table.

The following statements also close all files and devices:

- CLEAR
- END
- RESET
- RUN
- SYSTEM
- CHAIN (with BC /O command-line option)

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they first open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

See Also OPEN Statement (File I/O), RESET Statement, END Statement, STOP Statement.

CLS Method

Applies To Form, picture box.

Action Clears text output generated at run time from a form or picture box. Use the CLS statement for non-form functionality.

Syntax [*form.*][*picturebox.*]**CLS**

Remarks If no object is specified, CLS behavior is determined by its location in code:

- If CLS is in a form module, it is a method, and applies to the default form.
- If CLS is in a non-form module, it is a statement, and applies to the entire screen.

The CLS method clears text generated at run time with the PRINT method.

Background images and controls placed on the form are not affected by the CLS method.

Text placed on a form or picture box while AutoRedraw is True (-1) is not affected by CLS if AutoRedraw is set to False (0) before CLS is executed. By carefully manipulating the AutoRedraw property of a form or picture box, some degree of permanence can be achieved.

After the CLS method is executed, the CurrentX and CurrentY properties of the form or picture box are reset to 0.

See Also CLS Statement.

Example The example uses the CLS method to erase printed information on a form.

```
DECLARE SUB ClsDemo ()
```

```
SUB ClsDemo ()
    AutoRedraw = -1                 ' Turn on AutoRedraw.
    ForeColor = 15                  ' Set foreground to white.
    BackColor = 1                   ' Set background to blue.
    Msg$ = "This is information printed on the form background."
    PRINT Msg$                      ' Print message to form.
    Msg$ = "Choose OK to clear the information"
    Msg$ = Msg$ + " just displayed on the form"
    MSGBOX Msg$                     ' Display message.
    CLS                             ' Clear form background.
END SUB

SUB Form_Click ()
   CALL ClsDemo
END SUB
```

CLS Statement

Action

Clears various parts of the display screen; not available when forms are showing. Use the CLS method when forms are showing.

Syntax

CLS [0 | 1 | 2]

Valid settings are:

Value	Description
None	By itself, CLS clears either the text or graphics viewport; if a graphics viewport has been set (using VIEW), only the graphics viewport is cleared; otherwise, the text viewport or entire screen is cleared.
0	Clears the screen of all text or graphics.
1	Clears the graphics viewport or the entire screen if a graphics viewport has not been set.
2	Clears the text viewport.

Remarks

If no object is specified, CLS behavior is determined by its location in code:

- If CLS is used in a form module, it is a method, and applies to the default form.
- If CLS is in a non-form or code module, it is a statement, and applies to the entire screen.

If CLS is used as a statement (with the arguments 0, 1, or 2) when forms are showing, Visual Basic generates the error message, "Invalid when forms are showing."

To use the CLS statement with arguments 0, 1, or 2, you must first unload or hide any visible forms.

The behavior of a particular CLS statement depends on:

- The CLS statement argument (0, 1, 2, or none).
- The current screen mode.
- Whether or not a VIEW statement has been executed that has changed the graphics viewport from the default of the entire screen.
- Whether or not a VIEW PRINT statement has been executed that has established a user-defined text viewport.
- Whether or not a KEY ON statement has been executed that has established the bottom text line as a list of function-key assignments. The bottom line on the screen may be 25, 30, 43, or 60, depending on the current screen mode.

The following conditions determine how the CLS statement works with graphics and text viewports:

- Screen mode 0 has only a text viewport. All other screen modes have both a text and graphics viewport.
- The default graphics viewport is the entire screen.
- The default text viewport is the entire screen, except for the bottom line of text.
- After a user-defined graphics viewport (or "clipping region") has been established by executing a VIEW statement, it is reset to the default by executing any of the following statements:
  - VIEW without arguments.
  - CLEAR.
  - Any SCREEN statement that changes the screen resolution.
- After a user-defined text viewport has been established by executing a VIEW PRINT statement, it is reset to the default by executing any of the following statements:
  - VIEWPRINT without arguments.
  - Any text-oriented statement (such as WIDTH) that changes the number of text columns or text lines on the screen.
- If the KEY ON statement has been executed, CLS statement regenerates the bottom screen line as the function-key assignment list or as a line of blank characters.

The following table summarizes how the CLS statement works:

Statement	Condition	Operations Performed
CLS 0	Any screen mode or combination of user-defined viewports.	Clears entire screen of all text and graphics except bottom text line. Clears graphics viewport. Regenerates bottom text line. Returns cursor to home.
CLS 1	Graphics screen mode with no user-defined graphics viewport.	Clears default graphics viewport. Does not clear text viewport. Regenerates bottom text line. Returns cursor to home.
CLS 1	Graphics screen mode with user-defined graphics viewport.	Clears user-defined graphics viewport. Does not clear text viewport. Does not regenerate bottom text line. Does not return cursor to home.
CLS 1	In screen mode 0 (text mode).	No effect.
CLS 2	Any screen mode.	Clears text viewport. Does not clear graphics viewport. Does not regenerate bottom text line. Note: The bottom line of text will not be cleared even if included in the user-defined text viewport.
CLS	Graphics-screen mode with no user-defined graphics viewport.	Clears default graphics viewport. Regenerates bottom text line. Returns cursor to home.
CLS	Graphics-screen mode with user-defined graphics viewport.	Clears graphics viewport. Does not regenerate bottom text line. Does not return cursor to home.
CLS	In Screen mode 0 (text mode).	Clears text viewport. Regenerates bottom text line. Returns cursor to home.

See Also CLS Method, VIEW PRINT Statement, VIEW Statement, WINDOW Statement.

Example

This example uses the VIEW PRINT statement to set up graphics and text viewports. The CLS statement is used to clear the graphics viewport after 30 circles have been drawn and to clear the text viewport after "Hello" has been printed 45 times.

```
CLS                                 ' Clear the screen.
RANDOMIZE TIMER
SCREEN 1
VIEW (5, 5)-(100, 80), 3, 1         ' Set up a graphics viewport with a border.
VIEW PRINT 12 TO 24                 ' Set up a text viewport.
                                    ' Print a message on the screen outside
                                    ' the text viewport.
LOCATE 25, 1: PRINT "Press any key to stop."
Count = 0
DO
                                    ' Draw a circle with a random radius.
    CIRCLE (50, 40), INT((35 - 4) * RND + 5), (Count MOD 4)
                                    ' Clear the graphics viewport every 30
                                    ' circles.
    IF (Count MOD 30) = 0 THEN CLS 1
    PRINT "Hello. ";
                                    ' Clear the text viewport every 45 times.
    IF (Count MOD 45) = 0 THEN CLS 2
    Count = Count + 1
LOOP UNTIL INKEY$ <> ""
WIDTH 80
SCREEN 0
```

COLOR Statement

Action

Changes the default color of text output to the screen; not available when forms are showing. Use the BackColor and ForeColor properties with forms and controls.

Use the COLOR syntax that matches the screen mode for which you are programming.

Syntax

Syntax 1: Screen Mode 0 (Text Only)

COLOR [*foreground&*] [,[*background&*] [, *border&*]]

Syntax 2: Screen Mode 1

COLOR [*background&*] [,*palette%*]

Syntax 3: Screen Modes 4, 12, 13

COLOR [*foreground&*]

Syntax 4: Screen Modes 7, 8, 9, 10

COLOR [*foreground&*] [, *background&*]

Part	Description
foreground&	Number that sets the foreground screen color; in screen mode 0, a color attribute that sets text color; in other screen modes, a color attribute or 4-bit color value (screen mode *4* only) that sets the text and line-drawing color.
background&	Number that sets the background screen color; in screen mode 0, a color attribute; in screen mode 1, a 4-bit color value; in screen modes 7-10, a color value.
border&	Color attribute that sets the screen border color.
palette%	Number (0 or 1) specifying which of two sets of color attributes to use:

Palette%	Attribute 1	Attribute 2	Attribute 3
0	Green	Red	Brown
1	Cyan	Magenta	Bright white

Remarks

References to the COLOR statement are not allowed when forms are showing and cause Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this statement.

The available color attributes for COLOR depend on your graphics adapter and the screen mode set by the most recent SCREEN statement.

Visual Basic generates the error message "Illegal function call" if you attempt to either:

- Specify COLOR statement arguments outside allowable ranges.
- Use the COLOR statement in screen modes 2, 3, or 11.

The foreground can be the same color as the background, making both characters and graphics invisible.

The default background color is black for all display hardware configurations and all screen modes. This background is produced by the default display color of 0 associated with the background color attribute of 0.

With an EGA, VGA, or MCGA installed, the PALETTE statement gives you flexibility in assigning different display colors to the attribute color ranges for the foreground, background, and border colors.

The foreground color value must be an integer expression between 0 and 31, inclusive. Sixteen colors can be selected with the integers 0-15. A blinking version of a color is specified by adding 16 to the color number. For example, a blinking color 7 is equal to 7+16, or 23.

The background color value is an integer expression between 0 and 7, inclusive, and is the color of the background for each text character. Blinking background colors are not supported.

The border color, which fills the screen area outside the text area, is an integer expression between 0 and 15, inclusive.

The *border&* argument has no effect with:

- IBM Enhanced Graphics Adapter (EGA).
- IBM Video Graphics Array adapter (VGA).
- IBM Multicolor Graphics Array adapter (MCGA).
- Forms.

In screen mode 1, a COLOR statement overrides PALETTE statements that were executed previously.

The background color value must be an integer expression between 0 and 3, inclusive.

The *palette%* argument is an integer expression with an odd or even value, between 0 and 255, inclusive. An even value makes available one set of three default foreground colors; an odd value makes available a different default set.

The following Visual Basic code defines the two default sets of colors in terms of EGA PALETTE statements:

```
                ' Definition of the even palette-number set.
COLOR ,0        ' In screen mode 1, an even palette number
                ' is equivalent to next three EGA PALETTE
                ' statements:
PALETTE 1,2     ' EGA Attribute 1 = color 2 (green).
PALETTE 2,4     ' EGA Attribute 2 = color 4 (red).
PALETTE 3,6     ' EGA Attribute 3 = color 6 (yellow).

                ' Definition of the odd palette number set.
COLOR ,1        ' In screen mode 1, an odd palette-number
                ' is equivalent to:
PALETTE 1,3     ' EGA Attribute 1 = color 3 (cyan).
PALETTE 2,5     ' EGA Attribute 2 = color 5 (magenta).
PALETTE 3,7     ' EGA Attribute 3 = color 7 (white).
```

With screen mode 4, the argument *foreground&* is an integer expression between 0 and 15, inclusive. This expression sets the display color associated with color attribute *1* only. The screen background color is fixed as black.

The foreground (line-drawing) color is a color attribute.

The background (screen) color is a display color.

In screen mode 11, you can set both the foreground and background colors by assigning a color to attribute 0 with a PALETTE statement.

With screen modes 12 and 13, the argument *foreground&* is a color attribute.

In screen modes 12 and 13, you can set the background color by assigning a color to attribute 0 with a PALETTE statement. For example, to make the background color 8224 (olive green), use the following PALETTE statement:

```
PALETTE 0,8224
```

To make the background color bright red-violet, use:

```
PALETTE 0,4128831
```

Example This example illustrates how to use the COLOR statement.

```
SCREEN 0                            ' Text mode only.
                                    ' Foreground = 1 (blue).
                                    ' Background = 2 (green).
                                    ' Border = 3 (cyan).
                                    ' EGA and VGA boards do not support the
                                    ' border argument.
COLOR 1, 2, 3
CLS                                 ' Clear the screen.
LOCATE 12, 25: PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""
SCREEN 1                            ' Set screen to 320 x 200 graphics.
COLOR 1, 0                          ' Background = 1 (blue).
                                    ' Foreground = even palette number
                                    ' (red, green, yellow).
LINE (20, 20)-(300, 180), 3, B      ' Draw a brown box.
LOCATE 12, 7: PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""              ' Background = 2 (green).
COLOR 2, 1                          ' Foreground = odd palette (white, magenta,
                                    ' cyan).
LINE (20, 20)-(300, 180), 3, B      ' Draw a bright white box.
LOCATE 12, 7: PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""
SCREEN 0                            ' Set screen to text mode.
COLOR 7, 0, 0                       ' Foreground = 7 (white),
CLS                                 ' Background = 0 (black),
                                    ' Border = 0 (black).
SCREEN 0
```

```
SCREEN 7
FOR i% = 0 TO 15
    COLOR i%
    PRINT i%
NEXT i%
SLEEP 5
WIDTH 80
SCREEN 0
```

COM Statement

Action

Enables, disables, or suspends event trapping on a communications port; can be used with object-specific events.

Syntax

COM(*n*%) {ON | OFF | STOP}

Part	Description
n%	Number of a COM (serial) port (1 or 2).
ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a COM ON statement is executed.

Remarks

If your program contains event-handling statements and you are compiling from the Basic Compiler (BC.EXE) command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

COM(*n*%) ON enables communications-event trapping at COM port *n*%. If a character arrives at a communications port after a COM(*n*%) ON statement, the routine specified in the ON COM statement is executed.

COM(*n*%) OFF disables communications-event trapping at COM port *n*%. No trapping takes place until another COM(*n*%) ON statement is executed. Events occurring while trapping is off are ignored.

COM(*n*%) STOP suspends communications-event trapping at COM port *n*%. No trapping takes place until a COM(*n*%) ON statement is executed. Any ON COM events that occur while trapping is suspended are remembered and processed when the next COM(*n*%) ON statement is executed. However, remembered events are lost if a COM(*n*%) OFF statement is executed.

When a signal-event trap occurs (that is, the GOSUB is performed), an automatic COM(*n*%) STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a COM(*n*%) ON statement unless an explicit COM(*n*%) OFF was performed inside the routine.

See Also ON COM Statement, OPEN COM Statement, PRINTER Object, UEVENT Statement.

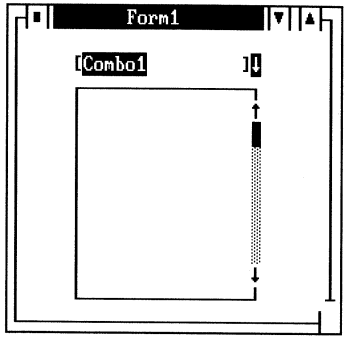
Example This example uses the OPEN COM statement to open the COM1 port for input and the COM statement to enable event trapping. The ON COM statement passes control to a subroutine when there is activity on a terminal.

Note To run this program, you must have a terminal connected to the COM1 port.

```
CLS                                          ' Clear the screen.
                                             ' Set up error handling in
ON ERROR GOTO ErrHandler                     ' case COM1 doesn't exist.
OPEN "COM1:9600,N,8,1,BIN" FOR INPUT AS #1 ' Open the COM port.
COM(1) ON                                    ' Turn on COM event processing.
ON COM(1) GOSUB Com1Handler                  ' Set up COM event handling.

' Wait for a COM event to occur or a key to be pressed.
DO
LOOP WHILE INKEY$ = ""
COM(1) OFF                                   ' Turn off COM event handling.
CLS
END

Com1Handler:
    PRINT A$; "Something was typed on the terminal attached to COM1."
    RETURN

ErrHandler:
    SELECT CASE ERR
         CASE 24: PRINT "COM1 is unavailable on this computer.": END
         CASE ELSE: END
    END SELECT
```

Combo Box Control

Action Allows the user to either enter a value or select from a list of options; similar to a text box and a list box combined.



Remarks

Events

Change	Click	DblClick	DragDrop
DragOver	DropDown	GotFocus	KeyDown
KeyPress	KeyUp	LostFocus	

Methods

ADDITEM	DRAG	MOVE	REFRESH
REMOVEITEM	SETFOCUS		

Properties

BackColor	CtlName	DragMode	Enabled
ForeColor	Height	Index	Left
List	ListCount	ListIndex	MousePointer
Parent	SelLength	SelStart	SelText
Sorted	Style	TabIndex	TabStop
Tag	Text	Top	Visible
Width			

A combo box combines the features of a text box with the features of a list box. The user can make a selection by either:

- Typing text into the text box.
- Selecting an item from the list below the text box.

Use the Style property to determine the behavior of the edit and list portions of a combo box.

When an item in the drop-down list is selected, the string is entered in the text box. Use the ADDITEM and REMOVEITEM methods to add and remove strings from the list portion of the combo box at run time.

The minimum size of a combo box is 1 x 5; dimensions correspond to the Height and Width property settings. Only the width of the list portion of a combo box can be changed; the height of a combo box control is read-only.

See Also List Box Control, Text Box Control.

Example This example shows the effect of using the Style property settings with combo box controls. To try this example, create a form containing three combo boxes. Set the Style properties of the second and third combo box:

- For Combo2, Style = 1 (Simple Combo).
- For Combo3, Style=2 (Dropdown List).

```
SUB Form_Load ()
   Combo1.ADDITEM "Luscious"              ' Add the entries.
   Combo1.ADDITEM "Lucy"
   Combo1.ADDITEM "Edwards"
   Combo2.ADDITEM "Edmark"
   Combo2.ADDITEM "Doo"
   Combo2.ADDITEM "Drew"
   Combo3.ADDITEM "Zoo"
   Combo3.ADDITEM "Zebra"
   Combo3.ADDITEM "Xylox"
END SUB
```

COMBUTTON_FORECOLOR Constant

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the foreground color of a command button.

Syntax **SCREEN.ControlPanel({COMBUTTON_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	COMBUTTON_FORECOLOR has a value of 4.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
0	(Default) Black.

Remarks The COMBUTTON_FORECOLOR constant sets the foreground color for all:

- Single-line commandbuttons (3D and non-3D).
- Multiline command buttons (non-3D only).

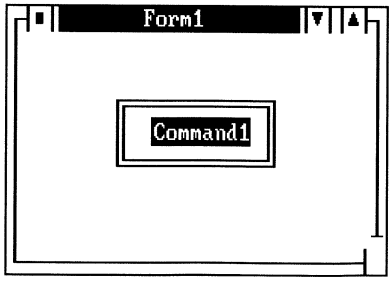
See Also Command Button Control.

Example This example illustrates the effect of changing the value of the COMBUTTON_FORECOLOR constant. To try this example, create a form containing a command button. Set the command button Caption property to "&DoIt."

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(COMBUTTON_FORECOLOR) = 4
END SUB
```

Command Button Control

Action Carries out a command or action when selected by the user.



Remarks

Events

Click	DragDrop	DragOver	GotFocus
KeyDown	KeyPress	KeyUp	LostFocus

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	Cancel	Caption	CtlName
Default	DragMode	Enabled	Height
Index	Left	MousePointer	Parent
TabIndex	TabStop	Tag	Top
Value	Visible	Width	

A command button triggers an action when the user chooses it. A command button is sometimes called a push button because it appears to be pushed in when it has been chosen.

Setting the Value property of a command button to True (-1) invokes the command button's Click event. The Value property is always equal to True (-1) inside the Click event and is reset to False (0) outside the Click event.

The user chooses a command button either by clicking the button or by pressing Enter when the button has the focus. For example, the user can choose a command button to indicate that the entries in a dialog box are correct. The application then continues processing based on the user's dialog selections.

To display text on a command button, set the Caption property.

If you want the user to activate a command button by pressing Esc, set the Cancel property to True (-1). Only one button per form can have this setting.

If you want the user to activate a command button that does not have the focus by pressing Enter, set the Default property to True (-1). Only one button per form can have this setting.

The minimum size of a command button is 1 x 3; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value. Height settings 1 and 2 change the appearance of the control:

Example

In this example, a control disapears and reappears when the command button is clicked. To try this example, create a form containing a picture box and a command button.

```
SUB Command1_Click ()
    IF Picture1.Visible = 0 THEN
          Picture1.Visible = -1
          Command1.Caption = "Visible"
    ELSE
          Picture1.Visible = 0
          Command1.Caption = "Invisible"
    END IF
END SUB
```

COMMAND$ Function

Action Returns a string containing the argument portion of the command line used to start Visual Basic or an application developed with Visual Basic.

Syntax **COMMAND$**

Remarks In the following example, cmdlinearg represents the argument information returned by the COMMAND$ function when you start Visual Basic with the /CMD switch:

```
VBDOS /CMD cmdlinearg
```

For applications developed with Visual Basic, COMMAND$ returns any argument that appears after the name of the application. For example:

```
MYAPP cmdlinearg
```

COMMAND$ converts all letters to uppercase (capital letters) and removes all leading blanks from the command line.

The COMMAND$ function can be used in stand-alone executable files, or if you are executing from the Visual Basic environment (VBDOS), by either:

- Using the /CMD command-line option.
- Choosing Modify COMMAND$ from the Run menu.

Example This example uses the COMMAND$ function to break the command line into separate arguments. Each argument is separated from adjoining arguments by one or more blanks or tabs on the command line.

```
DEFINT A-Z          ' Default variable type is integer in this module.

' Declare the Comline SUB procedure, as well as the number and type of
' its parameters.

DECLARE SUB Comline (N, A$(), Max)
CLS                         ' Clear the screen
```

```
DIM A$(1 TO 15)
CALL Comline(N, A$(), 10)      ' Get what was typed on the command line.
PRINT "Number of arguments = "; N ' Print out each part of the command line.
PRINT "Arguments are: "
FOR I = 1 TO N: PRINT A$(I): NEXT I

' The following is a sample command line and output for a stand-alone
' executable file (assumes program name is ARG.EXE):
'
' arg one  two   three    four     five      six

' Use SUB procedure to get command line and split into arguments.
' Parameters:  NumArgs : Number of command-line arguments found.
'              Args$() : Array in which to return arguments.
'              MaxArgs : Maximum number of arguments array can return.

STATIC SUB Comline (NumArgs, Args$(), MaxArgs)
CONST TRUE = -1, FALSE = 0

    NumArgs = 0: In = FALSE
    ' Get the command line using the COMMAND$ function.
    Cl$ = COMMAND$
    L = LEN(Cl$)
    ' Go through the command line a character at a time.
    FOR I = 1 TO L
         C$ = MID$(Cl$, I, 1)
         ' Test for character being a blank or a tab.
         IF (C$ <> " " AND C$ <> CHR$(9)) THEN
         ' Neither blank nor tab; test if you're already inside.
         ' an argument
              IF NOT In THEN
              ' You've found the start of a new argument.
                   ' Test for too many arguments.
                     IF NumArgs = MaxArgs THEN EXIT FOR
                     NumArgs = NumArgs + 1
                     In = TRUE
              END IF
              ' Add the character to the current argument.
              Args$(NumArgs) = Args$(NumArgs) + C$
         ELSE
         ' Found a blank or a tab.
              ' Set "Not in an argument" flag to FALSE.
              In = FALSE
         END IF
    NEXT I

END SUB
```

COMMON Statement

Action Defines global variables for sharing between modules or between chained programs.

Syntax **COMMON [SHARED]** [*/blockname/*] *variable*[()] [**AS** *type*] [, *variable*[()] [**AS** *type*]]...

Part	Description
SHARED	Indicates variable to be shared with all SUB or FUNCTION procedures in the module.
/blockname/	Name of memory block location where COMMON variables are stored (slashes required); if omitted, variables are stored in a blank common area.
variable	Any valid variable name.
()	Indicates array declaration.
AS *type*	Any valid data type; use a separate AS type clause for each variable defined.

Remarks Use DIM rather than COMMON when you want to share access to variables within a single module.

When using COMMON SHARED to share arrays between forms, the correct syntax is:

```
COMMON SHARED MyArray() AS INTEGER
```

This statement must appear at the module level of all forms that will share the array. Use REDIM in a Form_Load event procedure to dimension the array.

Errors caused by mismatched COMMON statements are subtle and difficult to find. You can avoid mismatched COMMON statements by:

- Placing COMMON declarations in a single include file.
- Using the $INCLUDE metacommand in each module.

The following code fragment shows how to use the $INCLUDE metacommand to share a file containing COMMON statements among programs:

```
'$INCLUDE:'COMDEF.BI'          ' This file is MENU.BAS.
.
.
.
CHAIN "Prog1"
END
```

```
'$INCLUDE:'COMDEF.BI'          ' This file is PROG1.BAS.
.
.
.
END
DIM A(100),B$(200)             ' This file is COMDEF.BI.
COMMON I,J,K,A()
COMMON A$,B$(),X,Y,Z           ' End COMDEF.BI.
```

When you compile a program using the Visual Basic Compiler (BC.EXE) with the /O option, common variables are not preserved across the chain. To preserve values across a chain, you must either:

- Compile without the /O option (using BC).
- Select the EXE Requiring Run-Time Module option from the Make EXE File dialog box (using VBDOS).

Unless it has been declared as a static array in a preceding DIM statement, an array variable in a COMMON statement is a dynamic array. Its dimensions must be set in a later DIM or REDIM statement.

Both static and dynamic arrays are placed in COMMON by using the array name followed by parentheses. Follow these rules when dimensioning an array:

- For static arrays, dimensions must be set with integer-constant subscripts in a DIM statement preceding a COMMON statement.
- For dynamic arrays, dimensions must be set in a later DIM or REDIM statement.

The elements of a dynamic array are not allocated in the COMMON block. Only an array descriptor is placed in COMMON.

A COMMON statement establishes storage for variables in a special area that allows them to be shared between modules or with other programs invoked with a CHAIN statement.

Because COMMON statements establish global variables for an entire program, they precede before any executable statements. All statements are executable except the following:

```
COMMON                         CONST
DATA                           DECLARE
DEFtype                        DIM (for static arrays)
$DYNAMIC                       $FORM
$INCLUDE                       OPTION BASE
OPTION EXPLICIT                REM
SHARED                         $STATIC
STATIC                         TYPE...END TYPE
```

When blockname is used, the COMMON block is a named COMMON block. When blockname is omitted, the block is a blank COMMON block. Items in a named COMMON block are not preserved across a chain to a new program.

The size of a common area can be different from that in another module or chained program if a blank COMMON block has been used. When a Visual Basic program shares COMMON blocks with a routine in the user library, the calling program may not redefine the COMMON block to a larger size.

Use the blockname argument to share only specific groups of variables. You cannot use a type-declaration character (%, &, !, #, @, $) as part of blockname.

Variables in COMMON blocks are matched by position and type, not by name. Thus, variable order is significant in COMMON statements. For example, in the following fragment, the order of variables in COMMON statements links the variables, not the names:

```
COMMON A, D, E                  ' Module level.
A = 5 : D = 8 : E = 10
.
.
.
COMMON A, E, D                  ' Common statement in another module.
.                               ' A = 5, E = 8, D = 10.
.
.
```

See Also $INCLUDE Metacommand, OPTION EXPLICIT Statement.

Example This example uses the COMMON SHARED statement to share access to a variable in a multiple-module application. To try this example, create a form containing a text box and a command button. Make the text box at least 35 characters wide.

Copy the first section of code example below to the form module. Choose New Module from the File menu to create and add the module MODULE.BAS to the project. Copy the second section of code example below to the code module, MODULE.BAS. Choose Code from the View menu and select FORM1.FRM to return to the start-up module and run the example.

```
' The section of code below goes in FORM1.FRM.

COMMON SHARED Sampletext$
COMMON SHARED i%

SUB Command1_Click ()
    i% = i% + 1                         ' Modifies and shares i%.
    CALL ChangeText
    Text1.Text = Sampletext$            ' Displays shared Sampletext$.
END SUB
```

```
' The section of code below goes in MODULE.BAS.

DECLARE SUB ChangeText ()
' $FORM Form1                          ' Share form properties.
COMMON SHARED Sampletext$
COMMON SHARED i%

SUB ChangeText ()
    SELECT CASE i%
    CASE 1
         Sampletext$ = "First there was a form"
         Form1.BackColor = 13
    CASE 2
         Sampletext$ = "Then there was a control"
         Form1.BackColor = 5
    CASE ELSE
         Sampletext$ = "Now there is a project"
         Form1.BackColor = 4
    END SELECT
END SUB
```

CONST Statement

Action Declare symbolic constants for use in place of values.

Syntax **CONST** *constantname* = *expression* [, *constantname* = *expression*]...

Part	Description
constantname	Name of the constant that consists of up to 40 characters and begins with a letter; valid characters are A-Z, 0-9, and period (.).
expression	Expression that is assigned to the constant; *expression* can consist of literals (such as 1.0), other constants, any arithmetic or logical operators except exponentiation (^), or a single literal string.

Remarks If you use a type-declaration character in the name, you can omit the character when the name is used, as shown in the following example:

```
CONST MAXDIM% = 250
.
.
.
DIM AccountNames$(MAXDIM)
```

If you omit the type-declaration character, the constant is given a type based on the expression in the CONST statement. Strings always yield a string constant. With numeric expressions, the expression is evaluated and the constant is given the simplest type that can

represent the constant. For example, if the expression gives a result that can be represented as an integer, the constant is given an integer type.

Names of constants are not affected by DEFtype statements such as DEFINT. A constant type is determined either by an explicit type-declaration character or by the type of the expression.

Constants must be defined before they are referred to. The following example produces an error because the constant ONE is not defined before it is used to define TWO (constants are defined from left to right):

```
CONST TWO = ONE + ONE, ONE = 1
```

Constants declared in a SUB or FUNCTION procedure are local to the SUB or FUNCTION. A constant declared outside a procedure is defined throughout the module. You can use constants anywhere that you would use an expression.

A common programming practice is to use a statement such as the following (any nonzero value represents "true"):

```
TRUE = -1
```

Using constants offers several advantages over using variables for constant values:

- You only define constants once for an entire module.
- Constants cannot be inadvertently changed.
- Using constants rather than variables produces faster and often more compact code in stand-alone programs.
- Constants make programs easier to modify.

Example

This example uses the CONST statement to declare symbolic constants for the ASCII values of nonprinting characters such as tab and linefeed. Constants also make programs easier to modify.

Note You must supply a text file when you run this example, either by:

- Using a text file you have already created.
- Creating a file with a text editor.
- Specifying the CONSTANT.BI text file.

```
' This program counts words, lines, and characters.
' A word is any sequence of nonblank characters.
DEFINT A-Z
CONST BLANK = 32, ENDFILE = 26, CR = 13, LF = 10
CONST TABC = 9, YES = -1, NO = 0

CLS                              ' Clear the screen.
FileName$ = COMMAND$             ' Get the file name from the command line.
```

```
IF FileName$ = "" THEN
    INPUT "Enter input file name: ", FileName$
    IF FileName$ = "" THEN END
END IF
OPEN FileName$ FOR INPUT AS #1
Words = 0
Lines = 0
Characters = 0
                        ' Set a flag to indicate you're not looking at a
InaWord = NO            ' word, then get the first character from the file.
DO UNTIL EOF(1)
    C = ASC(INPUT$(1, #1))
    Characters = Characters + 1
    IF C = BLANK OR C = CR OR C = LF OR C = TABC THEN
    ' If the character is a blank, carriage return,
    ' line feed, or tab, you're not in a word
        IF C = CR THEN Lines = Lines + 1
            InaWord = NO
        ELSEIF InaWord = NO THEN
                                ' The character is a printing character,
            InaWord = YES       ' so this is the start of a word.
            Words = Words + 1   ' Count the word and set the flag.
        END IF

LOOP
PRINT "Number of: "             ' Display results.
LOCATE 3, 13: PRINT "Characters = "; Characters
LOCATE 4, 13: PRINT "Words =      "; Words
LOCATE 5, 13: PRINT "Lines =      "; Lines
END
```

ControlBox Property

Applies To Form.

Action Determines if a Control-menu box appears on a form at run time; read-only at run time.

Syntax [*form.*]**ControlBox**

Valid settings are:

Value	Description
True (-1)	(Default) Displays Control-menu box.
False (0)	Removes Control-menu box.

Remarks Double-clicking the control-menu box closes the active window. If this is the last form displayed, this action will end the program.

The Control menu is often used to provide a window control keyboard interface for non-mouse users.

A Control-menu box cannot be displayed on a form with BorderStyle set to 0 (none).

Either a modal or a modeless window can have a Control-menu box.

See Also BorderStyle Property, MinButton Property, MaxButton Property.

ControlPanel Property

Applies To SCREEN object.

Action Sets or returns the value of many useful run-time settings. In general, ControlPanel constants are used to initialize global system colors and characteristics.

Syntax **SCREEN.ControlPanel**({*subscript* | *constant* })[= *setting%*]

Remarks

Part	Description
subscript *constant*	Valid constant, variable or numeric expression specifying the ControlPanel array subscript. Mapped array values are:

Subscript	Constant
0	ACCESSKEY_FORECOLOR
1	ACTIVE_BORDER_BACKCOLOR
2	ACTIVE_BORDER_FORECOLOR
3	ACTIVE_WINDOW_SHADOW
4	COMBUTTON_FORECOLOR
5	DESKTOP_BACKCOLOR
6	DESKTOP_FORECOLOR
7	DESKTOP_PATTERN
8	DISABLED_ITEM_FORECOLOR
9	MENU_BACKCOLOR
10	MENU_FORECOLOR
11	MENU_SELECTED_BACKCOLOR
12	MENU_SELECTED_FORECOLOR
13	SCROLLBAR_BACKCOLOR

Part	Description
subscript constant	Valid constant, variable or numeric expression specifying the ControlPanel array subscript. Mapped array values are:
setting%	Valid color or other setting as defined by the specified constant.

Subscript	Constant
14	SCROLLBAR_FORECOLOR
15	THREE_D
16	TITLEBAR_BACKCOLOR
17	TITLEBAR_FORECOLOR

ControlPanel constants are defined in the CONSTANT.BI include file. To use these constants, you must use the $INLUDE metacommand at the beginning of each module where the constants are used:

```
' $INCLUDE: 'CONSTANT.BI'
```

See Also BackColor Property, ForeColor Property.

Example See DESKTOP_BACKCOLOR Constant example.

COS, SIN, and TAN Functions

Action Returns the cosine (COS), sine (SIN), or tangent (TAN) of a specified angle.

Syntax

COS(*angle*)
SIN(*angle*)
TAN(*angle*)

Part	Description
angle	Any angle expressed in radians.

Remarks The cosine of an angle in a right triangle is the ratio between the length of the side adjacent to the angle and the length of the hypotenuse.

The sine of an angle in a right triangle is the ratio between the length of the side opposite the angle and the length of the hypotenuse.

The COS, SIN, and TAN functions are calculated in single precision if numeric-expression or angle is an integer or single-precision value. If you use any other numeric data type, they are calculated in double precision.

TAN (tangent) is the inverse of ATN (arctangent).

The tangent of an angle (the value returned by TAN) in a right triangle is the ratio between the length of the side opposite an angle and the length of the side adjacent to it.

To convert values from degrees to radians, multiply the angle (in degrees) by pi/180 (or .0174532925199433) where pi = 3.141593.

To convert a radian value to degrees, multiply radians by 180/pi (or 57.2957795130824).

Pi has the following approximate value:

```
pi = 3.14159265359      or      pi = 4.0 * ATN(1.0)
```

See Also ATN Function.

Example See ATN Function example.

CtlName Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, timer control, vertical scroll bar.

Action Name used to identify the control; not available at run time.

Remarks By default, Visual Basic defines the name of a control (the CtlName property) as the control type, plus a unique integer, starting at 1. The numbers assigned are cumulative — for example, Option1, Option2, Option3, and so on. This yields a unique naming scheme for each control.

Default control names are:

Control	Default	Control	Default
Check box	CheckN	Combo box	ComboN
Command button	CommandN	Directory list box	DirN
Drive list box	DriveN	File list box	FileN
Frame	FrameN	Horizontal scroll bar	HScrollN
Label	LabelN	List box	ListN
Menu	MenuN	Option button	OptionN
Picture box	PictureN	Text box	TextN
Timer	TimerN	Vertical scroll bar	VscrollN

To change the default, specify a unique name for the control. A valid control name has the following characteristics:

- Must start with a letter.

- Can include alphanumeric or underscore (_) characters.
- Cannot exceed 40 characters. Remember "control name" + "_" + "longest event name" must not exceed 40 characters total.

A CtlName property cannot be a Visual Basic reserved word.

To create a control array, specify the same CtlName property for each control in the array.

You cannot refer to the CtlName property itself in code. If you need to identify a control to be passed to a procedure, use the Tag property.

Changing the name of a control does not automatically change the name of related event procedures. You may need to manually rename event procedures to activate them for the renamed control.

See Also FormName Property, Index Property, Tag Property.

Ctype Functions (CCUR, CINT, CLNG, CSNG, CDBL)

Action Converts a numeric expression to one of the following data types:

- Currency (CCUR).
- Integer (CINT).
- Long (4-byte) integer (CLNG).
- Single-precision (CSNG).
- Double-precision (CDBL).

Syntax C*type*(*numeric-expression*)

Part	Description
type	Valid type values are:
numeric-expression	Any valid numeric expression

Type	Description
INT	Integer (2-byte).
LNG	Long integer (4-byte).
SNG	Single-precision floating-point (4-byte).
DBL	Double-precision floating-point (8-byte).
CUR	Currency (8-byte signed fixed-point number).

Remarks

The valid range for CINT and CLNG numeric expressions is:

Function	Range
CINT	-32,768 to 32,767, inclusive.
CLNG	-2,147,483,648 to 2,147,483,647, inclusive.
CSNG	<= 88.02969.
CDBL	<= 709.782712893.

If *numeric-expression* is outside the valid range, Visual Basic generates the run-time error message, "Overflow."

The CSNG function has the same effect as assigning the numeric expression to a single-precision variable. Likewise, the CDBL function has the same effect as assigning *numeric-expression* to a double-precision variable.

The results of CDBL are no more accurate than the original expression. The added digits of precision are not significant unless the expression is calculated with double-precision accuracy.

CSNG rounds the value, if necessary, before converting it.

See Also

INT Function.

Example

This example use the CSNG and CDBL functions to demonstrate how floating- point precision can be lost when a double-precision number is converted to single precision.

```
CLS                                    ' Clear the screen.
d# = 7# / 9#
PRINT "d# originally = "; TAB(40); d#
s! = CSNG(d#)                          ' Convert double (d#) to single (s!).
PRINT "s! assigned value from d#, now = "; TAB(40); s!
d# = CDBL(s!)                          ' Convert single (s!) back to double (d#).
PRINT "d# assigned value back from s!, now = "; TAB(40); d#
```

CURDIR$ Function

Action

Returns the path currently in use for the specified drive.

Syntax

**CURDIR$**[(*drive$*)]

Part	Description
drive$	String expression that specifies a drive.

Remarks

The argument *drive$* must be in the range A to lastdrive, where lastdrive is the maximum drive letter you may have set in your CONFIG.SYS file.

If no drive is specified or if *drive$* is a null string, CURDIR$ returns the path for the currently selected drive. This is similar to using the CHDIR command at the system prompt without specifying a path.

Visual Basic generates an error if:

- The first character *drive$* lies outside the range A to lastdrive, unless you are working in a networked environment.
- The first character in *drive$* corresponds to a drive that does not exist.
- The first letter in *drive$* is not a letter.

CURDIR$ is not case sensitive ("C" is the same as "c").

Since only the first letter of the *drive$* argument is used, the colon (:) is always optional.

Example See CHDRIVE Statement example.

CurrentX, CurrentY Properties

Applies To Form, picture box.

Action Sets or returns the current horizontal position or column number (CurrentX), or vertical position or row number (CurrentY).

Syntax [*form.*][*picturebox.*]**CurrentX**[= *coordinate%*]
[*form.*][*picturebox.*]**CurrentY**[= *coordinate%*]

Part	Description
coordinate%	Valid settings are:

Value	Description
0 - 254	Valid X or Y coordinate where 0 is the default.

Remarks The X and Y coordinates are expressed in characters and are relative to the upper-left corner of the specified form or control.

Invoking certain methods immediately alters CurrentX and CurrentY values:

Method	Action	Effect on CurrentX, CurrentY
CLS	Clears form or control.	Sets CurrentX and CurrentY to (0,0) respectively.
PRINT *exp*;	Prints expressionlist.	CurrentX and CurrentY take on coordinate values reflecting the position immediately after the last character printed.

Method	Action	Effect on CurrentX, CurrentY
PRINT *exp$*	Prints expressionlist.	CurrentY is increased by 1 and CurrentX is set to zero.
PRINT *exp$*,	Prints expressionlist.	CurrentX and CurrentY take on coordinate values reflecting the starting position of the next print zone on the same line.

Visual Basic generates the error "Overflow" when PRINT method output has exceeded the 0 to 254 print range governed by the CurrentX and CurrentY property values.

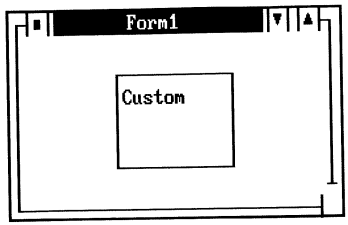
See Also CLS Method, PRINT Method.

Example This example displays the CurrentX and CurrentY property values of a picture box control when you click the form. To try this example, create a form containing a picture box control about 3 inches wide. Set the picture box's AutoRedraw property to True.

```
SUB Form_Load ()
     ' Display the current X and Y location.
    MsgA$ = "The CurrentX and CurrentY positions are:"
    MsgB$ = STR$(Picture1.CurrentX) + "," + STR$(Picture1.CurrentY)
    MsgC$ = "Choose OK, then click the form to print to the picture box."
    MSGBOX MsgA$ + MsgB$ + CHR$(10) + CHR$(13) + MsgC$
END SUB

SUB Form_Click ()
    Picture1.PRINT "This is just a test..." ' Print to the form.
     ' Display the current the new X and Y location.
    MsgA$ = "CurrentX and CurrentY values have changed to:"
    MsgB$ = STR$(Picture1.CurrentX) + "," + STR$(Picture1.CurrentY)
    MSGBOX MsgA$ + MsgB$
END SUB
```

Custom Control

Action Custom controls extend the set of standard controls available in Visual Basic for MS-DOS. You access this custom-control functionality in Visual Basic via custom-control libraries (.LIB) and Quick libraries (.QLB).



Remarks Like standard controls, custom controls recognize a predefined set of properties, events, and methods. For more information on the specific features and functionality of a custom control, refer to the documentation that came with the control.

Only one Quick library can be loaded at a time. Therefore, to use more than one custom control within your application, the custom-control Quick libraries must be combined. Use the LINK utility to combine Quick libraries.

When a custom control Quick library is loaded with Visual Basic, the control(s) defined within that Quick library appear in the Toolbox and on the Tools menu within the Form Designer.

At design time, custom controls appear as picture box controls with the control name displayed inside the control.

The actual "look" of a custom control as defined by the custom-control developer is not available until run time.

At design time, property settings other than Left, Top, Width, Height, and BorderStyle have no visible effects. In addition, no control-specific range checking is done for property values that you specify. Be sure to verify that your custom-control property settings will be valid at run time by referring to the specific custom control documentation.

See Also Chapter 19 in the *Programmer's Guide*, or Chapter 1 of *Professional Features* (available in Professional Edition only).

CVx Functions

Action Used with a FIELD statement to read numbers from a random-access file; each function takes a string and converts it to a value of the corresponding numeric type.

Syntax **CV***x*(*stringexpression$*)

Part	Description
x	One of the following letters, indicating the size and type of the converted value:

Function	Expression Type	Inverse Function
CVI	2-byte integer.	MKI$
CVL	4-byte long integer.	MKL$
CVS	4-byte single-precision.	MKS$
CVD	8-byte double-precision.	MKD$
CVC	8-byte currency.	MKC$

stringexpression$	Any valid string expression.

Remarks Visual Basic record variables provide a more efficient and convenient way of reading and writing random-access files than some earlier versions of Basic.

See Also CVxMBF Functions, FIELD Statement, MKx$ Functions.

Example This example uses the MKS$ and CVS functions to convert values to and from strings for storage in a random-access file.

Note If you run this program, you will be prompted to enter an account number. The account numbers in the sample data file ACCOUNT.INF are 1 - 5.

```
TYPE Buffer                         ' Define a user type for the data records.
    AccName AS STRING * 25
    Check   AS STRING * 4
END TYPE

DIM BankBuffer AS Buffer            ' Define a variable of the variable type.

CLS                                 ' Clear the screen.
OPEN "ACCOUNT.INF" FOR RANDOM AS #1 LEN = 29
CLS
RESTORE
READ AccName$, Check
I% = 0
DO WHILE UCASE$(AccName$) <> "END"
    I% = I% + 1
    BankBuffer.AccName = AccName$
    BankBuffer.Check = MKS$(Check)
    PUT #1, I%, BankBuffer
    READ AccName$, Check$
    IF AccName$ = "END" THEN EXIT DO
LOOP
CLOSE #1
```

```
DATA "Bob Hartzell", 300
DATA "Alice Provan", 150
DATA "Alex Landow", 75
DATA "Walt Riley", 50
DATA "Georgette Gump", 25
DATA "END", 0

OPEN "ACCOUNT.INF" FOR RANDOM AS #2 LEN = 29
FIELD #2, 25 AS AccName$, 4 AS Check$
DollarFormat$ = "$$######.##"
DO
    PRINT
    DO
        CLS                         ' Clear the screen.
        INPUT "Enter account # to update: ", Rec%
        GET #2, Rec%                ' Get the record
        PRINT : PRINT "This is the account for "; AccName$
        PRINT : INPUT "Is this the account you wanted"; R$
    LOOP WHILE UCASE$(MID$(R$, 1, 1)) <> "Y"
    Checkamt! = CVS(Check$)         ' Convert string to single-precision value.
    PRINT
    PRINT "The opening balance for this account is";
    PRINT USING DollarFormat$; Checkamt!
    PRINT : PRINT "Enter the checks and cash withdrawals for this"
    PRINT "account below. Enter 0 when finished."
    PRINT
    DO
        INPUT "Enter amount -> ", Checkout!
        Checkamt! = Checkamt! - Checkout!
    LOOP UNTIL Checkout! = 0
    PRINT
    PRINT "Enter the deposits for this account below."
    PRINT "Enter 0 when finished."
    PRINT
    DO
        INPUT "Enter amount ->", Checkin!: Checkamt! = Checkamt! + Checkin!
    LOOP UNTIL Checkin! = 0
    PRINT
    PRINT "The closing balance for this account is";
    PRINT USING DollarFormat$; Checkamt!
    LSET Check$ = MKS$(Checkamt!)' Convert single-precision number to string.
    PUT #2, Rec%                    ' Store the record.
    PRINT : INPUT "Update another"; R$
LOOP UNTIL UCASE$(MID$(R$, 1, 1)) <> "Y"
CLOSE #2
KILL "ACCOUNT.INF"
END
```

CVxMBF, MKxMBF$ Functions

Action Converts Microsoft-binary-format (MBF) numeric strings to IEEE-format numbers (CVSMBF, CVDMBF); converts IEEE-format numbers to Microsoft-binary-format numeric strings (MKSMBF$, MKDMBF$).

Syntax **CV***x***MBF**(*string$*)
MK*x***MBF$**(*numeric-expression*)

Part	Description
x	One of the following letters, indicating the type of the expression to convert:

Function	Expression Type	Return Value
CVSMBF	4-byte string.	Single-precision (IEEE).
CVDMBF	8-byte string.	Double-precision (IEEE).
MKSMBF$	Single-precision numeric.	4-byte (MBF).
MKDMBF$	Double-precision numeric.	8-byte (MBF).

string$	Any valid string expression

Remarks CVxMBF and MKx$ functions are used to write real numbers to random-access files using Microsoft binary format (MBF). They are particularly useful for maintaining data files created with earlier versions of Basic.

The MKSMBF$ and MKDMBF$ functions convert real numbers in IEEE format to strings containing MBF numbers so they can be written to the random-access file.

To write a real number to a random-access file in Microsoft binary format:

- Convert the number to a string using MKSMBF$ (for a single-precision number) or MKDMBF$ (for a double-precision number).
- Store the result in the corresponding field (defined in the FIELD statement).
- Write the record to the file using the PUT statement.

Currency type numbers are not stored in IEEE floating-point format, so they cannot be converted to strings that contain Microsoft binary format numbers.

See Also CVx Functions, MKx$ Functions.

Example This example uses the MKSMBF$ and CVSMBF functions.

```
CLS                                    ' Clear the screen.
TYPE Buffer
    SngNum AS STRING * 4
    DblNum AS STRING * 8
END TYPE
DIM RecBuffer AS Buffer
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN = 12
SNum = 98.9
DNum = 645.3235622#
RecBuffer.SngNum = MKSMBF$(SNum)
RecBuffer.DblNum = MKDMBF$(DNum)
PUT #1, 1, RecBuffer
GET #1, 1, RecBuffer
CLOSE #1
PRINT CVSMBF(RecBuffer.SngNum), CVDMBF(RecBuffer.DblNum)
```

DATA, READ, RESTORE Statements

Action Specifies values to be read by subsequent READ statements (DATA); reads values and assigns them to variables (READ); prepares values in specified DATA statements to be reread (RESTORE).

Syntax

DATA *constant* [,*constant*]...
READ *variablelist*
RESTORE [*label* | *linenumber*]

Part	Description
constant	Any valid numeric or string constant specifying the data to be read.
variablelist	One or more valid variables, separated by commas, that are assigned data values.
label *linenumber*	The label or line number of a DATA statement; if omitted, the next READ statement reads values in the first DATA statement found in the program.

Remarks A DATA statement can contain as many constants as will fit on a line (constants must be separated by commas). There is no limit to the number of DATA statements that can be used in a single program.

DATA statements are used in the order in which they appear in the source file. Consider the items in several DATA statements as one continuous list of items, regardless of how many items are in a statement or where the statement appears in the program.

Names of symbolic constants (defined in a CONST statement) that appear in DATA statements are interpreted as strings rather than as names of constants. For example, in the following code fragment, the second data item is a string (PI), and not the value 3.141593:

```
CONST PI = 3.141593            ' PI defined as constant = 3.141593.
.
.
.
DATA 2.20,PI,45,7              ' Statement recognizes PI, not value.
.
.
.
```

Null data items (indicated by a missing value) can appear in a data list:

```
DATA 1,2,,4,5
```

When a null item is read into a numeric variable, the variable has the value 0. When a null item is read into a string variable, the variable has the null string value ("").

When you are working in the programming environment, you can only enter DATA statements in module-level code. Visual Basic moves all DATA statements not in the module-level code to the module-level code when it reads a source file.

REM statements appearing at the end of DATA statements must be preceded by a colon (:). Otherwise, Visual Basic interprets trailing REM statements as string data.

Do not use the single-quotation form of REM (') in a DATA statement because it will be considered valid data.

READ statements must appear in the same program module as the DATA statement.

Each variable in a READ statement receives its value from a DATA statement. Which value the variable receives depends on how many values have previously been read.

Since the values of all DATA statements in a module can be considered a single list of values, each value in this list is assigned in turn to the variables specified in READ statements.

The number of values specified in a given DATA statement or variables specified in a READ statement doesn't matter: If you attempt to read more values than are specified in all the statements combined, Visual Basic generates the error message, "Out of Data."

Only individual elements of a record variable can appear in a READ statement.

Previously read DATA statements can be reread using the RESTORE statement.

When you attempt to read data of one data type into a variable with a different data type, the following results can occur:

Action	Result
String value is read into a numeric variable.	Run-time error.
Numeric value is read into a string variable.	Value is stored as a string of numerals; no error produced.
Numeric value is read into integer variables.	Value is rounded before it is assigned to the variable.
Numeric value is read into a variable not large enough to handle the numeric variable.	Run-time error.
String value is read into fixed-engths string variables.	Truncated if string is too long; left-justified and padded with blanks if string is shorter than the variable.

Example

This example uses READ to read in values from a DATA statement. RESTORE allows the data to be reread.

```
CLS                    ' Clear the screen.
FOR i% = 1 TO 3
    READ a%, b$
    PRINT a%, b$
    RESTORE            ' Restore so data can be read again.
NEXT i%
PRINT
FOR i% = 1 TO 3
    READ a%, b$
    PRINT a%, b$
NEXT i%

DATA 1, "first", 2, "second", 3, "third"
```

DATE$ Function, DATE$ Statement

Action

Returns the computer's current system date (DATE$ function — no parameters), or sets the current system date (DATE$ statement).

Syntax

**DATE$**[= *stringexpression$*]

Part	Description
stringexpression$	Date expression in a valid format and range (leading zeros may be omitted):

Description
mm/dd/yy
mm/dd/yyyy
mm-dd-yy
mm-dd-yyyy (returned by the DATE$ function)

Inclusive Ranges	Valid for
01 - 12	mm (month)
01 - 31	dd (day)
1980 - 2099	yy or yyyy (year)

Remarks When you change the date in your computer's system with the DATE$ statement, the change remains in effect until you change the date again.

The valid range for setting the system date differs from the valid range used by date/time functions to convert serial numbers.

See Also FORMAT$ Function.

Example This example uses the DATE$ statement to set the system date on your computer to the new date you enter.

Note If you change the system date, any files you create or revise will be stamped with the new date. Make sure to reset the system date to the current date.

```
CLS                     ' Clear the screen.
PRINT "The current system date is set to: " + DATE$
PRINT
PRINT "Enter the date below (default year is 1992)."
INPUT "    Month:    ", Month$
INPUT "    Date:     ", Day$
INPUT "    Year:     ", Year$
IF Year$ = "" THEN Year$ = "92"
DATE$ = Month$ + CHR$(47) + Day$ + CHR$(47) + Year$
PRINT
PRINT "The system date has been set to: " + DATE$
```

DATESERIAL Function

Action Returns a serial number that represents the date of the arguments.

Syntax **DATESERIAL**(*year%*, *month%*, *day%*)

Part	Description
year%	Year between 1753 and 2078, inclusive.
month%	Month between 1 and 12, inclusive.
day%	Day between 1 and 31, inclusive.

Remarks For the argument *year%*, values between 0 and 178 can be used to represent the years 1900 - 2078. For all other years, use the complete 4-digit year (for example, 1800).

Arguments outside valid ranges are evaluated to calculate a valid date. For example:

```
TheDate = DATESERIAL(92, 13, 1)
```

where the argument *month%* (13) evaluates to the first month of the next year (January 1, 1993).

Specifying arguments that return a serial number outside the valid range (-53,688 to 65,380) produces the Visual Basic error message, "Illegal function call." Note: This translates to an actual date range of January 1, 1753 to December 31, 2078, inclusive.

You can use negative numbers as arguments. For example, to find the serial number for the date one week before 9/1/92, you can use:

```
DATESERIAL(92, 9, 1-7)
```

Example This example uses the DATESERIAL function to calculate the date value for January 1st in the year of your birth. You are then told what day of the week January 1st fell on in that year. Information is displayed using the YEAR and WEEKDAY functions.

```
OPTION BASE 1
DEFINT A-Z
DIM DayOfWeek(7) AS STRING

CLS                                         ' Clear the screen.
FOR I = 1 TO 7                              ' Initialize the array.
    READ DayOfWeek$(I)
NEXT I
INPUT "What are the last two digits of your birth year"; Birthyear
Jan1Date = DATESERIAL(Birthyear, 1, 1)  ' Calculate January 1st of birthyear.
```

```
PRINT "January 1,"; YEAR(Jan1Date); "fell on a ";  ' Display results.
PRINT DayOfWeek$(WEEKDAY(Jan1Date))

DATA "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday"
DATA "Friday", "Saturday", "Sunday"
```

DATEVALUE Function

Action Returns a serial number that represents the date of the argument.

Syntax **DATEVALUE**(*date$*)

Part	Description
date$	Any valid date string between January 1, 1753 and December 31, 2078, inclusive.

Remarks The argument *date$* can accept date formats such as "December 30, 1988," "12/30/88," and "28-Dec-1988."

If you embed time data within the datetext$ argument, it is ignored. For example:

```
DATEVALUE("12/30/88 2:24 AM")
```

If the year part of date is omitted, DATEVALUE uses the current year from your computer's system date.

Example This example uses the DATEVALUE function to convert a string to a date value. A calculation is then made that tells you the date of two weeks before and after your birth. Date information is displayed using the WEEKDAY, DAY, MONTH, and YEAR functions.

```
CLS                                     ' Clear the screen.
OPTION BASE 1
DIM DayOfWeek(7) AS STRING
DIM MonthOfYear(12) AS STRING
FOR I = 1 TO 7                          ' Initialize the arrays.
   READ DayOfWeek(I)
NEXT I
FOR I = 1 TO 12
   READ MonthOfYear(I)
NEXT I
INPUT "Enter your birthdate as MM/DD/YYYY "; Birthdate$
```

```
' Convert input date to numbers and create a complete date string.
BDateVal = DATEVALUE(Birthdate$)      ' Get the unique date value.
BDay$ = DayOfWeek$(WEEKDAY(BDateVal)) + " " + MonthOfYear$(MONTH(BDateVal))
BDay$ = BDay$ + STR$(DAY(BDateVal)) + "," + STR$(YEAR(BDateVal))
EDateVal = BDateVal - 14              ' Calculate a date 14 days earlier.

' Put the date string together.
EDay$ = DayOfWeek$(WEEKDAY(EDateVal)) + " " + MonthOfYear$(MONTH(EDateVal))
EDay$ = EDay$ + STR$(DAY(EDateVal)) + "," + STR$(YEAR(EDateVal))
LDateVal = BDateVal + 14              ' Calculate a date 14 days later.

' Put the date string together.
LDay$ = DayOfWeek$(WEEKDAY(LDateVal)) + " " + MonthOfYear$(MONTH(LDateVal))
LDay$ = LDay$ + STR$(DAY(LDateVal)) + "," + STR$(YEAR(LDateVal))
PRINT "You were born on "; BDay$; "." ' Display the results.
PRINT
PRINT "What you may not have been able to immediately calculate was:"
PRINT
PRINT "Two weeks before your birth was "; EDay$; "."
PRINT "Two weeks after your birth was "; LDay$; "."
END

DATA "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday"
DATA "Friday", "Saturday"
DATA "January", "February", "March", "April", "May", "June", "July"
DATA "August", "September", "October", "November", "December"
```

DAY Function

Action Evaluates a date/time serial number and returns the day of the month, a value in the range 1 to 31, inclusive.

Syntax **DAY**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See DATEVALUE Function example.

DblClick Event

Applies To Combo box, file list box, form, label, list box, option button, picture box.

Action Occurs when the user presses and releases the mouse button, then presses it again within the double-click time limit set with the user's mouse.

Syntax

SUB Form_DblClick ()
SUB *ctlname*_**DblClick ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks For a form, the DblClick event occurs when the user double-clicks a disabled control or a blank area of a form using the left mouse button.

For a control, the DblClick event occurs when the user:

- Double-clicks a control with the left mouse button.
- Double-clicks an item in a simple combo box (Style = 1), file list box, or list box.
- Changes the Path property for a directory list box in code.
- Sets a file list box's FileName property in code to a valid existing file name (that is, a file name with no wildcard characters).

To distinguish between left and right mouse buttons, use the MouseDown and MouseUp events.

You can use a DblClick event procedure to carry out multiple steps with a single action, such as double-clicking to select an item in a list box and at the same time to close the dialog box.

A DblClick event always recognizes this event ordering: MouseDown, MouseUp, Click, and DblClick.

If the DblClick event does not occur within the system's double-click time limit, the object recognizes another Click event. The double-click time limit can be adjusted with the user's mouse settings.

See Also Click Event.

Example In this example, a selected list item is displayed in a text box when either:

- A command button is clicked.
- A list box control item is double-clicked.

To try this example, create a form containing a list box, a command button, and a text box.

```
SUB Command1_Click ()
    Text1.Text = List1.Text  ' Display selection in text box.
END SUB

SUB Form_Load ()
    List1.ADDITEM "John"     ' Add list box entries.
    List1.ADDITEM "Paul"
    List1.ADDITEM "George"
    List1.ADDITEM "Ringo"
END SUB

SUB List1_DblClick ()
    Command1.Value = -1      ' Trigger Click event for Command1.
END SUB
```

DECLARE Statement (Basic Procedures)

Action Declares a FUNCTION or SUB procedure and invokes argument data type checking.

Syntax **DECLARE** {**FUNCTION** | **SUB**} *name* [([[**BYVAL**] *variable*[( )] [**AS** *type*] [, [**BYVAL**] *variable*[( )] [**AS** *type*]]...])]

Part	Description
name	Name of the procedure.
BYVAL	Indicates the parameter is passed by value rather than by reference; cannot be used with a variable of a user-defined type.
variable	Visual Basic variable name; for array variables, empty parentheses are required.
AS type	Data type of the variable: INTEGER, LONG, SINGLE, DOUBLE, STRING, user-defined type, or two additional types, FORM and CONTROL.

Remarks If the variable defined in the *parameterlist* is an array, it can be followed by the number of dimensions in parentheses. For example:

```
DECLARE SUB DisplayText (A(2) AS STRING)
DIM Text$(100,5)
.
.
.
CALL DisplayText(Text$( ))
```

The number of dimensions is optional; however, the empty parentheses should be included.

If a parameter is passed by reference, any change to the parameter value inside the procedure changes its value in the calling program.

If a parameter is passed by value, any changes to the parameter value inside the procedure is local to that procedure and does not affect its value in the calling program.

For calls within Visual Basic, the DECLARE statement is required only if you call SUB procedures without the CALL keyword, or if you invoke a FUNCTION procedure defined in another module.

A DECLARE statement also causes the compiler to check the number and type of arguments used to invoke the procedure. Visual Basic automatically generates DECLARE statements when you save your program in the programming environment (VBDOS).

The DECLARE statement can appear only in module-level code (not in a SUB or FUNCTION procedure) and affects the entire module.

The form of *parameterlist* determines whether or not argument checking is done. For example:

Example	Description
DECLARE SUB First	You can omit the parentheses only if the SUB or FUNCTION procedure is separately compiled. No argument checking is done.
DECLARE SUB First ()	First has no parameters. Arguments in a call to First generate an error. An empty argument list indicates that the SUB or FUNCTION procedure has no parameters and that argument checking should be done.
DECLARE SUB First (X AS LONG)	First has one long-integer parameter. The number and type of the arguments in each call or invocation are checked when the argument list appears in the DECLARE statement.

You cannot have fixed-length strings in DECLARE statements. Only variable-length strings can be passed to SUB and FUNCTION procedures. Fixed-length strings can appear in an argument list but are converted to variable-length strings before being passed.

You can have arrays containing fixed-length strings in DECLARE statements. The lengths of the array in the CALL statement and the declaration must match.

See Also CALL Statement (Basic Procedures), DECLARE Statement (Non-Basic Procedures), FUNCTION Statement, SUB Statement.

Example This example uses the DECLARE statement to allow a SUB procedure to be invoked using the CALL keyword and the BYVAL keyword to pass an argument by value rather than by reference.

```
DECLARE SUB Mult (BYVAL X!, Y!)
CLS                          ' Clear the screen.
a = 1
b = 1
PRINT "Before subprogram call, A = "; a; ", B ="; b
CALL Mult(a, b)              ' A is passed by value, and
                             ' B is passed by reference.
PRINT "After subprogram call, A ="; a; ", B ="; b

SUB Mult (BYVAL X, Y)
    X = 2 * X
    Y = 3 * Y
    PRINT "In subprogram, X ="; X; " , Y ="; Y
END SUB
```

DECLARE Statement (Non-Basic Procedures)

Action Declares calling sequences for external procedures written in other languages.

Syntax **DECLARE FUNCTION** *name* [**CDECL**] [**ALIAS** *"aliasname"*] [([[{**BYVAL**|**SEG**}] *variable* [**AS** *type*][,[{**BYVAL**|**SEG**}] *variable* [**AS** *type*]]...])]

DECLARE SUB *name* [**CDECL**] [**ALIAS** *"aliasname"*] [([[{**BYVAL**|**SEG**}] *variable* [**AS** *type*][,[{**BYVAL**|**SEG**}] *variable* [**AS** *type*]]...])]

Part	Description
FUNCTION	External procedure that returns a value; can be used in an expression.
SUB	External procedure invoked like a SUB.
name	Name used to invoke the procedure; can have up to 40 characters, including an explicit type character (%, &, !, #, @, or $).
CDECL	Indicates that the procedure uses C-language argument order, naming conventions, and calling conventions.
ALIAS	Indicates that the procedure being called has another name in the .OBJ or library file.
aliasname	Name the procedure has in the file or library.

Part	Description
BYVAL	(Default) Indicates the parameter is passed by value rather than by reference; cannot be used with a variable of a user-defined type.
SEG	Indicates the argument is passed as a segmented address (far address).
variable	Any valid Visual Basic variable name; only variable type is significant.
AS *type*	Data type of the variable: INTEGER, LONG, SINGLE, DOUBLE, STRING, CURRENCY, ANY, user-defined type, or two additional types, FORM and CONTROL.

Remarks

The CDECL passes the arguments from right to left, rather than using the Visual Basic convention of left to right.

CDECL also affects the name used in searches of object files and libraries:

- If there is no ALIAS clause in the DECLARE statement, the type-declaration character is removed from the name of the procedure, and an underscore is added to the beginning. This becomes the name used when searching libraries and external files.
- If CDECL is used with ALIAS, *aliasname* is used as is.

BYVAL can be used only with INTEGER, LONG, SINGLE, DOUBLE, and CURRENCY types. When BYVAL appears in front of a parameter, the actual argument is converted to the type indicated in the DECLARE statement before being passed. The additional data types, FORM and CONTROL, can only be used when a form or control is passed as an argument to a SUB or FUNCTION procedure.

If the variable is an array, it can be followed by the number of dimensions in parentheses to maintain compatibility with earlier versions of Microsoft Basic. For example:

```
DECLARE SUB EigenValue (A(2) AS DOUBLE)
```

You also can indicate the variable type for *parameterlist* by either:

- Including an explicit type-declaration character (%, &, !, @, #, or $) in the variable name.
- Relying on the default type.

When declaring external procedures written in other languages, you can use the ANY keyword in the AS clause. ANY overrides type checking for that argument. You cannot use ANY with arguments passed by value.

When neither BYVAL nor SEG is used, arguments are passed as near addresses (offsets).

This form of the DECLARE statement allows you to refer to procedures written in other languages.

The DECLARE statement causes the compiler to check the number and type of arguments used to invoke the procedure.

A DECLARE statement can appear only in module-level code and affects the entire source file.

The form of *parameterlist* determines whether or not argument type checking is done, as described in the following declarations:

Example	Description
DECLARE SUB First CDECL	No argument checking is done when there is no parameter list.
DECLARE SUB First CDECL ( )	First has no parameters. Arguments in a call to First generate an error; empty parentheses indicate that SUB or FUNCTION has no parameters and that argument checking should be done.
DECLARE SUB First CDECL (X AS LONG)	First takes one long integer argument. When an argument list appears, the number and type of the arguments are checked in each invocation.

A procedure that appears in a DECLARE statement can be invoked without the CALL keyword.

You cannot have fixed-length strings in DECLARE statements because only variable-length strings can be passed to SUB and FUNCTION procedures. Fixed-length strings can appear in an argument list but are converted to variable-length strings before being passed.

Be careful when using the SEG keyword, because Visual Basic may move variables in memory before the called routine begins execution. Anything in a CALL statement's argument list that causes memory movement may create problems. You can safely pass variables using SEG if the CALL statement's argument list contains only simple variables or arithmetic expressions.

You cannot pass arrays using the SEG keyword.

See Also CALL[S] Statement (Non-Basic Procedures), DECLARE Statement (Basic Procedures).

Example This example uses the DECLARE statement to use a C function in a Basic program. The Basic program calls the function addone, which uses C argument passing and takes a single integer argument passed by value.

Note: To run this program, you must compile the C function separately and place it in a Quick library or link it to the Visual Basic program.

```
DEFINT A-Z
DECLARE FUNCTION addone CDECL (BYVAL n AS INTEGER)
CLS                  ' Clear the screen.
INPUT x
y = addone(x)
PRINT "x and y are "; x; y
END

/* C function addone. Returns one more than the value of its argument. */
int far addone(int n)
{
   return(n+1);
}
```

DEF FN Statement

Action

Defines and names a function.

Syntax

Syntax 1

DEF FN*name*[([**BYVAL**] *variable*[( )] [**AS** *type*] [, [**BYVAL**] *variable*[( )] [**AS** *type*]]...)] = *expression*

Syntax 2

DEF FN*name*[([**BYVAL**] *variable*[( )] [**AS** *type*] [, [**BYVAL**] *variable*[( )] [**AS** *type*]]...)]
    [*statementblock*]
FN*name* = *expression*
    [*statementblock*]
    [**EXIT DEF**]
    [*statementblock*]
END DEF

Part	Description
BYVAL	Indicates the parameter is passed by value rather than by reference; cannot be used with a variable of a user-defined type.
variable	Visual Basic variable name; for array variables, empty parentheses are required.
type	Data type of the variable: INTEGER, LONG, SINGLE, DOUBLE, STRING, or a user-defined data type.
expression	Return value of the function.
EXIT DEF	Immediately exits an executing DEF FN function.

Remarks

In both versions of syntax, the argument *expression* is evaluated and the result is the function's value. In Syntax 1 (single-line syntax), *expression* is the entire body of the function and is limited to one logical line.

When the function is called, Visual Basic assigns the value of each argument to its corresponding parameter. Function arguments are passed by value. Functions defined by DEF FN do not accept arrays, records, or fixed-length strings as arguments.

When no *expression* is assigned to the name, the default return values are 0 for a numeric DEF FN function, and the null string ("") for a string DEF FN function.

DEF FN must define a function before the function is used. If you call the function before it is defined by DEF FN, Visual Basic generates the error message, "Function not defined." DEF FN function definitions cannot appear inside other DEF FN definitions. In addition, functions defined by DEF FN cannot be recursive.

Functions defined by DEF FN can be used only in the module in which they are defined. The FUNCTION statement provides a better way of defining functions.

A DEF FN-defined function can share variables with the module-level code. Variables not in *parameterlist* are global — that is, their values are shared with the module-level code. To keep a variable value local to a function definition, declare it in a STATIC statement.

DEF FN can return either numeric or string values. DEF FN returns a string value if name is a string-variable name, and a numeric value if name is a numeric-variable name. If you assign a numeric value to a string function name or assign a string value to a numeric function name, Visual Basic generates the error message, "Type mismatch."

If the function is numeric, DEF FNname returns a value with the precision specified by name. For example, if name specifies a double-precision variable, then the value returned by DEF FNname is double precision, regardless of the precision of *expression*.

Because Visual Basic may rearrange arithmetic expressions for greater efficiency, avoid using DEF FN functions that change program variables in expressions that may be reordered. The following example may give unpredictable results:

```
DEF FNShort
    I=10
    FNShort=1
END DEF
I=1 : PRINT FNShort + I + I
```

If Visual Basic reorders the *expression* so FNShort is called after calculating (I+I), the result is 3 rather than 21. You usually can avoid this problem by isolating the DEF FN function call:

```
I = 1 : X = FNShort : PRINT X + I + I
```

Note: Embedding I/O operations in DEF FN-defined functions used in I/O statements, or embedding graphics operations in DEF FN-defined functions in graphics statements, may cause similar problems.

See Also DEF SEG Statement, FUNCTION Statement, SHARED Statement, STATIC Statement.

Example This example uses a DEF FN function to calculate the factorial of an integer (for example, the factorial of 3 is 3*2*1).

```
CLS                          ' Clear the screen.
DEF FNFactorial# (X%)
    STATIC Tmp#, I%
    Tmp# = 1
    FOR I% = 2 TO X%
         Tmp# = Tmp# * I%
    NEXT I%
    FNFactorial# = Tmp#
END DEF

INPUT "Enter an integer: ", Num%
PRINT : PRINT Num%; "factorial is"; FNFactorial#(Num%)
```

DEF SEG Statement

Action Sets the current segment address for a subsequent execution of either:

- Absolute routine.
- BLOAD or BSAVE statement.
- PEEK function.
- POKE statement.
- The segment address setting remains in effect until changed.

Syntax **DEF SEG** [= *address*]

Part	Description
address	Segment address value in the range 0 to 65,535, inclusive; if omitted, the current segment address is reset to the default data segment.

Remarks The *address* is a numeric expression with an unsigned integer value between 0 and 65,535, inclusive. DEF SEG sets the current segment to this value. If you use a value outside this range, Visual Basic generates the error message, "Illegal function call." The previous segment is retained if an error occurs.

To reset the current segment to the default data segment (DGROUP), use DEF SEG without any argument.

Be sure to separate DEF and SEG with a space. Otherwise, Visual Basic interprets the statement to mean, "assign a value to the variable DEFSEG."

To set the current segment address to the address of data stored in far memory, you can use DEF SEG with the SSEG or VARSEG functions. SSEG returns the current segment address of a string; VARSEG returns the current segment address of numeric data. For example, this statement sets the current address for a far string named a$:

```
DEF SEG = SSEG(a$)
```

Do not use DEF SEG to set the segment of an expanded-memory array. If you start Visual Basic (VBDOS.EXE) with the /Ea switch, any of the following arrays may be stored in expanded memory:

- Numeric arrays < 16K in size.
- Fixed-length string arrays < 16K in size.
- User-defined type arrays < 16K in size.

If you want to use DEF SEG to set the segment of an array, first start Visual Basic (VBDOS.EXE) without the /Ea switch. Note: Without the /Ea switch, no arrays are stored in expanded memory.

See Also Absolute Routine, BLOAD Statement, BSAVE Statement, DEF FN Statement, PEEK Function, POKE Statement, SSEG Function, SSEGADD Function.

Example This example uses the DEF SEG statement, the PEEK function, and POKE statement to turn the Caps Lock key on and off. The program uses the LINE INPUT statement to prompt the user to enter a string, demonstrating that the Caps Lock key is on.

Note: This program contains hardware-specific instructions. It works correctly on IBM PC, XT, and AT computers.

```
DECLARE SUB CapsOn ()
DECLARE SUB CapsOff ()
DECLARE SUB PrntMsg (R%, C%, M$)

CLS                                 ' Clear the screen.
CapsOn
PrntMsg 24, 1, "<Caps Lock On>"
LOCATE 12, 10
LINE INPUT "Enter a string (all characters are caps): ", S$
CapsOff
PrntMsg 24, 1, "                    "
PrntMsg 25, 1, "Press any key to continue..."
DO WHILE INKEY$ = "": LOOP
CLS                                 ' Clear the screen.
END
```

```
STATIC SUB CapsOff ()              ' Turn Caps Lock off.
  DEF SEG = 0
  ' Set Caps Lock off (turn off bit 6 of &H0417).
  POKE &H417, PEEK(&H417) AND &HBF
  DEF SEG
END SUB

STATIC SUB CapsOn ()               ' Turn Caps Lock on.
  ' Set segment to low memory.
  DEF SEG = 0
  ' Set Caps Lock on (turn on bit 6 of &H0417).
  POKE &H417, PEEK(&H417) OR &H40
  ' Restore segment.
  DEF SEG
END SUB

STATIC SUB PrntMsg (Row%, Col%, Message$)
   ' Print message at Row%, Col% without changing cursor.
   ' Save cursor position.
   CurRow% = CSRLIN: CurCol% = POS(0)
   LOCATE Row%, Col%: PRINT Message$;
   ' Restore cursor.
   LOCATE CurRow%, CurCol%
END SUB
```

Default Property

Applies To Command button.

Action Determines which command button is the default button on a form.

Syntax [*form.*]*commandbutton*.**Default**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	Specifies the control as the default command button; only one button on a form can be the default button.
False (0)	(Default) Disables this functionality.

Remarks

A command button with its Default property set to True (-1) is referred to as the form's default button.

Only one button on a form can be the default button.

When Default is set to True (-1) for one button, it is automatically set to False (0) for all other buttons.

When a command button's Default property is set to True (-1), pressing the Enter key simulates selection of the default button and invokes the Click event if:

- The button's container form is active.
- Another command button does not have the focus.

If another command button on the same form has the focus, pressing Enter invokes the Click event for that button rather than the Default button.

When a control other than a command button has the focus, pressing Enter causes a Click event for the default button.

For a form that supports an irreversible action such as a delete operation, make the Cancel button the default button.

See Also

Cancel Property, Click Event.

Example

This example shows how to use the Default property. When the Enter key is pressed, the Click event for the default command button is executed. To try this example, create a form containing a picture box and two command buttons.

```
SUB Command2_Click ()
    Msg$ = "The Default property is True for command button 2"
    Picture1.PRINT Msg$
END SUB
```

DEFtype Statements

Action

Sets the default data type for a range of variable names, DEF FN statements, and FUNCTION procedures.

Syntax

DEF*type letterrange* [**,** *letterrange*]**...**

Part	Description
type	Valid type values are:

Statement	Description
DEFINT	Integer (2-byte).
DEFLNG	Long (4-byte) integer.
DEFSGN	Single-precision floating-point (4-byte).
DEFDBL	Double-precision floating-point (8-byte).
DEFCUR	Currency (8-byte signed integer); up to 4 digits to the right of the decimal point and 15 digits to the left.
DEFSTR	String.

letterrange Letter or range of letters (for example, A-M). Visual Basic sets the default data type for variables, DEF FN statements, and FUNCTION procedures whose names begin with the specified letter(s).

Remarks By using one of the DEFtype statements (DEFINT, DEFLNG, DEFCUR, DEFSNG, DEFDBL, or DEFSTR), you can specify that all variables starting with a given letter or range of letters are one of the elementary variable types without using a type-declaration suffix.

DEFtype sets the default data type. For example, in the following code fragment, Message is a string variable:

```
DEFSTR A-Q
.
.
.
Message = "Out of stack space."
```

The case of the letters in *letterrange* is not significant. These three statements are equivalent:

```
DEFINT I-N
DEFINT i-n
DEFINT i-N
```

A type-declaration character (%, &, !, #, @, $) always takes precedence over a DEFtype statement. DEFtype statements do not affect record elements.

If you do not specify a data type in a DEFtype statement, single-precision is the default.

After you specify a DEFtype statement in your program, Visual Basic automatically inserts a corresponding DEFtype statement in each procedure you create.

DEFtype statements only affect variable names in the module in which they appear. A DEFtype statement affects all variables starting with the given letter or range of letters. Note: I%, I&, I!, I#, I@, and I$ are all distinct variables, and each can hold a different value.

Example See ABS Function example.

DESKTOP_BACKCOLOR, DESKTOP_FORECOLOR Constants

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the desktop background or foreground color.

Syntax **SCREEN.ControlPanel**({**DESKTOP_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({**DESKTOP_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	DESKTOP_BACKCOLOR has a value of 5; DESKTOP_FORECOLOR has a value of 6.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
0	(Default - DESKTOP_BACKCOLOR) Black.
7	(Default - DESKTOP_FORECOLOR) White.

See Also DESKTOP_PATTERN Constant.

Example This example illustrates the effect of changing the value of the DESKTOP_BACKCOLOR, DESKTOP_FORECOLOR, and DESKTOP_PATTERN constants. To try this example, create a form containing a command button.

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(DESKTOP_BACKCOLOR) = 4
    SCREEN.ControlPanel(DESKTOP_FORECOLOR) = 11
    SCREEN.ControlPanel(DESKTOP_PATTERN) = 192
END SUB
```

DESKTOP_PATTERN Constant

Applies To ControlPanel property, SCREEN object.

Action Determines the desktop fill pattern. The fill pattern consists of a repeating character that is displayed as an application's background.

Syntax **SCREEN.ControlPanel({DESKTOP_PATTERN** | *subscript%*})[= *pattern%*]

Part	Description
subscript%	DESKTOP_PATTERN has a value of 7.
pattern%	Valid settings are:

Value	Description
0 - 255	Valid ASCII character.
176	(Default) ▒.

Remarks If *pattern%* is set to 0 (the null character), the current screen image will be saved and used as the background image when forms are showing. For example, when forms are visible setting DESKTOP_PATTERN to 0 will save the current state (including forms) as the background image. Once this image has been saved, the screen can be cleared or changed and the saved background remains persistent.

When *pattern%* is changed from 0 to any other value, the saved image is cleared from memory.

When all forms are hidden or unloaded, the persistent background image (if any) is not automatically cleared. However, subsequent graphic or other output statements will overwrite it.

To change the screen image once stored in memory:

- Clear the screen (using CLS).
- Redraw the image.
- Set DESKTOP_PATTERN = 0.

Switching line modes (25 to 43, or vice versa) after saving an image will accordingly either pad or truncate the image buffer with null characters.

See Also DESKTOP_BACKCOLOR Constant, DESKTOP_FORECOLOR Constant.

Example See DESKTOP_BACKCOLOR, DESKTOP_FORECOLOR Constants example.

DIM Statement

Action Declares an array or specifies a data type for a nonarray variable.

Syntax **DIM [SHARED]** *variable*[([*lower* **TO**] *upper* [,[*lower* **TO**] *upper*]...)] [**AS** *type*][,*variable*[([*lower* **TO**] *upper* [,[*lower* **TO**] *upper*]...)] [**AS** *type*]]...

Part	Description
SHARED	Causes specified variable or array to be shared with all SUB or FUNCTION procedures in the module.
variable	Name of an array or nonarray variable.
lower	(Default - 0) Lower boundary of the array's subscripts.
upper	Upper boundary of the array's subscripts.
AS *type*	(Default - SINGLE) Declares data type of the array or variable; use a separate AS type clause for each variable defined.

Remarks The TO keyword provides a way to indicate both the lower and upper bounds of an array variable's subscripts. For example, the following statements are equivalent if there is no OPTION BASE statement:

```
DIM A(8, 3)
DIM A(0 TO 8, 0 TO 3)
DIM A(8, 0 TO 3)
```

Array subscripts can be negative. The TO keyword can be used to specify any range of subscripts between -32,767 and 32,767, inclusive. For example:

```
DIM A(-4, TO 10)
DIM B(-99 TO -5, -3 TO 0)
```

The amount of memory used by a numeric array is calculated by multiplying the number of elements in the array times the number of bytes required by the data type of the array. For example, an integer array requiring 2 bytes per element can contain twice as many elements as a long integer array requiring 4 bytes per element. String arrays are limited to almost 64K bytes.

If you use a subscript greater than the specified maximum or smaller than the specified minimum, an error message is generated. An error message is also generated if the size of the array (bytes of memory used) exceeds the limits described above.

The DIM statement initializes all:

- Elements of numeric arrays to zero.
- Elements of string arrays to null strings.
- Fields of user-defined type variables to zero, including fixed-length string elements.
- Fixed-length strings to null strings.

Use DIM to set the following properties of an array:

- Number of dimensions (maximum - 60).
- Upper subscript value (default - 10 for any array dimension).

Use DIM at the module or procedure level to make the following variable declarations:

- At the module level, declare variables that are available to all procedures throughout that module.
- At the procedure level, within a SUB or FUNCTION procedure, declare variables that are local to that procedure; it is acceptable to place DIM statements at the beginning of the procedure.

Use DIM with empty parentheses at either the module or procedure level to declare dynamic arrays. Use REDIM at the procedure level to actually define the number of elements and dimensions in the array.

You cannot use DIM to redeclare a dimension for an array variable whose size has already been declared. If you attempt to do so, an error message is generated.

DIM provides a way to declare variables of user-defined types. For example, the following user-defined type is defined at the module level. In a module or procedure, the variable TopCard is declared using the DIM statement:

```
TYPE Card                          ' At module level.
     Suit AS STRING * 9
     Value AS INTEGER
END TYPE
DIM TopCard As Card                ' At module or procedure level
```

You can also declare arrays of user-defined types. For example:

```
DIM Deck(1 TO 52) AS Card
```

The SHARED keyword used with DIM shares variables among all procedures in a module. In contrast, the SHARED statement shares variables between a single procedure and module-level code.

See Also OPTION EXPLICIT Statement, REDIM Statement.

DIR$ Function

Action Returns a file name that matches the specified pattern.

Syntax **DIR$**[(*filespec$*)]

Part	Description
filespec$	String expression that specifies a file name or path; may include a drive and MS-DOS wildcard characters.

Remarks

If you omit *filespec$* when you first call DIR$, Visual Basic generates the error message, "Illegal Function Call."

DIR$ returns the first file name that matches the *filespec$* you specify. To retrieve additional file names that match the *filespec$* pattern, call DIR$ again with no argument. When no file names match, DIR$ returns a null string.

You do not have to retrieve all the file names that match a given *filespec$* before calling DIR$ again with a new *filespec$*.

DIR$ is not case sensitive ("C" is the same as "c").

Because file names are retrieved in no particular order, you may want to store file names in a dynamic array and sort the array.

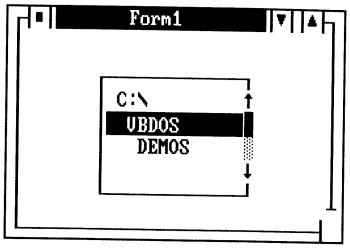
Example

This example uses the DIR$ function within the GetFileCount FUNCTION procedure to return the number of files that match the given file specification.

```
DECLARE FUNCTION GetFileCount& (filespec$)
filespec$ = "*.*"
count = GetFileCount(filespec$)
PRINT count; "files in the current directory match the file specification."

' This function evaluates a file specification and returns the
' number of files that match the specification. Wildcards ("*" and "?")
' are permitted. Drive and directory path specifications may also be included
' in filespec$.

CLS                                          ' Clear the screen.
FUNCTION GetFileCount& (filespec$)
    DIM FileCount AS LONG
    IF LEN(DIR$(filespec$)) = 0 THEN         ' Ensure filespec is valid.
         FileCount& = 0                      ' It's not.
    ELSE
         FileCount = 1                       ' It is, so count files.
         DO WHILE LEN(DIR$) > 0
              FileCount& = FileCount& + 1
         LOOP
     END IF
     GetFileCount = FileCount&
END FUNCTION
```

Directory List Box Control

Action Displays a hierarchical list of directories; allows the user to select directories and paths at run time.



Events

Change	Click	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
LostFocus	MouseDown	MouseMove	MouseUp

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	CtlName	DragMode	Enabled
ForeColor	Height	Index	Left
List	ListCount	ListIndex	MousePointer
Parent	Path	TabIndex	TabStop
Tag	Top	Visible	Width

Remarks Use the directory list box control to create a dialog box in your application that, for example, allows the user to select a subdirectory and open a file from that subdirectory.

Directories displayed within a directory list box control are not necessarily listed in alphabetical order. The order of the directories will be the same order displayed using the MS-DOS command DIR /AD.

Use the List, ListCount, and ListIndex properties to access items displayed in the list box at run time.

Use the Path property to pass the selected directory to a file list box. For example:

```
SUB Dir1_Change ()
    File1.Path = Dir1.Path
END SUB
```

Use the Drive property from the drive list box to update the directory list box when a new drive is selected. For example:

```
SUB Drive1_Change()
    Dir1.Path = Drive1.Drive
END SUB
```

The minimum size of a directory list box is 4 x 5; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value.

See Also Drive List Box Control, File List Box Control.

Example This example uses the DRAG method to drag a file name to a picture box where the path and the the name of the file are displayed. When the program begins you will be able to browse your file system and find the CONSTANT.BI file in the C:\VBDOS subdirectory using the drive, directory, and file list box controls. Once you've located the file, click the file name and drag it to the picture control.

```
SUB Dir1_Change ()                    ' Any change in Dir1
    File1.Path = Dir1.Path            ' is reflected in File1
END SUB

SUB Dir1_Dragover (Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State
➥AS INTEGER)
    ' Change pointer to no drop.
    IF State = 0 THEN Source.MousePointer = 12
    ' Use default mouse pointer.
    IF State = 1 THEN Source.MousePointer = 0
END SUB

SUB Drive1_Change ()              ' Any change in Drive1.
    Dir1.Path = Drive1.Drive      ' is reflected in Dir1.
END SUB

SUB Drive1_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE,
➥State AS INTEGER)
    ' Change pointer to no drop.
    IF State = 0 THEN Source.MousePointer = 12
    ' Use default mouse pointer.
IF State = 1 THEN Source.MousePointer = 0
END SUB
```

```
SUB File1_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State
➥AS INTEGER)
    ON ERROR RESUME NEXT
    IF State = 0 AND RIGHT$(File1.Filename, 3) = ".bi" THEN
       Label1.Text = File1.Path + "\" + File1.Filename
       IF ERR THEN MSGBOX "The file file could not be loaded."
   ELSEIF State = 1 THEN
   END IF
END SUB

SUB File1_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE,
➥Y AS SINGLE)
    DY = TEXTHEIGHT("A")                        ' Get height of one line.
    Label1.MOVE File1.Left, File1.Top + Y - DY / 2, File1.Width, DY
    Label1.DRAG                                 ' Drag label outline.
END SUB

SUB Form_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State
➥AS INTEGER)
    ' Change pointer to no drop.
    IF State = 0 THEN Source.MousePointer = 12
    ' Use default mouse pointer.
    IF State = 1 THEN Source.MousePointer = 0
END SUB

SUB Form_Load ()
    Label1.Visible = 0      ' Make the label invisible.
    File1.Pattern = "*.BI"  ' Set file patterns.
END SUB

SUB Text1_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
    ON ERROR RESUME NEXT
    Text1.Text = File1.Path + "\" + File1.Filename
    IF ERR THEN MSGBOX "The file could not be loaded."
END SUB
```

DISABLED_ITEM_FORECOLOR Constant

Applies To ControlPanel property, SCREEN object.

Action Determines the foreground color of disabled items.

Syntax **SCREEN.ControlPanel({DISABLED_ITEM_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	DISABLED_ITEM_FORECOLOR has a value of 8.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
8	(Default) Gray.

Example

This example illustrates the effect of changing the value of the DISABLED_ITEM_FORECOLOR constant. To try this example, create a form containing a command button.

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(DISABLED_ITEM_FORECOLOR) = 11
END SUB
```

DO...LOOP Statement

Action

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

Syntax

Syntax 1:

Condition evaluated; if True, *statementblock* executed, and so on (WHILE); if False, *statementblock* executed until True (UNTIL)

DO [{**WHILE** | **UNTIL**} *condition*]
    [*statementblock*]
    [**EXIT DO**]
    [*statementblock*]
LOOP

Syntax 2:

Statementblock executed, then condition evaluated; if True, *statementblock* re-executed, and so on (guaranteed one-time execution of statements).

DO
    [*statementblock*]
    [**EXIT DO**]
    [*statementblock*]
LOOP[{**WHILE** | **UNTIL**} *condition*]

Part	Description
condition	Numeric expression that evaluates as True (nonzero) or False (zero).
statementblock	One or more valid statements on one or more lines.
EXIT DO	Alternative exit from DO...LOOP; transfers control to the statement following the LOOP keyword.

Remarks

EXIT DO can be used only in a DO...LOOP statement.

When EXIT DO is used within nested DO...LOOP statements, control is transferred out of the immediately enclosing loop.

You can use a DO...LOOP statement instead of a WHILE...WEND statement. DO...LOOP is more versatile than WHILE...WEND because it can test for condition at the beginning or at the end of a loop.

See Also

FOR...NEXT Statement, WHILE...WEND Statement.

Example

This example uses a DO...LOOP statement to perform tests at the end of the loop because the entire array must be examined at least once to see if it is in order. In general, test at the end of a loop only if you know that you always want the statement block executed at least once.

```
CONST NOEXCH = -1       ' Set up a special value to indicate no exchanges.
DIM Exes(12)
CLS                     ' Clear the screen.
FOR i = 1 TO 12 STEP 2  ' Load the array and mix it up.
   Exes(i) = 13 - i
   Exes(i + 1) = 0 + i
NEXT i
Limit = 12
PRINT "This is the list of numbers to sort in ascending order:"
PRINT
FOR i = 1 TO 12
   PRINT USING " ### "; Exes(i);
NEXT i
PRINT

PRINT : PRINT "Press any key to continue..."
DO                       ' DO...LOOP with test at the bottom of the loop.
LOOP WHILE INKEY$ = ""
```

```
DO
    Exchange = NOEXCH
    FOR i = 1 TO Limit - 1              ' Make one pass over the array.
        IF Exes(i) > Exes(i + 1) THEN
             SWAP Exes(i), Exes(i + 1) ' Exchange array elements.
             Exchange = i              ' Record location of most
        END IF                         ' recent exchange.
    NEXT i                          ' Sort on next pass only to where
    Limit = Exchange                ' last exchange was done.
LOOP UNTIL Exchange = NOEXCH        ' Sort until no elements are exchanged.
PRINT : PRINT "Sorting is completed. This is the sorted list:": PRINT
FOR i = 1 TO 12
   PRINT USING " ### "; Exes(i);
NEXT i
END
```

DOEVENTS Function

Action Causes Visual Basic to process form events; returns the number of forms loaded.

Syntax **DOEVENTS()**

Remarks Form events are not processed within standard event-handling procedures. Some procedures may contain code which takes several seconds to run. Therefore, if you want to respond to form events while these procedures are running, you must intersperse calls to DOEVENTS in the procedure. For example:

- In response to a Click event, you may want to sort a list containing thousands of items.
- While the list is being sorted, you display a message box with a Cancel button that asks the user to wait until the sort has been completed.
- For the Cancel button to work, you must place a call to DOEVENTS in the sorting routine. This allows the Click event for the Cancel button to be processed during the sort. Without DOEVENTS, the Click event for the Cancel button cannot be processed until the sort is completed.

Be careful using DOEVENTS. If you use a DOEVENTS function in a Click event procedure, you can receive a second click (during DOEVENTS processing) before control returns from processing the first Click event. This type of re-entrancy error could occur in the sort example above if:

- The user is allowed to click the button that starts the sort routine while the sort is being performed.
- The second Click procedure attempts to open the list file, which has already been opened by the first Click procedure.

If the startup module in your program is a form module, form events processing begins automatically. If the start-up module in your program is not a form module, you must call DOEVENTS to begin form event handling. When the first form in the program is made visible, use DOEVENTS in a continual loop to begin processing form events. For example:

```
Form1.SHOW
WHILE DOEVENTS(): WEND
```

DOEVENTS causes form events to be processed until there are no forms loaded.

Be aware of global variables when using DOEVENTS. It is possible for an event procedure that is called during DOEVENTS processing to reset the value of a global variable upon which the original procedure depends.

Example

This example uses DOEVENTS to yield to other operating environment events that may be occurring at the same time as a CPU-intensive FOR...NEXT loop in Visual Basic processing. Click the form to start the DOEVENTS loop.

```
DECLARE SUB DoEventsDemo ()

SUB DoEventsDemo ()
    Msg$ = "A FOR...NEXT loop simulates a CPU intensive operation. "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"DOEVENTS is not used the first time the loop "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"executes. As a result, you are unable to do "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"anything else until the loop is finished. Choose "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"OK and try to select another window before the beep."
    MSGBOX Msg$
    FOR I% = 1 TO 5000                          ' Start 1st loop.
        N% = INT(5000 * RND + 1)                ' Generate random number.
    NEXT I%                                     ' Increment loop counter.
    BEEP                                        ' Notify user 1st loop is done.
    Msg$ = "DOEVENTS is used the second time the loop executes causing "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"it to yield control to other events that may occur "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"while the loop is executing. The processing appears "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) +
    ➥"to be running in the background. Now choose OK and "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) + "try to select another window."
    MSGBOX Msg$                                 ' Display explanation
    FOR I% = 1 TO 1000                          ' Start 2nd loop.
        N% = INT(5000 * RND + 1)                ' Generate random number.
        State% = DOEVENTS()                     ' Yield to other events.
    NEXT I%                                     ' Increment loop counter.
```

```
    BEEP: BEEP                          ' Notify user 2nd loop is done.
    MSGBOX "All Done!"                  ' Display final message.
END SUB

SUB Form_Click ()
    CALL DoEventsDemo
END SUB
```

DRAG Method

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Begins, ends, or cancels the dragging of a control.

Syntax [*form.*][*ctlname.*]**DRAG** [*action%*]

Part	Description
ctlname	Name of control to drag.
action%	Integer value that indicates the action to perform:

Value	Description
0	Cancel drag operation.
1	(Default) Begin dragging control.
2	End dragging (drop control).

Remarks Use the DRAG method to control dragging and dropping only when the DragMode property of the control to drag is set to 0 (Manual). However, you can still use DRAG on a control whose DragMode property is set to 1 (Automatic).

If you want the mouse pointer to change while the control is being dragged, use the MousePointer property to define your pointer.

See Also DragDrop Event, DragMode Property, DragOver Event, MousePointer Property.

Example See Directory List Box Control example.

DragDrop Event

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Occurs when a drag-and-drop operation is completed as a result of dragging a control over a form or control and releasing the mouse button or by using the DRAG method with action% = 2.

Syntax **SUB Form_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)**
SUB *ctlname*_**DragDrop ([Index AS INTEGER,] Source AS CONTROL, X AS SINGLE, Y AS SINGLE)**

Part	Description
Source	Source control being dragged. You can refer to properties and methods with the Source argument. For example: `Source.Visible = 0`
X, Y	Current horizontal (X) and vertical (Y) positions of the mouse pointer within the target form or control.
Index	Uniquely identifies a control in a control array.

Remarks Use a DragDrop event procedure to prescribe what happens after a drag operation is complete. For example, you can move the Source control to a new location or copy a file from one location to another.

When multiple controls can potentially be used as the Source argument, use the TYPEOF keyword with the IF statement to determine the type of the source control. Set and use the control's Tag property to uniquely identify a control, then write a DragDrop event procedure accordingly.

Use the DragMode property and the DRAG method to specify the way dragging is initiated. Once dragging has been initiated, you can handle events that precede a DragDrop event with a DragOver event procedure.

See Also DRAG Method, DragMode Property, DragOver Event.

Example This example demonstrates a visual effect of dropping a picture box into another picture box using the DragDrop event and the IF...TYPEOF statement. To try this example, create a form containing two picture box controls. Set the properties of Picture1 as follows:

- DragMode = 1-Automatic
- BackColor = 2-Green

```
' Set Picture2 backcolor to that of Source control (Picture1).
SUB Picture2_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
    IF TYPEOF Source IS PictureBox THEN
        Picture2.BackColor = Source.BackColor
    END IF
END SUB
```

DragMode Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Sets or returns manual or automatic dragging mode for drag-and-drop operations.

Syntax [*form.*]*ctlname*.**DragMode**[= *mode%*]

Part	Description
mode%	Valid settings are:

Value	Description
0	(Default) Manual; requires the DRAG method to initiate dragging on the source control.
1	Automatic; clicking the source control automatically initiates dragging.

Remarks When DragMode is set to Automatic (1), the control does not respond as usual to mouse input messages. Use the Manual (0) setting to determine when dragging begins or ends. This is useful for either:

- Initiating dragging in response to a keyboard or menu command.
- Allowing a source control to recognize a MouseDown event prior to dragging.

Clicking while the mouse pointer is over a target control or form during a drag operation generates a DragDrop event for the target object. This ends the drag operation. Dragging may also generate a DragOver event.

While a control is being dragged, it cannot recognize other user-initiated mouse or keyboard events (KeyDown, KeyPress, or KeyUp; MouseDown, MouseMove, or MouseUp). However, the control can receive events initiated by code.

See Also DRAG Method, DragDrop Event, DragOver Event.

Example This example enables and disables the ability to drag a command button each time a form is clicked. To try this example, create a form containing a command button.

```
SUB Form_Click ()
    IF Command1.DragMode = 0 THEN          ' Check DragMode.
        Command1.DragMode = 1              ' Turn it on.
   ELSE
      Command1.DragMode = 0                ' Or turn if off.
   END IF
END SUB
```

DragOver Event

Applies To

Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action

Occurs when a drag-and-drop operation is in progress. You can use the DragOver event to monitor when the mouse pointer enters, leaves, or is directly over a valid target. The position of the mouse pointer determines which target object receives this event.

Syntax

SUB Form_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State AS INTEGER)
SUB *ctlname*_**DragOver ([Index AS INTEGER,] Source AS CONTROL, X AS SINGLE, Y AS SINGLE, State AS INTEGER)**

Part	Description
Source	Control being dragged. You can refer to properties and methods with the Source argument. For example: Source.Visible = 0
X, Y	Current horizontal (X) and vertical (Y) positions of the mouse pointer within the target form or control.
State	Transition state of the control being dragged in relation to a target form or control:

Value	Description
0	Entering; source control is being dragged within the range of a target.
1	Leaving; source control is being dragged out of the range of a target.
2	Over; source has moved from one position over the target to another.

Index	Uniquely identifies a control in a control array.

Remarks Use a DragOver event procedure to determine what happens after dragging is initiated and before a control drops onto a target. For example, you might verify a valid target range by either:

- Highlighting the target (set the BackColor or ForeColor property in code).
- Displaying a special drag pointer (set the MousePointer property in code).

Use the State argument to determine actions at key transition points. For example, you might highlight a possible target on State 0 (Enter) and restore the object's previous appearance on State 1 (Leave).

When an object receives a DragOver event with State = 0 (Enter):

- If the Source control is dropped on a valid target, a DragDrop event is generated.
- If the Source control is not dropped on a valid target, another DragOver event is generated with State = 1 (Leave).

Use the DragMode property and the DRAG Method, or the DragDrop event using the Source argument, to specify the way dragging is initiated.

See Also DRAG Method, DragDrop Event, DragMode Property.

Example This example demonstrates one way to indicate a valid drop target. The mouse pointer changes from the default to a happy face icon when the text box is dragged over a picture box. The mouse pointer returns to the default when the source is dragged elsewhere. To try this example, create a form containing a small text box and a large picture box. Set the Text1 DragMode property to 1-Automatic.

```
SUB Picture1_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE,
➥State AS INTEGER)
    CONST INSIDE = 2
    SELECT CASE INSIDE
         CASE INSIDE        ' Load icon
         Picture1.MousePointer = 4
         CASE ELSE          ' Unload icon
         Picture1.MousePointer = 0
    END SELECT
END SUB
```

DRAW Statement

Action Draws an object; not available when forms are showing.

Syntax **DRAW** *stringexpression$*

Part	Description
stringexpression$	String expression containing one or more drawing commands (LINE, COLOR, PALETTE, SCREEN, VARPTR$).

Remarks

You must be in a graphics screen mode (for example, screen mode 1) to use the DRAW statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

The current position defaults to the center of the screen when a program begins execution.

The argument *stringexpression$* contains one or more drawing commands which, when combined with other graphics statements (such as LINE and COLOR), make up a graphics macro language. There are three types of drawing commands:

- Line-drawing and cursor-movement.
- Rotation, color, and scale-factor.
- The "X" substring.

Cursor-movement commands initiate movement from the current graphics position, which usually is the coordinates of the last point plotted with a DRAW macro-language command or other graphics command.

The following prefix commands can precede any other movement commands in *stringexpression$*:

Prefix	Action
B	Move, but do not plot any points.
N	Plot, but return to original position when done.

The following is a list of available line-drawing and cursor-movement commands:

Command	Action
D[*n*%]	Moves cursor down *n*% units.
E[*n*%]	Moves cursor up and right *n*% units.
F[*n*%]	Moves cursor down and right *n*% units.
G[*n*%]	Moves cursor down and left *n*% units.
H[*n*%]	Moves cursor up and left *n*% units.
L[*n*%]	Moves cursor left *n*% units.
M[{+|-}]*x*%,*y*%	Moves cursor to point *x*%,*y*%; if *x*% is preceded by + or -, moves relative to the current point.
R[*n*%]	Moves cursor right *n*% units.
U[*n*%]	Moves cursor up *n*% units.
[**B**]	(Optional) moves cursor without drawing.
[**N**]	(Optional) draws and returns cursor to its original position.

If n% is omitted from line-drawing and cursor-movement commands, the cursor moves 1 unit.

The following rotation, color, and scale-factor commands allow you to change the appearance of a drawing by rotating it, changing colors, or scaling it:

Command	Description	Range of Values	Comments
A *n*[1]	Set angle rotation.	n = 0 to 3, where: 0 is 0° 1 is 90° 2 is 180° 3 is 270°	Figures rotated 90° or 270° are scaled so they appear the same size on a monitor screen with a 4:3 aspect ratio.
TA *n*[1]	Turn an angle.	-360 <= n <= 360	If n is positive, counter-clockwise rotation.
S *n*[1]	Set scale.	1 <= n <= 255	Set scale factor n, which can range from 0 to 255, inclusive. Increase or decrease length of moves. The default for n is 4, which causes no scaling. The scale factor multiplied by movement-command arguments (divided by 4) gives the actual distance moved.
C *n*[1]	Set color		
P *p,b*			Option *p* is the paint color for a figure's interior; *b* is the paint color for the figure's border.

1 The commands above specify movement in terms of *n* units:
The default unit size is one pixel.
Unit size can be modified by the S command (set scale factor).
If no unit argument is supplied, the graphics cursor is moved one unit.

The following example uses TA to draw spokes:

```
SCREEN 1
FOR D = 0 TO 360 STEP 10
  DRAW "TA=" + VARPTR$(D) + "NU50"
NEXT D
```

To execute a DRAW command substring from a DRAW command string, use the "X" substring command. For example:

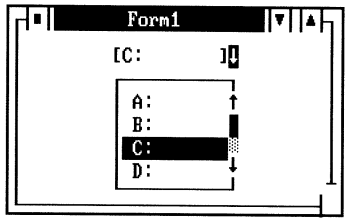
```
DRAW "X"+ VARPTR$(commandstring$)
```

Example This example uses the DRAW statement to draw a triangle's outline in magenta and to paint the interior cyan.

```
CLS                                    ' Clear the screen.
SCREEN 1
DRAW "C2"                              ' Set color to magenta.
DRAW "F60 L120 E60"                    ' Draw a triangle.
DRAW "BD30"                            ' Move down into the triangle.
DRAW "P1,2"                            ' Paint interior.
SLEEP 3
WIDTH 80
SCREEN 0
```

Drive List Box Control

Action Displays a list of all valid drives in the user's system; allows the user to find and switch between valid disk drives at run time.



Remarks

Events

Change	DragDrop	DragOver	GotFocus
KeyDown	KeyPress	KeyUp	LostFocus

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	Ctlname	DragMode	Drive
Enabled	ForeColor	Height	Index
Left	List	ListCount	ListIndex
MousePointer	Parent	TabIndex	TabStop
Tag	Top	Visible	Width

A drive list box displays all existing floppy drives, fixed media, and network connections in alphabetical order at run time.

Use the drive list box's Drive property to update the Path property of a directory list box. For example:

```
SUB Drive1_Change ()
    Dir1.Path = Drive1.Drive
END SUB
```

Use the List, ListCount, and ListIndex properties to access items displayed in the list box at run time.

The minimum size of a drive list box is 1 x 5; dimensions correspond to the Height and Width property settings. The maximum value for Height is 1 (read-only). Specified settings that are less than the minimum value are set to the minimum value.

See Also Directory List Box Control, File List Box Control.

Example This example displays a list of files for the current drive and directory using a drive list box and directory list box control. To try this example, create a form containing a drive list box, a file list box, and a directory list box.

```
SUB Dir1_Change ()
    File1.Path = Dir1.Path           ' When Dir changes, set File path.
END SUB

SUB Drive1_Change ()
  Dir1.Path = Drive1.Drive          ' When Drive changes, set Dir path.
END SUB
```

Drive Property

Applies To Drive list box.

Action Sets or returns the selected drive at run time; not available at design time.

Syntax [*form.*]*drivelistbox*.**Drive**[= *drivespec$*]

Part	Description
drivespec$	Valid range for drives, including all drives present in or connected to the system when the control is refreshed at run time; default is the current drive.

Remarks When reading this property, the selected drive is returned in the "C:" format.

When writing to this property:

- Only the first character of the string is significant (the string is not case-sensitive).
- Changing the setting for Drive causes a Change event.

Setting this property also regenerates the drive list. This provides a way to track network connections that were added since the control was created.

The Drive property returns a different value from ListIndex, which tracks the list box section.

See Also Directory List Box Control, File List Box Control, Path Property.

Example See Drive List Box Control example.

DropDown Event

Applies To Combo box.

Action Occurs when the list portion of a combo box is about to drop down; does not occur with simple combo boxes (Style = 1).

Syntax **SUB** *ctlname*_**DropDown** ([**Index AS INTEGER**])

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks Typically, you use a GotFocus event procedure to prescribe actions to occur when a control or form first receives the focus. For example, by attaching a GotFocus event procedure to each control on a form, you can guide the user by displaying brief instructions on the screen or status bar messages. You can also provide visual cues by enabling, disabling, or displaying other controls that depend on the control that has the focus.

An object can receive the focus only if its Enabled and Visible properties are set to True (-1).

To customize the keyboard interface so that the focus can be moved from one control to another, set the tab order or specify access keys for controls on a form.

A control that has the focus cannot receive a second GotFocus event even if the SETFOCUS method is used.

See Also ADDITEM Method, REMOVEITEM Method, Style Property.

Example The example updates a combo box based on the user's selection in an option button group. To try this example, create form containing a combo box and two option buttons. Set the CtlName property for both option buttons to "OptionGroup."

```
SUB Combo1_DropDown ()
    WHILE Combo1.ListCount              ' Delete existing items.
         Combo1.REMOVEITEM 0
    WEND
    IF OptionGroup(0).Value = -1 THEN
         Combo1.ADDITEM "Gray Goose Express", 0
         Combo1.ADDITEM "Wild Fargo Carriers", 1
     ELSE
         Combo1.ADDITEM "Pegasus Overnight"
     END IF
END SUB
```

$DYNAMIC, $STATIC Metacommands

Action Specifies that arrays declared in subsequent DIM statements are dynamic ($DYNAMIC) or static ($STATIC) arrays.

Syntax **{REM | '} $DYNAMIC**
{REM | '} $STATIC

Part	Description
REM | '	REM or remark character ('); must precede all metacommands
$DYNAMIC	Specifies array storage as dynamic; allocates space for arrays while the program is running.
$STATIC	Specifies array storage as static; sets aside storage for arrays during compilation.

Remarks The DIM and REDIM statements usually provide a better way to specify whether arrays are dynamic or static. You can also use REDIM to change the size of an array allocated using $DYNAMIC.

When $STATIC is used, the ERASE statement reinitializes all array values to zero (numeric arrays) or to a null string (string arrays), but does not remove the array from memory.

When $DYNAMIC is used, array storage is allocated while the program is running. Thus, the ERASE statement removes the array from memory and frees up space for other uses.

$STATIC and $DYNAMIC metacommands affect all arrays except implicitly dimensioned arrays (arrays not declared in a DIM statement). By default, implicitly dimensioned arrays are always allocated as if $STATIC had been used.

A DIM statement declared with a static array is a declarative statement; a DIM statement declared with a dynamic array is an executable statement.

All arrays inside a SUB or FUNCTION procedure are dynamic unless the STATIC keyword is included in the SUB or FUNCTION statement.

The STATIC statement, STATIC keyword, and $STATIC metacommand are all used to affect variables and arrays. Their usage differs as follows:

Usage	Description
STATIC keyword in SUB and FUNCTION statements	Declares the default for variables as fixed or static; variables with the same name as variables shared by module-level code are still shared.
STATIC statement	Makes specific variables static; overrides any variables shared by module-level code.
$STATIC metacommand	Affects how memory is allocated for arrays.

See Also REM Statement, STATIC Statement.

Example This example uses the ERASE statement with the $DYNAMIC and $STATIC metacommands:

```
CLS                         ' Clear the screen.
REM $DYNAMIC
DIM a(100, 100)
ERASE a                     ' Deallocates array A.
REDIM a(5, 5)               ' Redimension array A.
REM $STATIC
DIM B(50, 50)
PRINT B
ERASE B                     ' This sets all elements of B equal to zero.
                            ' B still has the dimensions assigned by DIM.
```

Enabled Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, timer control, vertical scroll bar.

Action Determines if the form or control can respond to user-generated events such as keypress and mouse events.

Syntax [*form.*][*ctlname.*]**Enabled**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	(Default) Allows the form or control to respond to events.
False (0)	Prevents the form or control from responding to events.

Remarks This property allows you to enable or disable forms and controls at run time. For example, you might disable a control used purely for display purposes, such as a text box that provides read-only information.

If Enabled is set to False (0):

- The control and any of its children are grayed and removed from the tab order.
- Child controls cannot receive or respond to events. Note: The only controls that can have children are forms, frames, and picture boxes.

Disabling a timer control by setting Enabled to False (0) suspends the countdown set up by the control's Interval property. Enabling the timer (setting Enabled to True (-1)) starts the countdown from the value of the object's Interval property, not the value at the point at which it was suspended.

See Also Visible Property.

Example This example alternates between enabling and disabling a command button when you type text into a text box. To try this example, create a form containing a text box and a command button.

```
SUB Text1_Change ()
    IF Command1.Enabled = -1 THEN
        Command1.Enabled = 0
        Command1.Caption = "Disabled"
    ELSE
        Command1.Enabled = -1
        Command1.Caption = "Enabled"
    END IF
END SUB
```

END Statement

Action Ends a program, procedure, statement block, or user-defined data type.

Syntax **END [DEF | FUNCTION | IF | SELECT | SUB | TYPE |** *n%***]**

Part	Description
DEF	Ends a multiline DEF FN function definition.
FUNCTION	Ends a FUNCTION procedure definition.
IF	Ends an IF...END IF or IF TYPEOF statement.
SELECT	Ends a SELECT CASE block.
SUB	Ends a SUB procedure.
TYPE	Ends a user-defined data type definition.
n%	Sets the MS-DOS error level variable used in batch files or non-Visual Basic programs; untrapped errors return a value of -1.

Remarks By itself, the END statement stops program execution, closes all files, and returns 0 to the operating system. In a compiled program, END returns control to the operating system; in Visual Basic, END returns to the programming environment.

The following table shows where END is required:

Location	Required
End of a program.	No
End of multiline DEF FN function definition.	Yes
End of a FUNCTION procedure definition.	Yes
End of a block IF...END IF statement.	Yes
End of a block IF TYPEOF statement.	Yes
End of a SELECT CASE block.	Yes
End of a Visual Basic SUB procedure.	Yes
End of a user-defined type definition (TYPE).	Yes

You can place any number of END statements anywhere in the program to to stop execution.

See Also DEF FN Statement, FUNCTION Statement, IF...END IF Statement, SELECT CASE Statement, STOP Statement, SUB Statement, SYSTEM Statement, TYPE Statement.

Example This example uses the ON ERROR statement to enable error trapping. The program attempts to write a large file to the disk. If an error occurs, control is passed to a handling routine, and the ERR function is used to determine an appropriate message to display. The ERROR, ERR, and RESUME statements are used to exit the handling routines, and the END, STOP, and SYSTEM statements demonstrate ways to exit a program.

```
DECLARE SUB ErrorMessage (Message$)
DECLARE SUB WriteBigFile (Filenum%)
ON ERROR GOTO ErrHandler
CLS                                 ' Clear the screen.
PRINT "This program will attempt to write a large file to a disk drive"
PRINT "you select. The file will be erased when the program ends."
PRINT
DO
    INPUT "Which drive"; DR$
    DR$ = UCASE$(DR$)
LOOP UNTIL LEN(DR$) >= 1 AND LEN(DR$) <= 2 AND DR$ >= "A" AND DR$ <= "Z"
IF LEN(DR$) > 1 THEN
    IF RIGHT$(DR$, 1) <> ":" THEN
         DR$ = LEFT$(DR$, 1) + ":"
    END IF
ELSE
    DR$ = DR$ + ":"
END IF

' Put together a complete file specification.
FileSpec$ = DR$ + "BIGFILE.XXX"
Filenum% = FREEFILE                     ' Get the next available file number.
OPEN FileSpec$ FOR OUTPUT AS Filenum%   ' Try to open the file.
WriteBigFile Filenum%
CLOSE Filenum%
CLS
PRINT "Everything was OK. No errors occurred."
PRINT "Deleting the file that was created."
KILL FileSpec$
'Same as END, returns to operating system.
SYSTEM

ErrHandler:
    SELECT CASE ERR
         CASE 52                    ' Bad file name or number.
         END
    CASE 53                         ' File not found.
         RESUME NEXT
    CASE 57                         ' Device I/O error.
         ErrorMessage "You should probably format the diskette."
         END
```

```
    CASE 64                         ' Bad file name.
         ErrorMessage "The drive name you specified was not correct."
         END
    CASE 68                         ' Device unavailable.
         ErrorMessage "The drive you named is unavailable."
         END
    CASE 71                         ' Drive not ready.
       ErrorMessage "The drive was not ready. Check the drive!"
         END
    CASE ELSE
         ErrorMessage "An unexpected FATAL error has occurred."
        STOP
END SELECT

SUB ErrorMessage (Message$)
    ON LOCAL ERROR GOTO MessageError
    CLS
    PRINT Message$
    PRINT "Cannot continue."
    PRINT
    PRINT "Press any key to exit."
    DO
    LOOP WHILE INKEY$ = ""
    EXIT SUB
MessageError:
    RESUME NEXT
END SUB

SUB WriteBigFile (Filenum%)
    ON LOCAL ERROR GOTO LocalHandler
    TEXT$ = STRING$(1024, "A")
    FOR i% = 1 TO 400
        PRINT #Filenum%, TEXT$
    NEXT i%
    EXIT SUB
LocalHandler:
    SELECT CASE ERR
        CASE 61                      ' Disk full.
            ErrorMessage ("There is no room remaining on the disk.")
            KILL "BIGFILE.XXX"
            END
        CASE ELSE
            ERROR ERR
    END SELECT
END SUB
```

ENDDOC Method

Applies To PRINTER object.

Action Terminates a document sent to the printer, and releases it to the print device.

Syntax **PRINTER.ENDDOC**

Remarks If ENDDOC is invoked immediately after NEWPAGE, no additional blank page is printed.

See Also NEWPAGE Method.

Example This example illustrates the use of the ENDDOC, PRINT, and NEWPAGE methods. ENDDOC is used to end a document after printing via the PRINT method. Two pages are printed (NEWPAGE), each with a centered line of text indicating the page number. Note: You must have a printer connected to LPT1: and it must be turned on before running this example.

```
DECLARE SUB PrinterDemo ()

SUB Form_Click ()
    CALL PrinterDemo
END SUB
SUB PrinterDemo ()
    ON LOCAL ERROR GOTO ErrorHandler          ' Set up error handler.
    Msg$ = "This is printed on page"
    FOR I% = 1 TO 2                           ' Set up two iterations.
        PRINTER.PRINT Msg$                    ' Print.
        PRINTER.NEWPAGE                       ' Send new page.
    NEXT I%
    PRINTER.ENDDOC                            ' Print done.
    Msg$ = "Two pages, each with a single, centered line of text, "
    Msg$ = Msg$ + CHR$(10) + CHR$(13) + "have been sent to your printer."
    MSGBOX Msg$                               ' Display message.
    EXIT SUB

ErrorHandler:
    MSGBOX "There was a problem printing to your printer."
    EXIT SUB
    END SUB
```

ENVIRON Statement, ENVIRON$ Function

Action Returns an MS-DOS environment string (ENVIRON); changes or adds an environment string in the environment-string table (ENVIRON$).

Syntax

**ENVIRON$** (*env-variable$*)
ENVIRON$ (*n%*)
ENVIRON *stringexpression$*

Part	Description
env-variable$	Name of an MS-DOS environment variable; must be all uppercase.
n%	Causes ENVIRON$ to return the nth string from the environment-string table.
stringexpression$	Name and setting of an MS-DOS environment variable (such as PATH or PROMPT) in one of the following forms: *env-variable$ = env-string$* *env-variable$ env-string$*

Remarks The following example sets a PATH environment variable:

```
ENVIRON "PATH=C:\CLIENTS;C:\RATES"
```

If *env-variable$* did not previously exist in the environment-string table, it is appended to the end of the table. If *env-variable$* exists in the table when the ENVIRON statement is executed, it is deleted, and the new *env-variable$* is appended to the end of the table.

If text is a null string ("") or a semicolon (";"), the existing parameter is removed from the environment-string table, and the remaining body of the table is compressed.

You cannot increase the size of the environment-string table with the ENVIRON statement. This means that before you can add a new environment variable or increase the size of an existing environment variable, you must first delete or decrease the size of existing environment variables.

Visual Basic generates the error message "Out of memory" when no more space can be allocated to the environment-string table. The amount of free space in the table is usually limited.

The environment variable must be expressed in all uppercase letters. For example:

This Statement...	Has This Effect...
ENVIRON "PATH=C:SALES"	Changes the path.
ENVIRON "path=C:SALES"	Does not change the path; creates a new environment variable that cannot be used by the operating system.

MS-DOS discards the environment-string table modified by the ENVIRON$ function when your program ends. The environment-string table is then the same as it was before your program ran.

You can use this statement to change the PATH parameter for a "child" process (a program or command started by a SHELL statement) or to pass parameters to a child by creating a new environment variable.

If you specify an environment-string name, but it cannot be found in the environment-string table, or there is no text following it, then ENVIRON$ returns a null string. Otherwise, ENVIRON$ returns the text following the equal sign in the environment-string table.

If you specify a numeric argument (n), the nth string in the environment-string table is returned. In this case, the string includes all of the text, including environmentstring$. If the nth string does not exist, ENVIRON$ returns a null string. The n argument can be any numeric expression; it is rounded to an integer.

Example

This example uses the ENVIRON$ function to get a copy of your current PATH variable. Your PATH is then changed using the ENVIRON statement. The program then uses the ENVIRON$ function to print out the contents of the environment string table. The example concludes by restoring your original PATH statement.

Note: For this example to run, the value of your path variable must be longer than the new path value in the ENVIRON statement.

```
CLS                                                ' Clear the screen.
DEFINT A-Z
PATH$ = "PATH="                                    ' Initialize.
I% = 1
OldPath$ = ENVIRON$("PATH")                        ' Get the old PATH.
ENVIRON "PATH=C:\BIN;C:\DOS;C:\BUSINESS"
PRINT "Your current environment settings are:"     ' Display the string table.
PRINT
DO WHILE ENVIRON$(I%) <> ""
    PRINT ENVIRON$(I%)
    I% = I% + 1
LOOP
ENVIRON PATH$ + OldPath$                           ' Change the PATH back.
PRINT
PRINT "Your PATH has been restored to:"            ' Verify the change.
PRINT
PRINT ENVIRON$("PATH")
```

EOF Function

Action Returns true (nonzero) at the end of a file or ISAM table (ISAM available in Professional Edition only).

Syntax **EOF**(*filenumber%*)

Part	Description
filenumber%	Number specifying an open file or ISAM table.

Remarks Use the EOF function with sequential files to test for the end of a file. Otherwise, Visual Basic generates the error message, "Input past end."

When used with random-access or binary files, EOF returns true if the last executed GET statement was unable to read an entire record.

When you use EOF with a communications device, the definition of the end-of-file condition depends on the mode (ASCII or binary) in which you opened the device:

Mode	Description
ASCII	EOF is false until you receive Ctrl+Z, after which it remains true until you close the device.
Binary	EOF is true when the input queue is empty (LOC(n)=0); it becomes false when the input queue is not empty.

The EOF function returns true (nonzero) if the end of a non-ISAM file has been reached or if the current position in an ISAM table is at the end of the table. The end of an ISAM table is the position immediately following the last record according to the current index.

EOF cannot be used with the Visual Basic devices SCRN, KYBD, CONS, LPTn, or PIPE.

See Also LOF Function, LOC Function, OPEN Statement (File I/O).

Example The example below uses the INPUT statement to receive input from the user and the EOF function to test for the end-of-file condition which will cause the program to end.

```
CLS                                     ' Clear the screen.
OPEN "LIST" FOR OUTPUT AS #1
DO
    INPUT "   NAME:       ", Name$      ' Read entries from the keyboard.
    INPUT "   AGE:        ", Age$
    WRITE #1, Name$, Age$
    INPUT "Add another entry"; R$
LOOP WHILE UCASE$(R$) = "Y"
CLOSE #1
```

```
OPEN "LIST" FOR INPUT AS #1          ' Echo the file back.
CLS                                  ' Clear the screen.
PRINT "Entries in file:": PRINT
DO WHILE NOT EOF(1)
    LINE INPUT #1, REC$              ' Read entries from the file.
    PRINT REC$                       ' Print the entries on the screen.
LOOP
    CLOSE #1
    KILL "LIST"
```

ERASE Statement

Action Reinitializes array elements or frees dynamic array storage space.

Syntax **ERASE** *arrayname* [, *arrayname*]...

Part	Description
arrayname	Any valid array name.

Remarks The ERASE statement sets the elements of a static array as follows:

Array Type	Element
Numeric static	Zeros.
String static	Null strings ("").
Array of records	Zeros (all elements of each word, including fixed strings).

Using ERASE on a dynamic array frees the memory used by the array. Before your program can refer to the dynamic array again, you must reallocate space for the array by redeclaring its dimensions with DIM or REDIM. For example:

- If you redeclare an array's dimensions with DIM (without first erasing the array), then Visual Basic generates the run-time error message, "Array already dimensioned."
- If you redeclare an array's dimensions with REDIM, then the ERASE statement is not required.

Example See $DYNAMIC Metacommand example

ERDEV, ERDEV$ Functions

Action

Returns an error code from the last device that generated a critical error (ERDEV); returns the name of the device that generated the error (ERDEV$).

ERDEV
ERDEV$

Remarks

For ERDEV, the low byte of the value contains the MS-DOS error code; the high byte contains device-attribute information.

ERDEV is set by the critical error handler (interrupt 24H) when MS-DOS detects a critical MS-DOS call error from a character or block device. It also is set by a timeout error on a communications port.

For block device errors, bits 0 - 7 (the low byte) of ERDEV contain the MS-DOS error code. Bits 8 - 15 (the high byte) contain the following device-attribute information:

Bit(s)	Value	Meaning
0 - 7	xxH	Contains MS-DOS error code xxH.
8	0	Read operation.
	1	Write operation.
9 - 10		Indicates the affected disk area:
	00	MS-DOS.
	01	File allocation table.
	10	Root directory.
	11	Files area.
11[1]	0	Fail response not allowed.
	1	Fail response allowed.
12[1]	0	Retry response not allowed.
	1	Retry response allowed.
13[1]	0	Ignore response not allowed.
	1	Ignore response allowed.
14		Not used.
15	0	Always zero.

1 Bits 11, 12, and 13 apply only to MS-DOS version 3.1 and later

For character device errors, bits 0-7 contain the MS-DOS error code. The upper bits are not significant.

For COM timeout errors, ERDEV returns a value indicating which option in the OPEN COM statement is experiencing the timeout. Therefore, it's not necessary to break ERDEV into low and high bytes:

ERDEV Value	OPEN COM Option Timeout
80H	CS (CTS timeout).
81H	DS (DSR timeout).
82H	CD (RLSD timeout).

ERDEV$ returns a string containing:

- 8-byte character device name if the error is on a character device, such as a printer.
- 3-byte block name (COM) if a communications port experiences a timeout.
- 2-byte block device name (A:, B:, and so on) if the device is not a character device.

Because ERDEV and ERDEV$ return meaningful information only after an error, they usually are used in error handlers specified by an ON ERROR statement.

ERDEV and ERDEV$ cannot be used on the left side of an assignment.

The following example extracts the low and high bytes from the ERDEV value. DosErrCode stores the block or character device error code, while DevAttr stores the block device attribute information:

```
DosErrCode = ERDEV AND &HFF                    ' Low byte of ERDEV.
DevAttr    = (ERDEV AND &HFF00) \ 256          ' High byte of ERDEV.
```

Example

This example prints the values of ERDEV and ERDEV$ after the program generates an error attempting to open a text file.

```
DEFINT A-Z

ON ERROR GOTO ErrorHandler       ' Indicate first line of error handler.
OPEN "A:JUNK.DAT" FOR INPUT AS #1' Attempt to open the file.
END

ErrorHandler:                              ' Error handling routine.
 PRINT "ERDEV value is "; ERDEV            ' Prints values of ERDEV and ERDEV$
 PRINT "Device name is "; ERDEV$           ' and returns error control to Visual
                                           ' Basic.
 SLEEP 2
 ON ERROR GOTO 0
```

ERR Statement

Action Assigns a value to ERR.

Syntax **ERR** = *n%*

Part	Description
n%	Integer expression with a value between 1 and 32,767, inclusive, that specifies a run-time error code, or 0, which means no run-time error has occurred.

Remarks When running an application program, Visual Basic uses ERR to record whether a run-time error has occurred and what the error was. When the program starts running, ERR is 0; when and if a run-time error occurs, Visual Basic sets ERR to the error code for that error.

Use the ERR statement to set ERR to a nonzero value to communicate error information between procedures. For example, you might use one of the run-time codes not used by Visual Basic as an application-specific error code.

Besides the ERR statement, the following Visual Basic statements set ERR whenever they execute:

This Statement...	Sets ERR to...
Any form of **RESUME**	0
EXIT SUB, **EXIT FUNCTION**,or **EXIT DEF**	0 (if executed within a procedure-level error handler)
All uses of **ON ERROR**	0
ON LOCAL ERROR	0
ERROR	Any value (as part of simulating any run-time error)

See Also ERR Function, ERROR Statement.

Example This example uses the ON ERROR statement to enable error trapping. The program attempts to write a large file to the disk. If an error occurs, control is passed to a handling routine, and the ERR function is used to determine an appropriate message to display. The ERROR, ERR, and RESUME statements are used to exit the handling routines, and the END, STOP, and SYSTEM statements demonstrate ways to exit a program.

```
DECLARE SUB ErrorMessage (Message$)
DECLARE SUB WriteBigFile (Filenum%)
ON ERROR GOTO ErrHandler
CLS                              ' Clear the screen.
```

```
PRINT "This program will attempt to write a large file to a disk drive"
PRINT "you select. The file will be erased when the program ends."
PRINT
DO
    INPUT "Which drive"; DR$
    DR$ = UCASE$(DR$)
LOOP UNTIL LEN(DR$) >= 1 AND LEN(DR$) <= 2 AND DR$ >= "A" AND DR$ <= "Z"
IF LEN(DR$) > 1 THEN
    IF RIGHT$(DR$, 1) <> ":" THEN
         DR$ = LEFT$(DR$, 1) + ":"
    END IF
ELSE
    DR$ = DR$ + ":"
END IF

' Put together a complete file specification.
FileSpec$ = DR$ + "BIGFILE.XXX"
Filenum% = FREEFILE                    ' Get the next available file number.
OPEN FileSpec$ FOR OUTPUT AS Filenum%  ' Try to open the file.
WriteBigFile Filenum%
CLOSE Filenum%
CLS
PRINT "Everything was OK. No errors occurred."
PRINT "Deleting the file that was created."
KILL FileSpec$
'Same as END, returns to operating system.
SYSTEM

ErrHandler:
    SELECT CASE ERR
         CASE 52                   ' Bad file name or number.
         END
    CASE 53                        ' File not found.
         RESUME NEXT
    CASE 57                        ' Device I/O error.
         ErrorMessage "You should probably format the diskette."
         END
    CASE 64                        ' Bad file name.
         ErrorMessage "The drive name you specified was not correct."
         END
    CASE 68                        ' Device unavailable.
         ErrorMessage "The drive you named is unavailable."
         END
    CASE 71                        ' Drive not ready.
       ErrorMessage "The drive was not ready. Check the drive!"
         END
```

```
    CASE ELSE
         ErrorMessage "An unexpected FATAL error has occurred."
        STOP
END SELECT

SUB ErrorMessage (Message$)
    ON LOCAL ERROR GOTO MessageError
    CLS
    PRINT Message$
    PRINT "Cannot continue."
    PRINT
    PRINT "Press any key to exit."
    DO
    LOOP WHILE INKEY$ = ""
    EXIT SUB
MessageError:
    RESUME NEXT
END SUB

SUB WriteBigFile (Filenum%)
    ON LOCAL ERROR GOTO LocalHandler
    TEXT$ = STRING$(1024, "A")
    FOR i% = 1 TO 400
        PRINT #Filenum%, TEXT$
    NEXT i%
    EXIT SUB
LocalHandler:
    SELECT CASE ERR
        CASE 61                     ' Disk full.
            ErrorMessage ("There is no room remaining on the disk.")
            KILL "BIGFILE.XXX"
            END
        CASE ELSE
            ERROR ERR
    END SELECT
END SUB
```

ERR, ERL Functions

Action

Returns the run-time error code for the most recent error (ERR); returns the line number where the error occurred, or the closest line number before the line where the error occurred (ERL).

Syntax

ERR
ERL

Remarks

After an error, the ERR function returns the latest setting of the run-time error code, and the ERL function returns the line number where the error occurred. Because ERR and ERL return meaningful values only after an error, they are usually used in error-handling routines to determine the error and the corrective action.

You can directly set the value returned by the ERR function using the ERR statement. The values returned by both ERR and ERL can be set indirectly with the ERROR statement.

The ERL function returns only the line number (not a line label) located at or before the line producing the error. If your program has no line numbers, or if there is no line number in the program before the point where the error occurred, ERL returns 0.

ERL values in programs compiled with BC differ from those run under VBDOS:

- In programs compiled with BC, ERL is set to the last numbered line in the source file preceding the line where the error occurred.
- When VBDOS detects an error in a procedure, it looks for a line number only in the immediate procedure. If the procedure doesn't contain a numbered line, VBDOS returns 0 for ERL.

You can make your programs run identically with both BC and VBDOS by always including a line number at the beginning of a procedure where an error might occur.

See Also

ERR Statement, ERROR Statement.

Example

This example shows how the ERR and ERL functions can be used in an error- handling routine.

In this example, line numbers are used for diagnostic purposes, not for essential program logic flow. When program logic is stable, remove all line numbers except those used to recover from trapped user errors.

```
     CLS              ' Clear the screen.
     Recnum% = 0      ' Redundant since default is 0.
     ON ERROR GOTO Errhandler
10 : ERROR 4          ' Force an error to check trap.

15 : OPEN "errtest.dat" FOR RANDOM AS #1 LEN = 80
     PRINT "Enter data <= 80 characters then return."
     PRINT "To Quit, press Enter at prompt."
     DO
         Recnum% = Recnum% + 1
25 :      INPUT "Your data"; aline$
30 :      PUT #1, Recnum%, aline$
     LOOP WHILE aline$ <> ""
     IF Recnum% = 1 AND aline$ = "" THEN
         PRINT "    Empty file."
```

```
35 :      ERROR 200
     END IF
     PRINT " This is your data file."
     FOR i% = 1 TO Recnum%
45 :      GET #1, i%, outaline$
         PRINT outaline$
     NEXT i%
60 :     CLOSE #1
     KILL "errtest.dat"                          ' Deletes file from disk.
     END

Errhandler:
     Errnum% = ERR
     Errlin% = ERL    ' Note that ERL returns only line NUMBERS Not LABELS!
     PRINT "Error number "; Errnum%; " occurred at line number "; Errlin%
     IF Errnum% = 200 THEN RESUME 60 ' If user quits before any
                                     ' input CLOSE and END.
     IF Errnum% = 59 THEN RESUME 25  ' If user exceeds LEN then try again!

' This returns to next executable statement after error.
     RESUME NEXT
```

ERROR Statement

Action

Simulates the occurrence of a Visual Basic error or a user-defined error.

Syntax

ERROR *expression%*

Part	Description
expression%	Error code of a Visual Basic or user-defined error; a value between 1 and 32,767, inclusive.

Remarks

If *expression%* is an error code already used by Visual Basic, the ERROR statement simulates the occurrence of that error.

To define your own error code, use a value that is greater than any used by the standard Basic error codes. Start by using error code 32,767 and work down to avoid compromising compatibility with future Microsoft Basic error codes. In general, the error codes used by Visual Basic are between 1 and 100, although not all these are used.

If an ERROR statement is executed when no error-handling routine is enabled, Visual Basic generates an error message and stops program execution. If the ERROR statement specified an error code that Visual Basic does not use, the message "Unprintable error" is generated.

See Also ERROR$ Function.

Example This example uses the ON ERROR statement to enable error trapping. The program attempts to write a large file to the disk. If an error occurs, control is passed to a handling routine, and the ERR function is used to determine an appropriate message to display. The ERROR, ERR, and RESUME statements are used to exit the handling routines, and the END, STOP, and SYSTEM statements demonstrate ways to exit a program.

```
DECLARE SUB ErrorMessage (Message$)
DECLARE SUB WriteBigFile (Filenum%)
ON ERROR GOTO ErrHandler
CLS                                 ' Clear the screen.
PRINT "This program will attempt to write a large file to a disk drive"
PRINT "you select. The file will be erased when the program ends."
PRINT
DO
    INPUT "Which drive"; DR$
    DR$ = UCASE$(DR$)
LOOP UNTIL LEN(DR$) >= 1 AND LEN(DR$) <= 2 AND DR$ >= "A" AND DR$ <= "Z"
IF LEN(DR$) > 1 THEN
    IF RIGHT$(DR$, 1) <> ":" THEN
        DR$ = LEFT$(DR$, 1) + ":"
    END IF
ELSE
    DR$ = DR$ + ":"
END IF

' Put together a complete file specification.
FileSpec$ = DR$ + "BIGFILE.XXX"
Filenum% = FREEFILE                     ' Get the next available file number.
OPEN FileSpec$ FOR OUTPUT AS Filenum%   ' Try to open the file.
WriteBigFile Filenum%
CLOSE Filenum%
CLS
PRINT "Everything was OK. No errors occurred."
PRINT "Deleting the file that was created."
KILL FileSpec$
'Same as END, returns to operating system.
SYSTEM

ErrHandler:
    SELECT CASE ERR
        CASE 52                     ' Bad file name or number.
        END
    CASE 53                         ' File not found.
        RESUME NEXT
```

```
    CASE 57                         ' Device I/O error.
         ErrorMessage "You should probably format the diskette."
         END
    CASE 64                         ' Bad file name.
         ErrorMessage "The drive name you specified was not correct."
         END
    CASE 68                         ' Device unavailable.
         ErrorMessage "The drive you named is unavailable."
         END
       CASE 71                          ' Drive not ready.
       ErrorMessage "The drive was not ready. Check the drive!"
         END
    CASE ELSE
         ErrorMessage "An unexpected FATAL error has occurred."
        STOP
END SELECT

SUB ErrorMessage (Message$)
    ON LOCAL ERROR GOTO MessageError
    CLS
    PRINT Message$
    PRINT "Cannot continue."
    PRINT
    PRINT "Press any key to exit."
    DO
    LOOP WHILE INKEY$ = ""
    EXIT SUB
MessageError:
    RESUME NEXT
END SUB

SUB WriteBigFile (Filenum%)
    ON LOCAL ERROR GOTO LocalHandler
    TEXT$ = STRING$(1024, "A")
    FOR i% = 1 TO 400
        PRINT #Filenum%, TEXT$
    NEXT i%
    EXIT SUB
LocalHandler:
    SELECT CASE ERR
        CASE 61                     ' Disk full.
            ErrorMessage ("There is no room remaining on the disk.")
            KILL "BIGFILE.XXX"
            END
        CASE ELSE
            ERROR ERR
    END SELECT
END SUB
```

ERROR$ Function

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

Syntax **ERROR$**[(*errorcode%*)]

Part	Description
errorcode%	Any valid error code in the range 1 - 32,767, inclusive; if omitted, returns the message string corresponding to the most recent run-time error. If there has been no run-time error, ERROR$ returns a null string ("").

See Also ERR Function, ERL Function, ERROR Statement.

Example The example uses the ERROR$ function to display the error messages associated with user-input error numbers.

```
DECLARE SUB ErrorDemo ()
CALL ErrorDemo

SUB ErrorDemo ()
    ON LOCAL ERROR GOTO Errorhandler          ' Set up error handler.
    NL$ = CHR$(13) + CHR$(10)                 ' Define new line.
    Msg$ = "Please enter an error number to see the associated error"
    Msg$ = Msg$ + " message."
    Usererror% = VAL(INPUTBOX$(Msg$))
    ERROR Usererror%                          ' Simulate error occurence.
    EXIT SUB

Errorhandler:
    Msg$ = "The error message for error number "
    Msg$ = Msg$ + LTRIM$(STR$(ERR)) + " is:" + NL$ + NL$
    Msg$ = Msg$ + """" + ERROR$(ERR) + """"
    MSGBOX Msg$                               ' Display message.
    RESUME NEXT

END SUB
```

EVENT Statement

Action When set to ON, event trapping is enabled until Visual Basic encounters an EVENT OFF statement. When set to OFF, event trapping is disabled. Use object-specific events with forms.

Syntax

EVENT {ON | OFF}

Remarks

The EVENT statement does not affect form- or control-specific events. Forms and controls use different event handlers. Use object-specific events with forms.

If your program contains event-handling statements and you are compiling from the Basic Compiler (BC.EXE) command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The EVENT statement affects compiled code generation. With EVENT OFF, the compiler will not generate any event-checking code.

The effect of the EVENT statement is always global.

EVENT ON and EVENT OFF can be used to bracket sections of code where events do not need to be detected and trapped, and fast performance is required.

If a trapped event occurs while EVENT OFF is in effect, that event is remembered and handled if and when an EVENT ON block is entered.

Example

This example uses the EVENT and STRIG statements to enable joystick event trapping for two buttons on each of two joysticks. The ON STRIG statement passes control to an event-handling routine when a button is pressed. The STRIG function is used to determine which button was pressed, and the STICK function returns the position of both joysticks.

```
CLS                                 ' Clear the screen.
EVENT ON                            ' Turn on event processing.

' Turn on event processing.
STRIG(0) ON                         ' Lower button, joystick A.
STRIG(2) ON                         ' Lower button, joystick B.
STRIG(4) ON                         ' Upper button, joystick A.
STRIG(6) ON                         ' Upper button, joystick B.

' Set up joystick event processing for each button.
ON STRIG(0) GOSUB JoyButtonHandler
ON STRIG(2) GOSUB JoyButtonHandler
ON STRIG(4) GOSUB JoyButtonHandler
ON STRIG(6) GOSUB JoyButtonHandler

LOCATE 22, 6
PRINT "Press a button on either joystick to see a complete joystick report."
LOCATE 23, 20
PRINT "Press ANY keyboard key to end the program."

' Infinite loop waiting for a joystick event or keyboard input.
DO
LOOP WHILE INKEY$ = ""
```

```
WrapItUp:
    CLS                              ' Clear the screen.
    STRIG(0) OFF
    STRIG(2) OFF
    STRIG(4) OFF
    STRIG(6) OFF
    END

JoyButtonHandler:
    LOCATE 10, 14: PRINT "Joystick A"       ' Print a label on screen.
    LOCATE 10, 45: PRINT "Joystick B"
    LOCATE 22, 1: PRINT STRING$(80, 32)
    DO
        IF STRIG(1) THEN                    ' Button 1, Joystick A.
            ButtonStatus$ = "DOWN"
        ELSE
            ButtonStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 16, 10: PRINT "Button 1 is "; ButtonStatus$
        EVENT ON
        IF STRIG(3) THEN                    ' Button 1, Joystick B.
            ButtonStatus$ = "DOWN"
        ELSE
            ButtonStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 16, 42: PRINT "Button 1 is "; ButtonStatus$
        EVENT ON
        IF STRIG(5) THEN                    ' Button 2, Joystick A.
            ButtonStatus$ = "DOWN"
        ELSE
            ButtonStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 18, 10: PRINT "Button 2 is "; ButtonStatus$
        EVENT ON
        IF STRIG(7) THEN                    ' Button 2, Joystick B.
            ButtonStatus$ = "DOWN"
        ELSE
            ButtonStatus$ = "UP  "
        END IF
        EVENT OFF
        LOCATE 18, 42: PRINT "Button 2 is "; ButtonStatus$
        EVENT ON
        GOSUB UpdateXY
    LOOP WHILE INKEY$ = ""
    RETURN WrapItUp
```

```
UpdateXY:
    EVENT OFF
    LOCATE 12, 10: PRINT USING "X Position = ###"; STICK(0)
    LOCATE 14, 10: PRINT USING "Y Position = ###"; STICK(1)
    LOCATE 12, 42: PRINT USING "X Position = ###"; STICK(2)
    LOCATE 14, 42: PRINT USING "Y Position = ###"; STICK(3)
    EVENT ON
    RETURN
```

EXIT Statement

Action Exits a DO or FOR loop, a FUNCTION or SUB procedure, or a DEF FN statement.

Syntax **EXIT {DEF | DO | FOR | FUNCTION | SUB}**

Part	Description
DEF	Exits a DEF FN statement.
DO	Exits a DO...LOOP statement.
FOR	Exits a FOR...NEXT loop.
FUNCTION	Exits a FUNCTION procedure.
SUB	Exits a SUB procedure.

Remarks EXIT statements do not define the end of the structure in which they are used. EXIT statements provide only an alternative exit from the structure.

EXIT DEF causes an immediate exit from the executing DEF FN function. Program execution continues where the DEF FN function was invoked.

EXIT DO provides an alternative exit from a DO...LOOP. It can be used only inside a DO...LOOP statement; EXIT DO transfers control to the statement following the LOOP statement. When used within nested DO...LOOP statements, it transfers out of the immediately enclosing loop.

EXIT FOR provides another way to exit a FOR...NEXT loop. It can appear only in a FOR...NEXT loop; EXIT FOR transfers control to the statement following the NEXT statement. When used within nested FOR...NEXT loops, transfers out of the immediately enclosing loop.

EXIT FUNCTION causes an immediate exit from a FUNCTION procedure. Program execution continues where the FUNCTION was invoked. It can be used only in a FUNCTION procedure.

EXIT SUB immediately exits a SUB procedure. Program execution continues with the statement after the CALL statement. It can be used only in a SUB procedure.

See Also DEF FN Statement, DO...LOOP Statement, FOR...NEXT Statement, FUNCTION Statement, SUB Statement.

Example This example uses the EXIT statement to exit a variety of control-flow structures. A loop is continuously executed until you press any key. Once a key is pressed, the next EXIT statement that executes will cause the program to end.

```
DECLARE SUB ExitDemo ()
CLS                                 ' Clear the screen.
DO
    PRINT : PRINT "Entering/Reentering ExitDemo"
    ExitDemo
    SLEEP 1
LOOP WHILE INKEY$ = ""
PRINT "Exiting 'EXIT' example program"

SUB ExitDemo ()
   DO
       FOR i% = 1 TO 1000
           Num% = INT(RND * 100)
           SELECT CASE Num%
               CASE 7
                   PRINT "Exiting For...Next Loop in ExitDemo SUB"
                   EXIT FOR
               CASE 29
                   PRINT "Exiting Do...Loop in ExitDemo SUB"
                   EXIT DO
               CASE 54
                   PRINT "Exiting ExitDemo SUB"
                   EXIT SUB
               CASE ELSE
           END SELECT
       NEXT i%
   LOOP
END SUB
```

EXP, LOG Functions

Action Returns e raised to a specified power, where e is the base of natural logarithms (EXP), or returns the natural logarithm of a numeric expression (LOG).

Syntax **EXP**(*numeric-expression*)
LOG(*numeric-expression*)

Part	Description
numeric-expression	For EXP, an expression less than or equal to 88.02969 (single-precision values) or 709.782712893 (double-precision values; for LOG, any valid expression.

Remarks

EXP and LOG are calculated in single precision if *numeric-expression* is an integer or a single-precision value. If you use any other numeric data type, EXP and LOG are calculated in double precision.

The valid range for EXP numeric expressions is <= 88.02969 (single-precision) or <= 709.782712893 (double-precision). If *numeric-expression* is outside the valid range, Visual Basic generates the run-time error message, "Overflow."

The constant e is approximately equal to 2.718282.

The natural logarithm is the logarithm to the base e.

You can calculate base-n logarithms for any number x by dividing the natural logarithm of x by the natural logarithm of n:

```
LOGn = LOG(x)/LOGn
```

For example, the following FUNCTION procedure calculates base-10 logarithms:

```
FUNCTION LOG10(X) STATIC
    LOG10 = LOG(X)/LOG(10)
END FUNCTION
```

Example

This example uses the EXP and LOG functions to print the value of e and calculates the natural logarithm of e taken to the first, second, and third powers.

```
CLS                                 ' Clear the screen.
PRINT EXP(1),
FOR I = 1 TO 3
    PRINT LOG(EXP(1) ^ I),
NEXT
```

FIELD Statement

Action

Allocates space for variables in a random-access file buffer.

Syntax

FIELD [#]*filenumber%*, *fieldwidth%* **AS** *stringvariable$*[,*fieldwidth%* **AS** *stringvariable$*] **...**

Part	Description
filenumber%	Number specified in the OPEN statement to open a file.
fieldwidth%	Number of characters in the field.

Part	Description
stringvariable$	String variable containing data read from a record.

Remarks

Visual Basic record variables and extended OPEN statement syntax provide a more convenient way to use random-access files.

The total number of bytes that you allocate in a FIELD statement must not exceed the record length that you specified when opening the file. Otherwise, Visual Basic generates the error message, "FIELD overflow." The default record length is 128 bytes.

Any number of FIELD statements can be executed for the same file. All FIELD statements that have been executed remain in effect at the same time.

All field definitions for a file are removed when the file is closed — that is, all strings defined as fields associated with the file are set to null.

Do not use a variable name defined as a field in an INPUT or assignment statement if you wish the variable to remain a field. Once a variable name is a field, it points to the correct place in the random-access file buffer. If a subsequent INPUT or assignment statement with that variable name is executed, the variable's pointer no longer refers to the random-access record buffer, but to string space.

See Also

GET Statement (File I/O), PUT Statement (File I/O), OPEN Statement (File I/O).

Example

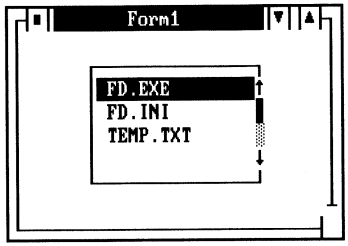
This example uses the FIELD statement to define a random-access file buffer. In the first FIELD statement, the 67-byte buffer is broken up into five separate variables for name, address, city, state, and ZIP code. In the second FIELD statement, the same buffer is assigned entirely to one variable, Plist$. The program then checks to see if Zip$, which contains the ZIP code, falls within a certain range; if it does, the complete address string is printed.

```
TYPE Buffer
   FuName AS STRING * 25
   Addr   AS STRING * 25
   City   AS STRING * 10
   State  AS STRING * 2
   Zip    AS STRING * 5
END TYPE
DIM RecBuffer AS Buffer

' Note: This part of the program creates a random-access file for use by the
' second part of the program, which demonstrates the FIELD statement.

OPEN "MAILLIST.DAT" FOR RANDOM AS #1 LEN = LEN(RecBuffer)
CLS                                  ' Clear the screen.
RESTORE
READ FuName$, Addr$, City$, State$, Zip$
I = 0
```

```
DO WHILE UCASE$(FuName$) <> "END"
    I = I + 1
    RecBuffer.FuName = FuName$
    RecBuffer.Addr = Addr$
    RecBuffer.City = City$
    RecBuffer.State = State$
    RecBuffer.Zip = Zip$
    PUT #1, I, RecBuffer
    READ FuName$, Addr$, City$, State$, Zip$
    IF FuName$ = "END" THEN EXIT DO
LOOP
CLOSE #1

DATA "Drew Fletcher","1200 Liberty St.","Bow","WA","98232"
DATA "Lynn Edmark","123 B St.","Bellevue","WA","98005"
DATA "Laura Bennett","14900 123rd","Bothell","WA","98011"
DATA "Jeff Webb","33 Minnow Lake Road","Lyman","WA","98263"
DATA "Georgette Gump","400 15th W.","Bellevue","WA","98007"
DATA "END",0,0,0,0,0

'This part of the program demonstrates use of the FIELD statement.

'Define field and record lengths with constants.
CONST FU = 25, AD = 25, CT = 10, ST = 2, ZP = 5
CONST RECLEN = FU + AD + CT + ST + ZP

OPEN "MAILLIST.DAT" FOR RANDOM AS #1 LEN = RECLEN
FIELD #1, FU AS FuName$, AD AS Addr$, CT AS City$, ST AS State$, ZP AS Zip$
FIELD #1, RECLEN AS Plist$

GET #1, 1
'Read the file, looking for zip codes in the range 98000 to 98015.
DO WHILE NOT EOF(1)
   Zcheck$ = Zip$
   IF (Zcheck$ >= "98000" AND Zcheck$ <= "98015") THEN
      Info$ = Plist$
      PRINT LEFT$(Info$, 25)
          PRINT MID$(Info$, 26, 25)
      PRINT RIGHT$(Info$, 17)
      PRINT
   END IF
   GET #1
LOOP
CLOSE #1
```

File List Box Control

Action Displays a list of file names for a specified directory at run time.



Remarks

Events

Click	DblClick	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
LostFocus	MouseDown	MouseMove	MouseUp
PathChange	PatternChange		

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

Archive	BackColor	CtlName	DragMode
Enabled	FileName	ForeColor	Height
Hidden	Index	Left	List
ListCount	ListIndex	MousePointer	Normal
Parent	Path	Pattern	ReadOnly
System	TabIndex	TabStop	Tag
Top	Visible	Width	

A file list box allows the user to select files from an alphabetical list in a specified directory. This list includes all files in the current directory unless otherwise specified by the Path and Pattern properties.

A Click event generated for a file list box recognizes the following event ordering: MouseDown, MouseUp, Click, DblClick.

A DblClick event generated for a file list box recognizes the following event ordering: MouseDown, MouseUp, Click.

The Hidden, Archive, ReadOnly, System, and Normal properties determine the types of files that are displayed in the list.

Use a file list box when you want to create file-selection dialog boxes.

Use the List, ListCount, and ListIndex properties to access items that are displayed in the list box at run time.

The minimum size of file list box is 4 x 5; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value.

See Also Directory List Box Control, Drive List Box Control.

Example This example demonstrates how to update a label control to reflect the current path for a file list box control. Double-clicking a directory name displays a list of that directory's files in the file list box; it also displays the directory's complete path in the label control. To try this example, create a form containing a label, a directory list box, and a file list box.

```
SUB Dir1_Change ()
    File1.Path = Dir1.Path
END SUB

SUB File1_PathChange ()
    Label1.Caption = "Path: " + Dir1.Path
END SUB
```

FILEATTR Function

Action Returns the file handle or file mode of an open file or ISAM table (ISAM available in Professional Edition only).

Syntax **FILEATTR**(*filenumber%*, *attribute%*)

Part	Description
filenumber%	Number specifying an open file
attribute%	Numeric expression, either 1 or 2, that indicates the type of information to return:

Number	Returns
2	MS-DOS file handle if *filenumber%* indicates a file, or 0 if *filenumber%* indicates an ISAM table.
1	File mode: 1 = INPUT 2 = OUTPUT 4 = RANDOM 8 = APPEND 32 = BINARY 64 = ISAM

Remarks The argument *filenumber%* is the number used in the OPEN statement to open the file. You can use a numeric expression as long as Visual Basic evaluates it to the number of an open file or device.

See Also OPEN Statement (File I/O).

Example This example uses the FILEATTR function to get the handles and modes from two opened files using the PRINT statement to print out the MS-DOS file.

```
CLS                                  ' Clear the screen.
OPEN "tempfile.dat" FOR APPEND AS #1
OPEN "tempfl2.dat" FOR RANDOM AS #2
PRINT "Number Handle Mode"
PRINT TAB(2); 1; TAB(10); FILEATTR(1, 2); TAB(15); FILEATTR(1, 1)
PRINT TAB(2); 2; TAB(10); FILEATTR(2, 2); TAB(15); FILEATTR(2, 1)
END
```

FileName Property

Applies To File list box.

Action Sets or returns the specified file name from the file list box. You can also set the drive, path, or pattern using this property; not available at design time.

Syntax [*form.*][*filelistbox.*]**FileName**[= *filename$*]

Part	Description
filename$	Any valid file name or wildcard pattern; default ("") is a null string.

Remarks At run time, this property returns the currently selected file name from the list. The path is retrieved separately, using the Path property. This returned value is functionally equivalent to:

```
List(ListIndex)
```

When writing to this property:

- Including a drive, path, or pattern in the string changes the Drive, Path, and Pattern properties accordingly.
- Including the name of an existing file (without wildcard characters) in the string selects the file.

Changing the value of this property may also cause one or more of these events:

Action	Event
Change the path.	PathChange
Change the pattern.	PatternChange
Enter an existing file name.	DblClick

See Also Drive Property, Path Property, Pattern Property.

Example This example displays a message in a label when a file name in a file list box is double-clicked.

```
SUB Dir1_Change ()
    File1.Path = Dir1.Path          ' When Dir1 path changes, set File1 path.
END SUB

SUB File1_DblClick ()               ' Display the selected file name.
    Label1.Caption = "Your selection: " + UCASE$(File1.FileName)
END SUB

SUB File1_PathChange ()
    Dir1.Path = File1.Path           ' When File1 path changes, set Dir1 END
SUB                                  ' path.
```

FILES Statement

See CHDIR Property.

FIX, INT Functions

Action Truncates (removes the fractional part of) a floating-point expression to its integer portion (FIX), or returns the largest integer less than or equal to a numeric expression (INT).

Syntax

FIX(*numeric-expression*)
INT(*numeric-expression*)

Part	Description
numeric-expression	For FIX, any numeric expression; for INT, any real number in the range -32,768 to 32,767, inclusive.

Remarks If *numeric-expression* is outside the valid range for INT (-32,768 to 32,767, inclusive), Visual Basic generates the run-time error message, "Overflow."

FIX(x) is equivalent to SGN(x)*INT(ABS(x)).

For a negative *numeric-expression*, FIX returns the first negative integer greater than *numeric-expression*; INT returns the first negative integer less than *numeric-expression*.

See Also CINT Function.

Example This example compares output from INT, CINT, and FIX:

- INT rounds down for positive numbers, up for negative numbers.
- FIX always rounds down (towards 0).
- CINT rounds off (at .5, CINT rounds towards the nearest even integer).

```
CLS                                ' Clear the screen
PRINT " N", "INT(N)", "FIX(N)", "CINT(N)"
N! = 99.3
FOR count% = 1 TO 3
    PRINT N!, INT(N!), FIX(N!), CINT(N!)
    N! = N! + .2
NEXT
PRINT
N! = -99.7
FOR count% = 1 TO 3
    PRINT N!, INT(N!), FIX(N!), CINT(N!)
    N! = N! + .2
NEXT
```

FOR...NEXT Statement

Action Repeats a block of statements a specified number of times.

Syntax

FOR *counter* = *start* **TO** *end* [**STEP** *increment*]
[*statementblock*]
[**EXIT FOR**]
[*statementblock*]
NEXT [*counter* [, *counter*]...]

Part	Description
counter	Numeric variable used as the loop *counter*; cannot be an array or record element.
start	Initial value of *counter*.
end	Final value of *counter*.
STEP *increment*	Amount *counter* is changed each time through the loop; if STEP is omitted, *increment* defaults to one.
statementblock	One or more valid statements on one or more lines.
EXIT FOR	Alternative exit from FOR...NEXT loop; transfers control to the statement following the NEXT keyword.

Remarks If you omit the variable in a NEXT statement, NEXT matches the most recent FOR...NEXT statement. If a NEXT statement is encountered before its corresponding FOR statement, Visual Basic generates the error message, "NEXT without FOR."

EXIT FOR can be used only in a FOR...NEXT statement.

When used within nested FOR...NEXT statements, EXIT FOR transfers out of the immediately enclosing loop.

A FOR...NEXT loop executes only if the arguments *start* and *end* are consistent with *increment*. If *end* is greater than *start*, *increment* must be positive. If *end* is less than *start*, *increment* must be negative. Visual Basic checks this at run time by comparing the sign of (*end* - *start*) with the sign of STEP. If both have the same sign, and *end* does not equal *start*, the FOR...NEXT loop is entered. If not, the entire loop is skipped.

Within a FOR...NEXT loop, the program lines following the FOR statement are executed until the NEXT statement is encountered. Then *counter* is changed by the amount specified by STEP and compared with the final value, *end*.

If *start* and *end* have the same value, the loop executes once, regardless of the value of STEP. Otherwise, the STEP value controls the loop as follows:

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

Avoid changing the value of *counter* within the loop. Changing the loop *counter* can make the program more difficult to read and debug.

You can nest FOR...NEXT loops; that is, you can place a FOR...NEXT loop within another FOR...NEXT loop. To ensure that the nested loops work properly, give each loop 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 correct form is:

```
FOR I = 1 TO 10
    FOR J = 1 TO 10
FOR K = 1 TO 10
.
.
.
NEXT K
    NEXT J
NEXT I
```

A NEXT statement with the form:

```
NEXT K, J, I
```

is equivalent to the following sequence of statements:

```
NEXT K
NEXT J
NEXT I
```

See Also DO...LOOP Statement, WHILE...WEND Statement.

Example This example uses FOR...NEXT loops to print the first 11 columns of Pascal's Triangle, in which each number is the sum of the number immediately above it and the number immediately below it in the preceding column.

```
DEFINT A-Z
CLS                                    ' Clear the screen.
CONST MAXCOL = 11
```

```
DIM A(MAXCOL, MAXCOL)
PRINT "Pascal's Triangle"
FOR M = 1 TO MAXCOL
    A(M, 1) = 1: A(M, M) = 1          ' Top and bottom of each column is 1.
NEXT M
FOR M = 3 TO MAXCOL
    FOR N = 2 TO M - 1
        A(M, N) = A(M - 1, N - 1) + A(M - 1, N)
    NEXT N
NEXT M
Startrow = 13                         ' Go to the middle of the screen.
FOR M = 1 TO MAXCOL
    Col = 6 * M
    Row = StartRow
    FOR N = 1 TO M
        LOCATE Row, Col: PRINT A(M, N)
    Row = Row + 2                     ' Go down 2 rows to print next number.
    NEXT N
    PRINT
    StartRow = StartRow - 1           ' Next column starts 1 row above
NEXT M                                ' preceding column.
```

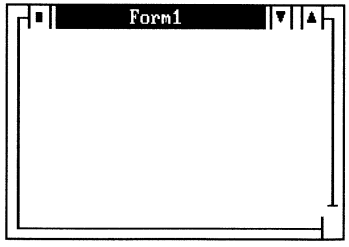
ForeColor Property

See BackColor Property.

Form

Action

Consists of a visual container for controls, such as a window or dialog box; provides a foundation for creating applications in Visual Basic.



Remarks

Events

Click	DblClick	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
Load	LostFocus	MouseDown	MouseMove
MouseUp	Paint	Resize	UnLoad

Methods

CLS	DRAG	HIDE	LOAD
MOVE	PRINT	PRINTFORM	REFRESH
SHOW	TEXTHEIGHT	TEXTWIDTH	UNLOAD

Properties

AutoRedraw	BackColor	BorderStyle	Caption
ControlBox	CurrentX	CurrentY	DragMode
Enabled	ForeColor	FormName	FormType
Height	Left	MaxButton	MinButton
MousePointer	ScaleHeight	ScaleWidth	Parent
Tag	Top	Width	WindowState
Visible			

Forms have an associated set of tools — properties, events, and methods — that you can use to define a form's appearance and response to different types of user input. You can uniquely determine the behavior and appearance of each individual form.

To create a form module, choose New Form from the File menu.

Form modules are private: Event and general procedures contain form modules that cannot be called from external modules or procedures. Form modules do not allow executable code at the module level.

Code modules are public: General procedures contain code modules that can be called from other modules or procedures. Executable code is allowed at the module level of code modules.

Multiple forms are often used to create an application's user interface.

You can place up to 254 controls on a form. Controls placed on a form are referred to as children, or child controls, of the form. A form cannot get the focus if any of its children are able to receive the focus.

The dimensions of a form correspond to the Height and Width property settings. Settings specified that are less than the minimum value are set to the minimum value.

A default-size form with no controls takes approximately 96 bytes of memory. You can use the FRE function to approximate the amount of memory taken up by your own forms. For example:

```
SUB Form_Click ()
     Loaded! = FRE(-1)
     UNLOAD Form1
     Unloaded! = FRE(-1)
     SHOW
     PRINT STR$(Unloaded! - Loaded!)
END SUB
```

See Also $FORM Metacommand.

$FORM Metacommand

Action Enables access to an external form's controls and properties; must be placed before any executable code.

Syntax {**REM** | '} **$FORM** *formname*

Part	Description
REM | '	REM or remark character ('); must precede all metacommands.
formname	Name of a form.

Remarks Each form module must contain the appropriate $FORM metacommands in order to share access to form properties. Use an include file to group $FORM metacommands.

See Also Form, $INCLUDE Metacommand, REM Statement, SHOW Method.

FORMAT$ Function

Action Converts a number to a string and formats it according to instructions contained in a format expression.

Syntax **FORMAT$**(*expression* [,*fmt$*])

Part	Description
expression	Numeric expression to format.
fmt$	String expression consisting of display-format characters that detail how the expression is displayed: Date and time formats Numeric expression formats

Remarks

If *fmt$* is omitted or is a null string, FORMAT$ provides the same functionality as the STR$ function — that is, it converts positive numbers to strings. However, numbers converted to strings using FORMAT$ do not have the leading space reserved to display the sign of the value, while numbers converted using STR$ retain the leading space. When no display formatting is required, use STR$ to convert numbers to strings.

A format expression (*fmt$*) can have up to three sections separated by semicolons (;). The first section formats positive values, the second section formats negative values, and the third section formats zeros:

Usage	Result
One section only.	The format applies to all numbers.
Two sections.	The first section applies to positive numbers and zeros; the second to negative numbers.
Three sections.	The first section applies to positive numbers, the second to negative numbers, and the third to zeros.

If you have semicolons with nothing between them, the missing section prevents that type of number from being displayed.

Visual Basic supports international formatting styles under MS-DOS version 3.31 or later.

Default settings for date and time formats are determined by the operating system.

See Also

PRINT Statement, STR$ Function.

Example

This example illustrates the use of the FORMAT$ function in formatting date/time values and numbers.

```
DIM message AS STRING
DIM newline AS STRING

newline$ = CHR$(13) + CHR$(10)          ' Define new line.
mess$ = "Today's date is " + FORMAT$(NOW, "ddddd") + "." + newline$
mess$ = mess$ + "The current time is " + FORMAT$(NOW, "ttttt")
mess$ = mess$ + "."
```

```
MSGBOX (mess$)                              ' Display date/time formatting.
 VALUE# = INT((4001) * RND - 2000)
 SELECT CASE SGN(VALUE#)                    ' Evaluate sign of random number.
     CASE -1: Desc$ = "negative"
     CASE 0: Desc$ = "zero"
     CASE 1: Desc$ = "positive"
 END SELECT
 mess$ = "The format expression determines how a number is displayed."
 mess$ = mess$ + "In this" + newline$ + "demonstration a random number"
 mess$ = mess$ + " between +2000 and -2000 is generated and"
 mess$ = mess$ + newline$ + "formatted. The number to be formatted is "
 mess$ = mess$ + Desc$ + "." + " Using the format" + newline$
 mess$ = mess$ + "expression provided, the number is displayed as: "
 mess$ = mess$ + FORMAT$(VALUE#, "#,##0.00;(#,##0.00)")
 MSGBOX (mess$)                             ' Display numeric formatting.
```

FormName Property

Applies To Form.

Action Name used to identify the form; not available at run time.

Remarks By default, Visual Basic defines the name of a form (FormName property) as "Form" plus a unique integer, starting at 1. Numbers assigned are cumulative — for example, Form1, Form2, and Form3. This yields a unique name for each form.

To change the default, specify a unique name for the form. FormName must start with a letter and can have a maximum of 40 characters, including alphanumeric or underscore (_) characters.

A form name cannot be a Visual Basic reserved word.

The FormName property is distinct from the Caption property. When you change the caption, the FormName property is not changed.

Using the FormName identifier with properties and methods is optional within a form's code section. Any reference to a form within its own module or file is implicit. For example, in the code module for Form1:

```
Form1.PRINT "Hello"
```

is equivalent to:

```
PRINT "Hello"
```

Visual Basic syntax requires the FormName identifier when referring to a form outside its own code section or a different module.

You cannot refer to the FormName property itself in code. For example:

```
FOO$ = Form1.FormName
```

causes an error. If you need to identify a form to pass to a procedure, use the Tag property.

See Also CtlName Property, Tag Property.

Example See CtlName Property example.

FormType Property

Applies To Form.

Action Determines whether a form is an MDI container or a standard form; not available at run time.

Value	Description
0	(Default) Standard.
1	MDI.

Remarks Only one form in an application can be a container form. If you try to set a second form's FormType property to MDI (1), Visual Basic generates the error message, "MDI container already exists." The previous container form must be set to a standard form (Normal = 0) before a new one can be set.

An MDI or container form functions like a standard form, with the following exceptions:

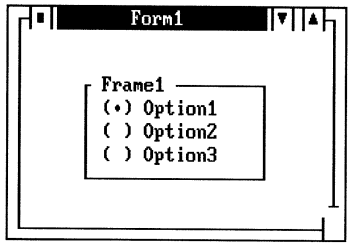
- The container form is always maximized. Move, Size, Minimize, and Restore are disabled when the form is visible.
- The container form can display a visible border style (BorderStyle <> 0). This allows display of a title bar and use of the Control-menu box to close the application. Note: If no border is specified (BorderStyle = 0), the client area will be the same as the screen size (minus the menu).
- The MinButton, MaxButton, WindowState, Height, Width, Top, and Left properties are read-only at design time and run time.
- The MOVE method is not supported at run time.
- The container form can only contain Menu controls. Child forms can contain menus, but they will be displayed on the container form.

All other forms in the project become children of the container form. The following rules apply:

- Loading a container does not automatically load its children; however, loading a child form automatically loads the container. Unloading the container unloads all children.
- Making the container invisible makes all children invisible. The visibility (Visible property) of a child form is independent of the container; however, the container must be visible before "visible" children can be seen. Note: Making the container visible only makes the container and any children previously visible appear.
- Disabling or enabling a child form is independent of the container form; however, the container must be enabled before children are truly enabled. Making the container form enabled only enables the container and any of its children that were previously enabled.

Frame Control

Action Appears as a rectangle; used to group option buttons or other controls.



Remarks

Events

DragDrop	DragOver

Methods

DRAG	MOVE	REFRESH

Properties

BackColor	Caption	CtlName	Enabled
ForeColor	Height	Index	Left
MousePointer	Parent	TabIndex	Tag
Top	Visible	Width	

A frame provides a visual and functional grouping for controls on a form. For example, you can separate one group of option buttons from another group to make a form more readable.

To group controls, draw the frame first, then draw controls inside the frame. This technique allows you to move the frame and the controls together as a single object.

The movement of child controls is always relative to the movement of the container or parent object.

To use a frame as a horizontal or vertical separator line, set its Height or Width property to 1.

Use the Caption property to assign a descriptive heading to a frame containing a group of controls.

You can place up to 254 controls in a frame.

The minimum size of a frame allowed is 1 x 1. Dimensions correspond to the Height and Width property settings; specified settings that are less than the minimum value are set to the minimum value. A Height setting of 1 changes the appearance of the control.

See Also Picture Box Control.

Example This example uses the frame control to group two option buttons. When you run the example and click the form, the selected option button will change. To try this example, create a form containing a frame and two option buttons. Make sure you create the frame control first. Set the frame control's Caption property to "Option Group."

```
SUB Form_Click ()
    IF Option1.Value = -1 THEN
         Option2.Value = -1
         Option1.Value = 0
    ELSE
         Option1.Value = -1
         Option2.Value = 0
    END IF
END SUB
```

FRE Function

Action Returns the amount (in bytes) of available or unused memory.

Syntax **FRE**(*numeric-expression%* | *stringexpression$*)

Part	Description
numeric-expression% *stringexpression$*	Values returned for different types of arguments:

Example	Return Value
FRE(a$)	Remaining space in a$'s segment (in bytes).
FRE("*string literal***")**	Remaining space for temporary strings (in bytes).
FRE(0)	Error message "Illegal function call."
FRE(-1)	Remaining space in far memory for data and code.
FRE(-2)	Remaining stack space (in bytes).
FRE(-3)	Remaining space in expanded memory (in K).
FRE(-4)	Remaining space within the data segment of the current form (as determined by ActiveForm).
FRE(*any other number***)**	Error message "Illegal function call."

Remarks

When a string expression is specified, the FRE function compacts the free string space into a single block, then returns the amount of available string space.

FRE(-2) returns meaningful values only when a program is executing. Values returned by FRE(-2) are not accurate when the function is called from the Immediate window, during program tracing, or when watching a variable. The unused stack space indicates how much stack is needed so that DGROUP can be reclaimed.

If expanded memory is not available, FRE(-3) returns the error message, "Feature unavailable."

If you are running a program with no forms loaded, FRE(-4) returns the error message "No current active form"

If you start Visual Basic (VBDOS.EXE) without the /E:n switch, you will be using all expanded memory available on your computer.

If you start Visual Basic (VBDOS.EXE) with the /E:n switch, you can limit the amount of expanded memory available to your program.

For example, if your computer has 5 MB of expanded memory but you want to use only 4MB of expanded memory, you can start Visual Basic with this:

```
VBDOS /E:4096
```

If you have specified 4MB of expanded memory as shown above, and then you use 2MB of expanded memory in your program, FRE(-3) returns 2048.

If you are running a compiled application using an earlier version of Basic expanded-memory overlays, the value returned by FRE(-3) is based on the total amount of expanded memory available. (When using overlays, you cannot limit the amount of expanded memory used by your program.)

You can use the FRE function to approximate the amount of memory taken up by a form or controls placed on your form. For example:

```
SUB Form_Click ()
     Loaded! = FRE(-1)
     UNLOAD Form1
     Unloaded! = FRE(-1)
     SHOW
     PRINT STR$(Unloaded! - Loaded!)
END SUB
```

You can write an event procedure similar to the one above to determine the amount of memory used by a given control before or after placing that control on a form. For example, if:

```
A = Size, in bytes, of default form with no controls
B = Size, in bytes, of form with one control
```

then:

```
B - A = Amount of memory, in bytes, taken up by the control
```

Example

This example uses the FRE function to report the availability of memory resources. The STACK function is used to show free DGROUP. The STACK statement is used to set the STACK size.

If Visual Basic is invoked with the /Ea switch, the integer arrays used in this example will be placed in available EMS memory space; otherwise, they will occupy far memory. If the arrays are placed in EMS, each array will reduce available EMS memory by 16K. If not in EMS, far memory will be reduced by the number of bytes occupied by the array rounded up to an even 16 bytes.

```
DECLARE SUB MakeSubString ()
DECLARE SUB Printit (a$)
DECLARE SUB Recursive (n%)
DECLARE SUB Pause ()
CLS                                 ' Clear the screen.
STACK 4096                          ' Set Stack size to 4k.
```

```
' Report the way things are to begin with.
PRINT "Remaining space (in bytes) to begin with is:"
CALL Printit(a$)
String1$ = STRING$(32767, 90)      ' Create a 32K string and give report.
PRINT "The remaining space after creating a 32K far string is:"
CALL Printit(String1$)

MakeSubString                      ' Make a substring and give report.
PRINT "The remaing space after returning from a sub is:"
CALL Printit(String2$)

n% = 60                            ' Do 60 recursive calls to a SUB.
CALL Recursive(n%)
PRINT "After performing 60 recursive calls, the space is:"
CALL Printit(a$)

REDIM a%(999)                      ' Dimension a 1001 element dynamic array.
                                   ' in EMS
PRINT "After dimensioning a 1000-element integer array, the space is:"
CALL Printit(a$)

PRINT "After dimensioning a 2nd 1000-element integer array, the space is:"
REDIM B%(999)
CALL Printit(a$)

SUB MakeSubString ()
    SHARED String1$
    String2$ = STRING$(32767, 90)
    PRINT "The space remaining after creating a 32K sub string is:"
    CALL Printit(String1$)
END SUB

SUB Pause ()
    PRINT "Press any key for next calculation"
    WHILE INKEY$ = "": WEND
END SUB

SUB Printit (a$)
    PRINT "Dgroup + Current Stack"; TAB(30); STACK
    ' STACK function returns what the largest stack that can be
    ' be dimensioned.  This is free DGROUP plus current stack.
    PRINT "Far Memory"; TAB(30); FRE(-1)
    PRINT "STACK 'low water' mark"; TAB(30); FRE(-2)
    ' FRE(-2) reports what the lowest amount of free stack space
    ' your program has had prior to executing the FRE(-2) function.

    PRINT "String Segment"; TAB(30); FRE(a$)
    PRINT "Temp. Segment"; TAB(30); FRE("")
```

```
    PRINT "EMS Memory"; TAB(30); FRE(-3); "K"
      PRINT
    CALL Pause
    PRINT
END SUB

SUB Recursive (n%)
    SHARED String1$
    n% = n% - 1
    IF n% = 0 THEN
        PRINT "The remaining space after 50 recursive calls is:"
        CALL Printit(String1$)
        EXIT SUB
    ELSE
        CALL Recursive(n%)
    END IF
END SUB
```

FREEFILE Function

Action Returns the next valid unused file number; use to supply a file number when you want to ensure that the number isn't already assigned.

Syntax **FREEFILE**

See Also OPEN Statement (File I/O).

Example See CLOSE Statement example.

FUNCTION Statement

Action Declares the name, arguments, code, and return type of a FUNCTION procedure.

Syntax **[STATIC] FUNCTION** *globalname* ([**[BYVAL]** *variable*[( )] **[AS** *type*] [, **[BYVAL]** *variable*[( )] **[AS** *type*]]...]) **[AS** *type*]
[statementblock]
globalname = *expression*
[EXIT FUNCTION]
[statementblock]
globalname = *expression*
END FUNCTION

Part	Description
STATIC	Indicates that the procedure's local variables are preserved between calls; does not affect variables declared outside the procedure, even if they are used in the FUNCTION procedure.
globalname	Name of the function; procedure names follow the same rules for naming Visual Basic variables; can end with a type-declaration character (%, &, !, #, @, or $).
BYVAL	Indicates the parameter is passed by value rather than by reference; cannot be used with a variable of a user-defined type.
variable	Name of variable to pass as an argument; for array variables, empty parentheses are required.
AS *type*	Keyword used when declaring the data type of the variable and the data type of the value returned by FUNCTION; may be INTEGER, LONG, SINGLE, DOUBLE, CURRENCY, or STRING.
statementblock	Any group of statements that executes within the body of the FUNCTION procedure.
expression	Return value of the function; a FUNCTION procedure returns a value by assigning that value to the function name.
EXIT FUNCTION	Causes an immediate exit from a FUNCTION procedure; execution continues with the statement after the statement that called the FUNCTION; a FUNCTION may have multiple EXIT FUNCTION statements.

Remarks

FUNCTION and END FUNCTION mark the beginning and end of a FUNCTION procedure. You can use the optional EXIT FUNCTION statement to exit a FUNCTION from any point within the procedure.

STATIC indicates that the value of a FUNCTION's local variables are saved between calls. Without STATIC, the local variables are allocated each time the FUNCTION is invoked, and the variable values are lost when the FUNCTION returns to the calling program.

Using the STATIC attribute increases execution speed slightly.

The STATIC attribute does not affect variables that are used in a FUNCTION but declared outside the FUNCTION in DIM or COMMON statements using the SHARED statement.

The FUNCTION name cannot appear in any other SUB or FUNCTION statement in the same application unless the FUNCTION is in form code. Note: Code within a form module is known only to the procedures in that module.

The name of the FUNCTION determines the data type the FUNCTION returns. For example, to create a FUNCTION procedure that returns a string, you include a dollar sign ($) in the name, or give it a name defined as a string name with a DEFSTR statement. Alternatively, you can use an AS clause following *parameterlist* to identify the data type.

BYVAL can be used only with simple numeric types (INTEGER, LONG, SINGLE, DOUBLE, or CURRENCY). The additional types, FORM and CONTROL, can only be used when a form or control is passed as an argument to a SUB or FUNCTION procedure.

Unless identified with the BYVAL keyword, individual arguments are passed by reference, so any change to an argument value inside the FUNCTION procedure changes its value in the calling procedure. If an argument passed to a function is an *expression*, it is treated the same as if it used the BYVAL keyword — that is, no part of the *expression* is altered by the function.

If a parameter is passed by reference, any change to the parameter value inside the procedure changes its value in the calling program.

If no value is assigned to the FUNCTION name, the procedure returns a default value: A numeric function returns zero; a string function returns the null string ("").

A FUNCTION can be recursive — that is, it can call itself to perform a given task. The STATIC keyword is usually not used with recursive FUNCTION procedures.

Any FUNCTION procedure variables or arrays are considered local to that FUNCTION procedure, unless they are explicitly declared as shared variables in a SHARED statement.

Avoid using I/O statements in a FUNCTION procedure called from an I/O statement; they can cause unpredictable results.

Because Visual Basic may rearrange arithmetic expressions to attain greater efficiency, avoid using FUNCTION procedures that change program variables in arithmetic expressions.

A FUNCTION procedure is a separate procedure, like a SUB procedure. It can accept arguments, perform a series of statements, and change the values of its arguments. However, unlike a SUB procedure, a FUNCTION procedure is used in an *expression* in the same manner as any Visual Basic intrinsic function (such as SQR, COS, or CHR$).

You cannot use an array of fixed-length strings as an argument when calling a FUNCTION procedure, because Visual Basic converts a simple fixed-length string argument to a variable-length string argument before passing the string to a FUNCTION procedure.

See Also CALL Statement (Basic Procedures), DECLARE Statement, SUB Statement.

Example See CALL Statement (Basic Procedures) example

GET, PUT Statements (File I/O)

Action

Reads from a file into a random-access buffer or variable (GET); writes a variable or random-access buffer to a file (PUT).

Syntax

GET [#]*filenumber%*[,[*recordnumber&*][,*variable*]]
PUT [#]*filenumber%*[,[*recordnumber&*][,*variable*]]

Part	Description
filenumber%	Number specified in the OPEN statement to open a file.
recordnumber&	For random-access files, the number of the record to read or write; for binary-mode files, the byte position where reading or writing starts.
variable	For GET, a variable used to receive input from the file; for PUT, a variable that contains output to write to the file; usually a variable of a user-defined data type.

Remarks

Do not use GET or PUT on ISAM files.

If you omit *recordnumber&*, the next record or byte (the one after the last GET or PUT, or the one pointed to by the last SEEK) is read into the buffer (GET) or written to (PUT). The largest possible record number is 2,147,483,647 (equivalent to 2^31 -1).

The first record or byte position in a file is 1.

A record cannot be longer than 32,767 bytes.

If you specify a variable, you do not need either:

CVI, CVL, CVS, CVD, or CVC to convert record fields to numbers

- MKI$, MKL$, MKS$, MKD$, or MKC$ to convert numeric fields before writing

You cannot use a FIELD statement with the file if you use the *variable* argument.

For random-access files, you can use any variable as long as the length of the variable is less than or equal to the length of the record. Usually, a record variable defined to match the fields in a data record is used.

For binary-mode files, you can use any variable. The GET statement reads as many bytes as there are in the variable; the PUT statement writes as many bytes to the file as there are bytes in the variable.

When you use a variable-length string variable, the statement reads (GET) or writes (PUT) as many bytes as there are characters in the string's value. For example, the following two statements use GET to read 10 bytes from file number 1:

```
VarStrings$=STRING$ (10, " ")
GET #1,,VarString$
```

The following two statements use PUT to write 15 bytes to file number 1:

```
VarString$=STRING$(15, "X")
PUT #1,,VarString$
```

You can omit the arguments *recordnumber&*, *variable*, or both. If you omit the *recordnumber&* but include *variable*, you must still include the commas:

```
GET #4,,FileBuffer
PUT #4,,FileBuffer
```

If you omit both arguments, do not include the commas:

```
GET #4
PUT #4
```

GET and PUT statements allow fixed-length input and output for Visual Basic communications files. Use GET carefully, because if there is a communications failure, GET waits indefinitely for *recordnumber&* characters.

When you use GET with the FIELD statement, you can use INPUT # or LINE INPUT # after GET to read characters from the random-access file buffer. You can use the EOF function after a GET statement to see if GET went beyond the end of the file.

When using a file buffer defined by a FIELD statement, LSET, RSET, PRINT #, and WRITE # can be used to put characters in the random-file buffer before executing a PUT statement.

In the case of WRITE #, Visual Basic pads the buffer with spaces up to the carriage return. If you attempt to read or write past the end of the buffer, Visual Basic generates the error message, "FIELD overflow."

See Also GET Statement (Graphics), PUT Statement (Graphics).

Example This example uses the GET statement to display the contents of a newly created random-access file that contains student names and corresponding test scores. The PUT statement is used to store the values in the file.

```
TYPE TestRecord
    Student AS STRING * 20
    Score AS SINGLE
END TYPE

DIM MyClass AS TestRecord

OPEN "FINAL.DAT" FOR RANDOM AS #1 LEN = LEN(MyClass)
MyClass.Student = "MarySa"
MyClass.Score = 99
PUT #1, 1, MyClass
CLOSE #1
```

```
OPEN "FINAL.DAT" FOR RANDOM AS #1 LEN = LEN(MyClass)
GET #1, 1, MyClass
PRINT "STUDENT:", MyClass.Student
PRINT "SCORE:", MyClass.Score
CLOSE #1
KILL "FINAL.DAT"
```

GET, PUT Statements (Graphics)

Action

Captures a graphics screen image (GET); displays an image captured by GET (PUT); not available when forms are showing.

Syntax

GET [**STEP**](*x1!*,*y1!*)-[**STEP**](*x2!*,*y2!*), *arrayname*[(*index%*)]
PUT [**STEP**](*x1!*,*y1!*), *arrayname*[(*index%*)] [,*actionverb*]

Part	Description
STEP	Specifies that coordinates are relative to the current graphics cursor position.
(*x1!*,*y1!*)	Upper-left coordinates of the image captured by GET, or of the screen location where PUT displays the image .
(*x2!*,*y2!*)	Lower-right coordinates of the captured image.
arrayname	Name of the array that stores the image.
index%	Numeric constants or variables indicating the element of the array where the saved image starts.
actionverb	Indicates how the image is displayed, using one of the following keywords:

Keyword	Description
AND	Transfers over existing image.
OR	Superimposes image on existing image.
PRESET	Transfers negative image point-by-point.
PSET	Transfers point-by-point.
XOR	(Default) Inverts the points.

Remarks

You must be in a graphics screen mode (for example, screen mode 1) to use the GET or PUT statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

Unless the array type is integer or long, the contents of an array after a GET operation appear meaningless when inspected directly. Examining or manipulating noninteger arrays that contain graphics images can cause run-time errors.

The STEP keyword makes the specified coordinates relative to the most recent point. For example, if the last point plotted were (10,10), then the actual coordinates referred to by STEP (5,10) would be (5+10,10+10) or (15,20). If the second coordinate pair in a GET statement has a STEP argument, it is relative to the first coordinate pair.

GET (*x1!*,*y1!*)-(*x2!*,*y2!*) specifies the screen coordinates of opposite corners of the rectangular image to store.

PUT (*x1!*,*y1!*) specifies the coordinates for the upper-left corner of the rectangle enclosing the image to be placed in the current output window. The entire rectangle must be within the bounds of the current viewport.

The argument *arrayname* can be of any numeric type; its dimensions must be large enough to hold the entire image.

The GET statement transfers a screen image into the array specified by *arrayname*. The PUT statement, associated with GET, transfers the image stored in the array onto the screen.

The following formula gives the required size of the array in bytes:

```
4+INT(((x2-x1+1)*(bits-per-pixel-per-plane)+7)/8)*planes*((y2-y1)+1)
```

The bits-per-pixel-per-plane and planes values depend on the screen mode set by the SCREEN statement. The following table shows the number of bits-per-pixel-per-plane ("Bits" below) and the number of planes ("Planes" below) for each screen mode:

Screen Mode	Bits	Planes
1	2	1
2	1	1
7	1	4
8	1	4
9 (> 64K EGA memory)	1	4
9 (64K EGA memory)	1	2
10	1	2
11	1	1
13	8	1

The bytes per element of an array are:

Bytes per Element	Array Type
2	Integer
4	Long-integer
4	Single-precision
8	Double-precision
8	Currency

For example, suppose you want to use the GET statement to store an image in high resolution (SCREEN 2). If the coordinates of the upper-left corner of the image are (0,0), and the coordinates of the lower-right corner are (32,32), then the required size of the array is:

```
4 + INT((33 * 1 + 7)/8) * 1 * (33)  ' or 169 bytes
```

This means that an integer array with 85 elements would be large enough to hold the image.

GET and PUT statements operating on the same image should be executed in matching screen modes. This can be either the same screen mode or any screen modes with the same values for planes and bits-per-pixel-per-plane.

GET and PUT arguments can be used to display a stored image with special effects such as animation.

The argument *actionverb* determines the interaction between the stored image and the one already displayed on screen:

Keyword	Description
XOR	(Default) Inverts the points on the screen where a point exists in the array image: When an image is placed on the screen against a complex background twice, the background is restored; this behavior is exactly like that of the cursor. You can move an object around the screen without erasing the background, thus creating animation effects.
PSET	Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen with GET.
PRESET	The same as PSET, except that a negative image is produced — for example, black on white.

Keyword	Description
AND	Transfers the image only if an image already exists under the transferred image: The resulting image is the result of a logical AND operation on the stored image and the screen. Points that have the same color in both the existing image and the stored image remain the same color. Points that do not have the same color in both the existing image and the stored image do not remain the same color.
OR	Superimposes the image onto an existing image: The resulting image is the product of a logical OR operation of the stored image and the screen image. The stored image does not erase the previous screen contents.

See Also GET Statement (File I/O), PUT Statement (File I/O).

Example This example uses GET to put a spaceship graphic into an array. The PUT statement is then used to display the ship array in different locations depending on user input.

```
ON ERROR RESUME NEXT

CONST g = .01                    ' Gravity.
CONST vThrust = .005             ' Horizontal thrust.
CONST hThrust = .01              ' Vertical thrust.

SCREEN 2
CLS

' Calculate size of array for ship array.
arraybytes = 4 + INT(((60 - 40 + 1) * 1 + 7) / 8) * 1 * ((60 - 50) + 1)
DIM ship(arraybytes / 2) AS INTEGER

LINE (50, 50)-(40, 60)                ' Draw ship.
LINE (40, 60)-(60, 60)
LINE (60, 60)-(50, 50)
GET (40, 50)-(60, 60), ship           ' Get ship image into array.
CLS

X = 310: Y = 10                       ' Set initial conditions.
vx = 0: vy = 0
ax = 0: ay = g
```

```
PRINT " i"                              ' Print help keys.
PRINT "j k"
PRINT " m"

LOCATE 25, 1                            ' Print initial conditions.
PRINT "ax="; USING "+#.###    "; ax;
PRINT "ay="; USING "+#.###    "; ay;
LINE (0, 190)-(639, 190)
PUT (X, Y), ship, PSET                  ' Draw ship.

DO
    vx = vx + ax: vy = vy + ay          ' Calculate velocity.
    newx = X + vx: newy = Y + vy        ' Calculate new x and y position.
    PUT (X, Y), ship, XOR               ' Erase ship at current position.
    PUT (newx, newy), ship, PSET        ' Draw ship at new location.
    X = newx: Y = newy

    a$ = INKEY$                         ' Check keys.
    SELECT CASE LCASE$(a$)
    CASE "m", "i", "j", "k":
        SELECT CASE a$
        CASE "m":                       ' Down key.
            ay = ay + vThrust
        CASE "i":                       ' Up key.
            ay = ay - vThrust
        CASE "j":                       ' Left key.
            ax = ax - hThrust
        CASE "k":                       ' Right key.
            ax = ax + hThrust
        END SELECT

    LOCATE 25, 1                            ' Print accelerations.
    PRINT "ax="; USING "+#.###    "; ax;
    PRINT "ay="; USING "+#.###    "; ay;
    END SELECT

    LOCATE 25, 40                           ' Print velocities.
    PRINT "vx="; USING "+##.###    "; vx;
    PRINT "vy="; USING "+#.###"; vy;
LOOP UNTIL Y > 180 OR ERR <> 0              ' Loop until landed or crashed.

IF ERR <> 0 OR vy > .2 THEN                 ' Crashed if velocity is greater
    CLS                                     ' than .2 or if ship went off
    LOCATE 12, 33: PRINT "You crashed!"     ' screen.
ELSE
    LOCATE 12, 33: PRINT "You made it!"
END IF
```

GETTEXT Method

Applies To CLIPBOARD object

Action Returns the text string found in the CLIPBOARD object.

Syntax **CLIPBOARD.GETTEXT()**

Remarks Use the CLIPBOARD object and applicable methods to provide cut, copy, and paste commands in your applications.

If the Clipboard is empty, a null string ("") is returned.

See Also CLEAR Method, SETTEXT Method.

Example See CLEAR Method example.

GOSUB...RETURN Statement

Action Branches to and returns from a subroutine. For a more structured alternative to GOSUB...RETURN subroutines, use SUB and FUNCTION statements.

Syntax **GOSUB** *line-label-to*

.
.
.

RETURN [*line-label-resume*]

Part	Description
line-label-to	Line number or label of the first line of the subroutine
line-label-resume	Line number or label where the subroutine returns; if omitted, execution continues at the statement following the GOSUB keyword

Remarks The following discussion of subroutines applies only to the targets of GOSUB statements, not to SUB procedures delimited by SUB statements. In most cases, use the CALL statement to enter and exit subroutines.

In addition to RETURN with no argument, Visual Basic supports RETURN with a line label or line number. This support allows a return from a subroutine to the statement having the specified line label or number, instead of a return to the statement after the GOSUB statement. Use this line-specific method with care, since it is often difficult to read and debug.

You can call a subroutine any number of times in a program. You also can call a subroutine from within another subroutine. How deeply you can nest subroutines is limited only by the available stack space. Note: You can increase the stack space with the CLEAR statement.

Recursive subroutines — that is, subroutines that call themselves — can easily run out of stack space.

RETURN with a line label or line number can return control to a statement only in the module-level code, not in procedure-level code.

A subroutine can contain more than one RETURN statement. A simple RETURN statement (without the *line-label-resume* option) in a subroutine makes Visual Basic branch back to the statement following the most recent GOSUB statement.

Subroutines can appear anywhere in the program, but it is good programming practice to make them readily distinguishable from the main program. To prevent inadvertent entry into a subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

See Also CALL Statement (Basic Procedures), FUNCTION Statement, SUB Statement.

GotFocus Event

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar

Action Occurs when an object receives the focus, either by user action — for example, when the user clicks or tabs to the object — or by changing the focus through the SETFOCUS method. Note: A form receives the focus only when it has no controls, or when all visible controls are disabled or invisible.

Syntax

SUB Form_GotFocus ()
SUB *ctlname*_**GotFocus ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array

Remarks Typically, you use a GotFocus event procedure to prescribe actions to occur when a control or form first receives the focus. For example, by attaching a GotFocus event procedure to each control on a form, you can guide the user by displaying brief instructions on the screen or status bar messages. You can also provide visual cues by enabling, disabling, or displaying other controls that depend on the control that has the focus.

An object can receive the focus only if its Enabled and Visible properties are set to True (-1).

To customize the keyboard interface so that the focus can be moved from one control to another, set the tab order or specify access keys for controls on a form.

A control that has the focus cannot receive a second GotFocus event even if the SETFOCUS method is used.

See Also ActiveControl Property, ActiveForm Property, Enabled Property, LostFocus Event, SETFOCUS Method, TabIndex Property, TabStop Property, Visible Property.

Example This example displays a status bar message when a button in an option group gets the focus. To try this example, create a form containing two option buttons and a label. Set the CtlName property for both option buttons to "OptionGroup."

```
SUB OptionGroup_GotFocus (Index AS INTEGER)
    SELECT CASE Index
         CASE 0
              Label1.Caption = "Option 1 has the focus."
         CASE 1
              Label1.Caption = "Option 2 has the focus."
    END SELECT
END SUB

SUB OptionGroup_LostFocus (Index AS INTEGER)
   Label1.Caption = ""
END SUB
```

GOTO Statement

Action Branches unconditionally to the line label or line number specified. A program with many GOTO statements can be difficult to read and debug; use structured control statements instead of GOTO statements.

Syntax **GOTO** *label-line*

Part	Description
label-line	Line label or line number to execute next.

Remarks The GOTO statement provides a way to branch unconditionally to another line (linelabel or linenumber). A GOTO statement can branch only to another statement at the same level of a program.

You cannot use GOTO to enter or exit a SUB, FUNCTION, or multiline DEF FN function. You can, however, use GOTO to control program flow within any of these program structures.

GOTO is also used as a keyword in the ON ERROR statement.

See Also DO...LOOP Statement, FUNCTION Statement, IF...END IF Statement, ON ERROR Statement, SELECT CASE Statement, SUB Statement.

Height, Width Properties

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, SCREEN object, text box, vertical scroll bar.

Action Sets or returns the dimensions of an object:

Object	Returns
Control	External height and width; combo box and drive list box values are read-only at design time.
Form	External height and width, including border and title bar.
SCREEN	Height and width of the screen (in characters); not available at design time and read-only at run time.

Use WIDTH statement for non-form functionality.

Syntax {[*form.*][*ctlname.*] | **SCREEN.**}**Height**[= *height%*]
{[*form.*][*ctlname.*] | **SCREEN.**}**Width**[= *width%*]

Part	Description
height%	Valid settings are:

Object	Range	Description
Form	MinHeight to ScreenHeight	MinHeight = 4 + Menubar (1-on, 0-off). ScreenHeight = 25, 43, or 50.
Control	1 to 254	Absolute maximum is 254 characters.
SCREEN	25, 43, or 50	Dependent on display characteristics of current screen mode.

Part	Description
width%	Valid settings are:

Object	Range	Description
Form	MinWidth to ScreenWidth	MinWidth = MinButton (3-on, 0-off) + MaxButton (3-on, 0-off) + ControlBox (3-on, 0-off). ScreenWidth = 80.
Control	1 to 254	Absolute maximum is 254 characters.
SCREEN	80	Dependent on display characteristics of current screen mode.

Remarks

Height and width values are always expressed in terms of characters.

For a or control, the values for these properties change as the object is sized by the user or by code.

A Height or Width setting less than 0 is an error. If Height or Width is changed to less than MinHeight or MinWidth, but greater than 0, the change is accepted, but ignored. For example, setting the Height of a resizable form to 1 sets Height to 4 (MinHeight); subsequent reads of the Height property will list 4.

Changing the number of displayable lines on screen may cause a form to be resized. For example, when all forms are invisible or hidden, the Height property of a form with Height greater than NewScreenHeight will change to the new ScreenHeight value when the form is made visible again.

By default, the values of the Height and Width properties at run time are equivalent to the height and width of the object at design time.

For forms and the SCREEN object, maximum Height and Width properties depend on the screen display characteristics of your computer.

The height and width of a control can range from 1 to an absolute maximum of 254. If moving a control causes Height or Width to exceed 254, the move attempt is ignored. If resizing a control causes Width to exceed 254, the maximum relative value within the 254 absolute range is used.

Use Height and Width and their related properties, Left and Top, for operations or calculations based on an object's total area, such as sizing or moving the object.

Use the ScaleHeight and ScaleWidth properties for operations or calculations based only on an object's internal area, such as printing or moving objects within a form or picture box.

See Also WIDTH Statement.

Example This example sets the size of a form to 75% of screen size, and centers the form when it is clicked.

```
SUB Form_Click ()
    Width = (SCREEN.Width * .75)          ' Set width of form.
    Height = (SCREEN.Height * .75)        ' Set height of form.
    Left = ((SCREEN.Width - Width) / 2)   ' Center form horizontally.
    Top = ((SCREEN.Height - Height) / 2)  ' Center form vertically.
END SUB
```

HEX$, OCT$ Functions

Action Returns a hexadecimal string representation of a number (HEX$); returns an octal string representation of a number (OCT$).

Syntax **HEX$**(*numeric-expression&*)
OCT$(*numeric-expression&*)

Part	Description
numeric-expression&	Any numeric expression.

Remarks HEX$ requires a numeric expression with a decimal value, while OCT$ converts any numeric expression.

Before a HEX$ or OCT$ function evaluates the numeric expression, it is rounded to an integer or long integer.

Example This example uses the HEX$ and OCT$ functions to display the hexadecimal and octal values of a number you supply.

```
CLS
DO                                  ' Clear the screen.
     LINE INPUT "Enter Decimal Number: ", Num$
     Decimal& = VAL(Num$)
     PRINT
```

```
        PRINT "Decimal", " ="; Decimal&
        PRINT "Hex", " = "; HEX$(Decimal&)
        PRINT "Oct", " = "; OCT$(Decimal&)
        PRINT
    LOOP UNTIL Num$ = ""
    END
```

Hidden Property

See Archive Property.

HIDE Method

Applies To Form, SCREEN object.

Action Removes a form from view by setting its Visible property to False (0).

Syntax [*form.*][**SCREEN.**]**HIDE**

Remarks The HIDE method has the same effect as setting a form's Visible property to False (0).

Use the SCREEN keyword with HIDE to set the Visible property of all visible forms to False (0). The SHOW method can then be used to display all hidden forms. For example:

```
SCREEN.HIDE
SCREEN.SHOW
```

You must hide all forms before using graphics statements such as CIRCLE, DRAW, and so on.

A hidden form's controls are not accessible to the user, but they are available to the running Visual Basic program, and to timer events.

When a form is hidden, control does not return to the user until all code in the event procedure that caused the form to be hidden has finished executing.

Use the HIDE method on an unloaded form to load the form without showing

When a form is hidden, only the displayed component is removed from memory. The code associated with the form module remains in memory.

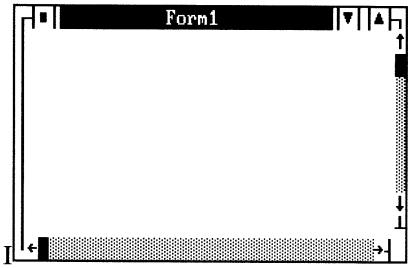
See Also SHOW Method, UNLOAD Statement, Visible Property.

Example The example uses the HIDE method to hide a form and the SHOW method to show the form.

```
DECLARE SUB HideShowDemo ()

SUB Form_Click ()
   CALL HideShowDemo
END SUB

SUB HideShowDemo ()
  HIDE                    ' Hide form.
  Msg$ = "Choose OK to make the form reappear."
  MSGBOX Msg$             ' Display message.
  SHOW                    ' Show form again.
END SUB
```

Horizontal and Vertical Scroll Bar Controls

Action Appear as scroll bars along a horizontal (left to right) or vertical (top to bottom) axis; provide a way of getting and displaying output.



Remarks

Events

Change	DragDrop	DragOver	GotFocus
KeyDown	KeyPress	KeyUp	LostFocus

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

Attached	CtlName	DragMode	Enabled
Height	Index	LargeChange	Left
Max	Min	MousePointer	Parent
SmallChange	TabIndex	TabStop	Tag
Top	Value	Visible	Width

Scroll bars are tools for navigating quickly through a long list of items or a large amount of information, or for indicating the current position on a scale.

Scroll bars can be used as input devices or as indicators of speed or quantity — for example, to control the volume of a computer game or to view the time elapsed in a timed process.

Use the Attached property to determine if a scroll bar is attached to a form.

When you're using a scroll bar as an indicator of speed or quantity or as an input device, use the Max and Min properties to set the range appropriate for the use of the control.

To specify the amount of change to report in a scroll bar, use the LargeChange property for clicking in the bar, and the SmallChange property for clicking the arrow at either end of the bar. The scroll bar's Value property increases or decreases by the amounts set for the LargeChange and SmallChange properties.

You can position the scroll bar at run time by setting the Value property between 0 and 32,767, inclusive.

Horizontal and vertical scroll bars differ only in their orientation. Dimensions correspond to the Height and Width property settings.

The minimum sizes for scroll bar controls are 1 x 2 (horizontal) and 2 x 1 (vertical). Specified settings that are less than the minimum value allowed are set to the minimum value.

See Also ScrollBars Property.

Example See Attached Property example.

HOUR Function

Action Evaluates a date/time serial number and returns the hour, a value between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive.

Syntax **HOUR**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example

This example calculates the time between now and midnight tonight. The time information is displayed using the TIMEVALUE, TIMESERIAL, HOUR, MINUTE, and SECOND functions.

```
CLS                                  ' Clear the screen.

' Get a time value for midnight tonight - 1 second.
Midnight = TIMEVALUE("23:59:59")

' Get a time value for right now.
Instant = NOW

' Get the difference in hours, minutes, seconds.
HourDiff% = HOUR(Midnight) - HOUR(Instant)
MinuteDiff% = MINUTE(Midnight) - MINUTE(Instant)

' Add in the odd second between 23:59:59 and midnight.
SecondDiff% = SECOND(Midnight) - SECOND(Instant) + 1

' Adjust so that seconds and minutes are less than 60.
IF SecondDiff% = 60 THEN
   MinuteDiff% = MinuteDiff% + 1
   SecondDiff% = 0
END IF
IF MinuteDiff% = 60 THEN
   HourDiff% = HourDiff% + 1
   MinuteDiff% = 0
END IF

' Calculate a total of seconds between now and midnight.
TotalMinDiff = (HourDiff% * 60) + MinuteDiff%
TotalSecDiff = (TotalMinDiff * 60) + SecondDiff%

' Put the difference back into a standard time format.
TotalDiff = TIMESERIAL(HourDiff%, MinuteDiff%, SecondDiff%)

' Display results.
CLS
PRINT "There are a total of"; TotalSecDiff; "seconds until midnight."
PRINT "That translates to"; HourDiff%; "hours,"; MinuteDiff%;
PRINT "minutes, and"; SecondDiff%; "seconds."
PRINT
PRINT "The difference can also be expressed in standard time notation as:"
PRINT FORMAT$(TotalDiff, "hh:mm:ss")
```

IF TYPEOF Statement

Action Allows conditional execution based on the evaluation of a Boolean expression and processing specific to a particular control type.

Syntax

IF {*condition* | **TYPEOF** *ctlname* **IS** *controltype*} **THEN**
[*statementblock-1*]
[**ELSEIF TYPEOF** *ctlname* **IS** *controltype* **THEN**
[*statementblock-2*]]...
[**ELSE**
[*statementblock-n*]]
END IF

Part	Description
condition	Any expression that can be evaluated as True (nonzero) or False (zero).
ctlname	Any valid control name.
controltype	One of the following valid control types:

```
CheckBox        ComboBox        CommandButton
Custom          DirListBox      DriveListBox
FileListBox     Frame           HScrollBar
OptionButton    Label           ListBox
Menu            PictureBox      TextBox
Timer           VScrollBar
```

statementblock	Any number of statements on any number of lines, separated by colons (:).

Remarks The IF...END IF statement provides the same type of conditional processing without the TYPEOF option.

The ELSE and ELSEIF blocks are optional. You can have as many ELSEIF clauses as you want in a block IF. Any of the statement blocks can contain nested block IF statements.

See Also IF...END IF Statement, SELECT CASE Statement.

Example See DragDrop Event example.

IF...END IF Statement

Action Allows conditional execution based upon the evaluation of a boolean expression.

Syntax

Syntax 1:

Block form:

IF *condition-1* **THEN**
    [*statementblock-1*]
[**ELSEIF** *condition-2* **THEN**
    [*statementblock-2*]]...
[**ELSE**
    [*statementblock-n*]]
END IF

Syntax 2:

Single-Line Form:

IF *condition* **THEN** *statements* [**ELSE** *statements*]

Part	Description
condition	Any expression that can be evaluated as True (nonzero) or False (zero).
statementblock	Any number of statements on any number of lines, separated by colons (:).

Remarks Use the object form of the IF statement to set up processing tasks specific to a particular control type.

The single-line form of the IF statement is best for short, straightforward tests where only one action is taken.

The block form of this statement provides several advantages:

- Adds structure and flexibility by allowing conditional branches across several lines.
- Tests more complex conditions.
- Allows you to use longer statements and structures.
- Allows program structure to be guided by logic rather than by the number of statements that fit on a line.
- Makes programs easier to read, maintain, and debug.

In executing a block-form IF, Visual Basic tests the first Boolean expression (condition-1):

- If the Boolean expression is True (nonzero), the statements following THEN are executed.
- If the first Boolean expression is False (zero), Visual Basic begins evaluating each ELSEIF *condition* in turn.

When Visual Basic finds a true *condition*, the statements following the associated THEN are executed. If none of the ELSEIF conditions is true, the statements following the ELSE are executed. After the statements following a THEN or ELSE are executed, the program continues with the statement following the END IF.

The ELSE and ELSEIF blocks are optional. You can have as many ELSEIF clauses as you want in a block IF. Any of the statement blocks can contain nested block IF statements.

Visual Basic looks at what appears after the THEN keyword to determine whether an IF statement is a block IF. If anything besides a comment appears after THEN, the statement is treated as a single-line IF statement.

A block IF statement must be the first statement on a line. The ELSE, ELSEIF, and END IF parts of the statement can have only a line number or line label in front of them. The block must end with an END IF statement.

Statement blocks can include GOTO *linenumber* or GOTO *linelabel* directives. Note: GOTO is optional with a line number, but is required with a line label.

The single-line form is never required. Any program using single-line IF...END IF statements can be written using block form.

You can have multiple statements with a single *condition*, but the statements must be on the same line and separated by colons:

```
IF A > 10 THEN A = A + 1: B = B + A: LOCATE 10,22: PRINT B,A
```

See Also IF TYPEOF Statement, SELECT CASE Statement.

Example The examples below illustrate use of the single-line and block form of IF...END IF statements to determine how many digits are in a number.

```
' Single-line form.
CLS                                    ' Clear the screen.
DO
    INPUT "Enter a number greater than 0 and less than 10,000:", X
    IF X >= 0 AND X < 10000 THEN EXIT DO ELSE PRINT X; "out of range"
LOOP
IF X < 10 THEN Y = 1 ELSE IF X < 100 THEN Y = 2 ELSE IF X < 1000 THEN
➥Y = 3 ELSE Y = 4
PRINT "The number has"; Y; "digits"
PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""
```

```
' Block form.
DO
    INPUT "Enter a number greater than 0 and less than 100,000:", X
    IF X >= 0 AND X < 100000 THEN
        EXIT DO
    ELSE
        PRINT X; "out of range"
    END IF
LOOP
IF X < 10 THEN
    Y = 1
ELSEIF X < 100 THEN
    Y = 2
ELSEIF X < 1000 THEN
    Y = 3
ELSEIF X < 10000 THEN
    Y = 4
ELSE
    Y = 5
END IF
PRINT "The number has"; Y; "digits"
```

$INCLUDE Metacommand

Action

Defines inclusion of information declared in an include file such as CONSTANT.BI, FINANCE.BI, or VBDOS.BI. $INCLUDE instructs the compiler to:

- Temporarily switch processing from one file to another.
- Read program statements from the Visual Basic file named in filespec.
- Return to processing the original file when the end of the included file is reached.

Syntax

{**REM** | '} **$INCLUDE:** *'filespec"*

Part	Description
REM | '	REM or remark character ('); must precede all metacommands .
:	Colon; required after metacommand, before filespec.
'filespec"	Name of a Visual Basic program file, including a path; must be enclosed in single quotation marks.

Remarks Include files are often used to group $FORM metacommands, declarations, and constants. It is easier to maintain a single include file than to update multiple Declaration sections in a project. For example:

Include files created from the programming environment must be in text (not binary) format.

When adding an include file to your project, be sure to select the Include File [*.BI] option in the Add File dialog box.

To view include-file text, choose Included File from the View menu.

When you are running a program from the Visual Basic programming environment (VBDOS.EXE), include files must not contain SUB or FUNCTION statements.

When you are compiling a program using the Visual Basic Compiler (BC.EXE), include files can contain SUB or FUNCTION statements.

See Also REM Statement.

Index Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, timer, vertical scroll bar

Action Sets or returns the number that uniquely identifies a control in a control array; read-only at run time.

Syntax [*form.*]*ctlname*[(*arrayindex%*)]**.Index**

Part	Description
arrayindex%	Valid settings are:

Value	Description
None 0 to 32,767	(Default) Not part of array; not available at design time. Part of array; specify an integer greater than or equal to 0 to assign a control to a control array. Or, if you specify the same name for two or more controls through CtlName, the first unique index available within the array is automatically assigned.

Remarks

Since control array elements share the same CtlName, you must use the Index property in code to refer to a particular control in the array. The index must appear as a number in parentheses next to the control array name. For example:

```
MyButton(3)
```

When a control in the array recognizes that an event has occurred, Visual Basic calls the array's event procedure and passes the applicable index as an additional argument.

Use Index when you are creating controls dynamically at run time with the LOAD statement, or removing them with the UNLOAD statement.

When you create a new element of a control array using the LOAD statement, the next available index is used.

To remove a control from a control array, change the control's CtlName property, or use the UNLOAD statement.

See Also

CtlName Property, Tag Property.

Example

This example illustrates use of the Index property and how controls can be added or deleted at run time. This example starts with two option buttons and adds new option buttons to the form each time the "Add" command button is clicked. Clicking the "Delete" command button deletes the last added control from the array. You can also click an option button to display the index value of the selected option button and change the background color in the picture box. To try this example, create a form containing two option buttons, a command button, and a large picture box. Set object properties:

- For Command1, Caption = "Add"
- For Command2, Caption = "Delete"
- For Option1, CtlName = "OptGroup"
- For Option2, CtlName = "OptGroup"

```
DIM Shared MaxIdx%

SUB Command1_Click ()
 IF MaxIdx% > 4 THEN EXIT SUB              ' Put only 5 buttons on form.
 IF MaxIdx% = 0 THEN MaxIdx% = 1           ' Preset MaxIdx%.
    MaxIdx% = MaxIdx% + 1                  ' Increment index.
    LOAD OptGroup(MaxIdx%)                 ' Create new item in array.
    ' Set location of new option button under previous button.
    OptGroup(MaxIdx%).Top = OptGroup(MaxIdx% - 1).Top + 1
    OptGroup(MaxIdx%).Visible = -1         ' Make new button visible.
END SUB
```

```
SUB Command2_Click ()
    IF MaxIdx% = 1 THEN EXIT SUB
    UNLOAD OptGroup(MaxIdx%)
    MaxIdx% = (MaxIdx% - 1)
END SUB

SUB OptGroup_Click (Index As Integer)
   Picture1.CLS                              ' Clear picture.
   Picture1.BackColor = OptGroup(Index).Index + 1
   Picture1.Print ("Button Index is #:" + STR$(OptGroup(Index).Index))
END SUB
```

INKEY$ Function

Action Reads a character from the keyboard or other standard input device; not available when forms are showing.

Syntax **INKEY$**

Remarks Input from the INKEY$ function cannot be directed to a form. Any reference to INKEY$ when forms are showing causes Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this function.

The INKEY$ function returns the following values:

Condition	Value
If there is no character	Null string.
For keys	1-byte string containing the character read.
For extended keys	2-byte string consisting of the null character (ASCII 0) and keyboard scan code.

The character returned by the INKEY$ function is never displayed on the screen; instead, all characters are passed through to the program except for the following key combinations:

Character	Visual Basic	Stand-Alone .EXE File
Ctrl+Break	Halts program execution.	Halts program execution if compiled with /D option; otherwise, passed through to program.
Ctrl+NumLock	Pauses program execution until a key is pressed; then next key pressed is passed through to program.	Same as running under VBDOS if compiled with /D option; otherwise, ignored; the program does not pause and no keystroke is passed.
Shift+PrtSc	Prints the screen contents.	Prints the screen contents.
Ctrl+Alt+Del	Reboots the system.	Reboots the system.

If you assign a string to a function key using the KEY statement and you press that function key when INKEY$ is waiting for a keystroke, INKEY$ passes the string to the program. Enabled keystroke trapping takes precedence over the INKEY$ function.

When you use INKEY$ with ISAM programs, Visual Basic performs implicit CHECKPOINT operations to minimize data loss in the event of a power failure. The CHECKPOINT is performed if INKEY$ fails to successfully retrieve a character after 65,535 calls, and 20 seconds has expired. A CHECKPOINT writes open database buffers to disk.

Example

This example uses the INKEY$ function to cause a program to pause until a key is pressed.

```
PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""
```

INP Function, OUT Statement

Action

Returns a byte read from a hardware I/O port (INP); sends a byte to a hardware I/O port (OUT).

Syntax

INP(*port%*)
OUT *port%*, *data%*

Part	Description
port%	Integer expression in the range 0 to 65,535, inclusive, that identifies the port address.
data%	Integer expression in the range 0 to 255, inclusive, to send to the port.

Remarks

INP and OUT give a Visual Basic program direct control over the system hardware through the I/O ports.

INP and OUT must be used carefully because they directly manipulate the system hardware.

See Also

WAIT Statement.

Example

This example uses the OUT statement and the INP function to control the timer and speaker to produce a note.

```
DECLARE SUB Sounds (Freq!, Length!)   ' Play a scale using speaker and timer.
CONST WHOLE = 5000!, QRTR = WHOLE / 4!
CONST C = 523!, D = 587.33, E = 659.26, F = 698.46, G = 783.99, A = 880!
CONST B = 987.77, C1 = 1046.5
CLS                                    ' Clear the screen.
CALL Sounds(C, QRTR): CALL Sounds(D, QRTR)
CALL Sounds(E, QRTR): CALL Sounds(F, QRTR)
CALL Sounds(G, QRTR): CALL Sounds(A, QRTR)
CALL Sounds(B, QRTR): CALL Sounds(C1, WHOLE)

STATIC SUB Sounds (Freq!, Length!)
    Clicks% = CINT(1193280! \ Freq!) ' Ports 66, 67, and 97 control timer
    LoByte% = Clicks% AND &HFF        ' and speaker; divide clock frequency
    HiByte% = Clicks% \ 256          ' by sound frequency to get number of
                                      ' "clicks" clock must produce.
    OUT 67, 182                       ' Tell timer that data is coming.
    OUT 66, LoByte%                   ' Send count to timer.
    OUT 66, HiByte%
    SpkrOn% = INP(97) OR &H3          ' Turn speaker on by setting bits 0 and
    OUT 97, SpkrOn%                   ' 1 of PPI chip.
    FOR I! = 1 TO Length!: NEXT I!    ' Leave speaker on.
    SpkrOff% = INP(97) AND &HFC       ' Turn speaker off.
    OUT 97, SpkrOff%
END SUB
```

INPUT #, LINE INPUT # Statement

Action Reads data items from a sequential device or file and assigns them to variables.

Syntax **[LINE] INPUT** *#filenumber%, stringvariable$*

Part	Description
filenumber%	Number specifying an open file.
stringvariable$	Holds a line of characters entered from the keyboard or read from a file.

Remarks Any reference to INPUT *#filenumber%* or LINE INPUT *#filenumber%* where filenumber% represents KYBD: or CONS: is not allowed with forms and causes Visual Basic to generate the error message, "Invalid when forms are showing."

When using INPUT #, data items in the file should appear just as they would if you were entering data in response to an INPUT statement. Separate numbers with a space, carriage return, linefeed, or comma. Separate strings with a carriage return or linefeed; leading spaces are ignored. The end-of-file character will end either a numeric or string entry.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return-linefeed sequence.

Use LINE INPUT # to read a text file a line at a time.

See Also File I/O Commands, INKEY$ Function, INPUT Statement, INPUT$ Function, INPUTBOX$ Function, LINE INPUT Statement, LINE Statement

Example This example uses the INPUT # statement to read a series of test scores from a sequential file, then calculates and displays the average score. The program uses the EOF function to determine when the last score has been read from the file.

```
DEFINT A-Z
DIM Total AS LONG

CLS                                 ' Clear the screen.
OPEN "class.dat" FOR OUTPUT AS #1   ' Create the required input file.
PRINT #1, 98, 84, 63, 89, 100
CLOSE #1
OPEN "class.dat" FOR INPUT AS #1    ' Open the input file just created.
DO WHILE NOT EOF(1)                 ' Do as long as the end-of-file
    Count = Count + 1               ' hasn't been reached.
    INPUT #1, Score                 ' Get a score from file #1.
    Total = Total + Score
    PRINT Count; Score
LOOP
PRINT
PRINT "Total students:"; Count; " Average score:"; Total / Count
END
```

INPUT$ Function

Action Returns a string of characters read from a specified file. Use the INPUTBOX$ function with forms.

Syntax **INPUT$(***n*[,[#]*filenumber%*])

Part	Description
n	Number of characters (bytes) to read.
filenumber%	Number of an open file; if omitted, INPUT$ reads from the keyboard.

Remarks Any reference to INPUT$ where *#filenumber%* represents KBYD: or CONS: is not allowed with forms and causes Visual Basic to generate the error message, "Invalid when forms are showing."

If the file is opened for random access, the argument *n* must be either:

- Less than or equal to the record length set by the LEN clause in the OPEN statement
- Less than or equal to 128 if the record length is not set

If the given file is opened for binary or sequential access, *n* must be less than or equal to 32,767.

If *filenumber%* is not specified, the characters are read from the standard input device. Note: If input has not been redirected, the keyboard is the standard input device.

You can use the MS-DOS redirection symbols (<, >, or >>) or the pipe symbol (|) to redefine the standard input or standard output for an executable file created with Visual Basic.

Unlike the INPUT # statement, INPUT$ returns all characters it reads. Use INPUT # to read data elements and assign them to variables.

See Also INPUT Statement, INPUT # Statement, LINE INPUT Statement, LINE INPUT # Statement.

Example This example uses the INPUT$ function to read one character at a time from a file. The input character is converted to standard printable ASCII, if necessary, and displayed on the screen.

Note You must supply a text file when you run this example. Use a text file you have already created, create a file with a text editor, or specify the README.TXT text file.

```
DEFINT A-Z
CONST HTAB = 9, LFEED = 10        ' ASCII codes for tab, and linefeed.
```

```
CLS                                 ' Clear the screen.
INPUT "Display which file"; Filename$
OPEN Filename$ FOR INPUT AS #1
DO WHILE NOT EOF(1)                 ' Input a single character from the file.
    S$ = INPUT$(1, #1)              ' Convert the character to an integer and
                                    ' turn off the high bit so WordStar files
                                    ' can be displayed.
    C = ASC(S$) AND &H7F            ' Is it a printable character?
    IF (C >= 32 AND C <= 126) OR C = HTAB OR C = LFEED THEN PRINT CHR$(C);
LOOP
END
```

INPUT, LINE INPUT Statement

Action

Reads input from the keyboard or a file (INPUT); reads a line of up to 255 characters from the keyboard or a file (LINE INPUT); not available when forms are showing. Use the INPUTBOX$ function with forms.

Syntax

[LINE] INPUT [;] ["*prompt*"{; | ,}] *stringvariable$*

Part	Description
;	Causes the cursor to remain on the same line after the user presses Enter; if omitted, the cursor skips to the next line.
prompt	Optional literal string that is displayed before the user enters data.
;	,	";" appends a question mark to the prompt; "," does not.
stringvariable$	Stores information entered from the keyboard or read from a file.

Remarks

References to the INPUT or LINE INPUT statement are not allowed with forms and cause Visual Basic to generate the error message, "Not available when forms are showing."

To use the INPUT or LINE INPUT statement with forms, you must first unload or hide any visible forms.

If the user is inputting only decimal integers, you can save between 1.6K and 11K in a non-stand-alone .EXE file by linking with the stub file NOFLTIN.OBJ. This places the following restrictions on user input:

- Decimal numbers only (no leading &H, &O, or &base specifiers)
- No trailing type specifiers (%, &, !, #, @, and $)
- No decimal point, E, or D (for example, 1.E2 cannot be used in place of the integer 100)

If the user is limited to using the Enter and Backspace keys for editing, you can reduce the size of the .EXE file by linking the stub file NOEDIT.OBJ.

The INPUT statement causes a running program to pause and wait for the user to enter data from the keyboard. The number and type of data items required from the user is determined by the structure of *stringexpression$*.

The number of entered data items must be the same as the number of variables in the list. The type of each entered data item must agree with the type of the variable. If either of these rules is violated, the program displays the prompt, "Redo from start" and input values are not assigned.

The INPUT statement determines the number of entered data items in a multiple-item list as follows:

- The first character encountered after a comma that is not a space, carriage return, or linefeed is assumed to be the start of a new item.
- If this first character is a quotation mark ("), the item is typed as string data and consists of all characters between the first quotation mark and the second.
- Not all strings have to start with a quotation mark. If the first character of the string is not a quotation mark, it terminates on a comma, carriage return, or linefeed.

Input stored in a record must be entered as single elements. For example:

```
TYPE Demograph
     FullName AS STRING * 25
     Age  AS INTEGER
END TYPE
DIM Person AS Demograph
INPUT "Enter name and age: "; Person.FullName, Person.Age
```

You can suggest to the user various ways to edit a line of input before pressing Enter to submit the line to the program. The following key combinations move the cursor, delete text, and insert text on the input line:

Keystrokes	Edit Function
Ctrl+M or Enter	Store input line.
Ctrl+H or Backspace	Delete character to the left of the cursor, unless cursor is at the beginning of the input, in which case delete character at the cursor.
Ctrl+\ or Right Arrow	Move cursor one character to the right.
Ctrl+] or Left Arrow	Move cursor one character to the left.
Ctrl+F or Ctrl+Right Arrow	Move cursor one word to the right.
Ctrl+B or Ctrl+Left Arrow	Move cursor one word to the left.
Ctrl+K or Home	Move cursor to beginning of input line.
Ctrl+N or End	Move cursor to end of input line.
Ctrl+R or Ins	Toggle insert mode on and off.

Keystrokes	Edit Function
Ctrl+I or Tab	Tab right and insert (insert mode on), or tab right and overwrite (insert mode off).
Del	Delete character at the cursor.
Ctrl+E or Ctrl+End	Delete to end of the line.
Ctrl+U or Esc	Delete entire line, regardless of cursor position.
Ctrl+T	Toggle function-key label display on and off at bottom of screen.
Ctrl+C or Ctrl+Break	Terminate input (exit compiled program).

When you use INPUT with ISAM programs, Visual Basic performs a CHECKPOINT operation every 20 seconds during an INPUT polling loop. A CHECKPOINT writes open database buffers to disk.

See Also INKEY$ Function, INPUT$ Function, INPUT # Statement, LINE INPUT # Statement, INPUTBOX$ Function, LINE Statement.

Example This example uses the INPUT statement to prompt the user for a value and assign it to a variable. The program calculates the area of a circle.

```
CLS                                          ' Clear the screen.
PI = 3.141593: R = -1                        ' Declare constants.
DO WHILE R
   PRINT "Enter radius (or 0 to quit)."
   INPUT ; "If radius = ", R
   IF R > 0 THEN
   A = PI * R ^ 2
   PRINT ", the area of the circle ="; A
   END IF
   PRINT
LOOP
```

INPUTBOX$ Function

Action Displays a prompt in a dialog box, waits for the user to input text or choose a button, and returns the contents of the edit box. INPUTBOX$ is available in screen mode 0 only.

Syntax **INPUTBOX$**(*prompt$* [, *title$* [, *default$* [, *xpos%*, *ypos%*]]])

Part	Description
prompt$	String expression containing the message to display in the dialog box; maximum length is 255 characters.
title$	String expression to display in the title bar of the dialog box; if omitted, no title is dispaly; *title$* text greater than 46 characters in length is truncated on the right.
default$	String expression to display in the edit box as the default response if no other input is provided; if omitted, the empty string is displayed ("").
xpos%	Numeric expression that specifies, in characters, the horizontal distance of the left edge of the dialog box from the left edge of the desktop; if omitted, dialog is centered.
ypos%	Numeric expression that specifies, in characters, the vertical distance of the top edge of the dialog box from the top edge of the desktop; if omitted, dialog is centered.

Remarks If the user chooses the OK button or presses Enter, INPUTBOX$ returns whatever is in the edit box. If the user chooses the Cancel button, INPUTBOX$ returns a null string ("").

If *prompt$* consists of more than one line, be sure to include a carriage return (CHR$(13)) and a linefeed (CHR$(10)) between each line.

If you omit *xpos%*, you must also omit *ypos%*. If *xpos%* and *ypos%* are omitted, the dialog box is horizontally and vertically centered.

The dialog box is movable and closable, but not sizable. Width and height are fixed at 46 x 16.

See Also INPUT Statement, INPUT # Statement, MSGBOX Function, MSGBOX Statement.

Example See ERROR$ Function example.

INSTR Function

Action Returns the position of the first occurrence of a string in another string.

Syntax **INSTR**([*start%*,]*stringexpression1$*, *stringexpression2$*)

Part	Description
start%	Sets the character position where the search begins; if omitted, INSTR starts at position 1.
stringexpression1$	String to be searched.
stringexpression2$	String to look for (substring).

Remarks

The arguments *stringexpression1$* and *stringexpression2$* can be string variables, string expressions, or string literals.

Use the LEN function to find the length of *stringexpression1$*.

The value returned by INSTR depends on the following conditions:

Condition	Return Value
stringexpression2$* found in *stringexpression1$	Position at which the match is found.
start% greater than length of *stringexpression1$*	0
stringexpression1$ is null string	0
stringexpression2$ cannot be found	0
stringexpression2$ is null string	*start%* (if given); otherwise, 1.

Example

This example uses INSTR and UCASE$ to determine a person's gender from the courtesy title (Mr., Mrs., or Ms.).

```
CLS                             ' Clear the screen.
DO                              ' Get a name
    INPUT "Enter name including courtesy title (Mr., Mrs., or Ms.): ", Nm$
LOOP UNTIL LEN(Nm$) >= 3

Nm$ = UCASE$(Nm$)               ' Convert lowercase letters to uppercase.
                                ' Look for MS, MRS, or MR to set Sex$.
IF INSTR(Nm$, "MS") > 0 OR INSTR(Nm$, "MRS") > 0 THEN
    Sex$ = "F"
ELSEIF INSTR(Nm$, "MR") > 0 THEN
    Sex$ = "M"
ELSE
    DO                          ' Can't determine sex, query user.
        INPUT "Enter sex (M/F): ", Sex$
        Sex$ = UCASE$(Sex$)
    LOOP WHILE Sex$ <> "M" AND Sex$ <> "F"
END IF
PRINT "Sex is "; Sex$           ' Print result
```

Interrupt[X] Routine

Action Enables Visual Basic programs to perform MS-DOS system calls. Requires VBDOS.QLB (VBDOS.LIB when compiling); include VBDOS.BI.

Syntax **CALL Interrupt[X]** (*interruptnum%*, *inregs*, *outregs*)

Part	Description
interruptnum%	MS-DOS interrupt number between 0 and 255, inclusive.
inregs	Register values before the interrupt is performed; declare *inregs* as type RegType (for Interrupt) or type RegTypeX (for InterruptX).
outregs	Register values after the interrupt is performed; declare *outregs* as type RegType (for Interrupt) or type RegTypeX (for InterruptX).

Remarks The Interrupt and InterruptX routines replace the INT86 and INT86X routines used in earlier versions of Microsoft QuickBasic. They provide a more convenient way for Visual Basic programs to use MS-DOS interrupts and services.

The register values before and after the interrupt are passed in variables declared as type RegType or type RegTypeX. Each element of the type corresponds to a CPU element.

The following statement defines RegType, available through Interrupt (the DS and ES registers are not included):

```
TYPE RegType
     AX AS INTEGER
     BX AS INTEGER
     CX AS INTEGER
     DX AS INTEGER
     BP AS INTEGER
     SI AS INTEGER
     DI AS INTEGER
     FLAGS AS INTEGER
END TYPE
```

The following statement defines RegTypeX, available through InterruptX:

```
TYPE RegTypeX
     AX AS INTEGER
     BX AS INTEGER
     CX AS INTEGER
     DX AS INTEGER
     BP AS INTEGER
```

```
    SI AS INTEGER
    DI AS INTEGER
    FLAGS AS INTEGER
    DS AS INTEGER
    ES AS INTEGER
END TYPE
```

InterruptX uses the values in the DS and ES registers. To use the current values of these registers, set the record elements to -1.

See Also Absolute Routine, $INCLUDE Metacommand.

Example See $INCLUDE Metacommand example.

Interval Property

Applies To Timer control.

Action Sets or returns the number of milliseconds in a timer's countdown interval. This interval represents the time delay before the associated Timer event is called.

Syntax [*form.*]*timer*.**Interval**[= *milliseconds&*]

Part	Description
milliseconds&	Valid settings are:

Value	Description
0	(Default) Timer disabled.
1 to 65,535	Sets an interval (in milliseconds) that takes effect once the timer's Enabled property is set to True (-1). For example, a value of 10,000 milliseconds equals 10 seconds; the maximum, 65,535 milliseconds, is equivalent to just over one minute.

Remarks You can set a timer's Interval property at design time or run time.

When using the Interval property, remember:

- The timer's Enabled property determines if a timer responds to the passage of time. Set Enabled to True (-1) to turn the timer on, and to False (0)) to turn it off.
- Write a Timer event procedure to tell Visual Basic what to do each time the interval has passed.

Example The example allows you to adjust the speed at which a form switches colors by dragging the scrollbox along the horizontal scroll bar.

```
SUB HScroll1_Change ()      ' Set interval according to scroll bar value.
    Timer1.Interval = 1000 - HScroll1.Value
END SUB

SUB Timer1_Timer ()         ' Switch BackColor between Red and Blue.
    IF Picture1.BackColor = 1 THEN
        Picture1.BackColor = 4
    ELSE
        Picture1.BackColor = 1
    END IF
END SUB
```

IOCTL$ Function, IOCTL Statement

Action Returns current status information from a device driver (IOCTL$); transmits a control data string to a device driver (IOCTL).

Syntax **IOCTL$**([#]*filenumber%*)
IOCTL [#]*filenumber%*, *string$*

Part	Description
filenumber%	Number of an open device.
string$	Control string sent to the device.

Remarks The IOCTL$ function is most frequently used to:

- Test whether an IOCTL statement succeeded or failed.
- Obtain current status information.

The IOCTL statement works only if all of the following conditions are met:

- The device driver is installed.
- The device driver states that it processes IOCTL strings.
- Visual Basic performs an OPEN operation on a file on that device and the file is still open.

An IOCTL control data string can be up to 32,767 bytes long.

ICOTL control strings and the information returned by IOCTL$ depend on the device driver. See your device-driver documentation for more information about IOCTL control strings and what is returned by IOCTL$.

Since most standard MS-DOS device drivers do not process IOCTL strings, you must determine if the specific driver accepts the command. If the driver does not process IOCTL strings, Visual Basic generates the error message, "Illegal Function Call."

Visual Basic devices (LPTn, COMn, SCRN, CONS, PIPE) and MS-DOS block devices (A - Z) do not support IOCTL.

You can use IOCTL$ to ask a communications device to return the current baud rate, information on the last error, logical line width, and so on. The exact information returned depends on the specific device driver.

Example

This example uses the IOCTL statement to set the data mode in the driver and the IOCTL$ function to test the data mode when communicating with a hypothetical device driver.

Note: The IOCTL$ function works only if the device driver is installed and states that it processes IOCTL strings, and only if an OPEN statement is performed on the device. Do not run this example in its current form.

```
OPEN "\DEV\ENGINE" FOR OUTPUT AS #1
IOCTL #1, "RAW"                 ' Tells the device that the data is raw.

' If the character driver "ENGINE" responds "false" from the raw data mode
' in the IOCTL statement, then the file is closed.

IF IOCTL$(1) = "0" THEN CLOSE 1
```

KEY Statement (Assignment)

Action

Assigns string values to function keys and, optionally, displays key values.

Syntax

KEY {*key%*, *stringexpression$* | **LIST** | **ON** | **OFF**}

Part	Description
key%	Number specifying a function key:

Value	Function Key
1 - 10	F1 - F10
30	F11
31	F12

Part	Description
stringexpression$	String of up to 15 characters that is returned when the function key is pressed.
LIST	Displays the assignments for each function key; not available when forms are showing.
ON	Turns on the function-key display line; not available when forms are showing.
OFF	Turns off the function-key display line.

Remarks

References to the KEY LIST or KEY ON statement are not allowed when forms are showing and cause Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing either of these statements.

Use the KEY statement to designate special "soft-key" functions — that is, strings that are returned when function keys are pressed.

Assigning a null string to a soft key disables the function key as a soft key.

If *stringexpression$* is longer than 15 characters, the extra characters are ignored.

If the function-key number is not in the correct range, Visual Basic generates the error message, "Illegal function call," and the previous key string expression is retained.

You can display soft keys with the KEY ON, KEY OFF, and KEY LIST statements:

Statement	Action
KEY ON	Displays the first five to six characters of the soft-key string values on the bottom line of the screen.
KEY OFF	Erases the soft-key display from the bottom line, making that line available for program use. It does not disable the function keys.
KEY LIST	Displays all soft-key values on the screen, with all 15 characters of each key displayed.

If a soft key is pressed, the effect is the same as if the user typed the string associated with the soft key.

INPUT$, INPUT, and INKEY$ all can be used to read the string produced by pressing the soft key.

See Also

KEY Statement (Event Trapping).

Example

This example uses the KEY statement to set up one-key equivalents of menu selections. For example, pressing F1 is the same as entering the string "Add."

```
CLS                             ' Clear screen.
DIM KeyText$(3)
DATA Add, Delete, Quit
```

```
FOR I = 1 TO 3              ' Assign soft-key strings from F1 to F3.
    READ KeyText$(I)
    KEY I, KeyText$(I) + CHR$(13)
NEXT I
PRINT "                 Main Menu": PRINT      ' Print menu.
PRINT "           Add to list (F1)"
PRINT "           Delete from list (F2)"
PRINT "           Quit (F3)": PRINT
DO
    LOCATE 7, 1: PRINT SPACE$(50);             ' Get input and respond.
    LOCATE 7, 1: INPUT "             Enter your choice:", R$
    SELECT CASE R$
        CASE "Add", "Delete"
            LOCATE 10, 1: PRINT SPACE$(15);
            LOCATE 10, 1: PRINT R$;
        CASE "Quit"
            EXIT DO
        CASE ELSE
            LOCATE 15, 1: PRINT "Enter first word exactly as shown"
            LOCATE 16, 1: PRINT "or press function key."
    END SELECT
LOOP
```

KEY Statement (Event Trapping)

Action

Enables, disables, or suspends event trapping of a key; not available when forms are showing; cannot be used with object-specific events.

Syntax

KEY(*n%*) {ON | OFF | STOP}

Part	Description
n%	Value that specifies a function key, arrow key, or user-defined key:

Value	Description
0	All keys listed (KEY(0) ON, KEY(0) OFF, and KEY(0) STOP only).
1-10	Function keys F1 - F10.
11	Up arrow key.
12	Left arrow key.

Part	Description
n%	Value that specifies a function key, arrow key, or user-defined key:

Value	Description
13	Right arrow key.
14	Down arrow key.
15-25	User-defined keys.
30, 31	Function keys F11 and F12.

ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a KEY ON statement is executed.

Remarks

References to KEY statements when forms are showing generate an implicit KEY OFF event. Use form- and control-specific events with forms.

To use the KEY statement with forms, you must first unload or hide any visible forms.

If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

KEY(*n%*) ON enables trapping of function keys, arrow keys, and user-defined keys. If key *n%* is pressed after a KEY(*n%*) ON statement, the routine specified in the ON KEY statement is executed.

KEY(*n%*) OFF disables trapping of key *n%*. No trapping takes place until another KEY(*n%*) ON statement is executed. Events occurring while trapping is off are ignored.

KEY(*n%*) STOP suspends trapping of key *n%*. No trapping takes place until a KEY(*n%*) ON statement is executed. Any ON KEY events that occur while trapping is suspended are remembered and processed when the next KEY(*n%*) ON is executed. However, remembered events are lost if KEY(*n%*) OFF is executed.

When a key event trap occurs (that is, the GOSUB is performed), an automatic KEY STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a KEY ON statement unless an explicit KEY OFF was performed inside the routine.

Visual Basic can also be used to trap shifted keys and other user-defined events.

See Also

KEY Statement (Assignment), KeyDown Event, KeyPress Event, KeyUp Event, ON KEY Statement.

Example This example uses the KEY and ON KEY statements to trap the Ctrl+s (control key and lowercase "s") key combination.

Note: This example requires Caps Lock and Num Lock to be off.

```
CONST ESC = 27

KEY 15, CHR$(&H4) + CHR$(&H1F)              ' Set up Ctrl+s as KEY 15
ON KEY(15) GOSUB PauseHandler
KEY(15) ON
WHILE INKEY$ <> CHR$(ESC)
    PRINT "Press Esc to stop, Ctrl+s to pause."
    PRINT
WEND
END

PauseHandler:
    SLEEP 1
    RETURN
```

KeyDown, KeyUp Events

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action Occurs when the user presses (KeyDown) or releases (KeyUp) a key while an object has the focus.

Syntax

SUB Form_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
SUB *ctlname*_**KeyDown ([Index AS INTEGER,]KeyCode AS INTEGER, Shift AS INTEGER)**

SUB Form_KeyUp (KeyCode AS INTEGER, Shift AS INTEGER)
SUB *ctlname*_**KeyUp ([Index AS INTEGER,]KeyCode AS INTEGER, Shift AS INTEGER)**

Part	Description
KeyCode	Key's key code. For example, KEY_F1 and KEY_HOME are the key codes for the F1 and Home keys, respectively. To specify key codes, use the constants in CONSTANT.BI, a file that you can include with each module using the $INCLUDE metacommand.
Shift	State of the Shift, Ctrl, and Alt keys at the time of the event. The Shift argument is a bit field, with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys is pressed.
Index	Uniquely identifies a control in a control array.

Remarks

Use KeyDown and KeyUp event procedures to handle keyboard routines when forms are showing. The ON KEY statement will not work when forms are showing.

For both KeyDown and KeyUp events, the object with the focus receives all keystrokes, but a form can have the focus only if it has no visible or enabled controls.

KeyCode values for ASCII-printable keys (A-Z, a-z, 0-9, ~, [,], {, }, +, =) are the same as their ASCII equivalents. In Visual Basic for DOS, the returned KeyCode value differentiates between uppercase and lowercase, and returns the same value as the KeyAscii parameter used in the KeyPress event procedure.

Use shortcut and access keys for keyboard shortcuts instead of code that responds to KeyDown and KeyUp events; these keys require no additional code. You can program these keys when creating menus in the Form Designer.

Although KeyDown and KeyUp events can detect all keys, they are most often used for:

- Extended character keys such as function keys.
- Navigation keys.
- Combinations of the Shift, Ctrl, and Alt keys (keyboard modifiers).
- Distinguishing between the numeric keypad and regular number keys.

Use KeyDown and KeyUp event procedures to write keyboard handlers if you want your application to respond to both the press and release of a key.

KeyDown and KeyUp are not invoked for the following keys and conditions:

Key	Condition
Enter	If the form has a command button with the Default property set to True (-1).
Esc	If the form has a command button with the Cancel property set to True (-1).
Tab	If two or more controls are on the form.

See Also KeyPress Event.

Example The example demonstrates a generic keyboard handler that responds to the F2 function key and to all of the associated Alt/Shift/Ctrl combinations. To find the value of the key constants, look at CONSTANT.BI, or just load it into your main module.

```
SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    CONST Key_F2 = &H71                    ' Set constants as in CONSTANT.BI.
    CONST SHIFT_MASK = 1
    CONST CTRL_MASK = 2
    CONST ALT_MASK = 4
    DIM txt AS STRING
    ShiftDown% = (Shift AND SHIFT_MASK) > 0
    AltDown% = (Shift AND ALT_MASK) > 0
    CtrlDown% = (Shift AND CTRL_MASK) > 0
    IF KeyCode = Key_F2 THEN               ' Display key combinations.
         IF ShiftDown% AND CtrlDown% AND AltDown% THEN
              txt$ = "Shift + Ctrl + Alt F2."
         ELSEIF ShiftDown% AND AltDown% THEN
              txt$ = "Shift + Alt F2."
         ELSEIF ShiftDown% AND CtrlDown% THEN
              txt$ = "Shift + Ctrl + F2."
         ELSEIF CtlDown% AND AltDown% THEN
              txt$ = "Ctl + Alt + F2."
         ELSEIF ShiftDown% THEN
              txt$ = "Shift F2."
         ELSEIF CtrlDown% THEN
              txt$ = "Ctrl F2."
         ELSEIF AltDown% THEN
              txt$ = "Alt F2."
         ELSEIF Shift = 0 THEN
              txt$ = "F2."
         END IF
              Text1.Text = "You pressed " + txt$
    END IF
END SUB
```

KeyPress Event

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action Occurs when the user presses an ASCII key. The object with the focus receives the event. A form can receive the event only if it has no visible or enabled controls.

Syntax

SUB Form_KeyPress (KeyAscii AS INTEGER)
SUB *ctlname*_KeyPress ([Index AS INTEGER] KeyAscii AS INTEGER)

Part	Description
KeyAscii	Returns a standard numeric ASCII character value.
Index	Uniquely identifies a control in a control array.

Remarks

A KeyPress event can involve:

- Any printable keyboard character.
- The Ctrl key + a character from the standard alphabet or one of a few special characters.
- The Enter key.
- The Backspace key.

A KeyPress event procedure is useful for intercepting keystrokes entered in a text box or combo box. It allows you to immediately test keystrokes for validity or to format characters as they are typed.

KeyAscii is passed by reference; changing it sends a different character to the object. This can be useful in a password scheme that uses a text box where you do not want the entered password to be displayed.

Changing KeyAscii to zero (0) cancels the keystroke so that the object does not receive the character.

You can convert the KeyAscii argument into a character by using the expression CHR$(KeyAscii). You can then perform string operations and translate the character back to an ASCII format that the control can understand by using this expression:

```
KeyAscii = ASC("x")
```

Use KeyDown and KeyUp event procedures to handle non-ASCII keystrokes such as:

- Function keys.
- Editing keys.
- Navigation keys.
- Any combination of the above with the Shift, Ctrl, and Alt keys (keyboard modifiers).

KeyPress interprets the uppercase and lowercase of each character as separate key codes and therefore as separate characters.

See Also

Change Event, KeyDown Event, KeyUp Event.

Example This example illustrates the use of the KeyPress event by showing a procedure that converts the text entered into a text box to uppercase. To try this example, create a form containing a text box.

```
SUB Text1_KeyPress (KeyAscii AS INTEGER)
    C$ = CHR$(KeyAscii)
    Keyascii = ASC(UCASE$(C$))
END SUB
```

KILL Statement

Action Deletes files from a disk.

Syntax **KILL** *filespec$*

Part	Description
filespec$	Specifies file(s) to delete; can include a path and the MS-DOS wildcards ? and *.

Remarks The argument *filespec$* is a string expression that may contain a path and the MS-DOS wildcard characters question mark (?) or asterisk (*). A question mark matches any single character in the file name or extension. An asterisk matches one or more characters starting at its position.

Be extremely careful when using wildcard characters with KILL. You can delete files unintentionally.

The KILL statement is similar to the MS-DOS ERASE or DEL command.

KILL is used for all types of disk files (program files, random-access data files, and sequential data files).

KILL deletes files only. To delete directories, use the MS-DOS RMDIR command or the Visual Basic RMDIR statement.

If you use KILL to delete a file that is currently open, Visual Basic generates the error message, "File already open."

See Also FILES Statement.

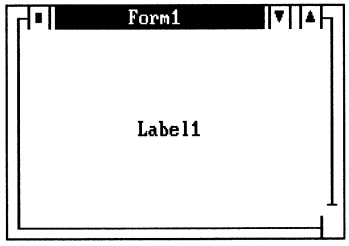
Example This example deletes the file specified on the command line. Note: Choose Modify COMMAND$ from the Run menu to specify a file to delete.

```
DEFINT A-Z
CLS                                 ' Clear the screen.
ON ERROR GOTO Errorhandle           ' Set up error handling.
FileName$ = COMMAND$                ' Get file name.
KILL FileName$
```

```
PRINT FileName$; " deleted"
END

Errorhandle:
     Number = ERR
     IF Number = 53 THEN
          PRINT "Couldn't delete "; FileName$;
          PRINT "; file does not exist in current directory"
     ELSE
          PRINT "Unrecoverable error:"; Number
          ON ERROR GOTO 0              ' Aborts program.
     END IF
RESUME NEXT
```

Label Control

Action Consists of text (called a caption) that the user cannot directly change.



Remarks

Events

Change	Click	DblClick	DragDrop
DragOver	MouseDown	MouseMove	MouseUp

Methods

DRAG	MOVE	REFRESH

Properties

Alignment	AutoSize	BackColor	BorderStyle
Caption	CtlName	DragMode	Enabled
ForeColor	Height	Index	Left
MousePointer	Parent	TabIndex	Tag
Top	Visible	Width	

You can use labels to display messages to the user by writing code that changes a label in response to events at run time. For example, if your application takes a few minutes to commit a change, you can create a message that displays its processing status.

You can set only one line of text with a label's Caption property at design time. To display multiple lines of text in a label, you must write code that includes carriage returns at line breaks.

A label can be used to identify a control, such as a text box, that does not have its own Caption property. Because you can set only one line of text with the Caption property at design time, you can use a label to define multiple line headings for controls that have their own Caption property.

The minimum size for a label is 1 x 1; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value. If the label has a border, Height property settings 1 and 2 change the appearance of the control.

See Also Caption Property.

Example See Autosize Property example.

LargeChange, SmallChange Properties

Applies To Horizontal scroll bar, vertical scroll bar.

Action Determines the amount of change to report in a scroll bar control when the user clicks between the scroll box and scroll arrow (LargeChange), or the amount of change to report in a scroll bar control when the user clicks the scroll arrow (SmallChange).

Syntax

[*form.*]{*hscrollbar.* | *vscrollbar.*}**LargeChange**[= *change%*]
[*form.*]{*hscrollbar.* | *vscrollbar.*}**SmallChange**[= *change%*]

Part	Description	
change%	Valid settings are:	
	Value	**Description**
	1 - 32,767	Valid LargeChange or SmallChange setting.
	1	(Default).

Remarks The operating environment automatically sets proportional scrolling increments for scroll bars on multiline text boxes, combo boxes, and list boxes. The increments are proportional to the amount of data in the control. For a scroll bar control, however, you must specify these increments.

Use LargeChange and SmallChange to set scrolling increments appropriate to how the scroll bar is used. Typically, you set LargeChange and SmallChange at design time. You can also reset them in code at run time when the scrolling increment must change dynamically.

The scroll bar's Value property increases or decreases by the change amount specified.

Use the Max and Min properties to set the maximum and minimum ranges of a scroll bar control.

See Also Change Event, Max Property, Min Property, Value Property.

Example This example uses a scroll bar to move a picture box control across a form. To try this example, create a form containing a picture box and a horizontal scroll bar.

```
SUB Form_Load ()
   HScroll1.Max = 100                ' Set maximum value.
   HScroll1.LargeChange = 20         ' Cross in 5 clicks.
   HScroll1.SmallChange = 5          ' Cross in 20 clicks.
   Picture1.Left = 0                 ' Start picture at left.
   Picture1.BackColor = 3            ' Set color of picture box.
END SUB

SUB HScroll1_Change ()
   ' Move picture according to scroll bar.
   Picture1.Left = (HScroll1.Value / 100) * ScaleWidth
END SUB
```

LBOUND, UBOUND Functions

Action Returns the lowest and highest element (smallest and largest available subscript) for the specified array dimension.

Syntax **LBOUND**(*array* [, *dimension%*])
UBOUND(*array* [, *dimension%*])

Part	Description
array	Any valid array name.
dimension%	(Default - 1) Array dimension whose lower or upper boundary is returned; use 1 for the first dimension, 2 for the second dimension, and so on.

Remarks

The UBOUND function can be used with the LBOUND function to determine the size of an array.

Use REDIM with the PRESERVE keyword to change the size of an array to the exact size you need. Use UBOUND to tell how many filled elements there are in the array since they will all be filled.

LBOUND and UBOUND return the values listed below for an array with the following dimensions:

```
DIM A(1 TO 100, 0 TO 3, -3 TO 4)
```

Usage	Value Returned
UBOUND(A,1)	100
UBOUND(A,2)	3
UBOUND(A,3)	4
LBOUND(A,1)	1
LBOUND(A,2)	0
LBOUND(A,3)	-3

You can use the shortened syntax UBOUND(*array*) or LBOUND(*array*) for one-dimensional arrays, because the default value for *dimension%* is 1.

The default lower bound for any dimension is either 0 or 1, depending on the setting of the OPTION BASE statement. If OPTION BASE is 0, the default lower bound is 0; if OPTION BASE is 1, the default lower bound is 1.

Arrays dimensioned using the TO clause in the DIM statement can have any integer value as a lower bound.

See Also

OPTION BASE Statement, REDIM Statement.

Example

This example uses the LBOUND and UBOUND functions in a SUB procedure to determine the size of an array passed to the procedure by a calling program.

```
DECLARE SUB PRNTMAT (a!())

CLS                                    ' Clear the screen.
DIM a(0 TO 3, 0 TO 3)
FOR I% = 0 TO 3
   FOR J% = 0 TO 3
   a(I%, J%) = I% + J%
   NEXT J%
NEXT I%
CALL PRNTMAT(a())
END
```

```
STATIC SUB PRNTMAT (a())
   FOR I% = LBOUND(a, 1) TO UBOUND(a, 1)
   FOR J% = LBOUND(a, 2) TO UBOUND(a, 2)
      PRINT a(I%, J%); " ";
   NEXT J%
   PRINT : PRINT
   NEXT I%
END SUB
```

LCASE$, UCASE$ Functions

Action Converts strings to all lowercase letters (LCASE$) or all uppercase letters (UCASE$).

Syntax **LCASE$**(*stringexpression$*)
**UCASE$**(*stringexpression$*)

Part	Description
stringexpression$	Any string expression.

Remarks Both functions are helpful in string-comparison operations that are not case sensitive.

Both functions can be used with variable- and fixed-length strings.

Example This example illustrates the LCASE$ and UCASE$ functions.

```
CLS                                  ' Clear the screen.
Test$ = "THE string"
PRINT Test$
PRINT LCASE$(Test$); " in lowercase"
PRINT UCASE$(Test$); " IN UPPERCASE"
```

LEFT$, RIGHT$ Functions

Action Returns a specified number of leftmost characters (LEFT$) or rightmost characters (RIGHT$) in a string.

Syntax **LEFT$**(*stringexpression$*, *n%*)
**RIGHT$**(*stringexpression$*, *n%*)

Part	Description
stringexpression$	Any string expression.
n%	Number of characters to return, beginning with the leftmost or rightmost string character.

Remarks

If *n%* is 0, the null string (length 0) is returned.

If *n%* is greater than or equal to the number of characters in *stringexpression$*, the entire string is returned.

To find the number of characters in *stringexpression$*, use:

```
LEN(stringexpression$)
```

Example

This example uses the LEFT$ function to print the leftmost five characters of a string and the RIGHT$ function to print the rightmost five characters of the same string.

```
CLS                              ' Clear the screen.
a$ = "Visual Basic for MS-DOS"
PRINT LEFT$(a$, 12)              ' Output is:  Visual Basic
PRINT RIGHT$(a$, 6)              ' Output is:  MS-DOS
```

Left, Top Properties

Applies To

Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action

Sets or returns the horizontal position or column number (Left) or the vertical position or row number (Top) of a form or control.

Syntax

[*form.*][*ctlname.*]**Left**[= *horipos%*]
[*form.*][*ctlname.*]**Top**[= *vertpos%*]

Part	Description
horipos%	Column number indicating the position of the object's left edge relative to its container; valid range for forms and controls is 0 - 254.
vertpos%	Row number indicating the position of the object's top edge relative to its container; valid range for forms and controls is 0 - 254.

Remarks Left and Top property values are expressed in terms of characters.

The values for these properties change as the object is moved by the user or by code.

Control values of the Left and Top properties are measured relative to the container form or parent control.

Use Left and Top, and their related properties, Height and Width, for measurements or calculations based on an object's total area, such as sizing or moving the object.

Use ScaleHeight and ScaleWidth for measurements based only on an object's internal area, such as drawing or moving objects within an object. Note: These properties apply only to forms and picture boxes.

(0,0) is the top left side of the screen when using Left or Top with forms.

(0,0) is the top left side of the container form or parent control when using Left or Top with controls.

See Also LEFT$ Function, Height Property, MOVE Method, Width Property.

Example See Height Property example.

LEN Function

Action Returns the number of characters in a string or the number of bytes required to store a nonstring variable.

Syntax **LEN**({*stringexpression$* | *variable*})

Part	Description
stringexpression$	Any string expression.
variable	Any nonstring variable.

Remarks Use the LEN function to return the number of bytes required by a variable when determining the correct record size for random-access files.

Use LEN to determine the length of string used in string operations.

Example This example uses the LEN function to print the length of a string and the size in bytes of several types of variables.

```
CLS                                    ' Clear screen.
TYPE EmpRec
    EmpName AS STRING * 20
    EmpNum AS INTEGER
END TYPE
```

```
DIM a AS INTEGER, B AS LONG, C AS SINGLE, D AS DOUBLE
DIM E AS EmpRec
PRINT "A string:"; LEN("A string.")
PRINT "Integer:"; LEN(a)
PRINT "Long:"; LEN(B)
PRINT "Single:"; LEN(C)
PRINT "Double:"; LEN(D)
PRINT "EmpRec:"; LEN(E)
END
```

LET Statement

Action Assigns the value of an expression to a variable.

Syntax [**LET**] *variable* = *expression*

Part	Description
variable	Any valid variable name.
expression	Any valid expression.

Remarks The LET keyword is always optional. You can assign an expression to a variable with or without LET.

LET statements can be used with record variables only when both variables are the same user-defined type. Use the LSET statement to assign record variables of different user-defined types.

See Also LSET Statement, RSET Statement.

Example This example uses the optional LET statement to assign values to variables.

```
CLS                             ' Clear the screen.
LET D = 12
LET E = 12 - 2
LET F = 12 - 4
LET SUM = D + E + F
PRINT D; E; F; SUM

D = 12                          ' The program lines to the left perform the
E = 12 - 2                      ' same function without using the LET keyword.
F = 12 - 4
SUM = D + E + F
PRINT D; E; F; SUM
```

LINE Statement

Action Draws a line or rectangle on the screen; not available when forms are showing.

Syntax **LINE** [[**STEP**](*x1!*,*y1!*)] - [**STEP**](*x2!*,*y2!*) [,[*color%*] [,[**B** | **BF**] [,*style%*]]]

Part	Description
STEP	Specifies that coordinates are relative to the current graphics cursor position.
(*x1!*,*y1!*), (*x2!*,*y2!*)	Screen coordinates of the start of the line and the end of the line.
color%	Color attribute that sets the line or rectangle color.
B	Draws a rectangle instead of a line; uses (*x1!*,*y1!*) and (*x2!*,*y2!*) as its diagonally opposite corners.
BF	Draws a filled box.
style%	16-bit value whose bits set whether or not pixels are drawn; use to draw dashed or dotted lines.

Remarks You must be in a graphics screen mode (for example, screen mode 1) to use the LINE statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

The available color attributes for LINE depend on your graphics adapter and the screen mode set by the most recent SCREEN statement.

When coordinates specify a point that is not within the current viewport, the line segment to that point is drawn to the border of the viewport.

LINE reads the bits in *style%* from left to right. If a bit is 0, then no point is plotted; if the bit is 1, a point is plotted. After plotting a point, LINE selects the next bit position in *style%*.

Using the *style%* argument with LINE is called "line styling."

Because a zero bit in *style%* does not change the point on the screen, you may want to draw a background line before styling, so that you have a known background.

The *style%* argument is used for normal lines and boxes, but has no effect on filled boxes.

STEP makes the specified coordinates relative to the most recent point. For example, if the most recent point referred to by the program were (10,10), the following would draw a line from (10,10) to the point with an x coordinate equal to 10+10 and a y coordinate equal to 10+5, or (20,15):

```
LINE -STEP (10,5)
```

You can establish a new most recent point by initializing the screen with the CLS and SCREEN statements. Using other graphics statements - for example, PSET, PRESET, CIRCLE, or DRAW - also establishes a new most recent point.

Variations of the STEP argument are shown below. For the following examples, assume that the last point plotted was (10,10):

Statement	Description
LINE -(50,50)	Draws from (10,10) to (50,50)
LINE -STEP(50,50)	Draws from (10,10) to (60,60)
LINE (25,25)-STEP(50,50)	Draws from (25,25) to (75,75)
LINE STEP(25,25)-STEP(50,50)	Draws from (35,35) to (85,85)
LINE STEP(25,25)-(50,50)	Draws from (35,35) to (50,50)

See Also LINE INPUT Statement, LINE INPUT # Statement.

Example This example uses the LINE statement to display a series of screens with different line graphics.

Note: To run this example, your screen must be 320 pixels wide by 200 pixels high and must support CGA screen mode.

```
SCREEN 1                    ' Sets up the screen mode.
LINE -(X2, Y2)             ' Draw a line (in the foreground color) from
                            ' the most recent point to X2, Y2.
DO
LOOP WHILE INKEY$ = ""      ' Press any key to continue.

CLS                         ' Clear the screen.
LINE (0, 0)-(319, 199)      ' Draw a diagonal line across the screen.
DO
LOOP WHILE INKEY$ = ""      ' Press any key to continue.

CLS                         ' Clear the screen.
LINE (0, 100)-(319, 100)    ' Draw a horizontal line across the screen.
DO
LOOP WHILE INKEY$ = ""      ' Press any key to continue.

CLS                         ' Clear the screen.
LINE (10, 10)-(20, 20), 2   ' Draw a line in color 2.
DO
LOOP WHILE INKEY$ = ""      ' Press any key to continue.

CLS                         ' Clear the screen.
FOR X = 0 TO 319            ' Draw an alternating pattern (line on/line off).
    LINE (X, 0)-(X, 199), X AND 1
NEXT
DO
LOOP WHILE INKEY$ = ""      ' Press any key to continue.
```

```
CLS                              ' Clear the screen.
LINE (0, 0)-(100, 100), , B      ' Draw a box in the foreground color.
DO
LOOP WHILE INKEY$ = ""           ' Press any key to continue.

CLS                                          ' Clear the screen.
LINE STEP(0, 0)-STEP(200, 200), 2, BF        ' Draws a filled box in color 2
                                             ' (coordinates are given as
                                             ' offsets with the STEP option).
DO
LOOP WHILE INKEY$ = ""                       ' Press any key to continue center of
                                             ' the screen in color 3.
DO
LOOP WHILE INKEY$ = ""                       ' Press any key to continue.
SCREEN 1                                     ' Final example.
LINE (110, 70)-(190, 120), , B
LINE (0, 0)-(320, 200), 3, , &HFF00
```

LINE INPUT Statement

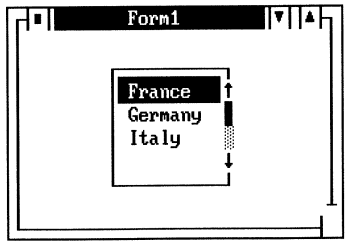
See INPUT Statement.

LINE INPUT # Statement

See INPUT# Statement.

List Box Control

Action Provides a list of choices to the user.



Remarks

Events

Click	DblClick	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
LostFocus	MouseDown	MouseMove	MouseUp

Methods

ADDITEM	DRAG	MOVE	REFRESH
REMOVEITEM	SETFOCUS		

Properties

BackColor	CtlName	DragMode	Enabled
ForeColor	Height	Index	Left
List	ListCount	ListIndex	MousePointer
Parent	Sorted	TabIndex	TabStop
Tag	Text	Top	Visible
Width			

Setting the Sorted property to True (-1) displays the list in alphabetical order.

You can add or delete items in a list with the ADDITEM or REMOVEITEM method.

Use the List, ListCount, and ListIndex properties to access items displayed in the list box at run time.

The minimum size of a list box is 4 x 5; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value.

See Also Combo Box Control.

Example See DblClick Event example.

List Property

Applies To Combo box, directory list box, drive list box, file list box, list box.

Action Sets or returns the items contained in a control's list. The list area is a string array in which each element is a list item. Property not available at design time; read-only for directory list box, drive list box, and file list box.

Syntax [*form.*]*ctlname*.**List**(*index%*)[= *stringexpression$*]

Part	Description
index%	Position in the list:

Value	Description
0 to ListCount-1	Valid string array index.
-1	(Default) Empty string array; no item selected.

Part	Description
stringexpression$	List item text.

Remarks

For all list box controls (except a directory list box), 0 is the index of the first item and ListCount-1 is the index of the last item. Enumerating a list from 0 to ListCount-1 returns all items in the list.

For a directory list box, enumerating the list from -n to ListCount-1 returns a list containing all directories and subdirectories visible from the currently expanded directory. In this case, n is the number of directory levels above the currently expanded directory.

The index number sequence for a directory list box is based on the directories and subdirectories present when the control is created at run time:

Index	Description
-1	The currently expanded directory.
-2 to -n	Directories above the currently expanded directory; represented by a negative index with higher absolute values. For example, -2 is the parent directory of the currently expanded directory, and -3 is the directory above that.
0 to ListCount-1	Directories below the currently expanded directory.

Initially, combo boxes and list boxes contain an empty list.

When the List index is outside the range of actual entries in the list box, an empty string is returned. For example, List(-1) returns an empty string for a combo box or list box.

The contents of the initial list for file-system controls are determined by conditions that exist when the control is created at run time:

Control	Description
Directory list box	Contains a list of all directories, using the range -n to ListCount-1.
Drive list box	Contains the list of drive connections in effect.
File list box	Contains the list of files in the currently expanded directory that match the Pattern property; the path is not included.

Use the ADDITEM method to specify items you want to display in a combo box or list box, often as part of the container form's Load event procedure. To keep items in alphabetical order, set the control's Sorted property to True (-1) before adding items to the list. To remove items, use the REMOVEITEM method.

See Also ADDITEM Method, ListCount Property, ListIndex Property, REMOVEITEM Method, Sorted Property.

Example This example loads a combo box with a list of sandwich names and displays the first item in the list. Drop down the list to select a different sandwich name. To try this example, create a form containing a combo box control (size combo box to 3" in width).

```
SUB Form_Load ()
   Combo1.ADDITEM "Denver Sandwich"      ' Add each item to list.
   Combo1.ADDITEM "Reuben Sandwich"
   Combo1.ADDITEM "Turkey Sandwich"
   Combo1.Text = Combo1.List(0)          ' Display first item.
END SUB
```

ListCount Property

Applies To Combo box, directory list box, drive list box, file list box, list box.

Action Returns the number of items in the list portion of a control; not available at design time and read-only at run time. When ListCount = 0, there are no items in the list.

Syntax [*form.*]*ctlname*.**ListCount**

Remarks ListCount provides specific information for each control:

Control	Returns
Combo box	The number of items in the list.
Directory list box	The number of subdirectories below the currently expanded directory.
Drive list box	The number of drive connections.
File list box	The number of files in the currently expanded directory that match the Pattern property.
List box	The number of items in the list.

See Also ADDITEM Method, List Property, ListIndex Property, REMOVEITEM Method.

Example See AutoRedraw Property example.

ListIndex Property

Applies To Combo box, directory list box, drive list box, file list box, list box.

Action Sets or returns the index of the currently selected item in the control; not available at design time.

Syntax [*form.*]*ctlname*.**ListIndex**[= *index%*]

Part	Description
index%	Valid settings are:

Value	Description
-1 to ListCount-1	Valid index range; an error is generated when you try to access an element greater than ListCount.
0	Top item in list box.
-1	No item selected; (all list box controls except Directory list box).
-1	(Directory list box only) Index of current directory path when control is loaded at run time.

Remarks You can use the List or Text properties to return the string for the currently selected item in a list. For example:

```
ctlname.Text
ctlname.List(ctlname.ListIndex)
```

Changing ListIndex selects a new item in the list; changing ListIndex also invokes a Click Event.

ListIndex returns -1 if:

- No item is selected (all controls except directory list boxes)
- The user enters a choice in a combo box instead of selecting an existing item in the list

ListIndex is set to -1 (no item selected) immediately after an ADDITEM or REMOVEITEM operation.

Use ListIndex to track the selected entry within a control. For each different control, the initial value of the ListIndex property has a specific meaning:

Control	Setting	Meaning
Combo box	-1	No item selected.
Directory list box	-1	Index of the current path when the control is loaded at run time;
	-2, -3, and so on	Index of directories above the currently expanded directory.
Drive list box	n	Index of the current drive when the control is created at run time.
File list box	-1	No item selected.
List box	-1	No item selected.

See Also ADDITEM Method, List Property, ListCount Property, REMOVEITEM Method.

Example This example displays the names of three players in a combo box and the corresponding salary of the selected player in a label. To try this example, create a form containing a combo box and a label. Set object properties:

- For Command1, Style = 2 Simple Combo
- For Command2, AutoSize = True

```
DIM SHARED Player$(0 TO 2)          ' General declarations.
DIM SHARED Salary$(0 TO 2)          ' Dimension two arrays.

SUB Combo1_Click ()                 ' Display corresponding salary for name.
    Label1.Caption = Salary$(Combo1.ListIndex)
END SUB

SUB Form_Load ()
    Player$(0) = "Murphy McPhoo"    ' Enter data into arrays.
    Player$(1) = "Alf Hinshaw"
    Player$(2) = "Woofer Dean"
    Salary$(0) = "$234,500"
    Salary$(1) = "$158,900"
    Salary$(2) = "$1,030,500"
    FOR I% = 0 TO 2                 ' Add names to list.
        Combo1.ADDITEM Player$(I%)
    NEXT I%
    Combo1.ListIndex = 0            ' Display first item in list.
END SUB
```

Load Event

Applies To Form

Action Occurs when a form is loaded or as the result of either the LOAD Statement or an implicit load. An "implicit load" is caused by a reference to an unloaded form's properties or controls.

Syntax **SUB Form_Load()**

Remarks Typically, you use a Load event procedure to include initialization code for a form Ä for example, specifying default settings for controls, indicating contents to be loaded into combo boxes or list boxes, or initializing form-level variables.

The Load event occurs when a form is first loaded. A form can be loaded implicitly or explicitly. A form is implicitly loaded when a reference is made to any of its properties or controls. A form is explicitly loaded with the LOAD Statement.

When a form is loaded, it is automatically made visible if it is the startup form and the Visible property is set to True (-1). If the form is not the startup form, it will not be made visible when loaded, even if its Visible property is set to True (-1). To display a form after it is loaded, you must use the SHOW method or explicitly set Visible to True (-1) after the form is loaded.

After the Load event, this event ordering occurs: Resize, GotFocus, Paint. Note: The Paint event only occurs if the AutoRedraw property is set to False (0).

See Also LOAD Statement, UnLoad Event, UNLOAD Statement.

Example This example illustrates the use of the Load event by specifying the contents of a combo box on a form when it is loaded. To try this example, create a form containing a combo box.

```
SUB Form_Load ()
    Combo1.ADDITEM "Medieval to Renaissance"    ' Add items to list.
    Combo1.ADDITEM "Baroque to Classical"
    Combo1.ADDITEM "Rock 'n Roll"
    Combo1.ADDITEM "Ethnic/Folk"
    Combo1.ListIndex = 2                        ' Set default selection.
END SUB
```

LOAD Statement

Action Loads a form into memory or adds a control to a control array.

Syntax **LOAD** {*form* | *ctlname*[*index%*]}

Part	Description
form	Name of form to create or load.
ctlname	Control array to which an element will be added.
index%	Position in the control array.

Remarks

It is not necessary to use the LOAD statement with forms. Any reference to a form automatically loads it if the form is not already loaded. For example, the SHOW method loads an unloaded form before displaying it.

When an application starts, Visual Basic automatically loads and displays the application's start-up form, if one exists. The start-up form's Visible property must be set to True (-1).

Use the HIDE method on an unloaded form to load the form without showing

Use the SHOW method rather than the LOAD statement to display hidden or invisible forms.

Once a form is loaded, all run-time properties for the form and its controls can be accessed by the program, whether or not the form is actually visible.

Attempting to load an already loaded form has no effect.

When Visual Basic loads a form, it sets form properties to their initial values and then executes the Load event procedure.

Visual Basic can only load one instance of a specific form when an application is running, but it can load and display multiple instances of a control that was defined as a control array at design time. For example, you can design a form with a single check box, but define that check box as a member of an array of check boxes. At run time, this allows your application to respond to specific conditions by loading new check boxes into the array and receiving events from them.

Standard dialog boxes produced by Visual Basic functions such as MSGBOX and INPUTBOX$ are invoked by direct reference. They cannot be loaded, shown, or unloaded.

See Also

Load Event, SHOW Method, UNLOAD Statement.

Example

This example uses the LOAD and UNLOAD statements with a form.

```
DECLARE SUB LoadUnloadDemo ()

SUB Form_Click ()
   CALL LoadUnloadDemo
END SUB
```

```
SUB LoadUnloadDemo ()
   UNLOAD Form1                  ' Unload form.
   Msg$ = "Form1 has been unloaded. Choose Yes to load and "
    Msg$ = Msg$ + CHR$(13) + CHR$(10) "display the form. Choose No to load "
    Msg$ = Msg$ + CHR$(13) + CHR$(10) "the form and leave it invisible. "
   Answer% = MSGBOX(Msg$, 4)     ' Get user response.
   IF Answer% = 6 THEN           ' Evaluate answer.
     SHOW                       ' If Yes, show form.
   ELSE
     LOAD Form1                 ' If No, just load it.
     Msg$ = "Form1 is now loaded. Choose OK to display it."
     MSGBOX Msg$                ' Display message.
     SHOW                       ' Show form.
  END IF
END SUB
```

LOC Function

Action Returns the current position in an open file or device.

Syntax **LOC**(*filenumber%*)

Part	Description
filenumber%	Number of an open file or device.

Remarks The LOC function cannot be used on ISAM tables, or the SCRN, KYBD, or LPTn devices.

For a COM device, LOC returns the number of characters in the input queue waiting to be read. The value returned depends on whether the device was opened in ASCII or binary mode.

In ASCII mode, the low-level routines stop queuing characters as soon as an end of file is received. The end of file itself is not queued and cannot be read. If you attempt to read the end of file, Visual Basic generates the error message, "Input past end of file."

In binary mode, the end-of-file character is ignored and the entire file can be read. The last byte read or written is returned.

For random-access files, LOC returns the number of the last record read from or written to the file.

For sequential files, LOC returns the current byte position in the file, divided by 128.

For a PIPE device, LOC returns 1 if any data is available in the PIPE queue.

Example This example demonstrates the use of the LOC function, the SEEK function, and the SEEK statement.

```
OPEN "TEST.DAT" FOR RANDOM AS #1
FOR i% = 1 TO 10
    PUT #1, , i%
NEXT i%
SEEK #1, 2
GET #1, , i%
PRINT "Data: "; i%; " Current record: "; LOC(1); " Next: "; SEEK(1)
```

LOCATE Statement

See POS Function.

LOCK, UNLOCK Statements

Action Controls access by other processes to all or part of an opened file:

- Limits or prevents access (LOCK).
- Releases any locks (UNLOCK).

Syntax

LOCK [#]*filenumber%* [,{*record&* | [*start&*] **TO** *end&*}]

.
.
.

UNLOCK [#]*filenumber%* [,{*record&* | [*start&*] **TO** *end&*}]

Part	Description
filenumber%	Number of an open file.
record&	Any record number in the range 1 to 2,147,483,647, inclusive.
start&	Number of the first record or byte to lock or unlock.
end&	Number of the last record or byte to lock or unlock.

Remarks LOCK and UNLOCK statements are used in networked environments where several processes might need access to the same file.

LOCK and UNLOCK statements are always used in pairs. The arguments to LOCK and UNLOCK must match exactly.

Be sure to remove all locks with an UNLOCK statement before closing a file or terminating your program. Failing to remove locks produces unpredictable results.

Do not use LOCK and UNLOCK on devices or ISAM tables.

A record can be up to 32,767 bytes long.

If you specify just one record, then only that record is locked or unlocked. If you specify a range of records and omit a starting record (*start&*), then all records from the first record to the end of the range (*end&*) are locked or unlocked. LOCK with no record arguments locks the entire file, while UNLOCK with no record arguments unlocks the entire file.

For binary-mode files, the arguments *record&*, *start&*, and *end&* represent the number of a byte relative to the beginning of the file. The first byte in a file is byte 1.

For random-access files, *record&*, *start&*, and *end&* are the number of a record relative to the beginning of the file. The first record is record 1.

If the file has been opened for sequential input or output, LOCK and UNLOCK affect the entire file, regardless of the range specified by *start&* and *end&*.

If you attempt to access a file that is locked, Visual Basic may generate the error message, "Permission denied."

LOCK and UNLOCK execute only at run time if you are using a version of MS-DOS that supports networking (version 3.1 or later). You must run the SHARE.EXE program to enable locking operations.

MS-DOS version 3.0 or earlier returns an error message that reads, "Advanced feature unavailable," if LOCK and UNLOCK are executed.

Example

This example illustrates the use of the LOCK and UNLOCK statements. A sample data record is created in a random-access file containing a customer record. The program allows you to update the file, locking the file while you use it to prevent access from another terminal.

Note: To use this program, you must have MS-DOS version 3.1 or later, and you must run the MS-DOS command SHARE (SHARE.EXE) to enable locking operations before you start Visual Basic.

```
TYPE AccountRec                          ' Define the record.
   Payer AS STRING * 20
   Address AS STRING * 20
   Place AS STRING * 20
   Owe AS SINGLE
END TYPE
DIM CustRec AS AccountRec
ON ERROR GOTO ErrHandler                 ' This section creates a
                                         ' sample record to use.
```

```
OPEN "MONITOR" FOR RANDOM SHARED AS #1 LEN = LEN(CustRec)
CustRec.Payer = "George Washington"
CustRec.Address = "1 Cherry Tree Lane"
CustRec.Place = "Mt. Vernon, VA"
CustRec.Owe = 12!
PUT #1, 1, CustRec                    ' Put one record in the file.
CLOSE #1

' This sections opens the sample record for updating.
OPEN "MONITOR" FOR RANDOM SHARED AS #1 LEN = LEN(CustRec)
DO
    Number% = 0                       ' Reset to zero.
    DO UNTIL Number% = 1              ' Force user to input 1.
        CLS : LOCATE 10, 10
       INPUT "Enter Customer Number? (enter 1)"; Number%
    LOOP
                        ' Lock the current record so another process
    LOCK #1, Number%    ' doesn't change it while you're using it.
    GET #1, Number%, CustRec
    LOCATE 11, 10: PRINT "Customer: "; CustRec.Payer
    LOCATE 12, 10: PRINT "Address:  "; CustRec.Address
    LOCATE 13, 10: PRINT "Currently owes: $"; CustRec.Owe
    LOCATE 15, 10: INPUT "Change (+ or - amount)", Change!
    CustRec.Owe = CustRec.Owe + Change!
    PUT #1, Number%, CustRec
    UNLOCK #1, Number%   ' Unlock the record so others can use it.
    LOCATE 17, 10: INPUT "Update another? ", Continue$
    Update$ = UCASE$(LEFT$(Continue$, 1))
LOOP WHILE Update$ = "Y"
CLOSE #1
KILL "MONITOR"                        ' Remove file from disk.
END

ErrHandler:
IF ERR = 70 THEN                      ' Permission denied error.
    CLS
    PRINT "You must run SHARE.EXE before running this example."
    PRINT "Exit the programming environment, run SHARE.EXE, and"
    PRINT "reenter the programming environment to run this"
    PRINT "example. Do not shell to MS-DOS to run SHARE.EXE or you"
    PRINT "may not be able to run other programs until you reboot."
END IF
END
```

LOF Function

Action Returns the size of an open file (bytes), the number of records in an ISAM table, or when used with the OPEN COM statement, the number of bytes free in the output buffer (ISAM available in Professional Edition Only).

Syntax **LOF**(*filenumber%*)

Part	Description
filenumber%	Number specifying an open file or ISAM table.

Remarks LOF can be used only on disk files, ISAM tables, or COMn: devices. It cannot be used with the Basic devices SCRN, KYBD, CONS, LPTn, or PIPE.

See Also EOF Function, LEN Function, LOC Function, OPEN Statement (File I/O).

Example This example uses LOF to determine the length of a message which is to be encrypted and then decrypted.

```
CLS                                          ' Clear the screen.
DIM C AS STRING * 1
OPEN "MESSAGE.DAT" FOR OUTPUT AS #1          ' Open a text file and put
CLS                                          ' the message "Hi there" in it
INPUT "Enter a message for encryption: ", message$
PRINT #1, message$
CLOSE #1
CLS
PRINT "Encrypting MESSAGE.DAT"
PRINT "   ";
OPEN "MESSAGE.DAT" FOR BINARY AS #1          ' Encrypt the text contained within
FOR i% = 1 TO LOF(1)                         ' MESSAGE.DAT; the encryption routine
    GET #1, i%, C                            ' replaces each ASCII character with
    C = CHR$((ASC(C) + 1) MOD 256)           ' the next ASCII character, for
    PUT #1, i%, C                            ' example, A becomes B, C becomes D
    PRINT C;                                 ' Print each encrypted character.
NEXT i%
LOCATE 3, 1
PRINT "Decrypting MESSAGE.DAT"
PRINT "   ";
FOR i% = 1 TO LOF(1)                         ' Decrypt the text in MESSAGE.DAT.
    GET #1, i%, C
    PRINT CHR$((ASC(C) + 255) MOD 256);      ' Print each decrypted character.
NEXT i%
```

```
LOCATE 5, 1
PRINT "File MESSAGE.DAT contains "; LOF(1); " bytes"
CLOSE #1
END
```

LostFocus Event

Applies To

Check box, combo box, command button, directory list box, drive list box, file list box, form, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action

Occurs when an object loses the focus, either by user action — for example, when the user clicks or tabs to another control or form — or by changing the focus through the SETFOCUS method. Note: A form receives the focus only when it has no controls, or when all visible controls are disabled or invisible.

Syntax

SUB Form_LostFocus ()
SUB *ctlname*_**LostFocus ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks

Use a LostFocus event procedure to verify and validate updates. LostFocus allows validation to take place as the user leaves the control.

You can use LostFocus for enabling, disabling, hiding, and displaying other objects, as you can with a GotFocus event. Or, you can reverse or change conditions you set up in the object's GotFocus event procedure.

To enhance an application visually, set the color of a control when it gains or loses the focus.

See Also

ActiveControl Property, ActiveForm Property, GotFocus Event, SETFOCUS Method, TabIndex Property, TabStop Property.

Example

This example changes the color of a text box when it receives or loses the focus and displays the appropriate text in the label. To try this example, create a form with two text boxes and a label.

In this example, the color of a text box changes when it receives (GotFocus event) or loses (LostFocus event) the focus. In addition, each time the scroll bar is clicked, the numeric setting of the scroll bar's Value property is displayed in a text box.

To try this example, create a form containing a text box, a label, and a horizontal scroll bar.

```
SUB Text1_GotFocus ()
    ' Show focus in red.
    Text1.BackColor = 4
    Label1.Caption = "Text1 has the focus."
END SUB

SUB Text1_LostFocus ()
    ' Show non-focus in green.
    Text1.BackColor = 2
    Label1.Caption = "Text1 does not have the focus."
END SUB
```

LPOS Function

Action Returns the number of characters sent to a printer since the last carriage return was sent.

Syntax **LPOS(*n%*)**

Part	Description
n%	Number specifying a printer port:

Value	Printer Port
0	LPT1
1	LPT1
2	LPT2
3	LPT3

Remarks The LPOS function does not necessarily give the current physical position of the print head since LPOS does not expand tab characters and since some printers may buffer characters.

Example This example uses the LPOS function is used to determine the position of the print head and avoid printing past the end of the line. Note: To run this program, you must have a printer connected to LPT1.

```
CLS                                          ' Clear the screen.
LPRINT "Team Members"; TAB(76); "TEAM"
LPRINT
INPUT "How many teams"; TEAMS
INPUT "How many players per team"; PPT
PRINT
```

```
FOR T = 1 TO TEAMS
    INPUT "Team name: ", TEAM$
    FOR P = 1 TO PPT
        INPUT "   Enter player name: ", PLAYER$
        LPRINT PLAYER$;
        IF P < PPT THEN
            IF LPOS(0) > 55 THEN        ' Print a new line if print
                                        ' head past column 55.
                LPRINT : LPRINT "      ";
            ELSE
                LPRINT ", ";            ' Otherwise, print a comma.
            END IF
        END IF
    NEXT P
    LPRINT STRING$(80 - LPOS(0) - LEN(TEAM$), "."); TEAM$
NEXT T
```

LPRINT, PRINT Statements

Action

Writes formatted or unformatted data to either the printer port LPT1 (LPRINT), or to the screen or a file (PRINT). The PRINT statement is not available when forms are showing. Use the PRINT method with objects.

Syntax

LPRINT [**USING** *formatstring$;*] *expressionlist* [{; | ,}]
PRINT [**USING** *formatstring$;*] [*expressionlist*] [{; | ,}]

Part	Description
USING *formatstring$;*	String expression containing: The literal string to print. One or more format specifiers that determine the field and format of the printed strings or numbers. A semi-colon following each format string is required.
expressionlist	List of one or more numeric or string expressions to print; string literals must be enclosed in quotation marks ("").
; | ,	Specifies where the next output begins:

Character	Description
;	Print immediately after the last value.
,	Print at the start of the next print zone; zones are 14 characters wide.

Remarks PRINT within a form module is treated as the PRINT method rather than the PRINT statement.

To use the PRINT statement form within a form module to print to the screen or perform other PRINT statement specific functions, you must place the PRINT statement in a general procedure and call it from the form module. For example:

```
SUB Form_Click                          ' In form module.
     SCREEN.HIDE
     CALL MyPrint("Hello")
     SCREEN.SHOW
END SUB

SUB MyPrint (temp$)                     ' In another module: Test.BAS
    PRINT temp$
END SUB
```

If the number to print is larger than the specified numeric field, a percent sign (%) is printed in front of the number. If rounding causes the number to exceed the field, a percent sign is printed in front of the rounded number. If the number of digits specified exceeds 24, Visual Basic generates the error message, "Illegal function call."

Because the LPRINT USING statement uses the LPT1 printer device, do not use LPRINT USING in a program that also contains an OPEN "LPT1" statement. Using these two statements together produces unpredictable results.

The LPRINT USING statement functions in the same way as the PRINT USING statement, except that output goes to the printer.

Printer output from an LPRINT USING statement will be the same as screen output from a PRINT USING statement, if both statements have the same values for formatstring$, *expressionlist*, and output-line width.

The LPRINT USING statement assumes an 80-character-wide printer. This width can be changed with a WIDTH statement.

If you use LPRINT USING with no arguments, a blank line is printed.

Item-format rules:

- A printed number is always followed by a space.
- If the number is positive, it is also preceded by a space; if the number is negative, it is preceded by a minus sign (-).
- If a single-precision number can be expressed as seven or fewer digits with no loss of accuracy, then it is printed in fixed-point format; otherwise, floating-point format is used. For example, the number 1.1E-6 is displayed as .0000011, but the number 1.1E-7 is displayed as 1.1E-7.

- If a double-precision number can be expressed as 15 or fewer digits and with no loss of accuracy, then it is printed in fixed-point format; otherwise, floating-point format is used. For example, the number 1.1D-14 is displayed as .000000000000011, but the number 1.1D-15 is displayed as 1.1D-15.

Print-line format rules:

- The print line is divided into print zones of 14 spaces each.
- The position of each printed item is determined by the punctuation used to separate the items in *expressionlist*:
  - A semicolon makes the next value print immediately after the last value.
  - A comma makes the next value print at the start of the next zone.
  - Using one or more spaces or tabs between expressions has the same effect as using a semicolon.
- If a comma or a semicolon terminates the list of expressions, the next PRINT statement to execute prints on the same line, after spacing accordingly.
- If *expressionlist* ends without a comma or a semicolon, a carriage return-linefeed sequence is printed at the end of the line.
- If the printed line is wider than the screen width, Visual Basic goes to the next physical line and continues printing.

To print information from a record, use individual record element names in the PRINT statement. For example:

```
TYPE MyType
     Word AS STRING * 20
     Count AS LONG
END TYPE
DIM Myrec AS MyType
PRINT Myrec.Word
```

The printer output from the LPRINT statement will be the same as the screen output from a PRINT statement if both statements have the same *expressionlist* values and output-line width.

The LPRINT statement assumes an 80-character-wide printer. This width can be changed with a WIDTH statement.

If you use LPRINT with no arguments, a blank line is printed.

An LPRINT CHR$(13) statement outputs both CHR$(13) and CHR$(10).

Because the LPRINT statement uses the LPT1 printer device, do not use LPRINT in a program that also contains an OPEN "LPT1" statement. Using these two statements together produces unpredictable results.

See Also MSGBOX Statement, PRINT Method, PRINT # Statement, SPC Function, TAB Function.

Example This example shows the effects of using data delimiters with PRINT # and uses LPRINT to send the results to a printer.

```
CLS                                     ' Clear the screen.
A$ = "CAMERA, AUTOFOCUS": B$ = "September 20, 1985": C$ = "42"
Q$ = CHR$(34)
OPEN "INVENT.DAT" FOR OUTPUT AS #1      ' Open INVENT.DAT for writing.
PRINT #1, A$; B$; C$                    ' Write A$, B$, C$ without delimiters.
                                        ' Write A$, B$, C$ with delimiters.
PRINT #1, Q$; A$; Q$; Q$; B$; Q$; Q$; C$; Q$
CLOSE #1
OPEN "INVENT.DAT" FOR INPUT AS #1       ' Open INVENT.DAT for reading.
FOR I% = 1 TO 2                         ' Read first two records and print.
    INPUT #1, First$, Second$, Third$
    LPRINT First$; TAB(30); Second$; TAB(60); Third$: LPRINT
NEXT
CLOSE #1
```

LSET, RSET Statements

Action Moves data into a random-access file buffer (in preparation for a PUT statement) and left- or right-justifies the value of a string variable; LSET also copies the contents of one record variable to another.

Syntax

LSET *stringvariable$* = *stringexpression$*
LSET *recordvariable1* = *recordvariable2*
RSET *stringvariable$* = *stringexpression$*

Part	Description
stringvariable$	Any string variable or random-access file field defined in a FIELD statement.
stringexpression$	For RSET, the right-justified version of *stringvariable$*; for LSET, the left-justified version of *stringvariable$*.
recordvariable1 *recordvariable2*	Record variables of any user-defined data type; use LSET to assign a record variable of one data type to a different user-defined data type.

Remarks If *stringexpression$* requires fewer bytes than were defined for *stringvariable$* in the FIELD statement, the RSET function right-justifies the string in the field; LSET left-justifies the string. Spaces are used to pad the extra positions.

If the string is too long for the field, both RSET and LSET truncate characters from the right.

Numeric values must be converted to strings before they are justified with the RSET or LSET statement.

RSET or LSET can be used with a string variable not defined in a FIELD statement to right- or left-justify a string in a given field. The following example right-justifies the string N$ in a 20-character field:

```
A$ = SPACE$(20)
RSET A$=N$
```

You can use LSET to assign one record variable to another. The following example copies the contents of RecTwo to RecOne:

```
TYPE TwoString
    StrFld AS STRING * 2
END TYPE

TYPE ThreeString
    StrFld AS STRING * 3
END TYPE
DIM RecOne AS TwoString, RecTwo AS ThreeString
.
.
.
LSET RecOne = RecTwoz
```

The example above demonstrates:

- LSET copying record variables of different data types.
- LSET copying only the number of bytes in the shorter of the two record variables Note: You also can use LSET to copy record variables of the same type.

When RSET is used with a variable-length string, the string is treated as a fixed field. The length of the field is the length of the value the variable had before the RSET statement.

See Also FIELD Statement, PUT Statement (File I/O).

Example This example uses the LSET and RSET statements to assign values to fixed- and variable-length strings.

```
DIM TmpStr2 AS STRING * 25

CLS                                    ' Clear the screen.

' Use RSET on variable-length string with a value.
TmpStr$ = SPACE$(40)
PRINT "Here are two strings that have been right and left "
PRINT "justified in a 40-character variable-length string."
PRINT
RSET TmpStr$ = "Right-|"
```

```
PRINT TmpStr$
LSET TmpStr$ = "|-Left"
PRINT TmpStr$
PRINT "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^"
PRINT "12345678901234567890123456789012345678901234567890"
PRINT "         1         2         3         4         5"

' Use RSET on fixed-length string of length 25.
PRINT
PRINT
PRINT "Here are two strings that have been right and left"
PRINT "justified in a 25-character fixed-length string."
PRINT
RSET TmpStr2 = "Right-|"
PRINT TmpStr2
LSET TmpStr2$ = "|-Left"
PRINT TmpStr2$
PRINT "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^"
PRINT "12345678901234567890123456789012345678901234567890"
PRINT "         1         2         3         4         5"
```

LTRIM$, RTRIM$ Functions

Action Remove leading spaces (LTRIM$) or trailing spaces (RTRIM$) from a string.

Syntax **LTRIM$**(*stringexpression$*)
**RTRIM$**(*stringexpression$*)

Part	Description
stringexpression$	Any string expression.

Remarks Both the LTRIM$ and RTRIM$ functions work with fixed- and variable-length string variables.

Example This example uses VAL and STR$ to convert a hexadecimal number to a decimal number. The LTRIM$ and RTRIM$ functions are used to trim blanks from the resulting string.

```
CLS                                 ' Clear the screen.

' Accept all data as ASCII text and assign it to a string.
LINE INPUT "Enter hexadecimal value to be converted to decimal: "; fathex$
hexnum$ = LTRIM$(RTRIM$(fathex$)) ' Trim blanks off both sides of string.
outhex$ = UCASE$(hexnum$)         ' Convert characters to uppercase.
num# = VAL("&H" + outhex$)        ' &H tells VAL that A,B,C,to F are OK.
dnum$ = STR$(num#)                ' STR$ converts decimal back to string.
```

```
PRINT "Your input is "; "&H " + outhex$
LOCATE 19, 1
PRINT "Hex input     "; "&H " + outhex$; " equals "; dnum$; " in decimal"
PRINT "Decimal value "; dnum$; " equals "; "&H " + outhex$; " in hex"
PRINT
END
```

Max, Min Properties

Applies To Horizontal scroll bar, vertical scroll bar.

Action Determines a scroll bar position's maximum (Max) or minimum (Min) value.

Syntax [*form.*]{*hscrollbar.* | *vscrollbar.*}**Max**[= *limit%*]
[*form.*]{*hscrollbar.* | *vscrollbar.*}**Min**[= *limit%*]

Part	Description	
limit%	Valid settings are:	
	Value	**Description**
	-32,768 to 32,767	Valid change setting.
	32,767	(Default) Max.
	0	(Default) Min.

Remarks The operating environment automatically sets ranges for scroll bars proportional to the contents of forms, combo boxes, and list boxes. For a scroll bar control, however, you must specify these ranges. Use Max and Min to set a range appropriate to how the scroll bar control is used — for example, as an input device, or to indicate speed, velocity, or quantity.

Max and Min determine the Value property settings depending on the position of the scroll box and the type of scroll bar used:

Scroll Box Position	Scroll Bar	Value Setting
Top	Vertical	Max
Bottom	Vertical	Min
Far left	Horizontal	Min
Far right	Horizontal	Max

If Max is set to a value less than Min, the setting is ignored and Max is instead set equal to Min. This visually sets the scroll bar at the leftmost or topmost position of a horizontal or vertical scroll bar, respectively.

Typically, you set Max and Min at design time. You can also set them in code at run time if the scrolling range must change dynamically — for example, when adding records to a database that can be browsed with a scroll bar.

You set the maximum and minimum scrolling increments for a scroll bar control with the LargeChange and SmallChange properties.

See Also LargeChange Property, SmallChange Property, Value Property.

Example This example displays the Value, Max, and Min property settings of a horizontal scroll bar in a text box. Create a form with a Wide text box and a Horizontal scroll bar. Set scroll bar properties: Max = 1000, LargeChange = 10.

```
SUB HScroll1_Change ()
   Text1.Text = "Value=" + STR$(HScroll1.Value) + " Min=" +
   ➥STR$(HScroll1.Min) + "Max=" + STR$(Hscroll1.Max)
END SUB
```

MaxButton Property

Applies To Form.

Action Determines whether a form has a Maximize button in the upper-right corner; read-only at run time.

Syntax [*form.*]**MaxButton**

Part	Description
	Valid settings are:

Value	Description
True (-1)	(Default) Maximize button.
False (0)	No Maximize button.

Remarks A Maximize button allows users to enlarge a form to full-screen size. When a form is maximized, the Maximize button automatically becomes a Restore button. Minimizing or restoring automatically changes the Restore button back to a Maximize button.

Maximizing a form at run time generates a Resize event.

If BorderStyle is set to 0, the MaxButton value is ignored even if set to True (-1).

The WindowState property reflects the current state of the window. If you set the WindowState property to 2 (Maximized), the form is maximized independently of whatever settings are in effect for the MaxButton and BorderStyle properties.

See Also Maximize Button, MinButton Property.

Menu Control

Action Displays a list of available commands at the top of a form when the user chooses the menu title. For example, the illustration below shows a File menu with the menu items New, Open, and Exit displayed. Menus are created in the Form Designer with the Menu Design window.

Remarks

Events

Click

Properties

Caption	Checked	CtlName	Enabled
Index	Parent	Separator	Tag
Visible			

With menu controls, you can design and display a customized menu in your application. Menu items can include commands the user can choose to carry out an action, menu titles, and separator bars.

A menu control exists for each menu title and menu item. These menu controls can be named uniquely or grouped together by creating a control array. Use the LOAD Statement to add a menu item to a control array at run time.

Menu controls that remain visible on a form and make up the menu bar are called menu titles. Menu items are the menu controls that make up a menu and become visible only when a top-level menu title is selected. You can define up to five levels of menus.

Menu items are created at design time using the Menu Design window in the Form Designer. When a menu item is selected, the Click event occurs. If the item selected is top level, the DropDown event also occurs for the first item in the cascading menu. This allows you to write code to initialize the menu before it is displayed.

When a menu title is selected, the associated menu is displayed. Any code written for the menu title's Click event is executed. It is not advisable to write menu title Click event code that will cause a Paint or any other events. A menu title's Click event should only be used to change property values of lower level menu items.

Menu controls have several unique property settings that affect the appearance and accessibility of a menu item. These settings are described in the following table:

Property	Setting	Description
Caption	- (hyphen)	Item displayed as a separator bar.[1]
Caption	& (before a letter)	Letter in caption appears underlined; pressing Alt+ the letter selects this menu item.
Checked	True (-1)	Item appears with small dot (•) to the left; dot does not appear if item is top level.[2]
Enabled	True (-1)	Item receives events and displays a cascading menu (if one exists) when selected.
Enabled	False (0)	Item is unavailable; no action is performed when selected.
Visible	False (0)	Item is not displayed on the menu; any gaps are filled in by remaining menu items.
Separator	True (-1)	Item displayed as a separator bar.[1]

[1] A separator bar is created when you enter a single hyphen (-) in the Caption box or select the Separator check box in the Menu Design window.

[2] A dot (•) is displayed next to a menu item when you select the Checked check box in the Menu Design window.

MENU_BACKCOLOR, MENU_FORECOLOR Constants

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the menu background or foreground color.

Syntax

SCREEN.ControlPanel({**MENU_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({**MENU_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	MENU_BACKCOLOR has a value of 9; MENU_FORECOLOR has a value of 10.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
7	(Default - MENU_BACKCOLOR) White.
0	(Default - MENU_FORECOLOR) Black.

See Also BackColor Property, ForeColor Property, MENU_SELECTED_BACKCOLOR Constant, MENU_SELECTED_FORECOLOR Constant.

Example This example illustrates the effect of changing the value of the MENU_BACKCOLOR, MENU_FORECOLOR, MENU_SELECTED_BACKCOLOR, and MENU_SELECTED_FORECOLOR constants. To try this example, create a form containing a command button. Open the Menu Design Window and enter the following settings:

Caption	CtlName	Index	Action
File	Main	1	Press Alt+N
&Edit	Main	2	Press Alt+N, Alt+R
Cut	Edit	1	Press Alt+N
Copy	Edit	2	Press Alt+N
Paste	Edit	3	Press Alt+N

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions

SUB Command1_Click ()
    SCREEN.ControlPanel(MENU_BACKCOLOR) = 4
    SCREEN.ControlPanel(MENU_FORECOLOR) = 2
    SCREEN.ControlPanel(MENU_SELECTED_BACKCOLOR) = 2
    SCREEN.ControlPanel(MENU_SELECTED_FORECOLOR) = 4
END SUB
```

MENU_SELECTED_BACKCOLOR, MENU_SELECTED_FORECOLOR Constants

Applies To Controlpanel property, SCREEN object.

Action Sets or returns the background or foreground color of the selected menu item.

Syntax

SCREEN.ControlPanel({**MENU_SELECTED_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({**MENU_SELECTED_FORECOLOR** | *subscript%*})[= *color%*]

Part	Description
subscript%	MENU_SELECTED_BACKCOLOR has a value of 11; MENU_SELECTED_FORECOLOR has a value of 12.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
0	(Default - MENU_SELECTED_BACKCOLOR) Black.
7	(Default - MENU_SELECTED_FORECOLOR) White.

See Also BackColor Property, ForeColor Property, MENU_BACKCOLOR Constant, MENU_FORECOLOR Constant.

Example See MENU_BACKCOLOR Constant example.

MID$ Function, MID$ Statement

Action Returns part of a substring from a string (MID$ function); replaces part of a string variable with another string (MID$ statement).

Syntax

**MID$**(*stringexpression$*, *start%* [, *length%*])
**MID$**(*stringvariable$*, *start%* [, *length%*]) = *stringexpression$*

Part	Description
stringexpression$	Any string expression.
start%	Position of the first character.
length%	Number of characters in the substring; if omitted, all characters to the right of the start position are returned or replaced.
stringvariable$	String variable being modified (MID$ statement).

Remarks

The arguments *start%* and *length%* must be numeric expressions between 1 and 32,767, inclusive.

If *length%* is omitted from the MID$ function or if there are fewer than *length%* characters in the string (including the start character), the MID$ function returns all characters from the position *start%* to the end of the string.

The optional *length%* refers to the number of characters from the argument *stringexpression$* that are used in the replacement. If *length%* is omitted from the MID$ statement, all of *stringexpression$* is used. However, regardless of whether *length%* is omitted or included, the replacement of characters never goes beyond the original length of *stringvariable$*.

If *start%* is greater than the number of characters in *stringexpression$*, MID$ returns a null string.

Use the LEN function to find the number of characters in *stringexpression$*.

Example

This example uses the MID$ statement to replace string characters.

```
CLS                          ' Clear the screen.
a$ = "Where is Paris?"
PRINT MID$(a$, 10, 5)        ' Output is:  Paris
Text$ = "Paris, France"
PRINT Text$                  ' Output is:  Paris, France
MID$(Text$, 8) = "Texas "
PRINT Text$                  ' Output is:  Paris, Texas
```

Min Property

See Max Property

MinButton Property

Applies To Form.

Action Determines whether a form has a Minimize button in the upper-right corner; read-only at run time.

Syntax [*form.*]**MinButton**

Valid settings are:

Value	Description
True (-1)	(Default) Minimize button.
False (0)	No Minimize button.

Remarks A Minimize button allows users to shrink a form window to an icon.

Shrinking a form at run time generates a Resize event.

If BorderStyle is set to 0, the MinButton value is ignored even if set to True (-1).

The WindowState property reflects the current state of the window. If you set the WindowState property to 1 (Minimized), the form is minimized independently of whatever settings are in effect for the MinButton and BorderStyle properties.

See Also MaxButton Property, Minimize Button.

MINUTE Function

Action Evaluates a date/time serial number and returns the minute, a value between 0 and 59, inclusive.

Syntax **MINUTE**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See HOUR Function example.

MKDIR Statement

See CHDIR Statement.

MKx$ Functions

Action Used with FIELD and PUT statements to write numbers to a random-access file; each function converts a numeric expression of a specific type to a string value.

Syntax **MK***x*$(*numeric-expression%*)

Part	Description
x	One of the following letters, indicating the size and type of the numeric expression to convert:

Function	Expression Type	Inverse Function
MKI$	2-byte integer.	CVI
MKL$	4-byte long integer.	CVL
MKS$	4-byte single-precision.	CVS
MKD$	8-byte double-precision.	CVD
MKC$	8-byte currency.	CVC

numeric-expression%	Any valid numeric expression.

Remarks Visual Basic record variables provide a more efficient and convenient way of reading and writing random-access files than some earlier versions of Basic.

See Also CVx Functions, FIELD Statement, MKxMBF$ Functions, PUT Statement (File I/O).

Example See CVx Function example.

MKxMBF$ Functions

See CVxMBF Function.

MONTH Function

Action Evaluates a date/time serial number and returns the month, a value in the range 1 to 12, inclusive.

Syntax **MONTH**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See DATEVALUE example.

MouseDown, MouseUp Events

Applies To Directory list box, file list box, form, label, list box, picture box.

Action Occurs when the user presses a mouse button (MouseDown) or releases a mouse button (MouseUp).

Syntax

SUB Form_{MouseDown | MouseUp} (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS SINGLE)
SUB *ctlname*_**{MouseDown | MouseUp} ([Index AS INTEGER,] Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS SINGLE)**

Part	Description
Index	Uniquely identifies a control in a control array.
Button	Button that was pressed (MouseDown) or released (MouseUp) to cause the event. The Button argument is a bit field with bits corresponding to the left button (bit 0) and right button (bit 1) Ä values 1, and 2, respectively. Only one of the bits is set, indicating which button caused the event.
Shift	State of the Shift, Ctrl, and Alt keys when the button specified in the Button argument is pressed or released. A bit is set if the key is down. The Shift argument is a bit field, with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys is pressed.
X,Y	Current location of the mouse pointer, relative to the control that has captured the mouse.

Remarks Use a MouseDown or MouseUp event procedure to prescribe actions to occur when a given mouse button is pressed or released.

Unlike the Click and DblClick events, MouseDown and MouseUp events allow you to distinguish between the left and right mouse buttons.

You can write code for mouse-keyboard combinations that use the Shift, Ctrl, and Alt keyboard modifiers and the MouseMove event.

The following applies to both Click and DblClick event procedures:

- If a mouse button is pressed while the pointer is over a form or control, that object "captures" the mouse and receives all mouse events up to and including the last MouseUp event. This implies that the X, Y mouse pointer coordinates given by a mouse event may not always be in the client area of the object that receives them; however, they will be relative to the object's Top and Left property values.
- If mouse buttons are pressed in succession, the object that captures the mouse after the first press receives all mouse events until all buttons are released.

Mouse events are detected within border areas of controls that have borders. CurrentX and CurrentY values can be used to indicate when the mouse is not in the client area of the control.

You can use a MouseMove event procedure to respond to an event caused by moving the mouse. The Button argument for MouseDown and MouseUp differs from the Button argument used for MouseMove. For MouseDown or MouseUp, Button indicates exactly one button per event; for MouseMove, it indicates the current state of all buttons.

See Also Click Event, DblClick Event, MouseMove Event, ScaleHeight Property, ScaleWidth Property.

Example This example demonstrates the MouseDown and MouseUp events. When you run the example, click the form to display messages generated in the mouse events procedures above.

```
SUB Form_Load ()
    ForeColor = 8
END SUB

SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS
➥ SINGLE)
   PRINT "The MouseDown event occurred"
END SUB

SUB Form_MouseUp (Button AS INTEGER, Shift as Integer, X AS SINGLE, Y
➥AS SINGLE)
    PRINT (CHR$(13) + CHR$(10) + "The MouseUp event occurred")
END SUB
```

MouseMove Event

Applies To File list box, form, label, list box, picture box, directory list box.

Action Occurs when the user moves the mouse.

Syntax

SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS SINGLE)
SUB *ctlname*_**MouseMove ([Index AS INTEGER,] Button AS INTEGER, Shift AS INTEGER, X AS SINGLE, Y AS SINGLE)**

Part	Description
Button	State of the mouse buttons, in which a bit is set if the button is down. The Button argument is a bit field, with bits corresponding to the left button (bit 0), and right button (bit 1), values 1, and 2, respectively. Button indicates the complete state of the mouse buttons; some, all, or none of these bits can be set, indicating that some, all, or none of the buttons are pressed.
Shift	State of the Shift, Ctrl, and Alt keys; bit is set if the key is down. The Shift argument is a bit field, with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Shift indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys is pressed.
X, Y	Current location of the mouse pointer.
Index	Uniquely identifies a control in a control array.

Remarks The MouseMove event is generated continually as the mouse pointer moves across objects. Unless another object has captured the mouse, an object recognizes a MouseMove event whenever the mouse position is within its borders.

Mouse events are detected in the border areas of controls that have borders. CurrentX and CurrentY values can be used to indicate when the mouse is not in the client area of the control.

See Also Click Event, DblClick Event, ScaleHeight Property, ScaleWidth Property.

Example See MouseDown example.

MousePointer Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, **SCREEN** object, text box, vertical scroll bar.

Action Sets or returns the type of mouse pointer displayed when the mouse is over a particular form or control at run time.

Syntax {[*form.*][*ctlname.*] | **SCREEN.**}**MousePointer**[= *char%*]

Part	Description
char%	Valid settings are:

Enumerated Value	Description	ASCII Number
0	(Default) Character determined by the underlying form or control; if no mouse pointer is specified, the block shape is used (ASCII 219).	
1	Block (▮)	219
2	Cross (┼)	197
3	IBeam (I)	073
4	Icon (▪)	002
5	Size (✲)	015
6	Left arrow (←)	027
7	Size north/south (↕)	018
8	Right arrow (→)	026
9	Size east/west (↔)	029
10	Up arrow (↑)	024
11	Hourglass (X)	088
12	Down arrow (↓)	025

Remarks The MousePointer property controls the shape of the mouse pointer. Use this property when you want to cue the user to changes in mouse functionality as the pointer passes over various controls on a form or dialog box.

When set for the SCREEN object, MousePointer overrides the MousePointer settings of all individual forms and controls.

Setting SCREEN.MousePointer to 0 restores the default behavior of the mouse pointer, based on the control or form it is passing over.

When a controls MousePointer value is 0 (default), the MousePointer value is inherited from its container (i.e. frame, form, or picture box control.)

See Also MouseMove Event.

Example The example changes the mouse pointer to an hourglass (X) while hearts are drawn on a form, and then changes it back to the default mouse pointer at the end of the procedure. The MousePointer property is used in conjunction with the SCREEN object.

```
SUB Form_Click ()
   SCREEN.MousePointer = 11        ' Change pointer to hourglass.
   FOR i% = 1 TO 10                ' Draw hearts (ASCII 3).
        FOR f% = 1 TO 32000        ' Create time delay.
        NEXT f%
   PRINT CHR$(3)
   NEXT i%
   SCREEN.MousePointer = 0         ' Return pointer to default.
END SUB
```

MouseUp Event

See MouseDown Event.

MOVE Method

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Moves and sizes a form or control.

Syntax [*form.*][*ctlname.*]**MOVE** *left%*[, *top%*[, *width%*[, *height%*]]]

Part	Description
left%	Horizontal coordinate for left edge.
top%	Vertical coordinate for top edge.
width%	Width of object.
height%	Height of object.

Remarks

Only the *left%* argument is required with the MOVE method. However, to specify any other arguments, you must specify all arguments that appear in the syntax before the argument you want to specify. For example, you cannot specify *width%* without specifying *left%* and *top%*. Any trailing arguments that are unspecified remain unchanged.

When a form or control is moved, the Left, Top, Width, and Height properties are automatically set to the new values specified by the MOVE method arguments. However, the height value for combo boxes (with Style = 0 or 2) and drive list boxes cannot be changed using the MOVE method. The Height property is read-only for these controls.

Coordinates are measured in characters. Moving a form on the screen or moving a control within a frame is always relative to the origin - that is, the upper-left corner of the container (0,0). Note: A "container" is usually a form or the screen; however, it can also be a frame or picture box.

You can move a hidden form or control. For example, moving a visible picture box control with several child controls can take a long time. To get the desired visual effect quickly:

- Set the Visible property of the picture box to False (0).
- Move the container.
- Set the Visible property to True (-1).

Example

This example uses the MOVE method to move a form around on the screen and the MSGBOX function to display information to the user.

```
DECLARE SUB MoveDemo ()

SUB Form_Click ()
   CALL MoveDemo
END SUB

SUB MoveDemo ()
  Msg$ = "Choose OK to resize and move this form by "
  Msg$ = Msg$ + "changing the value of properties."
  MSGBOX Msg$                          ' Display message.
  In% = 10                             ' Set in to 10.
  Width = 4 * In%                      ' Set width.
  Height = 2 * In%                     ' Set height.
  Left = 0                             ' Set left to origin.
  Top = 0                              ' Set top to origin.
  Msg$ = "Now choose OK to resize and move this form "
  Msg$ = Msg$ + "using the MOVE Method."
  MSGBOX Msg$                          ' Display message.
  Form1.MOVE SCREEN.Width - 2 * In%, SCREEN.Height - In%, 2 * In%, In%
END SUB
```

MSGBOX Function

Action

Displays a message in a dialog box, waits for the user to choose a button, and returns a value indicating which button the user chose.

Syntax

MSGBOX(*msg$* [, *type%* [, *title$*]])

Part	Description
msg$	String expression to display in message area of dialog box.
type%	Sum of the values that describe the number and type of buttons to display; also identifies the default button:

Value	Button Display
0	(Default) OK.
1	OK, Cancel.
2	Abort, Retry, Ignore.
3	Yes, No, Cancel.
4	Yes, No.
5	Retry, Cancel.
0	First button is default.
256	Second button is default.
512	Third button is default.

title$	String expression to display in title bar of dialog box; if omitted, no title is displayed.

Remarks

The return value of MSGBOX indicates which button was pressed. If MSGBOX has a Cancel button, pressing the Esc key is the same as pressing the Cancel button. The return values for MSGBOX are:

Value	Button Pressed
1	OK
2	Cancel (or Esc)
3	Abort
4	Retry
5	Ignore
6	Yes
7	No

No other form events can occur while the MSGBOX dialog is active.

MSGBOX displays a maximum of 1,024 characters. If your message is longer, it is truncated after the 1,024th character.

Dialog width is determined by the larger of:

- Msg$ width (maximum 76).
- Title$ width (maximum 74).
- Width necessary to display buttons.

MSGBOX truncates lines automatically at the right of the dialog box. To set line breaks for multiple line messages, use CHR$(13) (carriage return) and CHR$(10) (linefeed character) before the text that begins each new line. For example:

```
Msg$ = "This is the first line of my message"
Msg$ = Msg$ + CHR$(13) + CHR$(10) + "This is the second line"
x% = MSGBOX Msg$
```

The dialog box is movable but not sizable. It is closable only if the message box type contains a Cancel button.

See Also INPUTBOX$ Function, MSGBOX Statement.

Example See DIM Statement example.

MSGBOX Statement

Action Displays a message in a dialog box.

Syntax **MSGBOX** *msg$* [, *type%* [, *title$*]]

Part	Description
msg$	String expression to display in message area of dialog box.
title$	String expression to display in title bar of dialog box; if omitted, no title is displayed.
type%	Sum of the values that describe the number and type of buttons to display; also identifies the default button:

Value	Button Display
0	(Default) OK.
1	OK, Cancel.
2	Abort, Retry, Ignore.
3	Yes, No, Cancel.

Value	Button Display
4	Yes, No.
5	Retry, Cancel.
0	First button is default.
256	Second button is default.
512	Third button is default.

Remarks No other form events can occur while the MSGBOX dialog is active.

MSGBOX displays a maximum of 1,024 characters. If your message is longer, it is truncated after the 1,024th character.

Dialog width is determined by the larger of:

- Msg$ width (maximum 76).
- Title$ width (maximum 74).
- Width necessary to display buttons.

MSGBOX truncates lines automatically at the right of the dialog box. To set line breaks for multiple line messages, use CHR$(13) (carriage return) and CHR$(10) (linefeed character) before the text that begins each new line. For example:

```
Msg$ = "This is the first line of my message"
Msg$ = Msg$ + CHR$(13) + CHR$(10) + "This is the second line"
MSGBOX$ Msg$
```

Because the MSGBOX statement does not return a value, some combinations of *type%* values may not be useful. For example, placing more than one button on the message box dialog is meaningless, because you cannot determine which one was chosen by the user.

The dialog box is movable but not sizable. It is closable only if the message box type contains a Cancel button.

See Also MSGBOX Function, INPUTBOX$ Function.

Example See DIM Statement example.

MultiLine Property

Applies To Text box.

Action Determines if a text box can accept and display multiple lines of text; read-only at run time.

Syntax [*form.*]*textbox*.**MultiLine**

Part	Description
	Valid settings are:

Value	Description
True (-1)	Allows multiple lines of text.
False (0)	(Default) Ignores carriage returns; restricts data to a single line.

Remarks A text box control with its MultiLine property set to True (-1) is referred to as a multiline text box. A multiline text box automatically wraps text extending beyond the display area.

Scroll bars may also be added to a text box using the ScrollBars property.

On a form with no default button, pressing Enter in a multiline text box moves the cursor to the next line. If a default button exists, you must press Ctrl+Enter to move to the next line.

See Also ScrollBars Property.

Example This example illustrates the MultiLine property. To try this example, create a form containing a text box.

```
SUB Form_Load ()
    Text1.Text = "This multiline text box is great because no matter"
    ➥+ "what the width of the box, text automatically wraps!"
    Text2.Text = "This text box truncates text rather than wrapping it!"
END SUB
```

NAME Statement

Action Changes the name of a disk file or directory.

Syntax **NAME** *oldfilespec$* **AS** *newfilespec$*

Part	Description
oldfilespec$	Name of an existing file.
newfilespec$	New name for the file; may include a path.

Remarks The NAME statement is similar to the MS-DOS RENAME command.

NAME can move a file from one directory to another but it cannot move a directory.

The file *oldfilespec$* must exist and the file *newfilespec$* must not be in use. Both files must be on the same drive. If you use NAME with different drive designations in the old and new file names, Visual Basic generates the error message, "Rename across disks."

If you use NAME on an open file, Visual Basic generates the run-time error message, "File already open." You must close an open file before renaming it.

Example This example uses the NAME statement to rename your README.TXT file.

```
CLS                                     ' Clear the screen.

NAME "README.TXT" AS "READMINE.TXT"     ' Change README.TXT to READMINE.TXT.
FILES                                   ' Display the files in your directory
PRINT                                   ' to confirm name change.
PRINT "README.TXT is now called READMINE.TXT"
PRINT "Press any key to continue..."
DO
LOOP WHILE INKEY$ = ""

NAME "READMINE.TXT" AS "README.TXT"     ' Change name back to README.TXT and
FILES                                   ' confirm by displaying files.
PRINT
PRINT "README.TXT is back to its original name"
```

NEWPAGE Method

Applies To PRINTER object.

Action Terminates printing of the current page and advances to the next.

Syntax **PRINTER.NEWPAGE**

Remarks NEWPAGE resets the print position to the upper-left corner of the new page.

NEWPAGE sends a Ctrl + L (^L or formfeed) sequence to the printer.

See Also ENDDOC Method, PRINT Method.

Example See ENDDOC Method example.

Normal Property

See Archive Property.

NOW Function

Action Returns a serial number representing current system date and time.

Syntax **NOW**

Example See FORMAT$ Function example.

OCT$ Function

See HEX$ Function.

ON COM Statement

Action Branches to the specified routine when characters are received at a communications port; can be used with object-specific events.

Syntax **ON COM(***n%***) GOSUB** {*linenumber* | *linelabel*}

Part	Description
n%	Number of a COM (serial) port (1 or 2).
linenumber *linelabel*	First line of an event-handling routine to execute when characters are received at a communications port.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

Use RETURN *linenumber* or RETURN *linelabel* forms of RETURN to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR." In addition, if an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

If your program receives data using an asynchronous communications adapter, use the Visual Basic (VBDOS) command-line option /C to set the size of the data buffer.

The ON COM statement specifies only the start of an event-trapping routine. The COM statement determines whether the routine is called and how events are handled when trapping is off.

See Also COM Statement, PRINTER Object.

Example See COM Statement example.

ON ERROR Statement

Action Enables error handling and, when a run-time error occurs, directs your program to either branch to an error-handling routine or to resume execution.

Syntax **ON [LOCAL] ERROR {GOTO** *line* **| RESUME NEXT | GOTO 0}**

Part	Description
LOCAL	Indicates an error-handling routine in the same procedure; if omitted, module-level error handlers are indicated.
GOTO *line*	Branches to the first line of the error-handling routine, specified by a label or line number; to disable error handling, specify GOTO 0.
RESUME NEXT	Resumes execution with the statement following the statement that caused the run-time error; use the ERR function to obtain the error code for the error.
GOTO 0	Disables any enabled module-level error-handling routine within the current module; if used with the LOCAL keyword, disables any enabled error handler in the current procedure.

Remarks If no ON ERROR statement is used in a program, any run-time error will be fatal. Visual Basic generates an error message and program execution stops.

The LOCAL keyword is used to indicate an error-handling routine that is "local" to the procedure in which the error handler is located. A local error-handling routine:

- Overrides any enabled module-level error-handling routines.
- Is enabled only while the procedure it is located in is executing.

The local error handler remains enabled while any procedures execute that are directly or indirectly called by the procedure that contains the error handler. Error handlers within those procedures will supersede the local error handler.

If an error occurs in a procedure or module that does not have an enabled error-handling routine, Visual Basic searches for an enabled, inactive error handler to trap the error.

Visual Basic will search back through all the procedures and modules that have called the procedure where the error occurred. In a multiple-module program, this search includes error handlers at the module level of different modules.

The argument *line* is either in the module-level code, or, if the LOCAL keyword is used, in the same procedure. If not found in either place, Visual Basic generates the compile-time error message, "Label not defined."

The RESUME NEXT option causes program execution to resume with the statement immediately following the statement that caused the run-time error. Use RESUME NEXT to either:

- Continue execution of a block of code despite a run-time error, then check for the cause of the error.
- Build the error-handling code in line with the main program or procedure, rather than at a remote location in the program.

ON ERROR GOTO 0 disables error handling. It does not specify line 0 as the start of the error-handling code, even if the program or procedure contains a line numbered 0.

An error-handling routine is not a SUB or FUNCTION procedure or a DEF FN function. An error-handling routine is a block of code marked by a line label or line number that ends with a RESUME statement.

An error-handling routine will not be executed unless:

- It is enabled by execution of an ON ERROR GOTO *line* statement.
- A fault in the structure of the program causes program control to branch to or fall into the error-handling code.
- An error statement is used to cause an error condition.

Once an error handler is enabled, if a run-time error that can be trapped occurs, then program control jumps to the enabled error-handling routine, and the error handler becomes "active."

An error handler is active from the time a run-time error has been trapped until a RESUME statement is executed in the handler.

Error-handling routines must rely on the value in ERR to determine the cause of the error. The error-handling routine should test or save this value before any other error can occur or before calling a procedure that could cause an error. The value in ERR reflects only the last error that occurred.

If you use the BC command line to compile a program that has been developed in the Visual Basic programming environment (VBDOS.EXE) and has error-handling routines, compile with the /E or /X option. The VBDOS Make EXE command uses these options.

See Also ERL Function, ERR Function, ERROR$ Function.

Example See END Statement example.

ON KEY Statement

Action Branches to a routine when the specified key is pressed; not available when forms are showing; cannot be used with object-specific events.

Syntax **ON KEY(***n%***) GOSUB** {*linenumber* | *linelabel*}

Part	Description
n%	Value that specifies a function key, arrow key, or user-defined key:

Value	Description
0	All keys listed (KEY(0) ON, KEY(0) OFF, and KEY(0) STOP only).
1 - 10	Function keys F1 - F10.
11	Up arrow key.
12	Left arrow key.
13	Right arrow key.
14	Down arrow key.
15 - 25	User-defined keys.
30, 31	Function keys F11 and F12.

linenumber *linelabel*	First line of an event-handling routine to execute when a function key, arrow key, or user-defined key is pressed.

Remarks Use the KeyDown, KeyPress, and KeyUp events to handle keyboard input with forms. ON KEY will not work when forms are showing.

To use the ON KEY statement with forms, you must first unload or hide any visible forms.

If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The RETURN *linenumber* or RETURN *linelabel* forms of RETURN can be used to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

Keys are processed in the following order:

- The line printer's echo-toggle key is processed. Defining this key as a user-defined key trap does not prevent characters from being echoed to the line printer when pressed.
- Function keys and the arrow keys are examined. Defining a function key or arrow key as a user-defined key trap has no effect because these keys are predefined.
- The user-defined keys are examined.

The ON KEY statement can trap any key, including Break or System Reset (Ctrl+Alt+Del). This makes it possible to prevent accidentally breaking out of a program or rebooting the machine.

The ON KEY statement specifies only the start of an event-trapping routine. The KEY statement determines whether the routine is called and how events are handled when trapping is off.

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

After a key is trapped, the information specifying which key was trapped is no longer available. This means that you cannot subsequently use an INPUT statement or INKEY$ function to find out which key caused the trap. Because you do not know which keypress caused the trap, you must create a different event-handling routine for each key you want to trap.

See Also KEY Statement (Assignment), KEY Statement (Event Trapping), KeyDown Event, KeyPress Event, KeyUp Event.

Example See Key Statement (Event Trapping) example.

ON PEN Statement

Action Branches to the specified routine when a lightpen is activated; can be used with object-specific events.

Syntax **ON PEN GOSUB** {*linenumber* | *linelabel*}

Part	Description
linenumber *linelabel*	First line of the event-handling routine to execute when the lightpen is activated.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The RETURN *linenumber* or RETURN *linelabel* forms of RETURN can be used to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

The ON PEN statement specifies only the start of an event-trapping routine. The PEN statement determines whether the routine is called and how events are handled when trapping is off.

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

See Also PEN Statement.

Example This example uses the PEN statement to enable event trapping and the PEN function to display current light pen status and position. ON PEN GOSUB passes control to the PenReport subroutine when a pen event occurs.

Note: Do not run this example if your mouse driver is enabled.

```
CLS                             ' Clear the screen.
COLOR 0, 7                      ' Set black on white.
CLS                             ' Clear the screen.
PEN ON                          ' Enable light pen.
ON PEN GOSUB PenReport
DO
    LOCATE 23, 12
    PRINT "Press the light pen against the screen to see a report."
    LOCATE 24, 17
    PRINT "  Press the Spacebar to exit the program.     ";
LOOP UNTIL INKEY$ = " "
PEN OFF                         ' Disable light pen.
COLOR 7, 0                      ' Set back to black on white.
CLS                             ' Clear the screen.
END

PenReport:
     DO
          P = PEN(3)
          LOCATE 10, 27  ' Report light pen status and get X and Y position
          PRINT "A Pen event has occurred."
          LOCATE 24, 17
          PRINT "Press ANY key to exit the Light Pen report.";
          LOCATE 12, 30
          PRINT "Light Pen is ";
```

```
        IF P THEN
             PRINT "Down"
             X = PEN(4): Y = PEN(5)
        ELSE
             PRINT "Up  "
             X = 0: Y = 0
        END IF
        LOCATE 14, 22  ' Report the X and Y position.
        PRINT "X Position ="; X; "   "
      LOCATE 14, 40
      PRINT "Y Position ="; Y; "   "
LOOP WHILE INKEY$ = ""
RETURN
```

ON PLAY Statement

Action

Branches to the specified routine when there are fewer than *queuemin%* notes in the background-music queue; can be used with object-specific events.

Syntax

ON PLAY(*queuemin%*) **GOSUB** {*linenumber* | *linelabel*}

Part	Description
queuemin%	Integer expression that is the minimum number of notes to maintain in the background music queue between 1 and 32, inclusive.
linenumber *linelabel*	First line of an event-handling routine to execute when the number of notes in the music queue is less than *queuemin%*.

Remarks

If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The RETURN *linenumber* or RETURN *linelabel* forms of RETURN can be used to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

A play-event trap occurs only when music is playing in the background. Play-event traps do not occur when music is playing in the foreground.

A play-event trap does not occur if the background-music queue has already gone from having *queuemin%* notes to (*queuemin%* - 1) notes when a PLAY ON is executed.

If *queuemin%* is a large number, event traps may occur often enough to slow down the program.

The ON PLAY statement specifies only the start of an event-trapping routine. The PLAY statement determines whether the routine is called and how events are handled when trapping is off.

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

See Also PLAY Function, PLAY Statement (Event Trapping), PLAY Statement (Music).

Example This example below illustrates use of the PLAY function.

```
Music$ = "MBT180o2P2P8L8GGGL2E-P24P8L8FFFL2D"
PLAY Music$
WHILE PLAY(0) > 5: WEND
PRINT "Just about done!"
```

ON STRIG Statement

Action Branches to the specified routine when the joystick trigger is pressed; can be used with object-specific events.

Syntax **ON STRIG(***n%***) GOSUB** {*linenumber* | *linelabel*}

Part	Description
n%	Value that specifies a joystick trigger:

Value	Description
0	Lower trigger, joystick A.
2	Lower trigger, joystick B.
4	Upper trigger, joystick A.
6	Upper trigger, joystick B.

linenumber *linelabel*	First line of an event-handling routine to execute when joystick trigger *n%* is pressed.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The RETURN *linenumber* or RETURN *linelabel* forms of RETURN can be used to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR

statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

The ON STRIG statement specifies only the start of an event-trapping routine. The STRIG statement determines whether the routine is called and how events are handled when trapping is off.

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

See Also STICK Function, STRIG Function, STRIG Statement.

Example See EVENT Statement example.

ON TIMER Statement

Action Branches to the specified routine when *n&* seconds have passed; can be used with object-specific events. The Timer control and the Timer event are easier to use with forms.

Syntax **ON TIMER**(*n&*) **GOSUB** {*linenumber* | *linelabel*}

Part	Description
n&	Numeric expression with a value between 1 and 86,400, inclusive (24 hours); the number of seconds in the time interval.
linenumber *linelabel*	First line of an event-handling routine to execute when n% seconds have passed.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

The RETURN *linenumber* or RETURN *linelabel* forms of RETURN can be used to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

The ON TIMER statement specifies only the start of an event-trapping routine. The TIMER statement determines whether the routine is called and how events are handled when trapping is off.

See Also TIMER Function, TIMER Statement.

Example This example uses the TIMER and ON TIMER statements to enable timer event trapping. The program draws a polygon with a random shape, size, and location every three seconds.

```
CLS                                 ' Clear the screen.
SCREEN 1
DEFINT A-Z
DIM X(6), Y(6)
TIMER ON                            ' Enable timer event trapping.
ON TIMER(3) GOSUB Drawpoly          ' Draw a new polygon every 3 seconds.
PRINT "Press any key to end program"
INPUT "Press <RETURN> to start", Test$
DO
LOOP WHILE INKEY$ = ""              ' End program if any key pressed.
END

Drawpoly:
     CLS                             ' Erase old polygon.
     N = INT(5 * RND + 2)            ' N is random number from 2 to 6.
     FOR I = 0 TO N
          X(I) = INT(RND * 319)      ' Get coordinates of vertices of polygon.
          Y(I) = INT(RND * 199)
     NEXT
     PSET (X(N), Y(N))
     FOR I = 0 TO N
          LINE -(X(I), Y(I)), 2      ' Draw new polygon.
     NEXT
     RETURN
```

ON UEVENT Statement

Action Defines the event handler for a user-defined event; can be used with object-specific events.

Syntax **ON UEVENT GOSUB** {*linenumber* | *linelabel*}

Part	Description
linenumber *linelabel*	First line of an event-handling routine to execute when the user-defined event occurs.

Remarks The ON UEVENT statement does not affect form- or control-specific events. Forms and controls use different event handlers. Use object-specific events with forms.

If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

Use the RETURN *linenumber* or RETURN *linelabel* forms of RETURN to return to a specific line from a trapping routine. However, any GOSUB, WHILE, or FOR statements active at the time of the trap remain active. Visual Basic may generate error messages such as "NEXT without FOR."

If an event occurs in a procedure, a RETURN *linenumber* or RETURN *linelabel* statement cannot get back into the procedure, because the line number or label must be in the module-level code.

ON UEVENT enables your program to branch to an event-handling routine when a user-defined event occurs. The event usually is a hardware interrupt (although it also can be a software interrupt).

A user-defined event acts as an interrupt once it has been defined with an ON UEVENT statement. The program does not need to poll for the event.

Once ON UEVENT and UEVENT ON have been executed, the user-defined event automatically triggers execution of the Basic routine to handle it. The program does not have to poll.

Although ON UEVENT ties an event-handling routine to a user-defined event, it does not enable the event trap. The UEVENT statement is used to enable, disable, and suspend user-defined event trapping.

To set up a hardware interrupt as a user-defined event under MS-DOS, at least two (and sometimes three) pieces of code are needed:

- The interrupt-service routine.
- An initialization routine to insert the address of the service routine into the interrupt-vector table.
- The routine your program calls to retrieve the data (if any) collected by the interrupt-service routine.

If the initialization routine "steals" an interrupt used by another service routine, the original address must be restored before your program terminates.

These routines usually are written in assembly language. However, any language whose compiler can generate interrupt-service routines and whose object code can be linked with Visual Basic may be used.

There are four steps to creating a user-defined event:

- Write an event-handling routine and add it to your Visual Basic program.
- Execute the ON UEVENT statement to specify the user event-handling routine.
- Execute the UEVENT ON statement to enable user event trapping.
- Call the interrupt-initialization routine to insert the address of the interrupt-service routine into the interrupt-vector table.

Once a user-defined event is created, your program is ready for the interrupt when it occurs. The interrupt transfers execution to the interrupt-service routine. The service routine collects and stores the data the user wants, then calls SetUEvent.

SetUEvent sets a flag checked by Visual Basic before going to the next statement (or label if executing compiled code using /W instead of /V). When the flag is set, control transfers to the event-handling routine designated in ON UEVENT.

The ON UEVENT statement specifies only the start of an event-trapping routine. The UEVENT statement determines whether the routine is called and how events are handled when trapping is off.

See Also SetUEvent Routine, UEVENT Statement.

Example This example uses the UEVENT and ON UEVENT statements to enable user- defined event trapping. The SetUEvent routine signals a user-defined event and passes control to an event-handling routine. This example is a limited illustration of UEVENT statements, which are best suited for interfacing hardware to a computer.

Note: To use this program or any user-defined event trapping, you must load the Quick library VBDOS.QLB with the /L option when you start Visual Basic.

```
CLS                                ' Clear the screen.
PRINT "This program asks for 10 numbers between 0 and 9, inclusive."
PRINT "As each number is input, it is evaluated to determine whether"
PRINT "it is odd or even. Odd numbers cause a user-defined event"
PRINT "to occur. Even numbers do not.": PRINT
ON UEVENT GOSUB Event1
UEVENT ON
DO
    PRINT "Enter a number --> ";
    N = VAL(INPUT$(1)): PRINT N: PRINT
    SELECT CASE N
         CASE 1, 3, 5, 7, 9
              PRINT "An odd number was input causing a user-defined event.";
    CALL SetUEvent
         CASE ELSE
              PRINT "No user-defined event occurred.": PRINT
   END SELECT
   LoopCount = LoopCount + 1
LOOP UNTIL LoopCount = 10
END

Event1:
     PRINT "Now processing the UEVENT. The odd number was"; N
     PRINT
     RETURN
```

ON...GOSUB, ON...GOTO Statements

Action Branches to one of several specified lines, depending on the value of an expression. For a more powerful, convenient, and flexible way to do multiple branches, use the SELECT CASE statement.

Syntax

ON *expression%* **GOSUB** *line-label-list*
ON *expression%* **GOTO** *line-label-list*

Part	Description
expression%	Any numeric expression in the range 0 to 255, inclusive.
line-label-list	Set of labels or line numbers; if the value of *expression%* is 1, the program branches to the first line in the list; if *expression%* is 2, the program branches to the second line, and so on.

Remarks The value of *expression%* should be greater than or equal to 1 and less than or equal to the number of items in the list. If the value falls outside this range, one of the following results occurs:

Value	Result
Number equal to 0 or greater than number of items in list.	Control drops to the next Visual Basic statement.
Negative number or number greater than 255.	Visual Basic generates the error message, "Illegal function call"

You can mix line numbers and labels in the same list.

ON...GOTO accepts a maximum of 60 line labels and line numbers.

See Also SELECT CASE Statement.

OPEN COM Statement

Action Opens and initializes a communications channel for input or output (I/O); must be executed before a device can be used for communication using an RS232 interface.

Syntax

OPEN "COM*n***:** [*speed*] [,[*parity*] [,[*data*] [,[*stop*]]]]" [**FOR** *mode*] **AS** [#]*filenum%* [**LEN** = *reclen%*]

Part	Description
n	Communications port to open: 1 = COM1 2 = COM2
speed	Baud rate (bits per second). Allowed values: 75, 110, 150, 300, 600, 1200, 1800, 2400, 9600. Default: 300.
parity	Method of parity checking (none, even, odd, space, mark, error checking) Allowed values: N, E, O, S, M, PE. Default: E.
data	Number of data bits per byte. Allowed values: 5, 6, 7, or 8. Default: 7.
stop	Number of stop bits. Allowed values: 1, 1.5, or 2[1]. Default: 1.
mode	Keyword specifying mode: **OUTPUT** (sequential output mode.) **INPUT** (sequential input mode.) **RANDOM** (default — random-access mode.)
filenum%	Any unused file number between 1 and 255, inclusive.
reclen%	Size of a random-access-mode buffer; default is 128 bytes.

[1] The default value is 1 for baud rates > 110. For baud rates <= 110, the default value is 1.5 when data is 5; otherwise, the value is 2.

Remarks

If there is no OPEN COM statement in your program, you can reduce the .EXE file size by linking with the stub file NOCOM.OBJ.

Because Visual Basic uses complete bytes (8 bits) for numbers, you must specify 8 data bits when transmitting or receiving numeric data.

Additional options require comma placeholders even if all of the options are defaults. For example:

```
OPEN "COM1: ,,,,CD1500" FOR INPUT AS #1
```

If there are syntax errors in the OPEN COM statement, Visual Basic generates the error message, "Bad file name."

Specifying the argument *reclen%* is effective only in random-access mode. You can use any of the random-access I/O statements, such as GET and PUT, to treat the device as if it were a random-access file.

The OPEN COM statement performs the following steps in opening a communications device:

1. The communications buffers are allocated and interrupts are enabled.
2. The Data Terminal Ready line (DTR) is set high.

for the Data Set Ready line (DSR) to go high. If a timeout occurs, the process goes to step 6.

4. If the RS option is not specified, the Request To Send line (RTS) is set high.
5. If either of the OP or CD options is nonzero, the statement waits for the timeout period for the Data Carrier Detect line (DCD) to go high. If a timeout occurs, the process goes to step
6. Otherwise, the RS232 device has been successfully opened. 6. If there is a timeout, the open fails. The process deallocates the buffers, disables interrupts, clears all of the control lines, and generates the message, "Device timeout." In addition, for MS-DOS, the process sets the value of ERDEV$ to COM and sets ERDEV to a value that indicates the signal line that timed out, according to the following table:

ERDEV Value	Signal Line
128 (80H)	Clear to Send (CTS) timeout.
129 (81H)	Data Set Ready (DSR) timeout.
130 (82H)	Data Carrier Detect (DCD) timeout.

Note: Only the baud rates shown are supported. Any other value for speed is invalid. The PE parity value turns on error checking for parity errors. If you set data to eight bits per byte, you must specify no parity (N).

In addition to speed, parity, data, and stop, OPEN COM supports the options described below. Options can be specified in any order, and must be separated from one another by commas. There are three types of options: data mode, buffer size, and handshaking:

Data-mode options (ASC, BIN, and LF)

Option	Description
ASC or **BIN** (default)	Specifies treatment of tabs, carriage returns, and Ctrl+Z in the data stream: Tabs expanded to blanks (ASC = yes, BIN = no). CR forced at end of line (ASC = yes, BIN = no). Ctrl+Z means end-of-file (ASC = yes, BIN = no). Ctrl+Z sent when device closed (ASC = yes, BIN = no).
LF (ignored as part of BIN default)	Effective only with the ASC option. Used to print communication files on a serial line printer. Causes a linefeed character (0AH) to be sent automatically after each carriage return (0DH): The carriage return can be sent as a a result of the width setting for the device. INPUT and LINE INPUT, when used from a COM file opened with the LF option, ignore the linefeed character.

Buffer-size options for sequential modes (RB and TB)

Option	Description
RB[*n*]	Receive buffer size (in bytes). If the RB option is not used or n is omitted, the default is 512 bytes, unless overridden by the /C option in the C or VB command line. Maximum receive buffer size is 32,767 bytes.
TB[*n*]	Transmit buffer size (in bytes). If the TB option is not used or n is omitted, default is 512 bytes (max is 32,767).

Handshake and timing options (RS, CD, CS, DS, and OP)

Option	Description
RS	Suppresses detection of Request To Send (RTS).
CD[*m*]	Specifies the timeout period on the Data Carrier Detect line (DCD). If no signal appears on the DCD line (the DCD line remains low) for more than m milliseconds, a device timeout occurs. If a CD timeout occurs, ERDEV contains 130 (82H). The range for *m*, if specified, is 0-65,535 milliseconds, inclusive. The default value is 0, if the CD option is not used, or if *m* is omitted. When *m* is 0, either by default or assignment, it means that the state of the DCD line is to be ignored.
CS[*m*]	Specifies the timeout period on the Clear To Send line (CTS). If no signal appears on the CTS line (the CTS line remains low) for more than m milliseconds, a device timeout occurs. If a CS timeout occurs, ERDEV contains 128 (80H). The range for *m*, if specified, is 0-65,535 milliseconds, inclusive. The default value is 1,000 milliseconds, if the CS option is not used, or if *m* is omitted. When *m* is 0, either by default or assignment, it means the state of the CTS line is to be ignored.

Handshake and timing options (RS, CD, CS, DS, and OP) (*continued*)

Option	Description
DS[*m*]	Specifies the timeout period on the Data Set Ready line (DSR). If no signal appears on the DSR line (the DSR line remains low) for more than m milliseconds, a device timeout occurs. If a DS timeout occurs, ERDEV contains 129 (81H). The range for *m*, if specified, is 0-65,535 milliseconds, inclusive. The default value is 1,000 milliseconds, if the **DS** option is not used, or if *m* is omitted. When *m* is 0, either be default or assignment, it means that the state of the DSR lines is to be ignored.
OP[*m*]	Specifies how long the OPEN statement waits for all communications lines to become active: The range for m is 0-65,535 milliseconds inclusive. The default value for *m*, if the OP option is not used, is 10 times the CD or DS timeout value, whichever is greater. If OP is specified, but with *m* omitted, OPEN COM waits for 10 seconds. Use a relatively large value for the OP option compared to the CS, DS, or CD options.

Use a relatively large value for the OP option compared to the CS, DS, or CD options.

See Also OPEN Statement (File I/O).

Example See COM Statement example.

OPEN Statement (File I/O)

Action Enables I/O to a file, device, or ISAM table (ISAM available in Professional edition only.)

Syntax **OPEN** *file$* [**FOR** *mode*][**ACCESS** *access*][*lock*] **AS** [#]*filenumber%* [**LEN**=*reclen%*]
OPEN *database$* **FOR ISAM** *tabletype tablename$* **AS** [#]*filenumber%*

Part	Description
file$	File name (may include path).
database$	ISAM database file name (may include path).
mode	**RANDOM** | **BINARY** | **INPUT** | **OUTPUT** | **APPEND**
tabletype	Name of a type of table defined using TYPE.
tablename$	ISAM table name.
access	**READ** | **WRITE** | **READ WRITE**

Part	Description
lock	**SHARED | LOCK READ | LOCK WRITE | LOCK READ WRITE**
filenumber%	Integer expression with a value between 1 and 255, inclusive.
reclen%	For random-access files, the record length; for sequential files, the number of characters buffered.

Remarks

The arguments *file$* and *database$* specify an optional device, followed by a file name or path conforming to the MS-DOS naming conventions. The argument *database$* must be the name of an ISAM file. If *file$* or *database$* do not exist, they will be created.

The argument *tabletype* is the name of a user-defined type that defines a subset of the table definition. The argument *tablename$* is the name of the ISAM table being opened. It follows the ISAM naming conventions.

The argument *mode* is a keyword specifying one of the following file modes:

Keyword	Description
APPEND	Sequential output mode and sets the file pointer to the end of file and the record number to the last record of the file. A PRINT # or WRITE # statement then extends (appends to) the file.
BINARY	Specifies binary-file mode. In binary mode, you can read or write information to any byte position in the file using GET and PUT. In binary mode, if no ACCESS clause is present, three attempts are made to open the file. The attempts follow the same order as those for random files.
INPUT	Sequential input mode.
OUTPUT	Sequential output mode.
RANDOM	(Default) Specifies random-access file mode, the default mode. In random mode, if no ACCESS clause is present, three attempts are made to open the file when the OPEN statement is executed. Access is attempted in the following order: Read/write. Write-only. Read-only.

The argument *access* is a keyword that specifies the operation performed on the opened file:

Type	Description
READ	Opens the file for reading only.
WRITE	Opens the file for writing only.
READ WRITE	Opens the file for both reading and writing. This mode is valid only for random and binary files and files opened for append.

The *lock* argument works in a multiprocessing environment to restrict access by other processes to an open file. It can be one of the following keywords specifying the lock type:

Keyword	Description
SHARED	Any process on any machine can read from or write to this file. Do not confuse the shared lock type with the SHARED statement or the shared attribute that appears in other statements.
LOCK READ	No other process is granted read access to this file. This access is granted only if no other process has a previous read access to the file.
LOCK WRITE	No other process is granted write access to this file. This lock is granted only if no other process has a previous write access to the file.
LOCK READ WRITE	No other process is granted either read or write access to this file. This access is granted only if read or write access has not already been granted to another process, or if a lock read or lock write is not already in place.

If you do not specify a lock type, the file can be opened for reading and writing any number of times by this process, but other processses are denied access to the file while it is opened.

The *filenum%* argument is an integer expression whose value is between 1 and 255, inclusive. When an OPEN is executed, the file number is associated with the file as long as it is open. Other I/O statements can use the number to refer to the file.

The argument *reclen%* is an integer expression less than or equal to 32,767 bytes. It specifies different settings for random-access or sequential files:

File Mode	Description
Random access	*reclen%* sets the record length (the number of characters in a record); default is 128 bytes.
Sequential	*reclen%* specifies the number of characters to be loaded into the buffer before the buffer is written to, or read from, the disk. A larger buffer means more room taken from Visual Basic, but faster file I/O; a smaller buffer means more room in memory for Visual Basic, but slower I/O. Default is 512 bytes.

The LEN clause and *reclen%* are ignored if the file mode is binary.

You must open a file before any I/O operation can be performed on it. OPEN allocates a buffer for I/O to the file or device and determines the mode of access used with the buffer.

If the file is already opened by another process and the specified type of access is not allowed, the OPEN fails and Visual Basic generates the error message, "Permission denied."

The ACCESS clause works in an OPEN statement only if you are using a version of MS-DOS that supports networking (DOS version 3.0 or later). In addition, you must run the SHARE.EXE program (or the network startup program must run it) to perform any locking operation. If ACCESS is used with OPEN, earlier versions of MS-DOS return the error message, "Advanced feature unavailable."

For sequential files, *reclen%* need not correspond to an individual record size, because a sequential file may have records of different sizes.

The following devices are supported by Visual Basic and can be named and opened with the file argument:

```
KYBD:, SCRN:, COMn:, LPTn:, CONS:, PIPE:.
```

The Visual Basic file I/O system allows you to take advantage of user-installed devices. (See your MS-DOS manual for information on character devices.)

Character devices are opened and used in the same manner as disk files. However, characters are not buffered by Visual Basic as they are for disk files. The record length for the device files is set to one.

Visual Basic sends only a carriage return at the end of a line. If the device requires a linefeed, the driver must provide it. When writing device drivers, keep in mind that other Visual Basic users will want to read and write control information. Writing and reading of device-control data is handled by the IOCTL statement and IOCTL$ function.

None of the Visual Basic devices directly supports binary mode. However, line printer devices can be opened in binary mode by adding the BIN keyword:

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

Opening a printer in BIN mode eliminates printing a carriage return at the end of a line.

In input, random, and binary modes you can open a file under a different file number without first closing the file. In output or append mode, you must close a file before opening it with a different file number.

When you open an ISAM table, the next record is the first record in the table and the current index is the NULL index.

Any ISAM operation that closes a table causes transactions to be committed. For example, if a type mismatch occurs while you are opening an ISAM table, the table is closed and a pending transaction is committed.

You may wish to code your programs so they first open all tables, then perform all transactions, then close tables. Make sure any operation that can close a table occurs outside a transaction.

See Also OPEN COM Statement, CLOSE Statement, FREEFILE Function, TYPE Statement.

Example See CLOSE Statement example.

OPTION BASE Statement

Action Sets the default lower boundary for array subscripts.

Syntax **OPTION BASE {0 | 1}**

Remarks The TO clause in the DIM statement provides an easier, more flexible way to control the range of an array's subscripts.

The default base is 0. If the lower bound of an array subscript is not explicitly set, OPTION BASE can be used to change the default lower bound to 1. For example, if the following statement is encountered, the default lowest value of an array subscript is 1:

```
OPTION BASE 1
```

The OPTION BASE statement is never required. If it is used, however, it must precede any array declarations.

The OPTION BASE statement can be used only once in a module (source file) and can only appear in the module-level code.

Chained programs can have an OPTION BASE statement if no arrays are passed in COMMON between them in a COMMON block, or if the specified base is identical in the chained programs. The chained-to program inherits the OPTION BASE value of the chaining program if OPTION BASE is omitted in the latter.

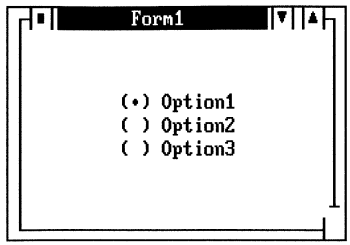
See Also LBOUND Function, Option Button Control, OPTION EXPLICIT Statement, UBOUND Function.

Example This example uses the OPTION BASE statement to override the default base array subscript value of 0. Subscripts in array A range from 1 to 20 rather than 0 to 20.

```
CLS                                ' Clear the screen
OPTION BASE 1
DIM A(20)
PRINT "The base subscript in array A is"; LBOUND(A)
PRINT "The upper bound subscript in array A is"; UBOUND(A)
```

Option Button Control

Action Represents an option in a group that the user can select or clear; only one option can be selected from a group.



Remarks

Events

Click	DblClick	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
LostFocus			

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	Caption	CtlName	DragMode
Enabled	ForeColor	Height	Index
Left	MousePointer	Parent	TabIndex
TabStop	Tag	Top	Value
Visible	Width		

An option button, sometimes called a radio button, displays an option that can be turned on or off. Option buttons are often used as part of an option group to display multiple choices to the user. Option buttons are automatically grouped together when you create and place them in the same container.

To group option buttons in a frame or picture box, first position the frame or picture box on the form and then place the option buttons inside. This grouping allows you to move the frame and the controls together instead of separately.

While option buttons and check boxes appear to function similarly, there is an important difference: When a user selects an option button, other option buttons in the same group are automatically deselected. By contrast, any number of check boxes in a group can be selected.

The minimum size of an option button is 1 x 3; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value.

See Also Check Box Control.

Example See Frame Control example.

OPTION EXPLICIT Statement

Action Requires that all variables be declared prior to use; variables can be declared using COMMON, CONST, DEF FN, DIM, FUNCTION, REDIM, SHARED, STATIC, or SUB statements.

Syntax **OPTION EXPLICIT**

Remarks This statement must precede executable code at the module level. For best results, place this statement in an include file. Use the $INCLUDE metacommand at the top of each module when you want access to the file containing the OPTION EXPLICIT statement.

See Also COMMON Statement, CONST Statement, DEF FN Statement, DIM Statement, FUNCTION Statement, REDIM Statement, SHARED Statement, STATIC Statement, OPTION BASE Statement.

Paint Event

Applies To Form, picture box.

Action Occurs when part or all of a form or control has been exposed because it has been moved or enlarged or because a window previously covering it has been moved; also invoked when the REFRESH method is used.

Syntax **SUB Form_Paint ()**
SUB *ctlname*_**Paint ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks Your application may need to redraw data in a newly uncovered area. This applies only when a related property, AutoRedraw, is set to False (0), the default setting. When AutoRedraw is set to True (-1), then repainting or redrawing is automatic; no Paint events occur.

A Paint event procedure is useful if you have output from the PRINT method in your code. With a Paint procedure, you can ensure that such output is repainted when necessary.

Using a Paint event procedure for certain tasks can cause a cascading event. Avoid using a Paint event for:

- Moving or sizing the current form or control.
- Changing variables that affect size or visual appearance, such as setting an object's BackColor property.
- Invoking the REFRESH method on the current form or control.
- Note: A Resize event procedure may be more appropriate for some of these tasks, but only when objects are being resized larger. Putting a REFRESH method in a Resize event procedure causes repainting every time a form is resized.

See Also AutoRedraw Property, PAINT Statement, REFRESH Method.

Example This example illustrates the Paint event. When the form is resized, the Paint event automatically occurs.

```
SUB Form_Paint ()
    PRINT "Paint event occurred"
END SUB
```

PAINT Statement

Action Fills a graphics area with a specified color or pattern; not available when forms are showing.

Syntax **PAINT** [**STEP**] (*x!*,*y!*)[,[{*color%* | *tile$*}] [,[*bordercolor&*] [, *background$*]]]

Part	Description
STEP	Specifies that coordinates are relative to the current graphics cursor position.
(*x!*,*y!*)	Screen coordinates where painting begins.
color%	Color attribute that sets the fill color.
tile$	Fill pattern 8 bits wide and up to 64 bytes long, defined as: `tile$ = CHR$(arg1) + CHR$(arg2) + ... + CHR$(argn%)` where arguments to CHR$ are numbers between 0 and 255, inclusive.
bordercolor&	Color attribute that specifies the color of the filled area's border.
background$	Specifies 1-byte, 8-pixel "background tile slice."

Remarks

You must be in a graphics screen mode (for example, screen mode 1) to use the PAINT statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

The available color attributes for PAINT depend on your graphics adapter and the screen mode set by the most recent SCREEN statement.

The coordinates (*x!*,*y!*) must be inside or outside a figure, not on the border itself. If this point is inside, the figure's interior is painted; if the point is on the outside, the background is painted.

The argument *background$* is a string value giving the "background tile slice" to skip when checking for boundary termination. Painting stops when adjacent points display the paint color. Specifying a background tile slice allows you to paint over an already painted area.

If you specify more than two consecutive bytes that match the tile string in the background tile slice, Visual Basic generates the error message, "Illegal function call."

When you omit *background$* in a PAINT statement, the default is CHR$(0).

Painting is complete when a line is painted without changing the color of any pixel; in other words, when the entire line is equal to the paint color. The PAINT statement permits coordinates outside the screen or viewport.

PAINT stops filling an area when it encounters a border of the specified color.

"Tiling" is the design of a PAINT pattern represented in string form; tiling can be done to produce varied patterns of different colors. The tile string is 8 bits wide and up to 64 bytes long. Each byte masks 8 bits along the x axis when plotting points. Use the following syntax to construct the tile mask:

```
A$ = CHR$(arg1) + CHR$(arg2) + ... + CHR$(argn)
PAINT (x,y), A$
```

The arguments to CHR$ are numbers between 0 and 255, represented in binary form across the x axis of the tile. Each CHR$(argn) defines a 1-byte, 8-pixel slice of the fill pattern based on the binary form of the number. There can be up to 64 of these CHR$ elements; each element generates an image of the bit arrangement of the code for that character, rather than an image of the assigned character.

For example, the decimal number 85 is binary 01010101. The graphic image line on a black-and-white screen generated by CHR$(85) is an 8-pixel line, with even-numbered points white and odd-numbered points black:

- Each bit equal to 1 turns the associated pixel on.
- Each bit equal to 0 turns the associated bit off. Note: The ASCII character CHR$(85), which is U, is not displayed in this case.

See Also

CHR$ Function, Paint Event.

Example

This example uses PAINT to create a magenta fish with a cyan tail. Note: This example requires a color graphics adapter.

```
 CONST PI = 3.1415926536#
CLS                                  ' Clear the screen.
SCREEN 1
CIRCLE (190, 100), 100, 1, , , .3    ' Outline fish body in cyan.
CIRCLE (265, 92), 5, 1, , , .7       ' Outline fish eye in cyan.
PAINT (190, 100), 2, 1               ' Fill in fish body with magenta.
LINE (40, 120)-STEP(0, -40), 2       ' Outline tail in magenta.
LINE -STEP(60, 20), 2
LINE -STEP(-60, 20), 2
PAINT (50, 100), 1, 2                ' Paint tail cyan & draw gills in black.
CIRCLE (250, 100), 30, 0, PI * 3 / 4, PI * 5 / 4, 1.5
FOR Y = 90 TO 110 STEP 4
   LINE (40, Y)-(52, Y), 0          ' Draw comb in tail.
NEXT
DO
LOOP WHILE INKEY$ = ""               ' Press any key to continue.
SCREEN 0
```

PALETTE, PALETTE USING Statement

Action

Changes the color assignments of color attributes in the current screen mode; works only on systems equipped with EGA, VGA, and MCGA adapters.

Syntax

PALETTE [*attribute%*, *color%*]
PALETTE USING *arrayname*[(*index%*)]

Part	Description
attribute%	Color attribute to change.
color%	Color value to assign to an attribute.
arrayname	Array of color values to assign to the current screen mode's set of attributes; must be large enough to assign colors to all the attributes.
index%	Index of the first array element to assign to an attribute.

Remarks

The argument *color%* must be a long integer expression for the IBM Video Graphics Array adapter (VGA) and IBM Multicolor Graphics Array adapter (MCGA) in screen modes 11 to 13. Integer or long integer expressions can be used with the IBM Enhanced Graphics Adapter (EGA).

The following table lists attribute and color ranges for various adapter types and screen modes:

Screen Mode	Monitor Attached	Adapter	Attribute Range	Color Range	PALETTE Supported
0	Monochrome	MDPA	0-15		No
	Monochrome	EGA	0-15	0-2	Yes
	Color	CGA	0-15		No
	Color/Enhanced[1]	EGA	0-15	0-15/0-63	Yes
	Analog	VGA	0-15	0-63	Yes
	Color/Analog	MCGA	0-15	0-63	No
1	Color	CGA	0-3	0-15[2]	No
	Color/Enhanced[1]	EGA	0-3	0-15	Yes
	Analog	VGA	0-3	0-15	Yes
	Color/Analog	MCGA	0-3	0-15	No
2	Color	CGA	0-1		No
	Color/Enhanced[1]	EGA	0-1	0-15	Yes
	Analog	VGA	0-1	0-15	Yes
	Color/Analog	MCGA	0-1	0-15	No
3	Monochrome	HGC	0-1		No
4	Color/Enhanced	OCGA[7] OEGA[7] OVGA[7]	0-1	0-15[3]	No
7	Color/Enhanced[1]	EGA	0-15	0-15	Yes
		VGA	0-15	0-15	Yes
8	Color/Enhanced[1]	EGA	0-15	0-15	Yes
		VGA	0-15	0-15	Yes
9	Enhanced[1]	EGA[4]	0-3	0-63	Yes
	Enhanced[1]	EGA[5]	0-15	0-63	Yes
	Analog	VGA	0-16	0-63	Yes
0	Monochrome	EGA	0-3	0-8	Yes
	Analog	VGA	0-3	0-8	Yes

Screen Mode	Monitor Attached	Adapter	Attribute Range	Color Range	PALETTE Supported
1	Analog	VGA	0-1	0-262,143[6]	Yes
	Analog	MCGA	0-1	0-262,143[6]	Yes
2	Analog	VGA	0-15	0-262,143[6]	Yes
3	Analog	VGA	0-255	0-262,143[6]	Yes
	Analog	MCGA	0-255	0-262,143[6]	Yes

[1] IBM Enhanced Color Display.

[2] Color range available for attribute 0 only.

[3] Color range available for attribute 1 only.

[4] With 64K of EGA memory.

[5] With more than 64K of EGA memory.

[6] Range of display colors is actually from 0 to 4,144,959, but only 262,144 of these can be displayed.

[7] Olivetti CGA, EGA and VGA.

The SCREEN statement lists the colors available for various screen modes, monitor, and graphics adapter combinations.

The argument *arrayname* is an array that contains more than one display color. It must be a long integer array for VGA and MCGA adapters in screen modes 11 to 13. Otherwise, the array can be either an integer or long-integer array.

The argument *index%* is the index of the first array element to use in setting the palette.

The PALETTE statement provides a way of mapping display colors (the actual binary values used by the adapter) to color attributes (a smaller set of values).

When a program enters a screen mode, the attributes are set to a series of default color values.

In the EGA, VGA, and MCGA adapters the default values have been selected so the display shows the same colors, even though the EGA uses different color values. Use the PALETTE statement to assign different colors to the attributes.

Use a PALETTE statement with no arguments to set the palette back to the default color values.

After a PALETTE statement is executed with arguments, the graphics adapter uses the display color associated with the attribute whenever an attribute argument appears in a statement such as DRAW or LINE.

Changing the display color assigned to an attribute changes those colors currently displayed on the screen associated with that attribute. For example, assume that the current palette contains colors 0, 1, 2, and 3 in the four attributes numbered 0, 1, 2, and 3. The

following DRAW statement selects attribute 3, and draws a line of 100 pixels using the display color associated with attribute 3, in this case also 3:

```
DRAW "C3L100"
```

If the following statement is executed, the color associated with attribute 3 is changed to color 2:

```
PALETTE 3,2
```

All text or graphics currently on the screen displayed using attribute 3, including the line that is 100 pixels long, are instantaneously changed to color 2. Text or graphics subsequently displayed with attribute 3 are also displayed in color 2. The new palette of colors contains 0, 1, 2, and 2.

Attribute 0 is always the screen background color. Under a common initial palette setting, points colored with the attribute 0 appear black on the display screen. For example, you can use the PALETTE statement to change the mapping of attribute 0 from black to white.

Use the COLOR statement to change the screen background color in modes 1 and 7-10.

The 64 EGA colors are derived from four levels each of red, green, and blue. For example, black is composed of red, green, and blue levels of (0,0,0), bright white is (3,3,3), dark gray is (1,1,1), and so on.

The best way to see the Visual Basic color code (0-63) associated with each combination of red, green, and blue levels is to run the following program:

```
' Display the EGA color codes 1 through 63
' using color code 0 (black) as background.
DEFINT A-Z
SCREEN 9                        'Establish EGA screen mode.
' Display a set of nine color bars.
FOR CC% = 1 TO 9
  LINE (((CC% * 64) - 24), 40)-STEP(60, 140), CC%, BF
NEXT CC%
 ' Display seven sets of nine color bars.
 ' A new set is displayed each time user presses a key.
FOR Set% = 0 TO 6
  FOR ColorBar% = 1 TO 9
    DisplayCode% = (Set% * 9) + ColorBar%
    LOCATE 15, (ColorBar% * 8)
    PRINT DisplayCode%
    PALETTE ColorBar%, DisplayCode%
  NEXT ColorBar%
  SLEEP
NEXT Set%
END
```

To calculate a VGA color value, select the intensities of red, green, and blue. The intensity of a color is a number from 0 (low intensity) to 63 (high intensity). Then use the following formula to calculate the actual color number:

```
color number = 65536 * blue + 256 * green + red
```

This formula yields integer values from 0 to 4,144,959; however, since there are gaps in the range of color numbers, it's better to use the formula rather than just select a number.

When used with the IBM Analog Monochrome Monitor, the VGA color values are converted to a gray-scale value by taking a weighted sum of the red, blue, and green intensities:

```
gray value = 11% blue + 59% green + 30% red
```

For example, if the blue, green, and red intensities are 45, 20, and 20, then the gray intensity value calculation would be:

```
gray value = (.11 * 45) + (.59 * 20) + (.30 * 20) = 22
```

Note: The fractional part of the result is dropped.

All entries in the palette can be modified in one PALETTE USING statement. Each attribute in the palette is assigned a corresponding color from the array.

A color argument of -1 in the array leaves the attribute unchanged in the palette. All other negative numbers are invalid values for color.

The dimensions for the array must be large enough to set all the palette entries after *index%*. For example, if you are assigning colors to a palette with 16 attributes and the argument *index%* is 5 (the first array element to use in resetting the palette), then the dimensions for the array must be large enough to hold at least 20 elements (because the number of elements from 5 to 20, inclusive, is 16):

```
DIM PAL%(20)                ' Other statements can go between
.                           ' DIM and PALETTE USING.
.
.
PALETTE USING PAL%(5)
```

Example

This example illustrates the use of the PALETTE and PALETTE USING statements. Note: This example requires a color graphics adapter.

```
' Using circles.
DEFINT A-Z
DIM SHARED PaletteArray(15)
CONST ASPECT = 1 / 3
SCREEN 8                          ' 640 x 200, 16 color resolution.
FOR i = 0 TO 15                   ' Initialize PaletteArray.
    PaletteArray(i) = i
NEXT i
FOR ColorVal = 15 TO 1 STEP -1    ' Draw and paint concentric ellipses.
```

```
    Radius = 20 * ColorVal
    CIRCLE (320, 100), Radius, ColorVal, , , ASPECT
    PAINT (320, 100), ColorVal
NEXT
DO                                  ' Shift the palette until a key is pressed.
    FOR i = 1 TO 15
        PaletteArray(i) = (PaletteArray(i) MOD 15) + 1
    NEXT i
    PALETTE USING PaletteArray(0)

' Map the black background to some random color.
    PALETTE 0, PaletteArray(INT(RND * 15))
LOOP WHILE INKEY$ = ""
WIDTH 80
SCREEN 0

' Using squares.
PALETTE 0, 1
SCREEN 1
FOR i% = 0 TO 3: a%(i%) = i%: NEXT i%
LINE (138, 35)-(288, 165), 3, BF
LINE (20, 10)-(160, 100), 2, BF
DO
    FOR i% = 0 TO 3
        a%(i%) = (a%(i%) + 1) MOD 16
    NEXT i%
    PALETTE USING a%(0)
    SLEEP 1
LOOP WHILE INKEY$ = ""
WIDTH 80
SCREEN 0
```

Parent Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, timer control, vertical scroll bar.

Action Returns the form on which a child control or child form is located; not available at design time and read-only at run time.

Syntax [*form.*][*ctlname.*]**Parent**

Remarks Use the Parent property to access properties, methods, or controls of a control's container *form*. For example:

```
MyButton.Parent.MousePointer = 4
```

The Parent property is useful in an application in which you pass controls as parameters. For example, you might pass a control variable to a general procedure in a module, and use the Parent property to access its container form.

Example See Check Box Control example.

Path Property

Applies To Directory list box, file list box.

Action Sets or returns the current path, including the drive name; not available at design time.

Syntax [*form.*]{*dirlistbox.* | *filelistbox.*}**Path**[= *pathstring$*]

Part	Description
pathstring$	String value representing the complete path, including the drive specification; default is the current path when the control is loaded.

Remarks Use this property when building an application's file-browsing and file-manipulation capabilities.

Setting the Path property affects the control in a manner similar to the MS-DOS Chdir command; relative paths are allowed with or without a drive. Setting only a drive, with a colon (:), selects the current directory on that drive.

Changing the value of Path has the following effects:

Control	Event Generated
Directory list box	Change
File list box	PathChange

For a directory list box, the return value of Path differs from that of List and ListIndex, which tracks only the selection:

```
List(ctlname.ListIndex)
```

See Also Archive Property, Change Event, Drive List Box Control, Drive Property, FileName Property, Hidden Property, List Property, ListCount Property, ListIndex Property, Normal Property, PathChange Event, Pattern Property, ReadOnly Property, System Property.

Example This example displays a list of files for the selected drive and directory. To try this example, create a form containing a drive list box, a directory list box, and a file list box.

```
SUB Dir1_Change ()
   File1.Path = Dir1.Path                    ' When Dir changes, set File path
END SUB

SUB Drive1_Change ()
   Dir1.Path = Drive1.Drive                  ' When Drive changes, set Dir path
END SUB
```

PathChange Event

Applies To File list box.

Action Occurs when the selected path changes by setting the FileName or Path property from code.

Syntax **SUB** *ctlname*_**PathChange** ([**Index AS INTEGER**])

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks Use a PathChange event procedure to respond to path changes in a file list box. When you assign a string containing a new path to the FileName property, the file list box control invokes the PathChange event.

See Also FileName Property, Path Property, Pattern Property, PatternChange Event.

Example See File List Box Control example.

Pattern Property

Applies To File list box.

Action Determines which file names are displayed in a file list box at run time; may include wildcards (? or *).

Syntax [*form.*]*filelistbox*.**Pattern**[= *filespec$*]

Part	Description
filespec$	Any valid file specification; default is *.*.

Remarks The Pattern property plays a key role in designing an application's file-browsing and manipulation capabilities. Use Pattern with other file properties to provide the user ways to explore files or groups of similar files in an application. For example, in an application dedicated to launching other programs, you might designate that only .EXE files be displayed in the file list box (*.EXE).

Changing the value of the Pattern property generates a PatternChange event.

See Also Archive Property, Drive Property, FileName Property, Hidden Property, Normal Property, Path Property, PatternChange Event, ReadOnly Property, System Property.

Example This example updates a text box with the new pattern selected in a file list box. The controls are set up so that when the user enters a pattern in the text box (such as *.TXT), it is reflected in the file list box, much like the interaction you see in a File Open dialog box. If a full path is entered in the text box (such as C:BIN*.EXE), the text is automatically parsed into path and pattern components by the file list box. To try this example, create a form containing a command button, a directory list box, a file list box, and a text box.

```
SUB Command1_Click ()          ' Text is parsed into path/pattern components
   File1.Filename = Text1.Text ' Pass the path to the directory list
   Dir1.Path = File1.Path
END SUB

SUB Dir1_Change ()
   File1.Path = Dir1.Path      ' Pass a path change to files list
END SUB

SUB File1_PatternChange ()
   Text1.Text = File1.Pattern  ' Set text to new pattern
END SUB
```

PatternChange Event

Applies To File list box.

Action Occurs when the pattern changes by setting the FileName or Pattern property in code.

Syntax **SUB** *ctlname*_**PatternChange ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks Use a PatternChange event procedure to respond to pattern changes in a file list box.

When you assign a string containing a new pattern to the FileName property, the file list box invokes the PathChange event.

See Also FileName Property, Path Property, PathChange Event, Pattern Property.

Example The example updates a file list box with files matching the pattern typed into the text box. If a full path is entered into the text box, such as C:\VBDOS*.exe, the text is automatically parsed into path and pattern components. The path is displayed by the label control. To try this example, create a form containing a text box, a label, a file list box, and a command button. Set command button properties:

- Caption = OK
- Default = True

```
SUB Command1_Click ()            ' OK button clicked.
    ' Text is parsed into path and pattern components.
    File1.FileName = Text1.Text
END SUB

SUB File1_PathChange ()
    Label1.Caption = "Path: " + File1.Path
END SUB

SUB File1_PatternChange ()
    Text1.Text = File1.Pattern         ' Set text to new pattern.
END SUB
```

PCOPY Statement

Action Copies one video memory page to another; not available when forms are showing.

Syntax **PCOPY** *sourcepage%, destinationpage%*

Part	Description
sourcepage%	Video memory page number to copy from.
destinationpage%	Video memory page number to copy to.

Remarks

References to the PCOPY statement are not allowed when forms are showing and cause Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this statement.

The value that identifies the video page is determined by the size of video memory and the current screen mode.

The number of video memory pages available depends on the current screen mode, the graphics adapter, and how much screen memory is on that adapter:

MDPA Screen Modes

Mode	Rows and Columns	Resolution	Pages Available
0	80x25	720x350	1

Hercules Adapter Screen Modes

Mode	Rows and Columns	Resolution	Pages Available
0	80x25	720x348	2
3	80x25	720x348	2

CGA Screen Modes

Mode	Rows and Columns	Resolution	Pages Available
0	40x25	320x200	8
	80x25	640x200	4
1	40x25	320x200	1
2	80x25	640x200	1

EGA Screen Modes

Mode	Rows and Columns	Display[a]	Resolution	Page Size	Pages Available
0	40x25	Color	320x200	-	8
		Enhanced	320x350	-	8
		Enhanced	320x350	-	-
		Color	640x200	-	-
		Enhanced	640x350	-	-
		Color	340x200	-	-
		Monochrome	720x350	-	-
		Enhanced	640x350	-	4[c]
		Monochrome	720x350	-	4[c]
1	40x25	-	320x200	16K	1
2	80x25	-	640x200	16K	1
7	40x25	-	320x200	32K	a
8	80x25	-	640x200	64K	a
9[d]	80x25	Enhanced	640x350	64K	1
	80x43	Enhanced	640x350	64K	1
	80x25	Enhanced	640x350	128K	a
	80x43	Enhanced	640x350	8K	a
10	80x25	Monochrome	640x350	64K	b
	80x43	Monochrome	640x350	64K	b

[a] Pages = Screen memory divided by page size (eight page maximum, one page minimum).

[b] Pages = Screen memory divided by 2 divided by page size (eight page maximum, one page minimum).

[c] Number of pages is halved with 64K.

[d] The first two entries under mode 9 are for an EGA with 64K of screen memory. The next two entries assume more than 64K of screen memory.

VGA Screen Modes

Mode	Rows and Columns	Resolution	Page Size	Pages Available
0	40x25	360x400	-	8
	40x43	320x350	-	8
	40x50	320x400	-	4
	80x25	720x400	-	8
	80x43	640x350	-	4
	80x43	720x350	-	4
	80x50	640x400	-	4
	80x50	720x400	-	4
1	40x25	320x200	16K	1

VGA Screen Modes (*continued*)

Mode	Rows and Columns	Resolution	Page Size	Pages Available
2	80x25	640x200	16K	1
7	40x25	320x200	32K	[a]
8	80x25	640x200	64K	[a]
9	80x25	640x350	128K	[a]
	80x43	640x350	128K	[a]
10	80x25	640x350	64K	[b]
	80x43	640x350	64K	[b]
11	80x30	640x480	64K	1
	80x60	640x480	64K	1
12	80x30	640x480	256K	1
	80x60	640x480	256K	1
13	40x25	320x200	64K	1

[a] Pages = Screen memory divided by page size (8-page maximum).

[b] Pages = Screen memory divided by 2 divided by page size (8-page maximum).

MCGA Screen Modes

Mode	Rows and Columns	Resolution	Page Size	Pages Available
0	40x25	320x400	-	8
	80x25	640x400	-	8
1	40x25	320x200	16K	1
2	80x25	640x200	16K	1
11	80x30	640x480	64K	1
	80x60	640x480	64k	1
13	70x25	320x20xx	64K	1

PEEK Function, POKE Statement

Action Returns a byte value stored at a specified memory location (PEEK); writes a byte value to a specified memory location (POKE).

Syntax

PEEK(*address*)
POKE *address*, *byte%*

Part	Description
address	Byte position relative to the current segment address set by DEF SEG; a value in the range 0 to 65,535, inclusive.
byte%	Byte value to write to the specified memory location; a value in the range 0 to 255, inclusive.

Remarks

Warning: Use POKE carefully. If used incorrectly, it can cause Visual Basic or the operating system to fail.

Direct string manipulation with PEEK should be used cautiously, because Visual Basic moves string locations during run time. The DEF SEG statement should be executed immediately before using the PEEK function.

Visual Basic moves string locations during run time. Therefore, the DEF SEG statement must be executed immediately before using the POKE statement.

If *address* is a single- or double-precision floating-point value or a long integer, it is converted to a 2-byte integer.

The PEEK function complements the POKE statement and vice versa.

When using PEEK to return a byte from a far-string array, use the SSEG and SADD functions to obtain the current segment and offset. For example:

```
DEF SEG = SSEG(a$)              ' Set the current segment address
                                ' to the address of a$
StringOffset = SADD(a$)         ' Determine the string's location
                                ' within the segment
PEEK(StringOffset)              ' Return the byte stored at this
                                ' location
```

Before using POKE to directly manipulate data stored in far memory, use the DEF SEG statement and the SSEG function to set the current segment address. For example:

```
DEF SEG = SSEG(a$)              ' Set the current segment address
                                ' to the address of a$
Offset1 = SADD(a$)              ' Determine the string's location
                                ' within the segment
POKE Offset1, PEEK(Offset2)     ' Write the byte stored
```

Do not use PEEK or POKE with an expanded-memory array. If you start Visual Basic (VBDOS.EXE) with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays < 16K in size.
- Fixed-length string arrays < 16K in size.
- User-defined type arrays < 16K in size.

If you want to use PEEK or POKE with an array, first start Visual Basic (VBDOS.EXE) without the /Ea switch. Note: Without the /Ea switch, no arrays are stored in expanded memory.

See Also DEF SEG Statement.

Example See DEF SEG Statement example

PEN Function

Action Returns the status of the light pen.

Syntax **PEN**(*n%*)

Part	Description
n%	Specifies the lightpen status information to return:

Value	Returns
0	Whether pen was used since last call (-1 if yes, 0 if no).
1	X screen coordinate of the last pen press.
2	Y screen coordinate of the last pen press.
3	Current pen switch status (-1 if down, 0 if up).
4	X screen coordinate where the pen last left screen.
5	Y screen coordinate where the pen last left screen.
6	Row number of the last pen press.
7	Column of the last pen press.
8	Row where the pen last left the screen.
9	Column where the pen last left the screen.

Remarks The PEN function does not work when the mouse driver is enabled because the mouse driver uses the PEN function's BIOS calls.

Use mouse function 14 to disable the driver's lightpen emulation. Mouse function 13 turns emulation back on.

See Also ON PEN Statement, PEN Statement, SCREEN Statement.

Example See ON PEN Statement example.

PEN Statement

Action Enables, disables, or suspends lightpen event trapping; can be used with object-specific events.

Syntax **PEN {ON | OFF | STOP}**

Part	Description
ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a PEN ON statement is executed.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

PEN ON enables lightpen-event trapping. A lightpen event occurs whenever the lightpen is activated by pressing the tip to the screen or pressing the touch ring. If a lightpen event occurs after a PEN ON statement, the routine specified in the ON PEN statement is executed.

PEN OFF disables lightpen-event trapping so no trapping takes place until another PEN ON statement is executed. Events occurring while trapping is off are ignored.

PEN STOP suspends lightpen-event trapping so no trapping takes place until a PEN ON statement is executed. Any ON PEN events that occur while trapping is suspended are remembered and processed when the next PEN ON statement is executed. However, remembered events are lost if PEN OFF is executed.

When a lightpen event trap occurs (that is, the GOSUB is performed), an automatic PEN STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a PEN ON statement unless an explicit PEN OFF was performed inside the routine.

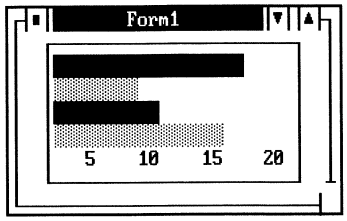
PEN ON must be executed before you use the PEN function. If you try to execute the PEN function when the lightpen is off, Basic generates the error message, "Illegal function call."

See Also ON PEN Statement, PEN Function.

Example See ON PEN Statement example.

Picture Box Control

Action Displays text or ASCII graphics output from the PRINT method or contains groups of option buttons.



Remarks

Events

Click	DblClick	DragDrop	DragOver
GotFocus	KeyDown	KeyPress	KeyUp
LostFocus	MouseDown	MouseMove	MouseUp
Paint			

Methods

CLS	DRAG	MOVE	PRINT
REFRESH	SETFOCUS	TEXTHEIGHT	TEXTWIDTH

Properties

AutoRedraw	BackColor	BorderStyle	CtlName
CurrentX	CurrentY	DragMode	Enabled
ForeColor	Height	Index	Left
MousePointer	Parent	ScaleHeight	ScaleWidth
TabIndex	TabStop	Tag	Top
Visible	Width		

Use a picture box when you want to surround text with custom border styles. To add a border to a picture box at design time:

- Place a picture box on a form.
- Place child controls in the picture box.
- Use the PRINT method to display the characters you want to use as a border at run time.

A picture box can be used as a visual container for controls, like a window or dialog box. You can place up to 254 controls in a picture box. A picture box can receive the focus even when it contains child controls.

Text written to a picture box at run time is clipped to fit within the border of the picture box.

To make picture box display persistent, set the AutoRedraw property to True (-1), or use the PRINT method in the Paint event procedure.

The minimum size of a picture box is 1 x 1; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are set to the minimum value.

See Also Frame Control, Option Button Control.

Example See BackColor Property example.

PLAY Function

Action Returns the number of notes in the background music queue.

Syntax **PLAY**(*n*)

Part	Description
n	Any numeric value.

See Also ON PLAY Statement, PLAY Statement (Event Trapping), PLAY Statement (Music).

Example This example below illustrates use of the PLAY function.

```
Music$ = "MBT180o2P2P8L8GGGL2E-P24P8L8FFFL2D"
PLAY Music$
WHILE PLAY(0) > 5: WEND
PRINT "Just about done!"
```

PLAY Statement (Event Trapping)

Action Enables, disables, or suspends play-event trapping; can be used with object-specific events.

Syntax **PLAY {ON | OFF | STOP}**

Part	Description
ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a PLAY ON statement is executed.

Remarks If your program contains event-handling statements and you are compiling from the BC command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

PLAY ON enables play-event trapping. A play event occurs when the number of notes in the background-music queue drops below the limit you set in the ON PLAY statement. If a play event occurs after a PLAY ON statement, the routine specified in the ON PLAY statement is executed.

PLAY OFF disables play-event trapping so no trapping takes place until another PLAY ON statement is executed. Events occurring while trapping is off are ignored.

PLAY STOP suspends play-event trapping so no trapping takes place until a PLAY ON statement is executed. Any ON PLAY events that occur while trapping is suspended are remembered and processed when the next PLAY ON statement is executed. However, remembered events are lost if PLAY OFF is executed.

When a play-event trap occurs (that is, the GOSUB is performed), an automatic PLAY STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a PLAY ON statement unless an explicit PLAY OFF was performed inside the routine.

See Also ON PLAY Statement, PLAY Function, PLAY Statement (Music).

Example See ON PLAY Statement example.

PLAY Statement (Music)

Action Plays musical notes.

Syntax **PLAY** *commandstring$*

Part	Description
commandstring$	String expression that contains one or more valid PLAY commands: Duration and tempo commands. Foreground and background operation commands. Octave and tone commands.

Remarks

The PLAY statement uses a concept similar to DRAW in that it embeds a music macro language in one statement - that is, a set of commands, used as part of the PLAY statement, specifies a particular action.

To execute a PLAY command substring from a PLAY command string, use the "X" substring command:

```
PLAY "X"+ VARPTR$(commandstring$)
```

Substring	Action
"X" + VARPTR$(*string***)**	Executes a substring.

In Visual Basic programs, use the VARPTR$(*variablename*) form for variables. For example:

```
PLAY "X" + VARPTR$(A$)
PLAY "O=" + VARPTR$(I)
```

See Also

BEEP Statement, PLAY Function, PLAY Statement (Event Trapping), SOUND Statement.

Example

This example uses the PLAY statement and VARPTR$ function to play music.

```
scale$ = "CDEFGAB"                ' Play scale in seven different octaves.
PLAY "L16"
FOR i% = 0 TO 6
   PLAY "O" + STR$(i%)
   PLAY "X" + VARPTR$(scale$)
NEXT i%
```

PMAP Function

Action

Returns the window coordinate equivalent to a viewport coordinate as defined by the WINDOW statement, or vice versa. Function not available when forms are showing.

Syntax

PMAP (*startcoordinate#*, *n%*)

Part	Description
startcoordinate#	X or Y coordinate in the window or viewport.
n%	Value indicating which coordinate is returned:

If *n%* is	and startcoordinate# is	then PMAP returns
0	Window x coordinate.	Viewport x coordinate.
1	Window y coordinate.	Viewport y coordinate.
2	Viewport x coordinate.	Window x coordinate.
3	Viewport y coordinate.	Window y coordinate.

Remarks

References to the PMAP function are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the HIDE method to hide any visible forms or controls before executing this function.

The four PMAP functions allow you to find equivalent point locations between the window coordinates created with the WINDOW statement and the absolute screen coordinates or viewport coordinates defined by the VIEW statement.

If no VIEW statement has been executed, or the most recently executed VIEW statement has no arguments, the viewport coordinates are equivalent to the absolute screen coordinates established by the most recently executed SCREEN statement.

See Also

POINT Function, VIEW Statement, WINDOW Statement.

Example

This example uses the VIEW and WINDOW statements to define a graphics viewport and window. The PMAP function is used to convert viewport coordinates to window coordinates. The program generates a fractal that shows a subset of the complex numbers called the "Mandelbrot Set."

```
DEFINT A-Z                              ' Default variable type is integer.
DECLARE SUB ScreenTest (EM%, CR%, VL%, VR%, VT%, VB%)

' Set maximum number of iterations per point.
CONST MAXLOOP = 30, MAXSIZE = 1000000
CONST FALSE = 0, TRUE = NOT FALSE       ' Boolean constants.

' Set window parameters.
CONST WLeft = -1000, WRight = 250, WTop = 625, WBottom = -625

' Call ScreenTest to find out if this is an EGA machine, and get
' coordinates of viewport corners.
ScreenTest EgaMode, ColorRange, VLeft, VRight, VTop, VBottom
```

```
' Define viewport and corresponding window.
VIEW (VLeft, VTop)-(VRight, VBottom), 0, ColorRange
WINDOW (WLeft, WTop)-(WRight, WBottom)
LOCATE 24, 10: PRINT "Press any key to quit.";
XLength = VRight - VLeft
YLength = VBottom - VTop
ColorWidth = MAXLOOP \ ColorRange

' Loop through each pixel in viewport and calculate whether or not it is
' in the Mandelbrot Set.
FOR Y = 0 TO YLength              ' Loop through every line in the viewport.
    LogicY = PMAP(Y, 3)           ' Get the pixel's logical y coordinate.
    PSET (WLeft, LogicY)          ' Plot leftmost pixel in the line.
    OldColor = 0                  ' Start with background color.
    FOR X = 0 TO XLength          ' Loop through every pixel in the line.
         LogicX = PMAP(X, 2)      ' Get the pixel's logical x coordinate.
         MandelX& = LogicX
         MandelY& = LogicY

' Do the calculations to see if this point is in the Mandelbrot Set.
         FOR I = 1 TO MAXLOOP
              RealNum& = MandelX& * MandelX&
              ImagNum& = MandelY& * MandelY&
              IF (RealNum& + ImagNum&) >= MAXSIZE THEN EXIT FOR
                   MandelY& = (MandelX& * MandelY&) \ 250 + LogicY
                   MandelX& = (RealNum& - ImagNum&) \ 500 + LogicX
         NEXT I
         PColor = I \ ColorWidth    ' Assign a color to the point.

' If color has changed, draw a line from the last point referenced to the
' new point, using the old color.
         IF PColor <> OldColor THEN
              LINE -(LogicX, LogicY), (ColorRange - OldColor)
              OldColor = PColor
         END IF
         IF INKEY$ <> "" THEN END
    NEXT X

' Draw the last line segment to the right edge of the viewport.
    LINE -(LogicX, LogicY), (ColorRange - OldColor)
NEXT Y
DO
LOOP WHILE INKEY$ = ""
SCREEN 0, 0                       ' Restore the screen to text mode,
WIDTH 80                          ' 80 columns wide.
END
```

```
BadScreen:                              ' Error handler that is invoked if
     EgaMode = FALSE                     ' there is no EGA graphics card.
     RESUME NEXT

' This procedure tests to see if the user has EGA hardware with SCREEN 8.
' If this causes an error, the EM flag is set to FALSE, and the screen
' is set with SCREEN 1. The procedure also sets values for the corners
' of the viewport (VL = left, VR = right, VT = top, VB = bottom), scaled
' with the correct aspect ratio so the viewport is a perfect square.
 STATIC SUB ScreenTest (EM, CR, VL, VR, VT, VB)
     EM = TRUE
     ON ERROR GOTO BadScreen
     SCREEN 8, 1
     ON ERROR GOTO 0

     IF EM THEN                         ' No error, so SCREEN 8 is OK.
          VL = 110: VR = 529
          VT = 5: VB = 179
          CR = 15                       ' 16 colors (0 - 15).

     ELSE                               ' Error, so use SCREEN 1.
          SCREEN 1, 1
          VL = 55: VR = 264
          VT = 5: VB = 179
          CR = 3                        ' 4 colors (0 - 3).
    END IF

 END SUB
```

POINT Function

Action Returns the current graphics cursor coordinates or the color attribute of a specified pixel; not available when forms are showing.

Syntax **POINT** {(*number%*) | (*x%*,*y%*)}

Part	Description
(*number%*)	Specifies type of coordinate to return:

Value	Returns
0	Current viewport x coordinate.
1	Current viewport y coordinate.
2	Current window x coordinate.
3	Current window y coordinate.

x%,y%)	Coordinates of the pixel that POINT checks for color; if coordinates are outside the current viewport, POINT returns -1.

Remarks

References to the POINT function are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the HIDE method to hide any visible forms or controls before executing this function.

When (*x%,y%*) are used to specify a pixel, the POINT function returns the color number of the indicated pixel. The expressions *x%* and *y%* are viewport coordinates; if no VIEW statement has been executed, or the most recently executed VIEW statement has no arguments, the viewport coordinates are the same as the absolute screen coordinates.

The meaning of the color number returned as well as the valid range of values for *x%* and *y%* depend on the current screen mode established by the most recently executed SCREEN statement. If the specified pixel is out of range, POINT returns the value -1.

See Also

COLOR Statement, PMAP Function, SCREEN Statement, VIEW Statement, WINDOW Statement.

Example

This example uses the POINT function to find the border of the ellipse by testing for a change in color while redrawing an ellipse drawn with the CIRCLE statement.

```
DEFINT X-Y
CLS                                  ' Clear the screen.
INPUT "Enter angle of tilt in degrees (0 to 90): ", Ang
SCREEN 1                             ' Medium resolution screen.
Ang = (3.1415926# / 180) * Ang       ' Convert degrees to radians.
Cs = COS(Ang): Sn = SIN(Ang)
CIRCLE (45, 70), 50, 2, , , 2        ' Draw ellipse.
PAINT (45, 70), 2                    ' Paint interior of ellipse.
FOR y = 20 TO 120
    FOR x = 20 TO 70
        IF POINT(x, y) <> 0 THEN     ' Check each point in rectangle.
        ' enclosing ellipse; if the point is in the ellipse, plot a
        ' corresponding point in the "tilted" ellipse
        Xnew = (x * Cs - y * Sn) + 200: Ynew = (x * Sn + y * Cs)
        PSET (Xnew, Ynew), 2
        END IF
    NEXT
```

```
NEXT
END
```

POKE Statement

See PEEK Function.

POS Function, CSRLIN Function, and LOCATE Statement

Action Returns the current column position of the cursor (POS); returns the current row position of the cursor (CSRLIN); moves the cursor to a specified position (LOCATE); not available when forms are showing. Use the CurrentX and CurrentY properties with forms and picture boxes.

Syntax

POS(*n%*)
CSRLIN
LOCATE [*row%*] [,[*column%*] [,[*cursor%*] [,*start%* [,*stop%*]]]]

Part	Description
n%	Any numeric expression; while it passes no information, required to maintain consistent function syntax.
row% *column%*	Number specifying the row and column to which the cursor moves.
cursor%	Specifies whether the cursor is visible: 0 = invisible. 1 = visible.
start% *stop%*	Integer expressions in the range 0 to 31, inclusive, that specify the first and last cursor scan lines; changing the cursor scan lines changes cursor size.

Remarks The POS and CSRLIN functions and the LOCATE statement are not allowed with forms. Any reference to POS, CSRLIN, or LOCATE when forms are showing causes Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls.

If the screen cursor is in the leftmost column, POS returns a value of 1; if the screen cursor is in the top row, CSRLIN returns a value of 1.

You can omit any argument from the LOCATE statement except *stop%*; if *stop%* is specified, *start%* also must be specified. When you omit *row%* or *column%*, LOCATE leaves the cursor at the row or column where it was moved by the most recently executed

input or output statement (such as LOCATE, PRINT, or INPUT). When you omit other arguments, Visual Basic assumes the previous value for the argument.

The start and stop lines are the CRT scan lines specifying which pixels on the screen are lit; a wider range between the start and stop lines produces a taller cursor (occupies an entire character block).

When *start%* is greater than *stop%*, LOCATE produces a two-part cursor:

- If the start line is given but the stop line is omitted, *stop%* assumes the same value as *start%*.
- A value of 8 for both *start%* and *stop%* produces the underline cursor.
- The maximum cursor size is determined by the character block size of the screen mode in use.
- Setting *start%* greater than *stop%* displays a full-height cursor on VGA-equipped systems.

See Also SCREEN Statement.

Example This example illustrates one use of the CSRLIN and POS function, and the LOCATE statement.

```
SUB Form_Click()
    SCREEN.HIDE                          ' Hide any visible forms.
    LOCATE Form1.Top + 1, Form1.Left + 1 ' Move cursor.
    col% = CSRLIN                        ' Get cursor column position.
    row% = POS(1)                        ' Get cursor row position.
    PRINT row%, col%
    SCREEN.SHOW                          ' Redisplay and hidden forms.
END SUB
```

PRESET, PSET Statements

Action Draws a specified point on the screen; not available when forms are showing. Available color attributes depend on your graphics adapter and screen mode. Coordinate values depend on your graphics adapter, screen mode, and most recent VIEW and WINDOW statements.

Syntax

PRESET [**STEP**] (*x!*,*y!*) [, *color%*]
PSET [**STEP**] (*x!*,*y!*) [, *color%*]

Part	Description
STEP	Specifies that *x!* and *y!* are expressed relative to the current graphics cursor location.

Part	Description
(*x!*,*y!*)	Screen coordinates of the pixel to set.
color%	Color attribute that sets the pixel color; if omitted, PRESET uses current background and PSET uses current foreground color.

Remarks

References to PRESET or PSET are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the HIDE method to hide any visible forms or controls before executing either of these statements.

The available color attributes for PRESET and PSET depend on your graphics adapter and the screen mode set by the most recent SCREEN statement.

Coordinate values for PRESET and PSET depend on the graphics adapter, screen mode, and most recent VIEW and WINDOW statements.

The STEP keyword indicates that the x and y coordinates are relative, not absolute. The coordinates are treated as distances from the most recent graphics cursor location, not distances from the (0,0) screen coordinate. For example, if the most recent graphics cursor location were (10,10), PRESET STEP (10,5) would draw a point at (20,15).

PRESET works exactly like PSET, except:

- If *color%* is not specified for a PSET statement, the current foreground color is the default.
- If *color%* is not specified for a PRESET statement, the current background color is the default.

If the coordinate is outside the current viewport, no action is taken, and no error message is generated.

See Also

SCREEN Statement, VIEW Statement, WINDOW Statement.

Example

This example uses the PSET and PRESET statements to draw a line 20 pixels long, then to move the line across the screen from left to right.

```
CLS                          ' Clear the screen.
SCREEN 1: COLOR 1, 1: CLS
FOR i = 0 TO 299 STEP 3
   FOR J = i TO 20 + i
      PSET (J, 50), 2     ' Draw the line in new location.
   NEXT
   SLEEP 1
   FOR J = i TO 20 + i
      PRESET (J, 50)      ' Erase the line.
   NEXT
NEXT
```

PRINT # Statement

Action Writes data to a sequential device or file.

Syntax **PRINT** *#filenumber%*, [**USING** *formatstring$;*] *expressionlist*[{;|,}]

Part	Description
filenumber%	Number used to open the sequential file.
formatstring$	Formatting characters that determine the appearance of *expressionlist*.
expressionlist	Numeric or string expressions to write to the file; if omitted, PRINT # prints a blank line to the file.
;|,	Specifies where the next output begins:

Character	Description
;	Print immediately after the last value.
,	Print at the start of the next print zone; print zones are 14 characters wide.

Remarks Any reference to PRINT *#filenumber%* where filenumber% represents SCRN: or CONS: is not allowed with forms and causes Visual Basic to generate the error message, "Invalid when forms are showing."

To use the PRINT # statement with forms, you must first unload or hide any visible forms.

Characters in formatstring$ have several meanings.

Spaces, commas, and semicolons in *expressionlist* have the same meaning in PRINT # as in a PRINT statement.

PRINT # works like PRINT and writes an image of the data to the file, just as the data would be displayed on the terminal screen. For this reason, be careful to delimit the data so it is output correctly. If you use commas as delimiters, the blanks between print fields are also written to the file.

See Also WRITE # Statement.

Example This example shows the effect of using data delimiters with the PRINT # statement.

```
CONST A$ = "CAMERA, AUTOFOCUS", B$ = "September 20, 1992", C$ = "42"
Q$ = CHR$(34)
CLS                                     ' Clear the screen.
OPEN "INVENT.DAT" FOR OUTPUT AS #1      ' Open INVENT.DAT for writing.
PRINT #1, A$; B$; C$                    ' Write A$, B$, C$ without delimiters.
                                        ' Write A$, B$, C$ with delimiters.
```

```
PRINT #1, Q$; A$; Q$; Q$; B$; Q$; Q$; C$; Q$
CLOSE #1
OPEN "INVENT.DAT" FOR INPUT AS #1      ' Open INVENT.DAT for reading.
FOR I% = 1 TO 2                        ' Read first two records and print.
   INPUT #1, First$, Second$, Third$
   PRINT First$; TAB(30); Second$; TAB(60); Third$: PRINT
NEXT
CLOSE #1
KILL "INVENT.DAT"                      ' Remove file from disk.
```

PRINT Method

Applies To Form, picture box, PRINTER object.

Action Prints a numeric or string expression on a form, picture box, or to the printer; printing starts at the position indicated by the CurrentX and CurrentY properties. Use the PRINT statement for non-form functionality.

Syntax [*form.*][*picturebox.* | **PRINTER.**]**PRINT** [**USING** *formatstring$;*][*expressionlist*]

Part	Description
USING *formatstring$;*	String expression containing: The literal string to print. One or more format specifiers that determine the field and format of the printed strings or numbers.
expressionlist	Numeric or string expressions to print. If omitted, PRINT prints a blank line. If multiple expressions are printed, they must be separated by one of the following characters:

Value	Description
; or space	Semicolon or space; places cursor immediately after the last character displayed.
,	Comma; places cursor at the start of the next print zone (every 14 columns).

Remarks The PRINT method is similar to the PRINT statement, except that output from the PRINT method may be sent to an object.

Normally, when *expressionlist* is printed, a carriage return (^M) is appended so that the next PRINT begins printing on the next line.

Carriage returns affect PRINT method behavior and CurrentX and CurrentY values, depending on the character separator used to print multiple expressions:

expressionlist Usage	PRINT Behavior	CurrentX and CurrentY
Appended with a semicolon (;).	No carriage return is appended; next PRINT displays on the same line as the current PRINT.	Set to the point immediately after last character printed.
Contains carriage returns.	The current PRINT displays and spaces text as defined in *expressionlist*.	Each embedded carriage return causes CurrentY to be increased by one and CurrentX to be set to zero.
Appended with a comma (,).	No carriage return is appended; next PRINT displays beginning at next print zone (a print zone is every 14 columns).	Set to next print zone on the same line.

Automatic scrolling does not occur if the printed string cannot fit in the specified form or picture box. Use horizontal and vertical scroll bars to view text that is printed beyond the boundaries of the specified form or picture box.

If the printed string is too long for the printer, it is either clipped on the right or wrapped to the next line, depending on your printer. If the string does not fit at the bottom of a printed page, a NEWPAGE occurs and the string is printed on the next page.

You can control the print position using the SPC or TAB functions.

See Also CurrentX Property, CurrentY Property, LPRINT Statement, PRINT Statement, PRINT # Statement, PRINTFORM Method, TAB Function, SPC Function.

Example See ENDDOC Method Example.

PRINTER Object

Action The PRINTER object is accessed by specifying the PRINTER keyword. This object controls text printed on a page and sends output directly to the default system printer.

Syntax

PRINTER.*property*[= *setting%*]
PRINTER.*method*[*arguments*]

Part	Description
property	Any applicable PRINTER object property.
method	Any applicable PRINTER object method.
setting%	Valid property setting.
arguments	Any arguments to the specified method.

Remarks

Methods

PRINT Method ENDDOC Method NEWPAGE Method

Properties

PrintTarget Property

When you use the PRINT method to print output which is larger than one page, background images are clipped at the bottom of the page and text is carried over to the next page.

Example

This example shows how the PRINTER object and the PrintTarget property can be used. To try this example, create a form containing a wide text box and a command button.

When you run the example, type a short sentence in the text box first, then choose the command button to print. (Note: Your printer must be set up to print on LPT1: before you run this example.)

```
COMMON SHARED Sentence$

SUB Command1_Click ()
    PRINTER.PrintTarget = "LPT1"
    PRINTER.PRINT Sentence$
END SUB
```

PRINTFORM Method

Applies To

Form.

Action

Sends an ASCII dump of the specified or current form (including visible controls and text) to a printer or file.

Syntax

[*form.*]**PRINTFORM**

Remarks

To print a form at design time, choose Print from the File menu in the Form Designer.

Printer output is sent to the device specified by the PrintTarget property.

PRINTFORM prints all controls and images that are visible on the form. Only raw text (that is, no attributes) will be printed. Normal alphanumeric characters are printed as they appear. Certain ASCII characters are translated to alternate characters for print representation. The table below identifies the translated characters and their printed equivalents:

Form Item	Character	Extended ASCII Code	Printed Character
Bottom right corners	┘,╝	217, 188	+
Bottom left corners	└,╚	192, 200	+
Combo box arrow	↓	25	v
Control box	■	179-254-179	|-|
Horizontal borders	─,═	196, 205	–
Maximize button	▲	179-30-179	|^|
Minimize button	▼	179-31-179	|v|
Option button selection	•	7	*
Resize button	╝	188	+
Restore button	↕	179-18-179	|+|
Scroll bar	▒	176	#
Scroll down arrow	↓	25	v
Scroll left arrow	←	27	<
Scroll right arrow	→	26	>
Scroll up arrow	↑	24	^
Shadow	▒	177	Not printed
Title bar border		255	=
Top left corners	┌,╔	218, 201	+
Top right corners	┐,╗	191, 187	+
Vertical borders	|,||	179, 186	|

See Also

PrintTarget Property.

Example The example uses the PRINTFORM method to print the current form.

```
DECLARE SUB PrintFormDemo ()

SUB Form_Click ()
    CALL PrintFormDemo
END SUB

SUB PrintFormDemo ()
    ON LOCAL ERROR GOTO ErrorHandler ' Set up error handler.
    PRINTFORM                        ' Print form.
    PRINTER.ENDDOC
    EXIT SUB

ErrorHandler:
    Msg$ = "The form could not be printed. "
    MSGBOX Msg$                      ' Display message.
    RESUME NEXT

END SUB
```

PrintTarget Property

Applies To PRINTER object.

Action Specifies the target device or file name for PRINT method or PRINTFORM method output.

Syntax **PRINTER.PrintTarget** = *target$*

Part	Description	
target$	Valid settings are:	
	Value	**Description**
	LPT1, LPT2, or LPT3	Valid printer port.
	filename	Valid MS-DOS file name or device.

See Also PRINT Method, PRINTFORM Method.

Example See PRINTER Object example.

PSET Statement

See PRESET Statement.

PUT Statement (File I/O)

See GET Statement (File I/O).

PUT Statement (Graphics)

See GET Statement (Graphics).

QBCOLOR Function

Action Returns the color value that you specify; provides compatibility with Visual Basic for Windows.

Syntax **QBCOLOR**(*x%*)

Part	Description
x%	Valid color attribute in the range 0 to 15, inclusive.

See Also RGB Function.

RANDOMIZE Statement, RND Function

Action Initializes the random-number generator (RANDOMIZE), or returns a single-precision random number between 0 and 1 (RND).

Syntax **RANDOMIZE** [*seed*]
RND[(*n*)]

Part	Description
seed	Any type of numeric expression; used to initialize the random-number generator; if omitted, the value of the TIMER function is used.
n	Determines how RND generates the next random number:

When *n* Is...	RND Returns...
< 0	Same number for any *n*.
= 0	Last number generated.
> 0 (or omitted)	Next random number in sequence.

Remarks

If the random-number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To avoid this, place a RANDOMIZE statement at the beginning of the program loop and change the argument with each iteration.

A convenient way to initialize the random-number generator is to use the TIMER function. Using TIMER ensures a new series of random numbers.

To produce random integers in a given range, use this formula:

```
INT ((upperbound - lowerbound + 1)*RND + lowerbound)
```

In the preceding example, upperbound is the highest number in the range, and lowerbound is the lowest number in the range.

See Also

TIMER Function.

Example

This example uses the RANDOMIZE statement to seed and reseed the random number generator and the RND function to simulate rolling a pair of dice.

```
RANDOMIZE TIMER     ' Use the timer as seed for the number
                      ' generator.
DO                   ' Simulate rolling two dice using RND.
     D1 = INT(RND * 6) + 1
     D2 = INT(RND * 6) + 1
     CLS                      ' Clear the screen. and report the roll.
     PRINT "You rolled a"; D1; "and a"; D2; "for a total of"; D1 + D2
     INPUT "Roll again (Y/N)"; Resp$
     PRINT
LOOP UNTIL UCASE$(MID$(Resp$, 1, 1)) = "N"
END
```

READ Statement

See DATA Statement.

ReadOnly Property

See Archive Property.

REDIM Statement

Action Changes the space allocated to an array that has been declared dynamic.

Syntax **REDIM [PRESERVE] [SHARED]** *variable*(*[lower* **TO]** *upper* [,*[lower* **TO]** *upper*]...) **[AS** *type*][,*variable*(*[lower* **TO]** *upper* [,*[lower* **TO]** *upper*]...)] **[AS** *type*]]**...**

Part	Description
PRESERVE	Causes array data to be preserved when the upper boundary is changed.
SHARED	Causes specified array to be shared with all SUB or FUNCTION procedures in the module.
variable	Any valid array name.
lower	Lower boundary of the array's subscripts (default - 0).
upper	Upper boundary of the array's subscripts.
AS *type*	(Default - SINGLE) Declares data type of an array or variable; use a separate AS type clause for each variable defined.

Remarks The REDIM statement is often used in the initial Form_Load event procedure to define the dimensions of an array shared between form modules.

Although you can change the size of an array's dimensions with REDIM, you cannot change the number of dimensions. For example, the following statements are legal:

```
DIM A(50,50)                    ' $DYNAMIC.
ERASE A
REDIM A(20,15)                  ' Array A still has two dimensions.
```

However, the following statements are illegal, and produce the error message, "Wrong number of dimensions":

```
DIM A(50,50)            ' $DYNAMIC.
ERASE A
REDIM A(5,5,5)          ' Changed number of dimensions
                        ' from two to three.
```

You can also use REDIM to declare or resize an array of user-defined types. In the following example, the variable Card is declared as a user-defined type; REDIM allocates space for an array called Deck that consists of 52 Card elements:

```
TYPE Card
    Suit AS STRING * 9
    Value AS INTEGER
END TYPE

DIM Deck()
REDIM Deck(1 TO 52) AS Card
```

Note: The user-defined type declaration (using the TYPE statement) can only exist at the module level; REDIM statement can only be used in a module or procedure.

If the PRESERVE keyword is not used, REDIM initializes all:

- Elements of numeric arrays to zero.
- Elements of string arrays to null strings.
- Fields of record variables to zero, including fixed-length string elements.

REDIM is used to size or resize a dynamic array that has already been formally declared using DIM with empty parentheses (no dimension subscripts).

If you first declare a dynamic array using a DIM statement and no dimension subscripts, the maximum number of dimensions you can later specify with REDIM is 8.

If your array requires more than 8 dimensions, you can use REDIM in a SUB or FUNCTION procedure to initially declare a local dynamic array variable; if this is the case, your array can have up to 60 dimensions.

When a REDIM statement is compiled, all arrays declared in the statement are treated as dynamic. At run time, when a REDIM statement is executed, the array is deallocated (if it is already allocated) and then reallocated with the new dimensions. Note: If the PRESERVE keyword is not used, old array element values are lost.

The PRESERVE keyword allows you to raise or lower the outer bounds of a dynamic array without erasing data. For example, the following statements keep any data entered in arrayX() and add space for more elements in the third dimension of the array:

```
REDIM X(10, 10, 10)
    .
    .
    .
REDIM PRESERVE X(10, 10, 15)
```

The third dimension of X() is the rightmost bound of the array. To change the leftmost bound, you must compile the program with /R.

Use UBOUND after REDIM PRESERVE to determine how many filled elements there are in an array since they will all be filled.

When used with DIM, the SHARED keyword shares variables among all procedures in a module. In contrast, the SHARE statement shares variables between a single procedure and the module-level code.

See Also DIM Statement.

Example This example uses the PRESERVE keyword in the REDIM statement to add records to an existing array as more data is entered.

```
DATA 23, -45, 6, 5, 1, 12, 7, 1, -9, 0

CLS                                          ' Clear the screen.
REDIM X(1 TO 10) AS INTEGER                  ' Create dynamic array.
FOR I% = 1 TO 10                             ' Read in data.
  READ X(I%)
NEXT I%
DO
  INPUT "Item to add to list (ENTER to finish)"; A$    ' Resize array X()
  IF LEN(A$) THEN                                      ' as items are added
    REDIM PRESERVE X(1 TO UBOUND(X) + 1) AS INTEGER     ' using REDIM with
    X(UBOUND(X)) = VAL(A$)                               ' PRESERVE keyword.
  END IF
LOOP WHILE LEN(A$)
FOR I% = 1 TO UBOUND(X)                      ' Print array.
  PRINT X(I%)
NEXT I%
END
```

REFRESH Method

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Redraws a form or control; in addition, updates the contents of the control if it is a file list box, directory list box, or drive list box.

Syntax [*form.*][*ctlname.*]**REFRESH**

Remarks Generally, a form or control is automatically refreshed after all events have been processed. However, there may be situations where you want the form or control refreshed immediately. For example, if you use a file list box, directory list box, or drive list box to show the current status of the file system, you can use REFRESH to update the status when a change is made.

See Also AutoRedraw Property.

Example The example uses the REFRESH method to refresh a file list box as test files are being created. To try this example, create a form containing a file list box control.

```
DECLARE SUB RefreshDemo ()

SUB Form_Click ()
   CALL RefreshDemo
END SUB

SUB RefreshDemo ()
     File1.Pattern = "TestFile.*"                ' Set file pattern.
     FOR I% = 1 TO 9                             ' Do 9 times.
         FileName$ = "TESTFILE." + LTRIM$(STR$(I%))
         OPEN FileName$ FOR OUTPUT AS FREEFILE   ' Create null file.
         File1.REFRESH                           ' Refresh file list box.
         CLOSE                                   ' Close file.
     NEXT I%
     Msg$ = "Choose OK to remove the created test files."
     MSGBOX Msg$                                 ' Display message.
     KILL "TESTFILE.*"                           ' Remove test files.
     File1.Pattern = "*.*"
     File1.REFRESH                               ' Update file list box.
END SUB
```

REM Statement

Action Allows explanatory remarks or metacommands to be inserted in a program.

Syntax {**REM** | '} *text*

Part	Description
text	Any valid remarks, comments, or metacommands.

Remarks Remarks are ignored when your program runs unless they contain metacommands.

REM statements are not compiled, but instead appear exactly as typed when the program is listed.

A remark can be inserted on a line after an executable statement if it is preceded by either:

- A single quotation mark (').
- REM preceded by a colon (:REM).

Do not use the single-quotation form of REM in a DATA statement because it will be considered valid data.

You can branch from a GOTO or GOSUB statement to a REM statement. Execution continues with the first executable statement after the REM statement.

Example This example shows two equivalent remark statements.

```
CLS     ' Clear the screen.
REM     This is a comment.
'       This is also a comment.
PRINT "Test1"         ' This is a comment after a PRINT statement.
PRINT "Test2":   REM   This is also a comment after a PRINT statement.
```

REMOVEITEM Method

Applies To List box, combo box.

Action Removes a list item from a list box or combo box at run time.

Syntax [*form.*]{*listbox* | *combobox*}.**REMOVEITEM** *index%*

Part	Description
index%	Integer specifying the position in the list of the item to remove; an *index%* of 0 represents the first position.

Remarks Use the List, ListCount, and ListIndex properties to access items displayed in the list box at run time.

ListIndex is set to -1 (no item selected) immediately after a REMOVEITEM operation.

If you specify an *index%* value that does exist, Visual Basic generates the error message "Invalid Property Array Index."

See Also ADDITEM Method.

Example See ADDITEM Method example.

RESET Statement

Action Closes all open files and devices.

Syntax **RESET**

See Also CLOSE Statement.

Example This example uses the RESET statement to close several files at once. The program attempts to write to the previously opened files, causing an error and demonstrating that all files are closed.

```
DEFINT A-Z
ON ERROR GOTO ErrHandler          ' Set up the error handling routine.

CLS                               ' Clear the screen.
FOR I = 1 TO 3
     OPEN "Test" + RIGHT$(STR$(I), 1) + ".dat" FOR OUTPUT AS FREEFILE
     PRINT "File #"; I; "has been opened for output."
NEXT I
PRINT : PRINT "Press any key to RESET all open files."
PRINT
Z$ = INPUT$(1)
RESET
```

```
FOR I = 1 TO 3
     PRINT "Trying to write to file #"; I
     PRINT #I, "Test data"
NEXT I
END

ErrHandler:                          ' Error 52 is "Bad File Name or Number."
     IF ERR = 52 THEN PRINT "  File #"; I; "not open. RESET closed it."
     RESUME NEXT
```

Resize Event

Applies To Form.

Action Occurs when the size of the form changes; also occurs when the form is first displayed.

Syntax **SUB Form_Resize ()**

Remarks Use a Resize event procedure to move or resize controls when the container form is resized.

A Resize event procedure can be used to recalculate variables or properties that depend on the size of the form.

Whenever the AutoRedraw property is set to False (0) and the form is resized, this event ordering occurs: Resize, Paint.

To invoke the Paint event every time a form is resized, use the REFRESH method in a Resize event.

The Top, Left, Height, Width, ScaleHeight, and ScaleWidth properties for a form are automatically adjusted when a Resize event occurs.

See Also AutoRedraw Property, LOAD Statement, Paint Event, REFRESH Method, SHOW Method.

Example This example automatically resizes a text box to fill the form whenever the form is resized. To try this example, create a form containing a text box control.

```
SUB Form_Load ()
   Text1.Text = ""                   ' Clear text.
END SUB

SUB Form_Resize ()
   Text1.Move 0, 0, ScaleWidth, ScaleHeight
END SUB
```

RESTORE Statement

See DATA Statement.

RESUME Statement

Action Resumes program execution after an error-handling routine is finished.

Syntax **RESUME** [{*line* | **NEXT**}]

Part	Description
line	Label or line number where execution resumes; if 0 or omitted, execution resumes with the statement that caused the error.
NEXT	Resumes execution at the statement following the statement that caused the error.

Remarks The location where execution resumes is based on the location of the error handler in which the error is trapped, not on the location where the error occurred.

If Visual Basic has to search for the error handler, and the error handler that contains the RESUME statement is in a procedure or module other than the one in which the error occurred, then the last statement executed in that procedure or module is the last call out of that procedure or module.

As a rule, avoid using a *line* argument with a RESUME statement in a module-level error handler, unless you expect errors to occur only at the module level.

A RESUME statement that is not in an error-handling routine produces the error message, "RESUME without error."

When an error-handling routine is active, if the end of the program text is encountered before executing a RESUME statement, Visual Basic generates the error message, "No RESUME." This is also true if an END, END SUB, END FUNCTION, or END DEF statement (for a local error handler) is executed before a RESUME statement.

When you are compiling from the Basic Compiler (BC.EXE) command line, programs that contain error-handling routines must be compiled with either the /E (ON ERROR) or /X (RESUME NEXT) option.

Example See END Statement example.

RGB Function

Action Returns the Basic color value that most closely approximates the RGB color generated in Visual Basic for Windows; provides compatibility with Visual Basic for Windows.

Syntax *x%* = **RGB**(*red%*, *green%*, *blue%*)

Part	Description
x%	Valid color attribute in the range 0 to 15, inclusive.
red% *green%* *blue%*	Integer value in the range 0 to 255, inclusive, that represents the intensity of the color.

See Also QBCOLOR Function.

RIGHT$ Function

See LEFT$ Function.

RMDIR Statement

See CHDIR Statement.

RND Function

See RANDOMIZE Statement.

RSET Statement

See LSET Statement.

RTRIM$ Function

See LTRIM$ Function.

RUN Statement

Action Runs the program currently in memory or executes a specified program.

Syntax **RUN** [*linenumber* | *filespec$*]

Part	Description
linenumber	Line number in the current program; if omitted, execution begins at the first executable line.
filespec$	Name of a Visual Basic source (.BAS) file or compiled executable (.EXE) file; if omitted, .BAS extension is assumed.

Remarks The line where execution begins must be at the module level of a non-form code module. No executable statements are allowed at the module level of a form module.

A RUN statement in a SUB or FUNCTION procedure must point to a line label at the module level. If no line label is given, execution always starts with the first executable line of the main module.

The .BAS extension is assumed in the programming environment; however, Visual Basic assumes the .EXE extension for compiled, stand-alone programs. If the program you want to run has a different extension, you must supply the extension. If the program name has no extension, the file name given must end with a period. For example, the following code executes CATCHALL.EXE from a BC-compiled program, and executes CATCHALL.BAS within the Visual Basic programming environment.

```
RUN "CATCHALL"
```

Program lines can have line numbers or alphanumeric labels. For example:

```
OpenWindow:
```

However, the RUN statement does not accept an alphanumeric label. If an alphanumeric label is the target of a RUN statement, Visual Basic generates the error message, "Type mismatch."

Programs running within the Visual Basic programming environment (VBDOS.EXE) must call only Basic program files (.BAS). The file is loaded and run as if it were a Basic program; if it is not in the Basic program format, execution halts. The error message that appears varies, depending on the file's contents. Likewise, programs compiled with BC must not invoke Basic source files (.BAS), since these files run only in the programming environment.

An executable file need not have been written in Basic. Any executable file can be run — for example, files with the extensions .COM and .EXE.

When a program is running in the Visual Basic programming environment, if a .BAS file matching the file name in the command line cannot be found, Visual Basic generates the error message, "File not found," and control returns to the programming environment.

When running a program compiled by BC, Visual Basic generates the error message, "File not found in module program name," and control returns to the operating system.

When the invoked program completes execution, control does not return to the invoking program. If the invoking program is run outside Visual Basic, control returns to the operating system. If the invoking program is run under the Visual Basic programming environment, control returns to the programming environment.

RUN closes all files and clears program memory before loading the designated program. If you want to run a different program, but leave open files open, use the CHAIN statement.

See Also CHAIN Statement.

Example This example uses the RUN statement to restart the program. This example shows that RUN resets all numeric variables to 0. As the line number following RUN increases in lines 60, 70, 80, and 90, the variables in the earlier statements lose their assigned values.

```
10 A = 9
20 B = 7
30 C = 5
40 D = 4
50 PRINT A, B, C, D
60 IF A = 0 THEN 70 ELSE RUN 20
70 IF B = 0 THEN 80 ELSE RUN 30
80 IF C = 0 THEN 90 ELSE RUN 40
90 IF D = 0 THEN END ELSE RUN 50
```

SADD Function

Action Returns the address of a string as an offset (2-byte integer) from the current data segment; used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

Syntax **SADD**(*stringvariable$*)

Part	Description
stringvariable$	String for which you want to find the offset; can be either a simple string variable or a single element of a string array; cannot be a fixed-length string argument.

Remarks Use the SADD function with care because strings can move in the Visual Basic string space (storage area) at any time.

In a non-Basic routine, do not add characters to the beginning or end of a string passed using the SADD and LEN functions. Adding characters may cause Visual Basic to generate a run-time error.

SADD can be used with far strings; however, to obtain the segment address of a far string, use the SSEG function.

In earlier versions of Basic, SADD was used only for near strings. Visual Basic does not support the use of near strings.

Example This example creates a string and then calculates its offset and segment using the SADD and SSEG functions. The information is then passed to a Basic SUB procedure which mimics the performance of a non-Basic print routine.

```
DECLARE SUB printit (Segment%, Offset%, Length%)
DEFINT A-Z
Text$ = ".... a few well-chosen words"      ' Create the string.

' Calculate the offset, segment and length of the string.
Offset = SADD(Text$)
Segment = SSEG(Text$)
Length = LEN(Text$)

' Pass these arguments to the print routine.
CALL printit(Segment, Offset, Length)
END

SUB printit (Segment, Offset, Length)
    CLS
    DEF SEG = Segment                       ' Set the segment for peeking.
    FOR i = 0 TO Length - 1
        ' Get each character from memory, convert to ASCII, and display.
        PRINT CHR$(PEEK(i + Offset));
    NEXT i
END SUB
```

ScaleHeight, ScaleWidth Properties

Applies To Form, picture box.

Action Returns the interior height (ScaleHeight) or width (ScaleWidth) of a form or picture box; read-only at run time and design time.

Syntax [*form.*][*picturebox.*]**ScaleHeight**
[*form.*][*picturebox.*]**ScaleWidth**

Valid values are:

Property	Border	Setting	Object
ScaleHeight	With	Height - MenuBar - 2	Form
		Height - 2	Picture box
	Without	Height - MenuBar	Form
		Height	Picture box
ScaleWidth	With	Width - 2	Form or picture box
	Without	Width	Form or picture box

Remarks

ScaleHeight and ScaleWidth can be used to position controls on forms or text in a picture box.

When used together, ScaleHeight and ScaleWidth allow you to make calculations based on the client area (the inside area) of a form or picture box.

ScaleHeight and ScaleWidth values vary depending on the existence of a border (form, picture box) and menus (form). In general, the values returned are:

```
ScaleHeight = Total Height - Borders (if any) - Menus (if any)
Scale Width = Total Width - Borders (if any)
```

See Also

Height Property, Width Property.

Example

This example uses the ScaleHeight and ScaleWidth properties to display the interior height and width of a form when it is selected.

```
SUB Form_Click ()
  PRINT "Interior height of form = " + STR$(ScaleHeight)
  ➥+ CHR$(13) + "Interior width of form = " + STR$(ScaleWidth)
END SUB
```

SCREEN Function

Action

Returns an ASCII character code or color code from the screen character location specified; not available when forms are showing. Use the SCREEN object with forms.

Syntax **SCREEN** (*line%*, *column%* [, *colorflag%*])

Part	Description
line%	Character location; integer expression whose valid range depends on the number of lines on the screen.
column%	Character location; integer expression with a value between 1 and 80, inclusive.
colorflag%	Numeric expression that determines which information is returned:

Expression	Return Value
Not used or 0	ASCII character code
1	Foreground color code

Remarks References to the SCREEN function are not allowed when forms are showing and cause Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this function.

If the character at (10,10) is A, the following statement returns 65, the ASCII code for A, to the variable X:

```
X = SCREEN(10,10)
```

In graphics screen modes, if the pattern on the screen at the given character location does not exactly match any pattern in the current ASCII character set, the SCREEN function returns the ASCII code for a space.

If *colorflag%* is nonzero, the SCREEN function returns 0.

See Also COLOR Statement, SCREEN Object, SCREEN Statement.

Example This example uses the SCREEN function to return the color attribute of the character in the upper-left corner of the screen.

```
CLS                              ' Clear the screen.
X = SCREEN(1, 1, 1)
PRINT "The color attribute is: "; X
```

SCREEN Object

Action

The SCREEN object is accessed by specifying the SCREEN keyword. It is used with specific properties and methods to perform the following tasks:

Task	Method or Property
Define windowing system default colors	ControlPanel
Determine the size of the screen	Height, Width
Toggles display of all visible forms	HIDE, SHOW
Track the location of the focus	ActiveControl, ActiveForm

Use the SCREEN function or the SCREEN statement for non-form functionality.

Syntax

SCREEN.*property*[= *setting%*]
SCREEN.*method*[*arguments*]

Part	Description
property	Any applicable SCREEN property.
setting%	Valid property setting.
method	Any applicable SCREEN method.
arguments	Any arguments to the specified method.

Remarks

Methods

HIDE	SHOW

Properties

ActiveControl	ActiveForm	AutoRedraw	ControlPanel
Height	MousePointer	Width	

The SCREEN object is accessed by including the keyword SCREEN when referencing applicable methods. For example:

```
SCREEN.Height
```

Use the SCREEN object with the HIDE method to hide all visible forms with a single line of code. Use the SCREEN statement to set the screen mode back to 0 before forms are displayed. In some cases, you will also need to use the WIDTH statement to reset the screen width to 80 columns. For example:

```
SUB Form_Click ()
    SCREEN.HIDE      ' Hide any visible forms.
    SCREEN 7
    COLOR 7,4
    CIRCLE STEP(0,0), 100, , , , 5/25
```

```
    SLEEP 2
    WIDTH 80
    SCREEN 0
    SCREEN.SHOW      ' Redisplay any visible forms.
END SUB
```

See Also SCREEN Function, SCREEN Statement.

Example See MousePointer Property example.

The example changes the mouse pointer to an hourglass (X) while hearts are drawn on a form, and then changes it back to the default mouse pointer at the end of the procedure. The MousePointer property is used in conjunction with the SCREEN object.

```
SUB Form_Click ()
   SCREEN.MousePointer = 11        ' Change pointer to hourglass.
   FOR i% = 1 TO 10                ' Draw hearts (ASCII 3).
        FOR f% = 1 TO 32000        ' Create time delay.
        NEXT f%
   PRINT CHR$(3)
   NEXT i%
   SCREEN.MousePointer = 0         ' Return pointer to default.
END SUB
```

SCREEN Statement

Action Sets a screen mode for a particular combination of display and adapter; not available when forms are showing. Use the SCREEN object with forms.

Syntax **SCREEN** *mode%* [,[*colorswitch%*] [,[*activepage%*] [,*visiblepage%*]]]

Part	Description
mode%	Integer constant or expression that selects a screen mode for a particular combination of display and adapter:

Display/Adapter	Screen mode
MDPA, CGA, Hercules, Olivetti, EGA, VGA, or MCGA text only.	0
CGA, EGA, VGA, or MCGA low- and medium-resolution graphics.	1, 2
Hercules and other adapters.	3, 4

Display/Adapter	Screen mode
EGA and VGA medium- and high-resolution graphics.	7, 8, 9
EGA and VGA monochrome high-resolution graphics.	10
VGA and MCGA high-resolution graphics.	11, 12, 13

colorswitch% Integer expression that switches composite monitor display between color and non-color (modes 0 and 1 only).

activepage% Integer expression that identifies the screen page that text or graphics output is written to.

visiblepage% Integer expression that identifies the screen page that is displayed.

Remarks References to the SCREEN statement are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the SCREEN object with forms.

To use the SCREEN statement with forms, you must first unload or hide any visible forms.

If *mode%* is an expression rather than a constant and if you know your program will not use certain screen modes, you can achieve a smaller .EXE file by linking with one or more stub files:

If Not Using Screen Modes	Link to File
1 or 2	NOCGA.OBJ
3	NOHERC.OBJ
4	NOOGA.OBJ
7, 8, 9, or 10	NOEGA.OBJ
11, 12, or 13	NOVGA.OBJ

The argument *colorswitch%* is effective only for screen modes 0 and 1 and for composite monitors:

Action	Screen Mode 0	Screen Mode 1
Disable color	0	A nonzero value
Enable color	A nonzero value	0

See Also SCREEN Function, SCREEN Object.

Example See PCOPY Statement example.

SCROLLBAR_BACKCOLOR, SCROLLBAR_FORECOLOR Constants

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the background and foreground colors for all non-attached scroll bars in a program.

Syntax **SCREEN.ControlPanel({SCROLLBAR_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({SCROLLBAR_FORECOLOR | *subscript%*})[= *color%*]

Remarks

Part	Description
subscript%	SCROLLBAR_BACKCOLOR has a value of 13; SCROLLBAR_FORECOLOR has a value of 14.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
7	(Default - SCROLLBAR_BACKCOLOR) White.
0	(Default - SCROLLBAR_FORECOLOR) Black.

See Also Horizontal Scroll Bar Control, ScrollBars Property, Vertical Scroll Bar Control.

Example This example illustrates the effect of changing the value of the SCROLLBAR_BACKCOLOR and SCROLLBAR_FORECOLOR constants. To try this example, create a form containing a command button, and a horizontal scroll bar.

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(SCROLLBAR_BACKCOLOR) = 2
    SCREEN.ControlPanel(SCROLLBAR_FORECOLOR) = 4
END SUB
```

ScrollBars Property

Applies To Text box.

Action Determines if a text box has horizontal scroll bars, vertical scroll bars, or both; read-only at run time.

Syntax [*form.*]*textbox*.**ScrollBars**

Valid settings are:

Value	Scroll Bars Present
0	(Default) None.
1	Horizontal only.
2	Vertical only.
3	Both horizontal and vertical.

Remarks Use this property to add scroll bars to a text box with the MultiLine property set to True (-1).

MultiLine must be set to True (-1) before the ScrollBars property is active at design time.

At run time, the operating environment automatically puts into effect a standard keyboard interface when vertical scroll bars are present. This enables navigation in text box controls with the arrow keys, Home and End keys, and so on.

Adding vertical scroll bars to a large multiline text box further facilitates browsing and navigating through the text; adding horizontal scroll bars affects word wrapping.

See Also Horizontal Scroll Bar Control, SCROLLBAR_BACKCOLOR Constant, Vertical Scroll Bar Control, SCROLLBAR_FORECOLOR Constant.

SECOND Function

Action Evaluates a date/time serial number and returns the seconds, a value between 0 and 59, inclusive.

Syntax **SECOND**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See HOUR Function example.

SEEK Function, SEEK Statement

Action Returns the current file position (SEEK function); sets the file position for the next read or write (SEEK statement).

Syntax

SEEK(*filenumber&*)
SEEK [#]*filenumber%*, *position&*

Part	Description
filenumber& *filenumber%*	Number of an open file.
position&	Position where the next read or write occurs; for random-access files, a record number; for other files, the byte position relative to the beginning of the file where the first byte is at position 1.

Remarks

SEEK returns a value between 1 and 2,147,483,647, inclusive (equivalent to 2^31 -1).

When used on random-access files, SEEK returns the number of the next record read or written.

For files opened in binary, output, append, or input mode, SEEK returns the byte position in the file where the next operation is to take place. The first byte in a file is 1.

SEEK returns zero when used on an ISAM table or on Visual Basic devices (SCRN, CONS, KYBD, COMn, LPTn, PIPE) that do not support SEEK.

The following code prints a message indicating whether the last read or write was done in the first, second, or final third of the file:

```
SELECT CASE (SEEK(1))
    CASE IS < .333*LOF(1)
        PRINT "In first third of file."
    CASE .333*LOF(1) TO .667*LOF(1)
        PRINT "In second third of file."
    CASE IS >= .667*LOF(1)
        PRINT "In last third of file."
END SELECT
```

The argument *filenumber%* is an integer used in the OPEN statement to open the file.

The argument *position&* is a numeric expression that indicates where the next read or write occurs. The position must be between 1 and 2,147,483,647, inclusive (equivalent to 2^31 -1).

The first byte in a file is at position 1; the second byte is at position 2, and so on. After a SEEK, the next file I/O operation starts at the byte position specified by SEEK.

If you attempt a SEEK operation to a negative or zero position, Visual Basic generates the error message, "Bad record number." Performing a file write after doing a SEEK beyond end of file extends the file.

Visual Basic leaves the file position unchanged when you use SEEK on an ISAM table or on Visual Basic devices (SCRN, CONS, KYBD, COMn, LPTn, PIPE) that do not support SEEK.

Record numbers on a GET or PUT override the file positioning done by SEEK.

See Also GET Statement (File I/O), OPEN Statement (File I/O), PUT Statement (File I/O), SEEKEQ Statement, SEEKGE Statement, SEEKGT Statement.

Example See LOC Function example.

SELECT CASE Statement

Action Executes one of several statement blocks, depending on the value of an expression.

Syntax

SELECT CASE *testexpression*
  CASE *expressionlist-1*
    [*statementblock-1*]
  [**CASE** *expressionlist-2*
    [*statementblock-2*]]
  [**CASE** *expressionlist-n*
    [*statementblock-n*]]
  [**CASE ELSE**
    [*statementblock-n*]]
END SELECT

Part	Description
testexpression	Any numeric or string expression.
expressionlist	Any of the following forms or a combination, separated by commas: *expression* [,*expression*]... *expression* **TO** *expression* **IS** *relational-operator expression*
expression	Any numeric or string expression compatible with *testexpression*.
relational-operator	One of the following relational operators: `< <= > >= <> =`
statementblock	One or more statements on one or more lines.

Remarks If *testexpression* matches the expressionlist associated with a CASE clause, the statement block following that CASE clause is executed up to the next CASE clause or, for the last one, up to END SELECT. Control then passes to the statement following END SELECT.

If you use the TO keyword to indicate a range of values, the smaller value must appear first. For example, the statements associated with the line:

```
CASE -1 TO -5
```

are not executed if *testexpression* is -4. The line should be written as:

```
CASE -5 TO -1
```

You can use a relational operator only when the IS keyword appears. If CASE ELSE is used, its associated statements are executed only if *testexpression* does not match any other CASE selections. It's a good idea to have a CASE ELSE statement in your SELECT CASE block to handle unforeseen *testexpression* values.

When there is no CASE ELSE statement and no expression listed in the CASE clauses matches *testexpression*, program execution continues at the statement following END SELECT.

You can use multiple expressions or ranges in each CASE clause. For example, the following line is valid:

```
CASE 1 TO 4, 7 TO 9, 11, 13, IS > MaxNumber%
```

You can also specify ranges and multiple expressions for strings:

```
CASE "everything", "nuts" TO "soup", TestItem$
```

Note: CASE matches strings that are exactly equal to "everything," the current value of TestItem$, or that appear between "nuts" and "soup" in alphabetical order.

Strings are evaluated according to the ASCII values of their characters. Lowercase letters have larger ASCII values than uppercase letters, so this statement is true:

```
nuts > Nuts > NUTS
```

If an expression appears in more than one CASE clause, only the statements associated with the first appearance of the expression are executed.

SELECT CASE statements can be nested. Each SELECT CASE statement must have a matching END SELECT statement.

See Also IF...END IF Statement, IF TYPEOF Statement.

Example This example determines an investment strategy according to risk using the SELECT CASE statement to take different actions based on the input value.

```
INPUT "Enter acceptable level of risk (1-10): ", Total
SELECT CASE Total
CASE IS >= 10
      PRINT "Maximum risk and potential return."
      PRINT "Choose stock investment plan."
CASE 6 TO 9
      PRINT "High risk and potential return."
      PRINT "Choose corporate bonds."
CASE 2 TO 5
      PRINT "Moderate risk and return."
      PRINT "Choose mutual fund."
```

```
CASE 1
      PRINT "No risk, low return."
      PRINT "Choose IRA."
CASE ELSE
      PRINT "Response out of range."
END SELECT
```

SelLength, SelStart, SelText Properties

Applies To Combo box, text box.

Action Sets or returns the number of characters selected in a combo or text box (SelLength); the starting point of the selected text or insertion point if no text is selected (SelStart); or the currently selected text or an empty string ("") if no text is selected (SelText).

Syntax

[*form.*]{*combobox.* | *textbox.*}**SelLength**[= *length%*]
[*form.*]{*combobox.* | *textbox.*}**SelStart**[= *point%*]
[*form.*]{*combobox.* | *textbox.*}**SelText**[= *stringexpression$*]

Part	Description
length% *point%*	Valid settings are:

Value	Description
0 to text length	Valid range for SelLength and SelStart; should not exceed number of characters in the control's edit area.

Part	Description
stringexpression$	Any string expression; the selected string in a combo box or text box control.

Remarks Use the SelLength, SelStart, and SelText properties for:

- Setting the cursor position.
- Establishing an insertion range.
- Selecting substrings in a control.
- Clearing text.

Setting SelLength to less than 0 causes a run-time error. Attempting to set SelStart to a value less than zero generates the run-time error message, "Invalid property value."

Setting SelStart greater than the text length sets the property to the existing text length.

Changing SelStart sets SelLength to 0.

Setting SelText replaces the currently selected text and places the cursor just after the inserted text. SelLength is always zero after this operation.

See Also Text Property.

Example This example allows the user to specify search text, searches the text, and if the text is found, selects it. To try this example, create a form containing a text box control.

```
SUB Form_Click ()
    DIM Search$                                 ' Text to search for.
    DIM Where%                                  ' Position where found.
    ' Get search string from user.
    Search$ = INPUTBOX$("Enter text to be found:")
    Where% = INSTR(Text1.Text, Search$)         ' Find string in text.
    IF Where% THEN                              ' If found...
         Text1.SelStart = Where% - 1            ' Set start of selection.
         Text1.SelLength = LEN(Search$)         ' Set length of selection.
    ELSE
         MSGBOX "String not found."             ' Notify user.
    END IF
END SUB

SUB Form_Load ()
    Text1.Text = "Two of the peak human experiences"
    Text1.Text = Text1.Text + " are good food and classical music."
END SUB
```

Separator Property

Applies To Menu.

Action Determines if a menu control is displayed as a separator bar. The Separator property is initially set at design time by selecting the Separator check box in the Menu Design window.

Syntax [*form.*]*menuitem*.**Separator**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	Menu item is displayed as separator bar.
False (0)	(Default) Menu item is not displayed as separator bar.

Remarks The default Separator value for a menu control is False. If a submenu item's Separator property is set to True at run time, its caption is displayed as a separator bar and the menu item does not receive events.

At design time, if you type a hyphen (-) in the Caption text box of an indented menu item, the Separator property is automatically set to True. If the property is set to False at run time, the submenu item is displayed with the caption "-" and events can be received.

If you try to set the Separator property for a top-level menu item, Visual Basic generates the error message, "Separator property can't be set on this control."

See Also Checked Property, Enabled Property, Visible Property.

SETFOCUS Method

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action Moves the focus to the specified form or control.

Syntax [*form.*][*ctlname.*]**SETFOCUS**

Remarks The "focus" is the ability to receive user input through the mouse or keyboard.

When SETFOCUS is used on a form that contains controls, the form's default control receives the focus.

When SETFOCUS is used on a form that has children, the focus always goes to the child form.

After SETFOCUS is executed, user input is directed to the specified form or control.

Disabled and invisible objects cannot receive the focus.

If SETFOCUS is used on a control that cannot accept the focus, then the focus goes to the next control in the tab order.

Using SETFOCUS without specifying an object sets the focus to the form containing the statement.

See Also GotFocus Event, LostFocus Event.

Example This example uses the SETFOCUS method to change focus between two command buttons. To try this example, create a form containing two command button controls.

```
DECLARE SUB SetFocusDemo ()

SUB Form_Click ()
   CALL SetFocusDemo
END SUB
```

```
SUB SetFocusDemo ()
   Msg$ = "Choose OK to watch the focus change to the other control."
   MSGBOX Msg$                          ' Display message.
   IF SCREEN.ActiveControl.Caption = "Command1" THEN
      Command2.SETFOCUS                 ' Set focus to button 2.
   ELSE
      Command1.SETFOCUS                 ' Set focus to button 1.
   END IF
END SUB
```

SETMEM Function

Action Changes the amount of memory used by the far heap (the area where far objects are stored) and returns information about the far heap.

Syntax **SETMEM**(*numeric-expression&*)

Part	Description
numeric-expression&	Specifies the number of bytes by which the far heap will increase or decrease; in addition, specifies the return value of the function:

Numeric Expression	Far Heap	Return Value
Negative	Decrease	Maximum size of far heap.
0	No change	Current size of far heap.
Positive	Increase	Maximum size of far heap.

Remarks The first call to SETMEM to increase the far heap has no effect because Visual Basic allocates as much memory as possible to the far heap when a program starts.

If SETMEM cannot change the far heap by the requested number of bytes, it reallocates as many bytes as possible.

SETMEM can be used in mixed-language programming to decrease the far heap space so procedures in other languages can dynamically allocate far memory.

Example This example uses the SETMEM function to free memory for a C function that uses the C routine malloc to get dynamic memory.

Note: To run this program, you must separately compile the C function and put it in a Quick library or link it to the Basic program. The C function must be compiled using the large memory model, so calls to malloc use the far space freed by the Basic program.

```
DECLARE SUB CFunc CDECL (BYVAL X AS INTEGER)

BeforeCall = SETMEM(-2048) ' Decrease the size of the far heap so
                           ' CFunc can use malloc to get dynamic memory.

CFunc (1024)               ' Call the C function.

AfterCall = SETMEM(3500)   ' Return the memory to the far heap; use a larger
                           ' value so all space goes back into the heap.

IF AfterCall <= BeforeCall THEN PRINT "Memory not reallocated."
END

/* C Function */
void far cfunc(bytes)
int bytes;
{
    char *malloc();
    char *workspace;

    /* Allocate working memory using amount BASIC freed. */
    workspace=malloc((unsigned) bytes);

    /* Working space would be used here. */

    /* Free memory before returning to BASIC. */
   free(workspace);
}
```

SETTEXT Method

Applies To CLIPBOARD object.

Action Puts a text string on the CLIPBOARD object.

Syntax **CLIPBOARD.SETTEXT** *stringexpression$*

Part	Description
stringexpression$	String data to place on the CLIPBOARD.

Remarks Use the CLIPBOARD object and applicable methods to provide cut, copy, and paste commands in your applications.

See Also CLEAR Method, GETTEXT Method.

Example See CLEAR Method example.

SetUEvent Routine

Action Sets the Basic entry point that causes a user-defined event; can be used with object-specific events.

Syntax **SetUEvent**

Remarks The SetUEvent routine does not affect form- or control-specific events. Forms and controls use different event handlers. Use object-specific events with forms.

SetUEvent signals an event for the ON UEVENT event-handling routine.

The SetUEvent routine is a part of Basic, and is automatically included in compiled applications or when starting Visual Basic (VBDOS) with the /L command-line option.

Your interrupt-service routine must call SetUEvent; it is the only way to alert your program that the event has occurred.

You can call SetUEvent from a non-Basic language.

To use SetUEvent in the Visual Basic programming environment, use any Quick library. However, you do not have to link to a library to use SetUEvent outside the programming environment.

See Also ON UEVENT Statement, UEVENT Statement.

Example See ON UEVENT Statement example.

SGN Function

Action Returns a value indicating the sign of a numeric expression.

Syntax **SGN**(*numeric-expression*)

Part	Description
numeric-expression	Any valid numeric expression:

Numeric Expression	Return Value
Equal to zero	0
Positive	1
Negative	-1

See Also ABS Function.

Example See ABS Function example.

SHARED Statement

Action Gives a SUB or FUNCTION procedure access to variables declared at the module level without passing them as parameters.

Syntax **SHARED** *variable*[()] [**AS** *type*] [, *variable*[()] [**AS** *type*]]...

Part	Description
variable	Any valid array or variable name.
()	Indicates array declaration.
AS *type*	(Default - SINGLE) Declares the data type of an array or variable; use a separate AS type clause for each variable defined.

Remarks The SHARED statement shares variables between a single procedure and the module-level code. In contrast, the SHARED keyword used with COMMON or DIM shares variables among all procedures in a module.

The SHARED statement can appear only in a SUB or FUNCTION procedure.

The SHARED statement shares variables only:

- Within a single compiled module; it does not share variables with programs in the Quick library or with procedures compiled separately and linked to the program.
- Between the module-level code and a SUB or FUNCTION procedure in the same module.

To access variables in a procedure without passing them as parameters, you can use either:

- A SHARED statement in a SUB or FUNCTION procedure.
- A SHARED keyword with COMMON or DIM in the module-level code.

Example This example uses the SHARED statement to share a variable between the main program and a SUB procedure. Each time the SUB is called, an asterisk (*) is added to a STATIC array.

```
DECLARE SUB foo ()
CLS                                          ' Clear the screen.
FOR i = 1 TO 9
    CALL foo
NEXT
END

SUB foo ()
    STATIC array() AS STRING                 ' "array" is static.
    STATIC flag                              ' "flag" is static.
    SHARED i                                 ' "i" is shared with main module
                                                 ' only.

    IF flag = 0 THEN                         ' Dimension array if not already
       DIM array(1 TO 9) AS STRING           ' dimensioned.
       flag = 1
    END IF

    array(i) = STRING$(i, "*")               ' Put data into array.

    IF i = 9 THEN                            ' Print contents of array when
       FOR j = 1 TO 9                        ' it is filled.
           PRINT j; ":"; array(j)
       NEXT
   END IF

END SUB
```

SHELL Statement

Action Suspends execution of a Visual Basic program, runs a .COM, .EXE, or .BAT program, or an MS-DOS command, and resumes execution of the Visual Basic program at the statement following the SHELL statement.

Syntax **SHELL** [*commandstring$*]

Part	Description
commandstring$	Name of a program or executable file.

Remarks

SHELL with no *commandstring$* gives you a new shell (COMMAND.COM.) You then can enter operating-system commands at the prompt. Use the EXIT command to return to Visual Basic.

Any .COM file, .EXE file, .BAT program, or MS-DOS command that runs under the SHELL statement is called a "child process." Child processes are executed by the SHELL statement, which loads and runs a copy of COMMAND.COM with the /C option.

The /C option allows any parameters in *commandstring$* to be passed to the child process. It also allows redirection of standard input and output, and execution of built-in commands such as DIR and PATH.

The program name in *commandstring$* can have any extension you wish. If no extension is supplied, COMMAND.COM looks for a .COM file, then an .EXE file, and finally, a .BAT file. If COMMAND.COM is not found, SHELL issues the error message, "File not found." Visual Basic does not generate an error if COMMAND.COM cannot find the file specified in *commandstring$*, but the operating system will generate an error message.

COMMAND.COM treats any text separated from the program name by at least one blank as a program parameter.

Visual Basic remains in memory while the child process is running. When the child process finishes, Visual Basic continues.

Example

This example uses the SHELL statement to start up a new COMMAND.COM. Type EXIT at the operating system prompt to return to the programming environment.

```
CLS                                    ' Clear the screen.
SHELL
```

SHOW Method

Applies To

Form, SCREEN object.

Action

Loads a form (if not already loaded), displays it, and gives the form the focus.

Syntax

[*form.*]**SHOW** [*style%*]
SCREEN.SHOW

Part	Description
style%	Determines if the form is modal or modeless; not valid with the SCREEN object. Valid settings are:

Value	Returns
0	(Default - Modeless) Does not require closing by an OK or Cancel action before the user can switch to another form or dialog box.
1	(Modal) Requires closing by an OK or Cancel action before the user can switch to another form or dialog box. Use the HIDE method or the UNLOAD statement to continue program execution.

Remarks Use SCREEN.SHOW to display all forms that have been hidden with the command SCREEN.HIDE.

If the specified form is not loaded when the SHOW method is invoked, Visual Basic automatically loads it.

When displaying a modal form, no user input (keyboard or mouse click) can occur in any other form until the modal form is hidden or unloaded. Your program must hide or unload a modal form (usually in response to some user action) before other forms can respond to events or accept user input. Because of this, it is not a good idea to use "SHOW 1" in a Form_Load event procedure unless you explicitly code an alternate way of removing or hiding the form.

When a modeless form is displayed, code that occurs after the SHOW method is executed as it is encountered. When a modal form is displayed, code after the SHOW method is not executed until the form is hidden or unloaded.

Standard dialog boxes produced by Visual Basic functions such as MSGBOX and INPUTBOX$ are invoked by direct reference. They cannot be loaded, shown, or unloaded.

See Also HIDE Method, LOAD Statement, UNLOAD Statement.

Example See HIDE Method example.

SLEEP Statement

Action Suspends execution of the calling program; not available when forms are showing.

Syntax **SLEEP** [*seconds%*]

Part	Description
seconds%	Integer value specifying length of time (in seconds) program execution will be suspended.

Remarks

Any reference to the SLEEP statement when forms are showing causes Visual Basic to generate the error message, "Invalid when forms are showing."

To use the SLEEP statement with forms, you must first unload or hide any visible forms.

An event (such as ON COM or ON TIMER) cannot interrupt a SLEEP suspension unless its trapping is active when the event occurs. This means that trapping must have been initialized with an ON event statement, turned on with an event ON statement, and not have been disabled with an event OFF statement or an event STOP statement.

If *seconds%* is 0, omitted, or a noninteger value, the program execution is suspended until a key is pressed or a trapped event occurs.

SLEEP responds only to keystrokes that occur after it executes.

SLEEP ignores characters in the keyboard buffer that were typed before it executed.

See Also

WAIT Statement.

Example

This example uses the SLEEP statement to suspend execution for 10 seconds. There is no ON event statement, so the only way to interrupt suspension before 10 seconds have passed is to press a key.

```
CLS                                  ' Clear the screen.
PRINT "Taking a 10 second nap..."
SLEEP 10
PRINT "Awake!"
END
```

SmallChange Property

See LargeChange Property.

Sorted Property

Applies To

Combo box, list box.

Action

Determines if the elements of a combo box or list box are automatically sorted; read-only at run time.

Syntax

[*form.*]{*combobox.* | *listbox.*}**Sorted**

Valid settings are:

Value	Description
True (-1)	List items are sorted alphabetically (case-insensitive).
False (0)	(Default) Specifies no sorting of list items.

Remarks

Warning: Be careful when specifying an index using the ADDITEM method. If the index causes an element to be added to a specific location that violates the sort order, subsequent additions may not be properly sorted.

Once you set this property, Visual Basic handles all necessary string processing to maintain alphabetical order, including reestablishing the index numbers that point to list items. This is often required when items are added or removed from a list.

See Also

ADDITEM Method, List Property, ListCount Property, ListIndex Property, REMOVEITEM Method.

Example

This example shows how the Sorted property can be used. When you run the example, click the command button and choose OK to add sorted items to the list box. Note that the items are added out of order but are automatically sorted. To try this example, create a form containing a list box and a command button control. Set the Sorted property of the list box control to True.

```
DECLARE SUB AddItemSortDemo ()

SUB AddItemSortDemo ()
   Msg$ = "Choose OK to add items to your list box."
   MSGBOX Msg$                                ' Display message.
    List1.ADDITEM "Luscious"                 ' Add the entries.
    List1.ADDITEM "Lucy"
    List1.ADDITEM "Edwards"
    List1.ADDITEM "Edmark"
    List1.ADDITEM "Doo"
    List1.ADDITEM "Drew"
    List1.ADDITEM "Zoo"
    List1.ADDITEM "Zebra"
END SUB

SUB Command1_Click ()
    CALL AddItemSortDemo
END SUB
```

SOUND Statement

Action Generates a sound through your computer's speaker.

Syntax **SOUND** *frequency*, *duration*

Part	Description
frequency	Frequency of the sound in hertz; a value in the range 37 to 32,767, inclusive.
duration	Specifies number of system clock ticks the sound lasts; a value in the range 0 to 65,535, inclusive; there are 18.2 clock ticks per second.

Remarks If *duration* is 0, any current SOUND statement running in the background is turned off. If no SOUND statement is running, a *duration* of 0 has no effect.

See Also BEEP Statement, PLAY Statement (Music).

Example This example uses the SOUND statement to produce rising and descending glissando.

```
CLS                                    ' Clear the screen.
FOR i = 440 TO 1000 STEP 5
    SOUND i, i / 1000
NEXT
FOR i = 1000 TO 440 STEP -5
    SOUND i, i / 1000
NEXT
```

SPACE$ Function

Action Returns a string of spaces.

Syntax **SPACE$**(*n%*)

Part	Description
n%	Integer value specifying the number of spaces in the string.

Example This example uses the SPACE$ function to create a string of spaces (ASCII 32). The program inserts the spaces on a printed line before the number of spaces that the string contains.

```
CLS                                    ' Clear the screen.
FOR I = 1 TO 5
    X$ = SPACE$(I)
    PRINT X$; I
NEXT I
```

SPC Function

Action Skips a specified number of spaces in a PRINT, LPRINT, or PRINT # statement, or in a PRINT method.

Syntax SPC(*n%*)

Part	Description
n%	Number of spaces to skip; a value in the range 0 to 32,767, inclusive.

Remarks For screen output, the SPC function moves the text cursor to a new print position and also overwrites any existing characters on a display screen with blanks.

The *n%* blank characters are printed starting at the current print position.

The leftmost print position on an output line is always 1; to have any effect, the value of *n%* must be >= 1.

The rightmost print position is the current line width of the output device.

The behavior of an SPC function depends on the relationship between three values:

- The argument *n%*.
- The output-line print position when the SPC function is executed.
- The current output-line width.

The following table summarizes this relationship:

If Current Position On Current Line Is...	Then SPC...
n% > output-line width	Calculates (*n%* MOD width) and skips that many blanks, starting at current print position
n% < output-line width (or *n%* MOD width)	Skips to the beginning of the next line and generates a number of blanks equal to: `(n%*(width*current print position))`

See Also LPRINT Statement, PRINT Method, PRINT Statement, PRINT # Statement, SPACE$ Function, TAB Function.

Example This example uses the SPC statement to insert a number of spaces within a printed line using the PRINT statement.

```
CLS                                        ' Clear the screen.
PRINT "The following line is printed using standard screen print zones."
PRINT : PRINT "Column 1", "Column 2", "Column 3", "Column 4", "Column 5"
PRINT : PRINT
PRINT "The next line is printed using the SPC statement to achieve the"
PRINT "same results."
PRINT
PRINT "Column 1"; SPC(6); "Column 2"; SPC(6); "Column 3";
PRINT SPC(6); "Column 4"; SPC(6); "Column 5"
```

SQR Function

Action Returns the square root of a numeric expression.

Syntax **SQR**(*numeric-expression*)

Part	Description
numeric-expression	Number greater than or equal to zero.

Example This example uses the SQR function to plot the graph of y = SQR(ABS(x)) for -9 <= x <= 9.

```
CLS                             ' Clear the screen.
SCREEN 1: COLOR 1               ' Low-resolution color graphics mode.
WINDOW (-9, -.25)-(9, 3.25)     ' Convert screen to Cartesian coordinates.
LINE (-9, 0)-(9, 0)             ' Draw X-axis.
LINE (0, -.25)-(0, 3.25)        ' Draw Y-axis.
FOR X = -9 TO 9
    LINE (X, .04)-(X, -.04)     ' Put tick marks on X-axis.
NEXT X
FOR Y = .25 TO 3.25 STEP .25
    LINE (-.08, Y)-(.12, Y)     ' Put tick marks on Y-axis.
NEXT Y
PSET (-9, 3)                    ' Plot the first point of function.
FOR X = -9 TO 9 STEP .25
    Y = SQR(ABS(X))             ' SQR argument cannot be negative.
    LINE -(X, Y), 2             ' Draw a line to the next point
NEXT X
```

SSEG Function

Action Returns the segment address for a string or 0 if the argument is a null string; used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

Syntax **SSEG**(*stringvariable$*)

Part	Description
stringvariable$	String variable for which you want to find the segment address; can be either a simple string variable or a single element of a string array; cannot be a fixed-length string argument.

Remarks The offset of a string can be found by using the SADD function; both the segment and offset can be found with the SSEGADD function.

See Also DEF SEG Statement, SSEGADD Function.

Example See SADD Function example.

SSEGADD Function

Action Returns the far address of a string variable (both segment and offset); used in mixed-language programming to obtain far addresses before passing far strings to procedures written in other languages.

Syntax **SSEGADD**(*stringvariable$*)

Part	Description
stringvariable$	String variable for which you want to find an address; can be either a simple string variable or a single element of a string array; cannot be a fixed-length string argument.

Remarks The SSEGADD function combines the segment and offset information into one long integer (4 bytes).

To find the offset of a string, use the SADD function; to find the segment of a string, use the SSEG function.

See Also DEF SEG Statement, SSEG Function.

Example This example uses the SSEGADD function to obtain the Basic string address of a string used by a C procedure. The Basic program calls the printmessage C procedure which, in turn, calls Basic to obtain the string address.

Note: To run this example, you must first compile the C procedure (shown below in remarks) and either incorporate it in a Quick library or link it to the compiled Basic program.

```
DEFINT A-Z
DECLARE SUB printmessage CDECL (BYVAL farstring AS LONG)

CLS                                    ' Clear the screen.
' Create the message as an ASCIIZ string, as required by the C printf
' function.
a$ = "This is a short example of a message" + CHR$(0)
CALL printmessage(SSEGADD(a$))         ' Call the C procedure with pointers.

 /* C Routine printmessage */
 /* Prints a BASIC far string on the screen.*/

 #include <stdio.h>

 /* Define a procedure which inputs a string far pointer */
 void  printmessage (char far *farpointer)
    {
    /* print the string addressed by the far pointer */
    printf( "%s\\n", farpointer);
    }
```

STACK Function

Action Returns the maximum stack size that can be allocated (free DGROUP).

Syntax **STACK**

Remarks Use FRE(-2) to determine the remaining stack space.

See Also STACK Statement.

Example See FRE Function example.

STACK Statement

Action Resets the size of the stack.

Syntax **STACK** [*longinteger&*]

Part	Description
longinteger&	Number of bytes to reserve for the stack; if omitted, stack is reset to its default size (3K); if = 0, the minimum stack size is 0.325K.

Remarks If you request a stack space that is too large, the STACK statement allocates as much stack space as possible.

The STACK statement is allowed only at the module level of a code (.BAS) module.

The STACK statement is useful in programs that contain recursion and lots of nesting of SUB and FUNCTION procedures.

See Also STACK Function.

Example See FRE Function example.

$STATIC Metacommand

See $DYNAMIC Metacommand.

STATIC Statement

Action Makes simple variables or arrays local to a SUB or FUNCTION procedure or a DEF FN statement and preserves values between calls.

Syntax **STATIC** *variable*[()] [**AS** *type*] [, *variable*[()] [**AS** *type*]]...

Part	Description
variable	Any valid array or variable name.
()	Indicates array declaration.
AS *type*	(Default - SINGLE) Declares the data type of an array or variable; use a separate AS type clause for each variable defined.

Remarks Variables declared in a STATIC statement override variables of the same name shared by DIM or COMMON statements in the module-level code. Variables in a STATIC statement also override global constants of the same name.

Usually, variables used in DEF FN statements are global to the module; however, you can use the STATIC statement inside a DEF FN function to declare a variable as local to only that function.

The STATIC statement can appear only in a SUB or FUNCTION procedure or a DEF FN statement.

The STATIC statement, the STATIC keyword, and the $STATIC metacommand are all used to affect variables and arrays, but differ in the following ways:

Usage	Description
STATIC attribute in SUB and FUNCTION statements	Declares the default for variables as static; variables having the same name as variables shared by module-level code are still shared.
STATIC statement	Makes specific variables static; overrides any variables shared by module-level code.
$STATIC metacommand	Affects how memory is allocated for arrays.

See Also DEF FN Statement, FUNCTION Statement, $STATIC Metacommand, SUB Statement.

Example This example uses the STATIC statement to preserve the value of a variable between calls to the SUB procedure in which the variable is used. The program searches for every occurrence of a certain string expression in a file and replaces it with another string.

```
DECLARE SUB Extension ()
DECLARE SUB Search ()

CLS                                          ' Clear the screen.
INPUT "Name of file"; F1$
INPUT "String to replace"; Old$
INPUT "Replace with"; Nw$
Rep = 0: Num = 0
M = LEN(Old$)
OPEN F1$ FOR INPUT AS #1
CALL Extension
OPEN F2$ FOR OUTPUT AS #2
DO WHILE NOT EOF(1)
    LINE INPUT #1, Temp$
    CALL Search
    PRINT #2, Temp$
LOOP
CLOSE
PRINT "There were "; Rep; " substitutions in "; Num; " lines."
PRINT "Substitutions are in file "; F2$
END
```

```
STATIC SUB Extension ()
SHARED F1$, F2$
    Mark = INSTR(F1$, ".")
    IF Mark = 0 THEN
         F2$ = F1$ + ".NEW"
    ELSE
         F2$ = LEFT$(F1$, Mark - 1) + ".NEW"
    END IF
END SUB
STATIC SUB Search ()
SHARED Temp$, Old$, Nw$, Rep, Num, M
STATIC R
    Mark = INSTR(Temp$, Old$)
    WHILE Mark
        Part1$ = LEFT$(Temp$, Mark - 1)
        Part2$ = MID$(Temp$, Mark + M)
        Temp$ = Part1$ + Nw$ + Part2$
        R = R + 1
        Mark = INSTR(Temp$, Old$)
    WEND
    IF Rep = R THEN
         EXIT SUB
    ELSE
         Rep = R
         Num = Num + 1
    END IF
END SUB
```

STICK Function

Action Returns the coordinates of a joystick.

Syntax **STICK**(*n%*)

Part	Description
n%	Specifies the joystick coordinate to return:

Value	Returns
0	X coordinate of joystick A.
1	Y coordinate of joystick A.
2	X coordinate of joystick B.
3	Y coordinate of joystick B.

Remarks You must use STICK(0) before you use STICK(1), STICK(2), or STICK(3) because STICK(0) not only returns the x coordinate of joystick A, but also records the other joystick coordinates. These recorded coordinates are returned by calling STICK(1), STICK(2), or STICK(3).

Joystick coordinates range from 1 to 200 in both directions.

See Also ON STRIG Statement, STRIG Function, STRIG Statement.

Example See EVENT Statement example.

STOP Statement

Action Terminates the active program or suspends trapping of several ON event statements.

Syntax **STOP** [*n%*]

Part	Description
n%	Sets the MS-DOS error level variable used in batch files or non-Visual Basic programs (untrapped errors return a value of -1); if omitted, STOP returns a value of 0 when successful.

Remarks You can place STOP statements anywhere in a program to terminate execution.

In a program compiled with the /O option, STOP closes all files and returns to the operating system. In Visual Basic, STOP leaves files open and returns to the programming environment.

If you use the /D, /E, or /X compile option on the Basic Compiler (BC.EXE) command line, the STOP statement prints line numbers as follows:

If Your Program Has...	STOP Prints...
Line numbers	Number of the line where execution stopped
No line number for STOP	Most recent line number
No line numbers	0

In the programming environment, STOP always returns an error level of 0, even if you specify a different error level.

In previous versions of Microsoft Basic, STOP statements were used for debugging. In Visual Basic, you do not have to use STOP for debugging.

See Also END Statement, SYSTEM Statement.

Example See END Statement example.

STR$, VAL Functions

Action Returns a string representation of a number (STR$); converts a string representation of a number to a numeric value (VAL).

Syntax **STR$**(*numeric-expression*)
VAL(*stringexpression$*)

Part	Description
numeric-expression	Any numeric expression.
stringexpression$	String representation of a number.

Remarks If *numeric-expression* is positive, the string returned by the STR$ function contains a leading blank.

The argument *stringexpression$* is a sequence of characters that can be interpreted as a numeric value.

The VAL function stops reading the string at the first character that it cannot recognize as part of a number. The VAL function also strips blanks, tabs, and linefeeds from the argument string. For example, the code below returns the value 1615198:

```
VAL("1615 198th Street")
```

Example See LTRIM$ Function example.

STRIG Function

Action Returns the status of a joystick trigger; -1 is returned if the condition is true, otherwise, 0 is returned; can be used with object-specific events.

Syntax **STRIG**(*n%*)

Part	Description
n%	Value that specifies a joystick-status condition:

Value	Description
0	Lower joystick A trigger was pressed since last STRIG(0).
1	Lower joystick A trigger is currently pressed.
2	Lower joystick B trigger was pressed since last STRIG(2).

Value	Description
3	Lower joystick B trigger is currently pressed.
4	Upper joystick A trigger was pressed since last STRIG(4).
5	Upper joystick A trigger is currently pressed.
6	Upper joystick B trigger was pressed since last STRIG(6).
7	Upper joystick B trigger is currently pressed.

Remarks The argument *n%* is an integer between 0 and 7 that indicates the joystick and trigger to check. The following list describes the values returned by the STRIG function for different values of *n%*:

Argument	Value Returned	Description
0	-1	Joystick A's lower trigger was pressed since the last STRIG(0) call.
	0	A's lower trigger was not pressed.
1	-1	Joystick A's lower trigger is currently down.
	0	A's lower trigger is not down.
2	-1	Joystick B's lower trigger was pressed since the last STRIG(2) call.
	0	B's lower trigger was not pressed.
3	-1	Joystick B's lower trigger is currently down.
	0	B's lower trigger is not down.
4	-1	Joystick A's upper trigger was pressed since the last STRIG(4) call.
	0	A's upper trigger was not pressed.
5	-1	Joystick A's upper trigger is currently down.
	0	A's upper trigger is not down.
6	-1	Joystick B's upper trigger was pressed since the last STRIG(6) call.
	0	B's upper trigger was not pressed.
7	-1	Joystick B's upper trigger is currently down.
	0	B's upper trigger is not down.

You cannot use the STRIG function inside a joystick event trap, because the trigger information used by STRIG function will not be available for ON STRIG.

In previous versions of Microsoft Basic, the statement STRIG ON enabled testing of the joystick triggers; STRIG OFF disabled joystick trigger testing. The current version of Basic ignores STRIG ON and STRIG OFF statements.

The earlier STRIG ON and STRIG OFF should not be confused with the current STRIG(*n*%) ON, STRIG(*n*%) OFF, and STRIG(*n*%) STOP statements. Note: The STRIG(*n*%) statements enable, disable, and inhibit trapping of joystick events.

See Also ON STRIG Statement, STICK Function, STRIG Statement.

Example See EVENT Statement example.

STRIG Statement

Action Enables, disables, or suspends joystick event trapping; can be used with object-specific events.

Syntax **STRIG(***n*%**) {ON | OFF | STOP}**

Part	Description
n%	Value that specifies a joystick trigger:

Value	Description
0	Lower trigger, joystick A.
2	Lower trigger, joystick B.
4	Upper trigger, joystick A.
6	Upper trigger, joystick B.

ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a STRIG ON statement is executed.

Remarks If your program contains event-handling statements and you are compiling from the Basic Compiler (BC.EXE) command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

STRIG(*n*%) ON enables joystick-event trapping. If joystick trigger *n*% is pressed after a STRIG(*n*%) ON statement, the routine specified in the ON STRIG statement is executed.

STRIG(*n*%) OFF disables joystick-event trapping. No trapping takes place until another STRIG(*n*%) ON statement is executed. Events occurring while trapping is off are ignored.

STRIG(*n*%) STOP suspends joystick-event trapping. No trapping takes place until a STRIG(*n*%) ON statement is executed. Any ON STRIG events that occur while trapping is suspended are remembered and processed when the next STRIG(*n*%) ON statement is executed. However, remembered events are lost if STRIG(*n*%) OFF is executed.

When a joystick event trap occurs (that is, the GOSUB is performed), an automatic STRIG(*n*%) STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a STRIG(*n*%) ON statement unless an explicit STRIG(*n*%) OFF was performed inside the routine.

In previous versions of Basic, the statement STRIG ON enabled testing of the joystick triggers; STRIG OFF disabled joystick-trigger testing. The current version of Basic ignores the earlier versions of the STRIG ON and STRIG OFF statements.

See Also ON STRIG Statement, STICK Function, STRIG Function.

Example See EVENT Statement example.

STRING$ Function

Action Returns a string of a specified length made up of a repeating character.

Syntax **STRING$**(*length%*, {*ascii-code%* | *stringexpression$*})

Part	Description
length%	Length of the string.
ascii-code%	ASCII code of the repeating character.
stringexpression$	Any string expression; the string is filled with the first character in the expression.

Example This example uses the STRING$ function to generate a bar graph.

```
CLS                                     ' Clear the screen.
PRINT TAB(7); "Daily Mean Temperature in Seattle"
PRINT
FOR MMonth = 1 TO 12 STEP 2             ' Get data for each month and graph
    READ Month$, Temp
     PRINT Month$; " +"; STRING$(Temp - 35, "*")  ' Print Temp-35 stars
    PRINT "    |"
NEXT MMonth
PRINT "    +";                          ' Print horizontal line
FOR X = 1 TO 7
    PRINT "----+";
NEXT X
PRINT
```

```
FOR X = 4 TO 39 STEP 5                          ' Print temperature labels
    PRINT TAB(X); X + 31;
NEXT X
PRINT

DATA Jan, 40, Mar, 46, May, 56
DATA Jul, 66, Sep, 61, Nov, 46
```

StringAddress Routine

Action Used by a non-Basic routine to return the far address of a Visual Basic variable-length string.

Syntax *far-address* = **StringAddress**(*string-descriptor%*);

Part	Description
string-descriptor%	Near address of a string descriptor within a non-Basic routine.

Remarks The StringAddress syntax is for the C language.

If you are not doing mixed-language programming, there is usually no reason to use StringAddress. In Visual Basic, you can use SSEGADD to find the far address of a variable-length string.

Calls to the StringAddress routine are always made from a non-Basic routine to find the address of a string that was transferred to Visual Basic from the non-Basic routine.

For example, assume that you have passed a string from a MASM routine to Visual Basic's string space using StringAssign. The descriptor for the string exists at offset descriptor. MASM can find the far address of the string data by using the following code:

```
.model
extrn       stringaddress: far

.code
lea         ax, descriptor    ; offset of descriptor
push        ax
call        stringaddress
```

The far address is returned in DX:AX. DX holds the segment and AX holds the offset.

See Also StringAssign Routine, StringLength Routine, StringRelease Routine.

Example This example uses the StringAddress routine to pass a Basic string to an assembly-language program. The StringLength routine is used to determine the length of the string.

Note To run this program, you must compile the following code and link it with the assembled MASM program (shown below in remarks) and the appropriate run-time library (VBDCL10.LIB).

```
DEFINT A-Z

CLS                                  ' Clear the screen.
' Create a variable-length string.
Message1$ = "This string was sent to MASM for printing."
CALL PrintMessage1(Message1$)        ' Pass it to MASM to be printed

; ************************PrintMessage1************************
;   This MASM procedure prints a string passed to it by Basic.

            .model      medium, basic       ;Use same model as Basic
;Define external (BASIC library) procedures
             extrn   StringAddress: far
             extrn   StringLength: far

            .code
;Define procedure with one word argument
PrintMessage1  proc     uses ds,  descriptor

             mov     ax, descriptor          ;Find length of string.
             push        ax
             call        StringLength
             push        ax                  ;Save length on stack.

             mov         ax, descriptor      ;Go find out the
             push    ax                      ;address of the string
             CALL    StringAddress           ;data.

             mov         ds, dx              ;Address of string.l.
             mov         dx, ax
             mov     bx, 01                  ;Handle of printer
             pop     cx                      ;Length of string
             mov     ah, 40h                 ;Have MS-DOS print it
             int         21h
             ret
 PrintMessage1   endp

             end
```

StringAssign Routine

Action Transfers a string from Visual Basic to a non-Basic routine, or from a non-Basic routine to Visual Basic.

Syntax **StringAssign**(*sourceaddress&*, *sourcelength%*, *destaddress&*, *destlength%*);

Part	Description
sourceaddress& *destaddress&*	Long integer value determined by string type:

String Type	Argument Value
Fixed-length	Far address of the start of string.
Variable-length	Far address of the string descriptor.

Part	Description
sourcelength% *destlength%*	Integer value determined by string type:

String Type	Argument Value
Fixed-length	String length
Variable-length	0

Remarks The StringAssign syntax is for the C language. However, the order of arguments in the syntax follows Basic, Pascal, and FORTRAN calling conventions rather than the C calling convention. Note: In Basic, Pascal, and FORTRAN, the arguments are passed in the order in which they appear in the source code. In C, the arguments are passed in the reverse order.

Calls to the StringAssign routine are usually made from a C, MASM, Pascal, or FORTRAN routine that performs string processing functions on Basic strings. Seldom, if ever, do you call StringAssign from inside a Visual Basic program.

StringAssign can be used to transfer far strings. Using StringAssign, you can write far string independent mixed-language programs.

MASM, C, Pascal, and FORTRAN deal only with fixed-length strings. When programming in these languages you can use StringAssign to create a new Basic variable-length string and transfer fixed-length string data to it. For example, to transfer a MASM fixed-length string containing the word "Hello" to a Basic variable-length string, you use this data structure:

```
fixedstring     db      "Hello"     ; source of data
descriptor      dd       0          ; descriptor for destination
```

The second data element, descriptor, is a 4-byte string descriptor initialized to zero. Visual Basic interprets this to mean that it should create a new variable-length string and associate it with the address labeled descriptor.

When creating a new variable-length string, you must allocate 4 bytes of static data for a string descriptor, as shown above. Allocating the data on the stack will not work.

To create a new Basic variable-length string and transfer fixed-length data to it, the StringAssign routine requires four arguments:

- The far address of the fixed-length string in the MASM data segment.
- The length of the fixed-length string in the MASM data segment.
- The far address of the string descriptor in the MASM data segment.
- 0 (indicating the string in the Basic data segment will be a variable-length string).

The following MASM code pushes these arguments on the stack and calls StringAssign:

```
.model
extrn      stringassign:  far

.code
push       ds               ; segment of fixed-length string
lea        ax, fixedstring  ; offset of fixed-length string
push       ax
mov        ax, 5            ; length of "Hello"
push       ax
push       ds               ; segment of descriptor
lea        ax, descriptor   ; offset of descriptor
push       ax
mov        ax, 0            ; 0 = variable-length string
push       ax
call       stringassign
```

When the call to StringAssign is made, Visual Basic fills in the double-word descriptor with the correct string descriptor.

See Also StringAddress Routine, StringLength Routine, StringRelease Routine.

Example This example uses the StringAssign and StringRelease routines to get a string from an assembly-language program.

Note: To run this program, you must compile the following code and link it with the assembled MASM program (shown below in remarks) and the appropriate run-time library (VBDCL10.LIB).

```
DEFINT A-Z

' Declare external MASM procedures.
DECLARE FUNCTION MakeString$
DECLARE SUB ReleaseIt
CLS                                  ' Clear the screen.
```

```
' Get string from MASM and print it.
PRINT MakeString

' Have MASM release the variable-length string it created.
CALL ReleaseIt

; ********************** MakeString ********************************

 ;This MASM program creates a fixed-length string and then assigns
 ;it to a Basic variable-length string. It then releases the string
 ;after Basic has used it.

                .model  medium, basic   ;Use same model as BASIC

 ;Declare routines to be used in Basic library.
                extrn   StringAssign: far
                extrn   StringRelease: far

                .data
 ;Create MASM string and a place for Basic to create a variable-
 ;length string descriptor.
 String2        db      "This is a string created by MASM."
 Descriptor     dd      0

                .code
 MakeString     proc                    ;Push arguments:
                push    ds          ;MASM string segment.
                lea     ax, String2     ;MASM string offset.
                push    ax
                mov     ax, 33          ;MASM string length.
                push    ax
                push    ds          ;Basic descriptor segment.
                lea     ax, Descriptor
                push    ax          ;BASIC descriptor offset
                xor     ax, ax          ;variable-length indicator.
                push    ax
                call        StringAssign   ;Assign the string.

                lea     ax, descriptor  ;Return with descriptor
                ret                     ;address in AX.
 MakeString     endp
```

```
ReleaseIt       proc                ;Routine to release
                lea     ax, Descriptor  ;the string.
                push    ax
                call    StringRelease
                ret
ReleaseIt       endp
                end
```

StringLength Routine

Action Used by a non-Basic routine to return the number of characters in a Visual Basic variable-length string.

Syntax *length* = **StringLength**(*string-descriptor%*);

Part	Description
string-descriptor%	Near address of a string descriptor within a non-Basic routine.

Remarks The StringLength syntax is for the C language.

If you are not doing mixed-language programming, there is usually no reason to use StringLength. Instead, use LEN to find the length of a variable-length string.

Calls to the StringLength routine are usually made from a non-Basic routine.

For example, assume that a MASM routine takes a pointer to a string descriptor as a parameter. MASM can find the length of the string data by using the following code:

```
.model
extrn       stringlength: far

.code
mov         ax, psdparm          ; offset of descriptor
push        ax
call        stringlength
```

The length of the string is returned in AX.

See Also LEN Function, StringAddress Routine, StringAssign Routine, StringRelease Routine.

Example See StringAddress Routine example.

StringRelease Routine

Action Used by a non-Basic routine to deallocate variable-length strings that have been transferred to Visual Basic's string space from the non-Basic routine.

Syntax **StringRelease**(*string-descriptor%*);

Part	Description
string-descriptor%	Near address of a string descriptor within a non-Basic routine.

Remarks The StringRelease syntax is for the C language.

Use the StringRelease routine only to deallocate variable-length strings that have been transferred to Visual Basic's string space from a non-Basic routine. Never use it on strings created by Visual Basic.

Calls to the StringRelease routine are made from a non-Basic routine to free up space in Visual Basic's data area.

Visual Basic automatically deallocates strings allocated by Basic. However, strings that have been transferred to Visual Basic from a non-Basic routine should be deallocated from the non-Basic routine using the StringRelease routine.

Using StringRelease to deallocate strings from a non-Basic routine is necessary because StringAssign transfers only strings, not string descriptors. Without the string descriptor, Visual Basic cannot deallocate the string.

For example, assume that you have passed a string from a MASM routine to Visual Basic's string space using StringAssign. To deallocate this string, assuming a descriptor for the variable-length string exists at offset descriptor, the code is:

```
.model
extrn        stringrelease: far

.code
lea          ax, descriptor       ; offset of descriptor
push         ax
call         stringrelease
```

See Also StringAddress Routine, StringAssign Routine, StringLength Routine.

Example See StringAssign Routine example.

Style Property

Applies To Combo box.

Action Determines the type of combo box and the behavior of the list box portion of the control; read-only at run time.

Syntax [*form.*]*combobox*.**Style**

Valid property settings are:

Value	Type	Includes	User Action
0	Dropdown Combo	(Default) Edit area and a list.	Display list by clicking the down scroll arrows. The user can select from the list or type in the edit area. The list closes when the user makes a selection.
1	Simple Combo	Edit area and a list always displayed.	Select from a list or type in the edit area.
2	Dropdown	Drop-down list with no edit capabilities.	Click scroll arrow to display the list. This style permits selection of a list item only. The list closes when the user makes a selection.

Remarks Use setting 0 (Dropdown Combo) or setting 1 (Simple Combo) to allow the user to enter a choice in the edit area.

Style 0 saves space on a form because the list portion closes when the user makes a selection.

Use setting 2 (Dropdown List) to display a fixed list of choices from which the user can select one. The list portion closes when the user selects an item.

The size of a Simple combo box (Style = 1) includes both the text and list portions of the control. By default, a Simple combo box is sized so that none of the list shows. Increase the Height property to show more of the list.

The DropDown event and the SelLength, SelStart, and SelText properties do not apply when Style = 1.

When Style = 2, no Change event is generated.

See Also Change Event, Click Event, DblClick Event, DropDown Event.

Example See Combo Box Control example.

SUB Statement

Action Declares the name, arguments, and code that form the body of a SUB procedure.

Syntax

[**STATIC**] **SUB** *globalname* [([**BYVAL**]*variable*[( )][**AS** *type*][,[**BYVAL**]*variable*[( )] [**AS** *type*]]...)]
    [*statementblock*]
    [**EXIT SUB**]
    [*statementblock*]
END SUB

Part	Description
STATIC	Indicates that the SUB procedure's local variables are preserved between calls; does not affect variables declared outside the procedure, even if they are used in the SUB.
globalname	Name of the SUB; a variable name up to 40 characters long that cannot appear in any other SUB or FUNCTION statement in the same project.
BYVAL	Indicates the parameter is passed by value rather than by reference; cannot be used with a variable of a user-defined type.
variable	Name of variable to pass as an argument; for array variables, empty parentheses are required.
AS *type*	Data type of the variable: INTEGER, LONG, SINGLE, DOUBLE, STRING, CURRENCY, user-defined type, or two additional types, FORM and CONTROL.
statementblock	Any group of statements to execute within the body of the SUB procedure.
EXIT SUB	Causes an immediate exit from a SUB procedure; execution continues with the statement after the statement that called the SUB; a SUB may have multiple EXIT SUB statements.

Remarks SUB and END SUB mark the beginning and end of a SUB procedure. You can use the optional EXIT SUB statement to exit a SUB from any point within the procedure.

STATIC indicates that the values of a SUB's local variables are saved between calls. Without STATIC, the local variables are allocated each time the SUB is invoked, and the variables' values are lost when the SUB returns to the calling program.

Using the STATIC attribute increases execution speed slightly.

The STATIC attribute does not affect variables that are used in a SUB, but declared outside the SUB in DIM or COMMON statements using the SHARED statement.

The SUB name cannot appear in any other SUB or FUNCTION statement in the same application unless the SUB is in form code. Code within a form module is known only to the procedures in that module.

BYVAL can be used only with simple numeric types (INTEGER, LONG, SINGLE, DOUBLE, or CURRENCY). The additional types, FORM and CONTROL, can only be used when a form or control is passed as an argument to a SUB or FUNCTION procedure.

Unless identified with the BYVAL keyword, individual arguments are passed by reference, so any change to an argument's value inside the SUB procedure changes its value in the calling procedure.

If a parameter is passed by reference, any change to the parameter's value inside the procedure changes its value in the calling program.

If no value is assigned to the SUB name, the procedure returns a default value: A numeric value returns zero; a string function returns the null string ("").

A SUB can be recursive — that is, it can call itself to perform a given task. The STATIC keyword is usually not used with recursive SUB procedures.

Any SUB procedure variables or arrays are considered local to that SUB procedure, unless they are explicitly declared as shared variables in a SHARED statement.

Avoid using I/O statements in a SUB procedure called from an I/O statement; they can cause unpredictable results. Also, because Visual Basic may rearrange arithmetic expressions to attain greater efficiency, avoid using SUB procedures that change program variables in arithmetic expressions.

A SUB procedure is a separate procedure, like a FUNCTION procedure. It can accept arguments, perform a series of statement, and change the values of its arguments. However, unlike a FUNCTION procedure, which returns a value, a SUB procedure cannot be used in an expression.

SUB procedures are called by a CALL statement or by using the SUB procedure name followed by the argument list.

Any SUB procedure variables or arrays are considered local to that SUB procedure, unless they are explicitly declared as shared variables in a SHARED statement.

When you call the SUB procedure, you can specify that an argument's value not be changed by the procedure by enclosing the argument in parentheses.

You cannot use GOSUB, GOTO, or RETURN to enter or exit a SUB procedure.

You cannot use an array of fixed-length strings as an argument when calling a SUB procedure because Visual Basic converts a simple fixed-length string argument to a variable-length string argument before passing the string to a SUB procedure.

See Also CALL Statement (Basic Procedures), DECLARE Statement, FUNCTION Statement.

Example This example demonstrates buffered input. User input is allowed to the screen, but the total number of typed or input characters is limited. The computer beeps if you exceed the limit.

```
DECLARE SUB BufInput (StringVar$, Limit%)

CLS                                     ' Clear the screen.
Limit% = 10 'Number of characters to input

LOCATE 10, 5                            ' Prompt for the input string.
PRINT "PLEASE INPUT A STRING OF TEN CHARACTERS: ";
COLOR 0, 7: PRINT SPACE$(Limit%)
LOCATE 10, 46

CALL BufInput(a$, Limit%)               ' Input a buffered string.

COLOR 7, 0
CLS

LOCATE 10, 10                           ' Print the result.
PRINT "THE STRING YOU INPUT WAS: ";
COLOR 0, 7
PRINT a$
COLOR 7, 0
END

SUB BufInput (StringVar$, Limit%)
BackSp$ = CHR$(8)
Enter$ = CHR$(13)
Length% = 0

'Loop waiting for input. ENTER terminates the routine.
WHILE char$ <> Enter$
char$ = INKEY$
   IF char$ <> "" THEN
     IF (Length% < Limit%) OR (char$ = BackSp$) THEN
        SELECT CASE char$
        CASE " " TO "~"  ' Input printable characters only.
           StringVar$ = StringVar$ + char$
           PRINT char$;
           Length% = Length% + 1
        CASE BackSp$  'If BackSpace is pressed, erase.
          IF Length% <> 0 THEN
             Length% = Length% - 1
             StringVar$ = LEFT$(StringVar$, Length%)
             CurrX% = CSRLIN
```

```
            CurrY% = POS(0) - 1
            LOCATE CurrX%, CurrY%
            PRINT " "
            LOCATE CurrX%, CurrY%
         END IF
      END SELECT
   ELSE
      PLAY "o1AL32"
   END IF
 END IF
WEND
END SUB
```

SWAP Statement

Action Exchanges the values of two variables.

Syntax **SWAP** *variable1*, *variable2*

Part	Description
variable1 *variable2*	Two variables of the same data type.

Remarks SWAP is an assignment statement that exchanges the values of two variables.

Any type of variable can be swapped (integer, long, single-precision, double-precision, string, currency, or user-defined). However, if the two variables are not exactly of the same type, Visual Basic generates the error message, "Type mismatch."

Example This example uses the SWAP statement to exchange the values of two variables.

```
CLS                              ' Clear the screen.
a% = 1: b% = 2
PRINT "Before: "; a%, b%
SWAP a%, b%
PRINT "After:  "; a%, b%
```

System Property

See Archive Property.

SYSTEM Statement

Action Closes all open files and returns control to the operating system.

Syntax **SYSTEM** [*n%*]

Part	Description
n%	Sets the MS-DOS error level variable used in batch files or non-Basic programs; untrapped and fatal errors return a value of -1.

Remarks A program containing a SYSTEM statement exits to the operating system if run from the VBDOS command line with the /RUN option.

Entering a SYSTEM statement in the Immediate window terminates Visual Basic.

In a program compiled with the /O option, SYSTEM returns control to the operating system, returns the value of *n%* to the operating system, and closes all files. In Visual Basic, SYSTEM stops program execution and closes all files.

See Also END Statement, STOP Statement, System Property.

Example See END Statement example.

TAB Function

Action Moves the text cursor to a specified print position when used in the PRINT, LPRINT, and PRINT # statements, or with the PRINT method.

Syntax **TAB**(*column%*)

Part	Description
column%	Column number of the new print position.

Remarks The leftmost print position is always 1. The rightmost position is the current line width of the output device, which can be set with the WIDTH statement.

Here is an example of using TAB in the PRINT function:

```
PRINT  TAB(10); City$; TAB(40); State$; TAB(45); Zip$
```

The behavior of a TAB function depends on the relationship between three values:

- The argument *column%*.
- The current print position on the current output line when TAB is executed.
- The current output-line width.

The following table summarizes this relationship:

If Current Position On Current Line Is...	Then TAB...
> *column%*	Skips to *column%* on next output line.
column% < 1	Moves print position to column 1.
column% > output-line width	Calculates print position equal to: `column% MOD width`
Calculated position < current print position	Jumps to next line at the calculated print position.
Calculated position > current print position	Moves to calculated print position on the same line.

See Also LPRINT Statement, PRINT Method, PRINT Statement, PRINT # Statement, SPACE$ Function, SPC Function.

Example This example uses the TAB function with the PRINT and PRINT USING statements to center a text string and position a page number at a particular location on an 80-column screen.

```
CLS                                  ' Clear the screen.
INPUT "What is your full name "; Name$
PRINT
PRINT "Do you want the page number <C>entered or <R>ight-justified? "
DO
    Answer$ = UCASE$(INKEY$)
LOOP UNTIL Answer$ = "C" OR Answer$ = "R"
CLS
PageNumber% = INT(RND * 10)
' Divide the length of Name$ by 2 and subtract from screen center point.
PRINT TAB(40 - (LEN(Name$) \ 2)); Name$
LOCATE 23, 1
IF Answer$ = "C" THEN
    PRINT TAB(35); USING "Page ##"; PageNumber%
ELSE
    PRINT TAB(72); USING "Page ##"; PageNumber%
END IF
END
```

TabIndex Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, frame, horizontal scroll bar, label, list box, option button, picture box, text box, vertical scroll bar.

Action Sets or returns a control's position in the tab order within its parent form.

Syntax [*form.*]*ctlname*.**TabIndex**[= *index%*]

Part	Description
index%	Valid settings are:

Value	Description
0 to (n-1)	Valid range of settings where n is the number of controls on a form with a TabIndex property.

Remarks Assigning a TabIndex value of less than 0 generates an error.

By default, Visual Basic assigns a tab order to the controls in the order in which you draw them on a form.

TabIndex for all controls on a form are always unique and consecutive (from 0 to n-1) within that form.

If you change the value of a control's TabIndex property to adjust the default tab order, Visual Basic automatically renumbers the TabIndex of the other controls to reflect insertions and deletions.

A form cannot get the focus if any of its children are able to receive the focus. In contrast, a picture box can receive the focus even when it contains child controls.

See Also TabStop Property.

Example The example reverses the tab order of a group of buttons by changing the TabIndex property on a command button array when you click the form. To try this example, create a form containing four command button controls. Set the CtlName property to "CommandX" for all four command buttons.

```
SUB Form_Click ()
    DIM X%          ' Reverse tab order by setting start value of X%.
    IF CommandX(0).TabIndex = 0 THEN X% = 4 ELSE X% = 1
    FOR I% = 0 TO 3
```

```
            CommandX(I%).Caption = STR$(X%)      ' Set caption.
            CommandX(I%).TabIndex = X% - 1       ' Set tab order.
            IF CommandX(0).TabIndex = 3 THEN     ' Increment or decrement X%.
                 X% = X% - 1
            ELSE
                 X% = X% + 1
            END IF
        NEXT I%
    END SUB
```

TabStop Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, horizontal scroll bar, list box, option button, picture box, text box, vertical scroll bar.

Action Determines if tabbing will stop at the control.

Syntax [*form.*]*ctlname*.**TabStop**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Setting	Description
True (-1)	(Default) Designates control as a tab stop.
False (0)	Bypasses control when the user is tabbing; the control still holds its place in the actual tab order as determined by the TabIndex property.

Remarks Instead of having the Tab key move through each button in an option button group, you can designate that the whole group be treated as one tab location. To do this, set TabStop to True (-1) for the first option button in the group and set TabStop to False (0) for the other option buttons in the group.

A form cannot get the focus if any of its children are able to receive the focus. In contrast, a picture box can receive the focus even when it contains child controls.

See Also TabIndex Property.

Tag Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, timer, vertical scroll bar.

Action Enables identification of a control or form that is passed to a procedure; stores unique data with an object.

Syntax [*form.*][*ctlname.*]**Tag**[= *stringexpression$*]

Part	Description
stringexpression$	String enclosed in quotes, string variable, or any other string expression; default is an empty string ("") .

Remarks Use this property to assign a unique identification string to an object without affecting any of its other property settings or causing side effects.

Use the Tag property when you need to identify a specific control that is passed to a procedure. This is done because CtlName and FormName cannot be used as unique identifiers at run time.

The Tag property is for generic data storage. You can use the Tag property to attach information to a control that is not already handled by any of its other properties.

See Also CtlName Property, FormName Property.

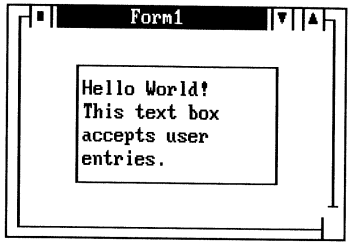
Example The example uses the Tag property to change the backcolor of a picture box based upon the Tag of the control that is dragged over it.

```
SUB Picture3_DragOver (Source AS CONTROL, X AS SINGLE, Y AS SINGLE,
➥State AS INTEGER)
    CONST ENTER = 0, LEAVE = 1, OVER = 2
    IF State = ENTER THEN
         ' Select based on Tag since CtlName is not available at run time.
         SELECT CASE Source.Tag
         CASE "Pic1"
              Picture3.BackColor = 4
         CASE "Pic2"
              Picture3.BackColor = 10
         END SELECT
    ELSEIF State = LEAVE THEN
         Source.BackColor = Picture3.BackColor
    END IF
END SUB
```

Text Box Control

Action

Provides an area (called an edit field) for the user to enter text; an application can manipulate the contents of a text box using the Text property.



Events

Change	DragDrop	DragOver	GotFocus
KeyDown	KeyPress	KeyUp	LostFocus

Methods

DRAG	MOVE	REFRESH	SETFOCUS

Properties

BackColor	BorderStyle	CtlName	DragMode
Enabled	ForeColor	Height	Index
Left	MousePointer	MultiLine	Parent
ScrollBars	SelLength	SelStart	SelText
TabIndex	TabStop	Tag	Text
Top	Visible	Width	

Remarks

A text box, sometimes called an edit field or edit control, is used to enter or display up to 32K of text. Data displayed in a text box can be defined at design time, or at run time by requesting input from the user.

Use the built-in capabilities of the CLIPBOARD object to write simple cut-and-paste routines using a text box.

At design time, only one line of text can be displayed in a text box regardless of the text box's size. To display multiple lines of text in a text box at run time, set the MultiLine property to True (-1).

Use the VAL function to convert a string in a text box to an integer.

The minimum size of a text box is 1 x 2; dimensions correspond to the Height and Width property settings. Specified settings that are less than the minimum value are ignored.

When BorderStyle property = 1, Height settings 1 and 2 change the appearance of the control.

Even if the ScrollBars property is turned on, scroll bars attached to a text box will not be displayed unless the following settings are made:

Scroll Bar	Border	Text Box Property
Horizontal	Present	Width > 3
Horizontal	Not present	Width > 1
Vertical	Present	Height > 3
Vertical	Not present	Height > 1

Example

This example uses three text boxes and a command button. As the user types into Text1, Text2 displays the text as lowercase and Text3 displays the text as uppercase. Clicking Command1 clears all three text boxes.

```
SUB Command1_Click ()                    ' Delete text.
   Text1.Text = ""
END SUB

SUB Text1_Change ()
   Text2.Text = LCASE$(Text1.Text)  ' Display text as lowercase.
   Text3.Text = UCASE$(Text1.Text)  ' Display text as uppercase.
END SUB
```

Text Property

Applies To

Combo box, list box, text box.

Action

Sets or returns the text contained in a text box or combo box, or the selected string in a list box; read-only at run time when used with a list box (or combo box if Style = 2); text can be changed by the user at run time.

Syntax

[*form.*]{*combobox.* | *listbox.* | *textbox.*}**Text**[= *stringexpression$*]

Part	Description
stringexpression$	Text or selected item; at design time only, defaults for this property are:
@	

Controls	Default
Combo box or text box.	CtlName
List box (or combo box with Style = 2).	Empty string ("")

Remarks

A text box can contain up to 32K of text.

The value returned from a list box is always equivalent to the value returned by the expression:

```
List(ctlname.ListIndex)
```

The Text property can be used to manipulate the string contained in the edit area of combo box controls with Style = 0 (Dropdown Combo) or Style = 1 (Simple Combo).

Use the Text property to determine the currenlty selected item in the list for combo box controls with Style = 2 (Dropdown List).

See Also

Caption Property, SelLength Property, SelStart Property, SelText Property.

Example

See Text Box Control example.

TEXTHEIGHT, TEXTWIDTH Methods

Applies To

Form, picture box.

Action

Returns the height in lines (TEXTHEIGHT) or length in characters (TEXTWIDTH) of a text string.

Syntax

[*form.* | *picturebox.*]**TEXTHEIGHT**(*stringexpression$*)
[*form.* | *picturebox.*]**TEXTWIDTH**(*stringexpression$*)

Part	Description
stringexpression$	Text string for which the height or length is determined.

Remarks

TEXTHEIGHT returns 1 + the number of CR+LF sequences (the number of lines) contained in *stringexpression$*.

If *stringexpression$* contains embedded carriage returns, TEXTWIDTH returns the length of the longest line.

If *stringexpression$* does not contain any embedded carriage returns, then TEXTWIDTH returns the display length of *stringexpression$*. This value is equivalent to:

```
LEN(stringexpression$)
```

See Also LEN Function.

Example This example uses the TEXTHEIGHT and TEXTWIDTH methods to determine the display height and width of a string expression. To try this example, create a form containing a picture box control.

```
SUB Form_Click ()
  MsgA$ = "A mistake is evidence that"
  MsgB$ = CHR$(13) + CHR$(10) + " someone has tried to do something."
  Msg$ = MsgA$ + MsgB$
  Picture1.PRINT Msg$
  PRINT "The TEXTHEIGHT of the picture box string is: ";
  Picture1.TEXTHEIGHT(Msg$)
  PRINT "The TEXTWIDTH of the picture box string is: ";
  Picture1.TEXTWIDTH(Msg$)
END SUB
```

THREE_D Constant

Applies To ControlPanel property, SCREEN object.

Action Toggles three-dimensional (3D) display effect for controls with borders.

Syntax **SCREEN.ControlPanel({THREE_D** | *subscript%*})[= *setting%*]

Part	Description
subscript%	THREE_D has a value of 15.
setting%	Valid settings are:

Value	Description
True (-1)	(Default) 3D effect is enabled.
False (0)	3D effect is disabled.

Remarks The three-dimensional (3D) effect applies only to controls with borders around them.

A form containing 3D controls becomes, in effect, a 3D form. However, the THREE_D constant has no effect on a form's border.

When THREE_D is set to True (-1), half of the control border is drawn in black, and the other half is drawn in the same color, but opposite intensity of the control's background.

When THREE_D is set to False (0), control borders are drawn in the foreground color of the control.

Command buttons have the black border along the bottom and right edges, thus producing a "raised" look. All other framed controls have the black border along the top and left edges, producing an "sunken" look.

The type of control and its BackColor property setting determine the border color when the 3D display is in effect:

Control	BackColor Setting	Top/Left Borders	Bottom/Right Borders
Command button	0	7	8
	1 - 7	BackColor + 8	0
	8	7	0
	9 - 15	BackColor - 8	0
All other controls (except timer and scrollbars)	0	8	7
	1-7	0	BackColor + 8
	8	0	7
	9 - 15	0	BackColor -8

Example

This example illustrates the effect of changing the value of the THREE_D constant. To try this example, create a form containing a command button control.

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(THREE_D) = 0
END SUB
```

TIME$ Function, TIME$ Statement

Action

Returns the computer's current system time (TIME$ function — no parameters), or sets the current system time (TIME$ statement).

Syntax

**TIME$**[= *stringexpression$*]

Part	Description
stringexpression$	Time expression in a valid format and range (leading zeros may be omitted):

Format

hh

hh:mm

hh:mm:ss (returned by the TIME$ function)

Inclusive Ranges	Valid for
00 - 23	hh (hour); a 24-hour clock is used, so 8:00 P.M. would be entered as 20:00:00.
00 - 59	mm (minute).
00 - 59	(Default - 0) ss (second).

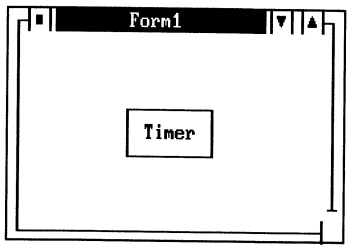
See Also FORMAT$ Function.

Example This example uses the TIME$ function to display the computer's current system time. The TIME$ statement is then used to reset the system's time based on user input.

```
DO UNTIL answer$ = "y"
    CLS                          ' Clear the screen.
    PRINT "The current time is: " + TIME$
    PRINT
    DO
         INPUT "Enter the hour: ", hour$
    LOOP UNTIL (VAL(hour$) < 24 AND VAL(hour$) >= 0)
    DO
         INPUT "Enter minutes: ", minutes$
    LOOP UNTIL (VAL(minutes$) < 60 AND VAL(minutes$) >= 0)
    DO
         INPUT "Enter seconds: ", seconds$
    LOOP UNTIL (VAL(seconds$) < 60 AND VAL(seconds$) >= 0)
    TIME$ = hour$ + ":" + minutes$ + ":" + seconds$
    LOCATE 10, 1
    PRINT "The new time is: " + TIME$
    INPUT "Is this ok (y/n)? ", answer$
LOOP
```

Timer Control

Action Executes event procedures that respond to the passage of a period of time specified by the timer's Interval property; not visible at run time.



Events

Timer

Properties

CtlName	Enabled	Index	Interval
Parent	Tag		

Remarks A timer allows you to run procedure code at a set time interval. The code to be executed is placed in the Timer event.

The time interval at which the Timer event occurs is determined by the Interval property.

To activate a timer, set its Enabled property to True (-1) and set the Interval property to a nonzero value. At run time, the timer event occurs at the specified interval, although the control itself is invisible to the user. To deactivate a timer, set its Enabled property to False (0).

Visual Basic supports up to 16 timers. Attempting to start a timer when none are available causes a run-time error.

See Also TIMER Function, TIMER Statement.

Example This example uses a label to display the current system time on a form and update the time at the interval specified using the Timer event. To try this example, create a form containing a label control.

```
SUB Timer1_Timer ()
   Label1.Caption=TIME$
END SUB
```

Timer Event

Applies To Timer.

Action Occurs when a preset interval for a timer control has elapsed. The interval's frequency is stored in the control's Interval property, which specifies the length of time in milliseconds.

Syntax **SUB** *ctlname*_**Timer ([Index AS INTEGER])**

Part	Description
Index	Uniquely identifies a control in a control array.

Remarks Use the Timer event procedure to tell Visual Basic what to do at every timer interval.

The Interval property specifies the interval between Timer events in milliseconds. For example, an Interval setting of 1 executes the Timer event every one-thousandth of a second.

A timer control's Enabled property must be set to True (-1) for the Timer event to occur.

See Also Enabled Property, Interval Property, ON TIMER Statement, TIMER Function, TIMER Statement.

Example See Timer Control example.

TIMER Function

Action Returns a real number (to 2 decimal places) representing the number of seconds elapsed since midnight; used to time programs or parts of programs, or with the RANDOMIZE statement to seed the random-number generator. The Timer control and the Timer event are easier to use with forms.

Syntax **TIMER**

See Also ON TIMER Statement, RANDOMIZE Statement, TIMER Statement.

Example This example searches for the prime numbers from 3 to 10,000 using a variation of the Sieve of Eratosthenes. The TIMER function is used to time the program.

```
DEFINT A-Z
CONST UNMARK = 0, MARKIT = NOT UNMARK
DIM Mark(10000)
```

```
CLS                                 ' Clear the screen.
Start! = TIMER
Num = 0
FOR N = 3 TO 10000 STEP 2
    IF NOT Mark(N) THEN
         'PRINT N,                  ' To print the primes, remove the
                                    ' remark delimiter in front of the
                                    ' PRINT statement.
         Delta = 2 * N
         FOR I = 3 * N TO 10000 STEP Delta
              Mark(I) = MARKIT
         NEXT
         Num = Num + 1
    END IF
NEXT
Finish! = TIMER
PRINT
PRINT "Program took"; Finish! - Start!;
PRINT "seconds to find the"; Num; "primes"
END
```

TIMER Statement

Action

Enables, disables, or suspends timer event trapping; can be used with object-specific events. The Timer control and the Timer event are easier to use with forms.

Syntax

TIMER {ON | OFF | STOP}

Setting	Description
ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a TIMER ON statement is executed.

Remarks

If your program contains event-handling statements and you are compiling from the Basic Compiler (BC) command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

TIMER ON enables timer-event trapping. A timer event occurs when n& seconds have elapsed (as specified in the ON TIMER statement). If a timer event occurs after a TIMER ON statement, the routine specified in the ON TIMER statement is executed.

TIMER OFF disables timer-event trapping. No timer-event trapping takes place until another TIMER ON statement is executed. Events occurring while trapping is off are ignored.

TIMER STOP suspends timer-event trapping. No trapping takes place until another TIMER ON statement is executed. Any ON TIMER events that occur while trapping is suspended are remembered and processed when the next TIMER ON statement is executed. However, remembered events are lost if TIMER OFF is executed.

When a timer-event trap occurs (that is, the GOSUB is performed), an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically performs a TIMER ON statement unless an explicit TIMER OFF was performed inside the routine.

See Also ON TIMER Statement, TIMER Function.

Example See ON TIMER Statement example.

TIMESERIAL Function

Action Returns a serial number that represents the time of the arguments.

Syntax **TIMESERIAL**(*hour%*, *minute%*, *second%*)

Part	Description
hour%	Hour between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive
minute%	Minute between 0 and 59, inclusive
second%	Second between 0 and 59, inclusive

Remarks You can use negative numbers as arguments as long as the resulting serial number is positive. For example, to find the serial number for the time 50 seconds before 2:00:02, you can use:

```
TIMESERIAL(2, 0, 2-50)
```

Example See HOUR Function example.

TIMEVALUE Function

Action Returns a serial number that represents the time of the argument.

Syntax **TIMEVALUE**(*time$*)

Remarks

Part	Description
time$	A time between 0:00:00 (12:00:00 A.M.) and 23:59:59 (11:59:59 P.M.), inclusive; *time$* value can be entered as "2:24PM" or "14:24"; date information, if included in *time$*, is ignored

Example See HOUR Function example.

TITLEBAR_BACKCOLOR, TITLEBAR_FORECOLOR Constants

Applies To ControlPanel property, SCREEN object.

Action Sets or returns the background and foreground colors for the active title bar in a program.

Syntax

SCREEN.ControlPanel({**TITLEBAR_BACKCOLOR** | *subscript%*})[= *color%*]
SCREEN.ControlPanel({**TITLEBAR_FORECOLOR** | *subscript%*})[= *color%*]

Remarks

Part	Description
subscript%	TITLEBAR_BACKCOLOR has a value of 16; TITLEBAR_FORECOLOR has a value of 17.
color%	Valid color settings are:

Value	Description
0 - 15	Valid color attribute.
0	(Default - TITLEBAR_BACKCOLOR) Black.
15	(Default - TITLEBAR_FORECOLOR) Bright White.

Example This example illustrates the effect of changing the value of the TITLEBAR_BACKCOLOR and TITLEBAR_FORECOLOR constants. To try this example, create a form containing a command button. Set the command button Caption property to "&DoIt."

```
' $INCLUDE: 'CONSTANT.BI'          ' Include ControlPanel definitions.

SUB Command1_Click ()
    SCREEN.ControlPanel(TITLEBAR_BACKCOLOR) = 2
    SCREEN.ControlPanel(TITLEBAR_FORECOLOR) = 4
END SUB
```

Top Property

See Left Property.

TRON, TROFF Statements

Action Traces the execution of program statements (TRON); disables trace execution (TROFF); not available when forms are showing.

Syntax

TRON
TROFF

Usage	Description
TRON	Enables trace.
TROFF	Disables trace.

Remarks To use the TROFF statement with forms, you must first unload or hide any visible forms.

In the Visual Basic programming environment, executing a TRON statement has the same effect as choosing the Trace On command from the Debug menu — each statement is highlighted on the screen as it executes.

To display line numbers when using TRON and TROFF statements:

- Choose Make EXE File from the Run menu and select the Run-Time Error Checking option in the Make EXE File dialog, or
- Specify the /D option on the Basic Compiler (BC.EXE) command line.

TYPE Statement

Action Defines a data type or ISAM table type that contains one or more elements or table columns (ISAM available in Professional Edition only) .

Syntax

TYPE *usertype*
    elementname **AS** *typename*
    [*elementname* **AS** *typename*]
    .
    .
    .
END TYPE

Part	Description
usertype	User-defined type or ISAM table type.
elementname	Element of the user-defined type or table column of the user-defined type.

Part	Description
typename	User-defined type, nested user-defined type, an array, or one of the following data types: INTEGER, LONG, SINGLE (non-ISAM types only), DOUBLE, CURRENCY, or fixed-length string.

Remarks

If the argument *usertype* is a table type, any *elementname* arguments are the names of columns in the table. The names must be exact matches to existing column names and must follow the ISAM naming conventions.

The *typename* can be a user-defined data type, a nested user-defined type (data types only), or an array.

Visual Basic now supports:

- Currency data type for dollars and cents math.
- Static arrays in user-defined types.
- User-defined types for ISAM tables.

Before you can use an ISAM table, you must declare a record type for the records that make up the table. Instances of this type are used to pass records to and from the table.

The following TYPE statement illustrates the use of static arrays. The record StateData includes the CityCode static array, and the record Washington has the same structure as StateData:

```
TYPE StateData
   CityCode (1 TO 100) AS INTEGER     'Declares a static array.
   County AS STRING * 30
END TYPE

DIM Washington(1 to 100) AS StateData
```

When you declare a static array within a user-defined type, its dimensions must be declared with numeric constants rather than variables.

For efficiency, make sure that arrays within a user-defined type start on even offsets.

You can create very large records when you include static arrays within records. Putting one of these records within a SUB procedure can use large amounts of stack space.

Strings in user types must be fixed-length strings. String lengths are indicated by an asterisk and a numeric constant. For example, the following code defines an element named Keyword in a user-defined type as a string with length 40:

```
TYPE
   Keyword AS STRING * 40
END TYPE
```

A user-defined type must be declared in a TYPE declaration before it can be used in the program. Although a user-defined type can be declared in the module-level code, you can

declare a variable to be of a user-defined type anywhere in the module, even in a SUB or FUNCTION.

Use the DIM, REDIM, COMMON, STATIC, or SHARED statements to declare a variable as a user-defined type.

The keyword REM cannot be used as a field name in a TYPE statement. The text that follows is treated as a comment.

If you have defined a table to have columns A, B, C, and D, you can use a user-defined type that has only the columns you need (any subset of A, B, C, and D).

See Also OPEN Statement (File I/O).

Example This example use the PUT statement to store test scores in a random-access file. The TYPE statement is used to create a user-defined variable type.

```
TYPE TestRecord                          ' Define record fields.
   NameField  AS STRING * 20
   ScoreField AS SINGLE
END TYPE

DIM FileBuffer AS TestRecord
DIM I AS LONG                            ' Open the test data file.
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN = LEN(FileBuffer)
CLS                                      ' Clear the screen.
I = 0
DO                                       ' Read pairs of names and
    I = I + 1                            ' scores from the console.
    INPUT "Name ? ", FileBuffer.NameField
    INPUT "Score? ", FileBuffer.ScoreField
    INPUT "-->More (y/n)? ", Resp$
    PUT #1, I, FileBuffer
LOOP UNTIL UCASE$(MID$(Resp$, 1, 1)) = "N"
PRINT I; " records written."
CLOSE #1
KILL "TESTDAT.DAT"                       ' Remove file from disk.
```

UBOUND Function

See LBOUND Function.

UCASE$ Function

See LCASE$ Function.

UEVENT Statement

Action Enables, disables, and suspends trapping of user-defined events; can be used with object-specific events.

Syntax **UEVENT {ON | OFF | STOP}**

Setting	Description
ON	Enables event trapping.
OFF	Disables event trapping; if an event takes place, it is ignored.
STOP	Suspends event trapping; any events that occur are remembered and processed as soon as a UEVENT ON statement is executed.

Remarks The UEVENT statement does not affect form- or control-specific events. Forms and controls use different event handlers. Use object-specific events with forms.

If your program contains event-handling statements and you are compiling from the Basic Compiler (BC.EXE) command line, use the /W or /V option. If you do not use these options and your program contains event traps, Visual Basic generates the error message, "ON event without /V or /W on command line."

UEVENT ON enables trapping of a user-defined event. A user-defined event is usually a hardware interrupt (although it can be a software interrupt). If a user-defined event occurs after a UEVENT ON statement, the routine specified in the ON UEVENT statement is executed.

UEVENT OFF disables trapping of a user-defined event. No trapping takes place until another UEVENT ON statement is executed. Events occurring while trapping is off are ignored.

UEVENT STOP suspends trapping of a user-defined event. No trapping takes place until a UEVENT ON statement is executed. Any defined user events that occur while trapping is suspended are remembered and processed when the next UEVENT ON statement is executed. However, remembered events are lost if a UEVENT OFF statement is executed before a UEVENT ON statement is encountered.

When a user-defined trap occurs (that is, the GOSUB is performed), an automatic UEVENT STOP is executed so that recursive traps cannot take place. The RETURN from a trapping routine automatically executes a UEVENT ON statement unless an explicit UEVENT OFF was performed inside the routine.

See Also EVENT Statement, ON UEVENT Statement, SetUEvent Routine.

Example See ON UEVENT Statement example.

Unload Event

Applies To Form.

Action Occurs when a form is about to be unloaded from memory. The Unload event is triggered either by user action — for example, closing the form using the Control-menu box — or by the UNLOAD Statement.

Syntax **SUB Form_Unload (Cancel AS INTEGER)**

Part	Description
Cancel	Any nonzero value; setting Cancel to any nonzero value prevents the form from being unloaded.

Remarks An Unload event procedure is a good place to include form-level validation code for closing the form or saving data contained in it.

When a form is reloaded, the contents of all its controls are reinitialized.

See Also Load Event, LOAD Statement, UNLOAD Statement.

Example This example demonstrates a simple procedure to close a form while querying the user with various message boxes. In an actual application, you could add calls to general-purpose SUB procedures that emulate the processing of the Exit, Save, and Save As menu items on Visual Basic's File menu.

```
SUB Form_Unload (Cancel AS INTEGER)
    Msg$ = "Save Data before closing?"
    Response% = MSGBOX(Msg$, 3 + 32, "Save Dialog")
    SELECT CASE Response%
         CASE 2
              ' Do not allow close.
              Cancel = -1
              Msg$ = "Command has been canceled."
         CASE 6
              ' Enter code to save data here.
              Msg$ = "Data saved."
         CASE 7
              Msg$ = "Data not saved."
    END SELECT
    MSGBOX Msg$, 0, "Confirm"
END SUB
```

UNLOAD Statement

Action Removes a form or control array element from memory.

Syntax **UNLOAD** {*form* | *ctlname*[*index%*]}

Part	Description
form	Name of form to unload.
ctlname	Control array from which an element will be removed.
index%	Indicates the position in the control array.

Remarks Unloading a form or control frees memory and resets properties to their original value.

When a form is unloaded, all controls placed on the form at run time are no longer accessible. Controls placed on the form at design time remain intact, although any run-time changes to those controls and their properties are lost when the form is reloaded. All changes to form properties are also lost.

Control array elements added to a form at run time can only be removed with the UNLOAD statement. The properties of unloaded controls are reinitialized when the controls are reloaded.

When a modal form is unloaded, program execution returns to the point in code where the SHOW method was used to display the modal form.

Referencing a control on a form that has been unloaded causes the form to be reloaded.

See Also LOAD Statement, Unload Event.

Example See LOAD Statement example.

Value Property

Applies To Check box, command button, horizontal scroll bar, option button, vertical scroll bar.

Action Sets or returns the current state of a control (check box, option button), whether the control is pressed (command button), or the current position of a control (scroll bar); not available at design time with command buttons.

Syntax [*form.*][*ctlname.*]**Value**[= *setting%*]

Part	Description
setting%	Valid settings are:

Control	Setting	Description
Check box	0	(Default) Unchecked (off or False).
	1	Checked (on or True).
	2	Grayed (not available).
Command button	False (0)	(Default) Button is not pressed.
	True (-1)	Button is pressed; setting from code invokes the button's Click event; not available at design time.
Option button	False (0)	(Default) Button is not selected.
	True (-1)	Button is selected.
Scroll bar	Min to Max (-32,768 to 32,767 inclusive)	(Default) Min.

See Also Max Property, Min Property.

Example See Max, Min Properties example.

VARPTR$ Function

Action Returns a string representation of a variable's address for use in DRAW and PLAY statements.

Syntax **VARPTR$(***variablename***)**

Part	Description
variablename	String variable containing DRAW or PLAY commands; if an array element, the dimensions for the array must be specified before VARPTR$ is used and the array must consist of variable-length strings.

Remarks To guarantee correct results, use the value returned by VARPTR$ immediately after invoking the function.

See Also DRAW Statement, PLAY Statement (Music), VARPTR Function.

Example See PLAY Statement (Music) example.

VARPTR, VARSEG Functions

Action Returns the address of a variable or string descriptor.

Syntax **VARPTR**(*variablename*)
VARSEG(*variablename*)

Part	Description
variablename	Any valid variable, including a record variable or record element.

Remarks VARPTR and VARSEG return different values depending on the type of variable specified in *variablename*:

Variable Type	Return Value (VARPTR)
Numeric	Offset of the variable within its segment as an unsigned integer.
String	Offset of string descriptor.
Not defined	Creates the variable and returns its address.

Variable Type	Return Value (VARSEG)
Numeric	Segment address as an unsigned integer.
String	Segment address.
Not defined	Creates the variable and returns its address.

Because many Visual Basic statements move variables in memory, use the values returned by VARPTR and VARSEG immediately after the functions are used.

VARPTR and VARSEG are often used with Absolute, BLOAD, BSAVE, Interrupt, PEEK, POKE, or when passing arrays to procedures written in other languages.

When using VARPTR or VARSEG to get the address of an array, use the first element of the array as the argument:

```
DIM A(150)
.
.
.
ArrAddress=VARPTR(A(1))
```

You can use VARPTR alone to get the address of a numeric variable stored in DGROUP. You must use both VARPTR and VARSEG to get the complete address of a numeric variable stored in far memory.

It is not meaningful to use VARSEG and VARPTR for strings. Since Visual Basic stores strings far, the format of the descriptor is different from previous versions of Basic. Use SSEG, SADD, and SSEGADD to return the address of string variables in Basic code. Use StringAddress to return the address of string variables from procedures written in other languages.

You cannot use VARPTR to get the address of a file's buffer. Instead, use the FILEATTR function to get information about a file.

Programs written in Microsoft Basic version 3.0 or earlier that used VARPTR to access numeric arrays may no longer work. You must now use a combination of VARPTR and VARSEG. For example, the following Microsoft QuickBasic fragment no longer works correctly:

```
DIM Cube(675)
.
.
.
BSAVE "graph.dat",VARPTR(Cube(1)),2700
```

The fragment can be rewritten as follows to work in the current version of Visual Basic:

```
DIM Cube(675)
.
.
.                                       ' Change segment to segment
DEF SEG=VARSEG(Cube(1))                 ' containing Cube.
BSAVE "graph.dat",VARPTR(Cube(1)),2700
DEF SEG                                 ' Restore Basic segment.
```

The VARSEG function, combined with VARPTR, replaces the PTR86 subprogram used in earlier versions of Basic.

Do not pass expanded memory arrays to non-Basic procedures. If you start Visual Basic (VBDOS.EXE) with the /Ea switch, any of these arrays may be stored in expanded memory:

- Numeric arrays < 16K in size
- Fixed-length string arrays < 16K in size
- User-defined-type arrays < 16K in size

If you want to pass arrays to non-Basic procedures, first start Visual Basic (VBDOS.EXE) without the /Ea switch. Note: Without the /Ea switch, no arrays are stored in expanded memory.

See Also SADD Function, SSEG Function, VARPTR$ Function.

Example This example uses the VARPTR and VARSEG functions in a CALL statement to pass a Basic array to a C function.

Note: To run this example, you must separately compile the C function with the large-model (/AL) switch and either place the object module in a Quick library or link it to the Basic program.

```
DEFINT A-Z
DECLARE SUB AddArr CDECL (BYVAL Offs, BYVAL Segm, BYVAL Num)
DIM A(1 TO 100) AS INTEGER

CLS                          ' Clear the screen.
FOR I = 1 TO 15              ' Fill the array with the numbers 1 to 15.
    A(I) = I
NEXT I

' AddArr expects a far address (segment and offset). Because CDECL puts
' things on the stack from right to left, place the offset VARPTR(A(1)) first
' in the list, followed by the segment VARSEG(A(1)).

CALL AddArr(VARPTR(A(1)), VARSEG(A(1)), 15)
FOR I = 1 TO 15              ' Print the modified array
    PRINT A(I)
NEXT I
END

  /* C Function AddArr */
  /* Add one to the first num elements of array arr.*/
  void far addarr(arr,num)
  int far *arr;
  int num;
  {
      int i;
      for(i=0;i<num;i++) arr[i]++;
  }
```

Vertical Scroll Bar Control

See Horizontal Scroll bar Control.

VIEW PRINT Statement

Action Sets the boundaries of the screen text viewport; not available when forms are showing.

Syntax **VIEW PRINT** [*toprow%* **TO** *bottomrow%*]

Part	Description
toprow%	Number specifying the top row of the text viewport.
bottomrow%	Number specifying the bottom row of the text viewport.

Remarks References to VIEW PRINT are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the HIDE method to hide any visible forms or controls before executing this statement.

Without the arguments *toprow%* and *bottomrow%*, the VIEW PRINT statement initializes the whole screen area as the text viewport.

The number of text lines on the screen depends on the screen mode and whether or not Visual Basic (VBDOS.EXE) was started with the /H option.

See Also VIEW Statement.

Example See CLS Statement example.

VIEW Statement

Action Defines the size and location of a viewport where graphics are displayed on the screen; not available when forms are showing.

Syntax **VIEW** [[**SCREEN**] (*x1!*,*y1!*)-(*x2!*,*y2!*) [,[*color%*] [, *border&*]]]

Part	Description
SCREEN	Specifies that coordinates are relative to the screen rather than to the viewport; only graphics within the viewport are plotted.
(*x1!*,*y1!*)-(*x2!*,*y2!*)	Coordinates of diagonally opposite corners of the viewport.

Part	Description
color%	Color attribute that sets the viewport fill color.
border&	Color attribute that sets the viewport border color.

Remarks

You must be in a graphics screen mode (for example, screen mode 1) to use the VIEW statement; otherwise, Visual Basic may generate the error message, "Illegal function call."

The available color attributes for VIEW depend on your graphics adapter and the screen mode set by the most recent SCREEN statement.

If SCREEN is omitted, all points are plotted relative to the viewport.

The argument (*x1!*,*y1!*)-(*x2!*,*y2!*) must be within the physical bounds of the screen in the current screen mode.

If *color%* is omitted, the viewport area is not filled — that is, it has the same color as the screen background color.

If *border&* is omitted, no line is drawn around the viewport.

If VIEW is used with no arguments, the entire screen is defined as the viewport. RUN and SCREEN also define the entire screen as the viewport.

See Also VIEW PRINT Statement.

Example See PMAP Function example.

Visible Property

Applies To Check box, combo box, command button, directory list box, drive list box, file list box, form, frame, horizontal scroll bar, label, list box, menu, option button, picture box, text box, vertical scroll bar.

Action Determines whether an object is visible on the screen.

Syntax [*form.*][*ctlname.*]**Visible**[= *boolean%*]

Part	Description
boolean%	Valid settings are:

Value	Description
True (-1)	(Default) Object is visible.
False (0)	Object is invisible.

Remarks To hide a control at startup, set the Visible property to False at design time.

Setting the Visible property in code allows you to hide and later redisplay a control at run time in response to a particular event. For example, in response to a Click or KeyPress event in one control, you might redisplay another control.

Generally, the Visible property is used for control display, while the SHOW and HIDE methods are used to manipulate form display. However, using the SHOW and HIDE methods on a form is the same as setting the form's Visible property in code to True (-1) and False (0), respectively.

No user events are processed while controls are invisible except Timer events.

See Also HIDE Method, SHOW Method.

Example See Command Button Control example.

WAIT Statement

Action Suspends program execution until a specified bit pattern is input from an input port.

Syntax **WAIT** *portnumber%*, *and-expression%* [, *xor-expression%*]

Part	Description
portnumber%	Number specifying the input port.
and-expression%	Integer expression that WAIT combines with the bit pattern value using an AND operator; when the result is nonzero, WAIT stops monitoring the port.
xor-expression%	Turns line bits on and off in the bit pattern before the AND operation is applied (optional).

Remarks The WAIT statement is an enhanced version of the INP function; it suspends execution until a specified bit pattern is input from an input port:

- The data byte read from the port is combined, using an XOR operation, with *xor-expression%*. If *xor-expression%* is omitted, it is assumed to be 0.
- The result is combined with *and-expression%* using an AND operation.
- If the result is zero, the first two steps are repeated.
- If the result is nonzero, execution continues with the next program statement.

It is possible to enter an infinite loop with the WAIT statement if the input port fails to develop a nonzero bit pattern. In this case, you must manually restart or "cold boot" the machine.

The following program line illustrates the syntax of the WAIT statement:

```
WAIT  HandShakePort, 2
```

WAIT causes Visual Basic to do an AND operation on the bit pattern received at MS-DOS I/O port HandShakePort with the bit pattern represented by 2 (00000010).

See Also INP Function, ON COM Statement, OUT Statement, SLEEP Statement.

Example This example uses the WAIT statement to suspend program execution while monitoring the status of COM1.

Note: To run this example, you must have an ASCII terminal connected to COM1 at 9600 baud, 8 data bits, 1 stop bit, and no parity.

```
CLS                                                    ' Clear the screen.

OPEN "COM1:9600,N,8,1,BIN" FOR RANDOM AS #1            ' Open and close the port
CLOSE #1                                               ' at the proper baud rate.
PortNum% = &H3F8 'COM1
NonPrintCharMask% = 96

' Wait until there's some activity on COM1.
' Mask out characters less than 32 for the first input character.
LOCATE 12, 15: PRINT "Waiting for a printable input character. "
WAIT PortNum%, NonPrintCharMask%

' Once a printable character comes in, accept all characters.
DO
    A% = INP(PortNum%)                 ' Get a character from the port.
    SELECT CASE A%                     ' Evaluate the character.
        CASE 7
            Character$ = "   BELL    "
        CASE 8
            Character$ = "BACK SPACE "
        CASE 9
            Character$ = "    TAB    "
        CASE 13
            Character$ = "    CR     "
        CASE 32
            Character$ = "   SPACE   "
        CASE 127
            Character$ = "    DEL    "
        CASE IS < 31, IS > 127
            Character$ = "unprintable"
        CASE ELSE
            Character$ = SPACE$(5) + CHR$(A%) + SPACE$(5)
    END SELECT
    LOCATE 12, 15
```

```
    PRINT "Numeric Value                    ASCII Character"
    LOCATE 14, 20
    PRINT A%; TAB(50); Character$       ' Report the input character.
    IF A% > 127 THEN STOP
    LOCATE 23, 25
    PRINT "Press ANY key to quit."
LOOP WHILE INKEY$ = ""
END
```

WEEKDAY Function

Action Evaluates a date/time serial number and returns the day of the week.

Syntax **WEEKDAY(***serial***)**

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See DATEVALUE Function example.

WHILE...WEND Statement

Action Executes a series of statements in a loop as long as a specified *condition* is true. The DO...LOOP statement provides a more powerful and flexible loop control structure.

Syntax

WHILE *condition*
statementblock
WEND

Part	Description
condition	Numeric expression that evaluates as True (nonzero) or False (zero).
statementblock	One or more valid statements on one or more lines.

Remarks If *condition* is true — that is, if it does not equal zero — then any intervening statements are executed until the WEND statement is encountered. Visual Basic then returns to the WHILE statement and checks the *condition*. If it is still true, the process is repeated. If *condition* is not true or equals zero, execution resumes with the statement following the WEND statement.

WHILE...WEND loops can be nested to any level. Each WEND matches the most recent WHILE. When Visual Basic encounters an unmatched WHILE statement, it generates the error message, "WHILE without WEND." If Visual Basic encounters an unmatched WEND statement, it generates the error message, "WEND without WHILE."

Do not branch into the body of a WHILE...WEND loop without executing a WHILE statement. Doing so may cause run-time errors or program logic errors that are difficult to locate.

See Also DO...LOOP Statement, FOR...NEXT Statement.

Example This example uses the WHILE...WEND statement to perform a bubble sort of an array A$.

```
CLS                     ' Clear the screen.
CONST FALSE = 0, TRUE = -1
DIM I AS INTEGER
DIM A$(4)
A$(1) = "New York"
A$(2) = "Boston"
A$(3) = "Chicago"
A$(4) = "Seattle"
Max = UBOUND(A$)
Exchange = TRUE         ' Force first pass through the array.
WHILE Exchange          ' Sort until no elements are exchanged.
    Exchange = FALSE    ' Compare array elements by pairs; when two are
                        ' exchanged, force another pass by setting Exchange
                        ' to TRUE.
    FOR I = 2 TO Max
        IF A$(I - 1) > A$(I) THEN
            Exchange = TRUE
            SWAP A$(I - 1), A$(I)
        END IF
    NEXT
WEND
CLS
FOR I = 1 TO 4
    PRINT A$(I)
NEXT I
END
```

Width Property

See Height Property.

WIDTH Statement

Action Assigns an output-line width to a device (such as a printer) or file, or changes the number of text lines and columns displayed on the screen. Use the Width property with objects.

Syntax

WIDTH {#*filenumber%* | *device$*}, *width%*
WIDTH LPRINT *width%*
WIDTH {*screenwidth%* | , *screenheight%* | *screenwidth%*, *screenheight%*}

Part	Description
filenumber%	Number specifying an open file or device.
device$	One of the following device names: SCRN:, COM1:, COM2:, LPT1:, LPT2:, or LPT3:.
width%	For WIDTH, the width of the output line for the file or device; for WIDTH LPRINT, an integer value between 1 and 255, inclusive; default value is 80.
screenwidth%	Desired width in columns; screen display width must be 40 or 80 columns; not available when forms are showing.
screenheight%	Desired screen-display height; value can be 25, 30, 43, 50, or 60, depending on the available display adapter and screen mode; not available when forms are showing.

Remarks Any reference to WIDTH *device$* where *device$* represents SCRN: or CONS: is not allowed with forms and causes Visual Basic to generate the error message, "Invalid when forms are showing."

To use the WIDTH statement with forms, you must first unload or hide any visible forms.

The following are examples of WIDTH statement usage:

WIDTH #*filenumber%*, *width%*

- Sets the line width of an output device already opened as a file — for example, LPT1: or CONS:.
- The argument *filenumber%* is the file number associated with the device through the OPEN statement.
- The "#" character in front of *filenumber%* is not optional.
- The width assignment takes place immediately.

WIDTH *device$, width%*

- Sets to *width%* the line width of *device$* (a device file name). The device should be a string expression — for example, "LPT1:".
- The width assignment is deferred until the next OPEN statement affecting the device.
- The assignment does not affect output for an already open device.

WIDTH LPRINT *width%*

- Sets the line width of the line printer for use by subsequent LPRINT statements.
- Equivalent to the statement:

  WIDTH "LPT1:", *width%*

WIDTH {*screenwidth%* | ,*screenheight%* | *screenwidth%*, *screenheight%*}

- Sets the number of text character columns and lines to display on the screen.
- The value of *screenwidth%* must be either 40 or 80; default is 80.
- The value of *screenheight%* may be 25, 30, 43, 50, or 60, depending on the display adapter used and the screen mode.

The number of lines that can be displayed when a program is started is listed below:

Adapter(1)	Screen Mode(2)	Screenwidth x Screenheight	Color Display
MDPA	0	80x25	Monochrome
Hercules	0	80x25	Monochrome
	3	80x25	Monochrome
CGA	0	40x25, 80x25	Color
	1	40x25	Color
	2	80x25	Color
EGA	0	40x25, 40x43, 80x25, 80x43	Color
	0	40x25, 40x43, 80x25, 80x43	Enhanced color
	0	40x25, 40x43, 80x25, 80x43	Monochrome
	1	40x25	Color or Enhanced
	2	80x25	Color or Enhanced

Adapter(1)	Screen Mode(2)	Screenwidth x Screenheight	Color Display
	7	40x25	Color or Enhanced
	8	80x25	Color or Enhanced
EGA	9	80x25, 80x43	Enhanced color
EGA or VGA	10	80x25, 80x43	Monochrome
VGA	0	40x25, 40x43, 40x50	Analog
	0	80x25, 80x43, 80x50	Analog
	1	40x25	Analog
	2	80x25	Analog
	7	40x25	Analog
	8	80x25	Analog
	9	80x25, 80x43	Analog
	10	80x25, 80x43	Analog
	11	80x30, 80x60	Analog
	12	80x30, 80x60	Analog
	13	40x25	Analog
MCGA	0	40x25, 80x25	Color or Analog
	1	40x25	Color or Analog
	2	80x25	Color or Analog
	11	80x30, 80x60	Analog
	13	40x25	Analog

See Also LPRINT Statement, PRINT Statement, SCREEN Statement, VIEW PRINT Statement, Width Property.

Example This example uses the WIDTH statement to set different widths for File #1 (a terminal on COM1).

```
CLS                                 ' Clear the screen.

' Open the port at the proper baud rate.
OPEN "COM1:9600,N,8,1,ASC,LF" FOR OUTPUT AS #1
Test$ = "1234567890"                ' Set up a test string.
WIDTH #1, 3                         ' Change width to 3.
PRINT #1, Test$
WIDTH #1, 4                         ' Change width to 4.
```

```
PRINT #1, Test$
SLEEP 3                          ' Delay three seconds so you can observe
                                 ' the results on the terminal.
CLOSE #1                         ' Close the file.
```

WINDOW Statement

Action Defines logical dimensions for the current graphics viewport — for example, your own viewport coordinate system. Statement not available when forms are showing.

Syntax **WINDOW** [[**SCREEN**] (*x1!*,*y1!*)-(*x2!*,*y2!*)]

Part	Description
SCREEN	Inverts the normal Cartesian direction of the y screen coordinates so that y values increase from the top of the screen to the bottom.
(*x1!*,*y1!*)	Logical coordinates that map to the upper-left screen coordinates of the viewport.
(*x2!*,*y2!*)	Logical coordinates that map to the lower-right screen coordinates of the viewport.

Remarks References to the WINDOW statement are not allowed with forms and cause Visual Basic to generate the error message, "Invalid when forms are showing." Use the HIDE method to hide any visible forms or controls before executing this statement.

Using the WINDOW statement with no arguments disables the logical coordinate system.

Use the VIEW statement to change the size of the viewport.

If a WINDOW statement without a SCREEN argument appears in a program before the PUT statement, the coordinates refer to the lower-left corner of the rectangle.

The WINDOW statement allows the user to create a customized coordinate system to draw lines, graphs, or objects without being constrained by the graphics viewport physical coordinate values and orientation.

WINDOW defines a coordinate system that is mapped to the physical coordinates of the viewport. All subsequent graphics statements use the window coordinates and are displayed within the current viewport. The viewport may be the entire screen or a clipping area defined in physical coordinates by the VIEW statement.

The window coordinate system is disabled by:

- A WINDOW statement with no arguments.
- A RUN statement.

See Also CLS Statement, PMAP Function, POINT Function, SCREEN Statement, VIEW Statement, WIDTH Statement.

Example See PMAP Function example.

WindowState Property

Applies To Form.

Action Sets or returns the visual state of a form at run time.

Syntax [*form.*]**WindowState**[= *state%*]

Part	Description
state%	Valid settings are:

Value	Description
0	(Default) Normal.
1	Minimized (minimized to an icon).
2	Maximized (enlarged to maximum size).

See Also Load Event, MaxButton Property, MinButton Property, Paint Event, Resize Event.

Example See $FORM Metacommand example.

WRITE # Statement

Action Writes data to the screen or a sequential file.

Syntax **WRITE** #*filenumber%* [, *expressionlist*]

Part	Description
filenumber%	Number of an open sequential file; if omitted, WRITE writes to the screen.
expressionlist	One or more variables or expressions, separated by commas, whose values are written to the screen or file.

Remarks Any reference to WRITE #*filenumber%* where *filenumber%* represents SCRN: or CONS: is not allowed with forms and causes Visual Basic to generate the error message, "Invalid when forms are showing."

To use the WRITE # statement with forms when printing to the screen, you must first unload or hide any visible forms.

The WRITE # statement, unlike the PRINT # statement, inserts commas between items and quotation marks around strings as they are written to the file. You do not have to put explicit delimiters in the list. A new line is inserted once the last item in the list has been written to the file.

If you attempt to write data to a sequential file restricted by a LOCK statement, Visual Basic generates the error message, "Permission denied" unless it is trapped by the program. All of Visual Basic's usual error-handling routines can trap and examine this error.

See Also PRINT # Statement, WRITE Statement.

Example This example uses the WRITE # statement to write customer information to a data file. The LINE INPUT # statement is then used to read the records from the file.

```
CLS                                          ' Clear the screen.
OPEN "LIST" FOR OUTPUT AS #1
PRINT "CUSTOMER INFORMATION:"
DO                                           ' Get customer information.
    PRINT
    INPUT "   LAST NAME:  ", LName$
    INPUT "   FIRST NAME: ", FrName$
    INPUT "   AGE:        ", Age$
    INPUT "   SEX:        ", Sex$
    Sex$ = UCASE$(Sex$)
    WRITE #1, LName$, FrName$, Age$, Sex$
    INPUT "Add another (Y/N)"; R$
LOOP WHILE UCASE$(R$) = "Y"
CLOSE #1
OPEN "LIST" FOR INPUT AS #1                  ' Echo the file back.
CLS
PRINT "Records in file:": PRINT
DO WHILE NOT EOF(1)
    LINE INPUT #1, REC$                      ' Read records from file.
    PRINT REC$                               ' Print the records on the screen.
LOOP

'Remove file from disk
CLOSE #1
KILL "LIST"
```

WRITE Statement

Action Sends data to the screen; not available when forms are showing. The values written on the screen are separated by commas, with double quotation marks around strings and no spaces around numbers.

Syntax **WRITE** [*expressionlist*]

Part	Description
expressionlist	Specifies one or more values to write.

Remarks Output from a WRITE statement cannot be directed to a form. Any reference to WRITE when forms are showing causes Visual Basic to generate an error. Use the HIDE method to hide any visible forms or controls before executing this statement.

If *expressionlist* is omitted, a blank line is written; if included, the values of the expressions are written to the screen.

The expressions in the list can be numeric and/or string expressions. They must be separated by commas.

When the printed items are written, each item is separated from the last by a comma.

Printed strings are delimited by double quotation marks (""). After the last item in the list is printed, Visual Basic inserts a carriage return-linefeed sequence.

The WRITE statement writes numeric values without leading or trailing spaces.

See Also PRINT Statement, WRITE # Statement.

Example This example use the WRITE statement to output delimiters between data items.

```
CLS                                    ' Clear the screen.
A = 80: B = 90: C$ = "That's all.": D = -1E-13
WRITE A, B, C$, D
PRINT A, B, C$, D
```

YEAR Function

Action Evaluates a date/time serial number and returns the year, a value in range 1753 to 2078, inclusive.

Syntax **YEAR**(*serial*)

Part	Description
serial	Serial number in the range -53,688 to 65,380 that represents a date and/or time.

Example See DATEVALUE Function example.

Appendixes

ASCII and Scan Code Tables

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	SLE
^Q	17	11	◄	CS1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SIB
^]	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

- ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL-BKSP key.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▒
178	B2	▒
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	ø
238	EE	∈
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	n
253	FD	2
254	FE	■
255	FF	

Key	Scan Code		ASCII or Extended•			ASCII or Extended• with SHIFT			ASCII or Extended• with CTRL			ASCII or Extended• with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
ESC	1	01	27	1B		27	1B		27	1B				
1!	2	02	49	31	1	33	21	!				120	78	NUL
2@	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3#	4	04	51	33	3	35	23	#				122	7A	NUL
4$	5	05	52	34	4	36	24	$				123	7B	NUL
5%	6	06	53	35	5	37	25	%				124	7C	NUL
6^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7&	8	08	55	37	7	38	26	&				126	7E	NUL
8*	9	09	56	38	8	42	2A	*				127	7F	NUL
9(	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
-_	12	0C	45	2D	-	95	5F	_	31	1F		130	82	NUL
=+	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
]}	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
;:	39	27	59	3B	;	58	3A	:						
'"	40	28	39	27	'	34	22	"						
`~	41	29	96	60	`	126	7E	~						

• Extended codes return NUL (ASCII 0) as the intial character. This is a signal that a second (extended) code isavailable in the keystroke buffer.

Key	Scan Code		ASCII or Extended•			ASCII or Extended• with SHIFT			ASCII or Extended• with CTRL			ASCII or Extended• with ALT		
	Dec	Hex	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
L SHIFT	42	2A												
|	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
,<	51	33	44	2C	,	60	3C	,						
.>	52	34	46	2E	.	62	3E	.						
/?	53	35	47	2F	/	63	3F	/						
R SHIFT	54	36												
*PRTSC	55	37	42	2A	*		INT	5§	16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY-	74	4A	45	2D		45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY-	78	4E	43	2B		43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

• Extended codes return NUL (ASCII 0) as the initial character. This is a signal that a second (extended) code is available in the keystroke buffer.

§ Under MS-DOS, SHIFT + PRTSC causes interrupt 5, which prints the screen unless an interrupt handler has been defined to replace the default interrupt 5 handler.

Reserved Words

The words in the following list are reserved in Escher. They may not be used as labels, or as names of variables or procedures. Reserved words include keywords (functions, statements, methods, and operators) and certain other words that are recognized by the Programming Environment. Words with asterisks (*) are new to the Basic language since QuickBasic version 4.5.

ABS
ACCESS
ADDITEM*
ALIAS
ALL*
AND
ANY
APPEND
AS
ASC
ATN
BASE
BEEP
BEGINTRANS*
BINARY
BLOAD
BOF*
BSAVE
BYVAL
CALL
CALLS
CASE
CCUR*
CDBL
CDECL
CHAIN
CHDIR
CHDRIVE*
CHECKPOINT*
CHR$
CINT
CIRCLE
CLEAR
CLIPBOARD*
CLNG
CLOSE
CLS
COLOR
COM
COMMAND$
COMMITTRANS*
COMMON
CONST
CONTROL*
COS
CREATEINDEX*
CSNG
CSRLIN
CURDIR*
CURRENCY*
CVC*
CVD
CVDMBF
CVI
CVL
CVS
CVSMBF
DATA
DATE$
DATESERIAL*
DATEVALUE*
DAY*
DECLARE
DEF
DEFCUR*
DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR
DELETE*
DELETEINDEX*
DELETETABLE*
DIM
DIR$*

DO
DOEVENTS*
DOUBLE
DRAG*
DRAW
ELSE
ELSEIF
END
ENDDOC*
ENDIF
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$
ERL
ERR
ERROR
ERROR$
EVENT *
EXIT
EXP
EXPLICIT*
FIELD
FILEATTR
FILES
FIX
FN
FOR
FORM*
FORMAT$*
FRE
FREEFILE
FUNCTION
GET
GETINDEX$*
GETTEXT*
GO
GOSUB
GOTO
HEX$
HIDE*
HOUR*
IF
IMP
INKEY$
INP
INPUT
INPUT$
INPUTBOX$*
INSERT*
INSTR
INT
INTEGER
IOCTL
IOCTL$
IS
ISAM*
KEY
KILL
LBOUND
LCASE$
LEFT$
LEN
LET
LINE
LIST
LOAD*
LOC
LOCAL
LOCATE
LOCK
LOF
LOG
LONG
LOOP
LPOS
LPRINT
LSET
LTRIM$
MID$
MINUTE*
MKC$*
MKD$
MKDIR
MKDMBF$
MKI$
MKL$
MKS$
MKSMBF$
MOD
MONTH*
MOVE*
MOVEFIRST*
MOVELAST*
MOVENEXT*
MOVEPREVIOUS*

MSGBOX*
NAME
NEWPAGE*
NEXT
NOT
NOW*
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
OUTPUT
PAINT
PALETTE
PCOPY
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESERVE*
PRESET
PRINT
PRINTER*
PRINTFORM*
PSET
PUT
QBCOLOR*
RANDOM
RANDOMIZE
READ
REDIM
REFRESH*
REM
REMOVEITEM*
RESET
RESTORE
RESUME
RETRIEVE*
RETURN
RGB*
RIGHT$
RMDIR
RND
ROLLBACK*
RSET
RTRIM$
RUN
SADD
SAVEPOINT*
SCREEN*
SECOND*
SEEK
SEEKEQ*
SEEKGE*
SEEKGT*
SEG
SELECT
SETFOCUS*
SETINDEX*
SETMEM
SETTEXT*
SGN
SHARED
SHELL
SHOW*
SIGNAL
SIN
SINGLE
SLEEP
SOUND
SPACE$
SPC
SQR
SSEG*
SSEGADD*
STACK*
STATIC
STEP
STICK
STOP
STR$
STRIG
STRING
STRING$
SUB
SWAP
SYSTEM
TAB
TAN
TEXTCOMP*
TEXTHEIGHT*
TEXTWIDTH*
THEN
TIME$

TIMER
TIMESERIAL*
TIMEVALUE*
TO
TROFF
TRON
TYPE
TYPEOF*
UBOUND
UCASE$
UEVENT
UNLOAD*
UNLOCK
UNTIL
UPDATE*
USING
VAL
VARPTR
VARPTR$
VARSEG
VIEW
WAIT
WEEKDAY*
WEND
WHILE
WIDTH
WINDOW
WRITE
XOR
YEAR*

Data Types

The following elementary data types are supported by Visual Basic:

Suffix	Data type	Size of data	Range
%	INTEGER	2 bytes	–32,768 to 32,767
&	LONG integer	4 bytes	–2,147,483,648 to 2,147,483,647
!	SINGLE-precision floating-point	4 bytes	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
#	DOUBLE-precision floating-point	8 bytes	–1.797693134862315D308 to –4.94066D-324 for negative values; 4.94066D-324 to 1.797693134862315D308 for positive values
@	CURRENCY (Scaled integer)	8 bytes	–922337203685477.5808 to 922337203685477.5807
$	STRING	1 byte per character	0K to approximately 32K (Some storage overhead is required.)
None	User-defined (using TYPE..END TYPE)	Number of bytes required by elements	
None	FORM	"	
None	CONTROL	"	

INTEGER Data Type

Integers are stored as 16-bit binary numbers (2 bytes) ranging in value from –32,768 to 32, 767.

In addition to representing numbers in this range, integers are used to represent Boolean values, that is, those values that can have only two states. For Boolean values, –1 represents True and 0 represents False. Integers are also used to represent enumerated values. Enumerated values can contain a finite set of unique integers, each of which has special meaning to the property with which it is used. Enumerated values provide a convenient way to select between a known number of choices.

Important At design time, the Property bar lists Boolean settings as True and False. When referring to a Boolean property setting in code, you should use –1 for True and 0 for False, unless you have specified the CONSTANT.BI file (an include file provided with Visual Basic in a $INCLUDE metacommand.). Loading the CONSTANT.BI file defines TRUE and FALSE as global constants.

LONG Integer Data Type

Long integers are stored as signed 32-bit binary numbers (4 bytes) ranging in value from –2,147,483,648 to 2,147,483,647.

SINGLE Data Type

Single-precision floating-point numbers are stored 32-bits (4 bytes) ranging in value from –3402823E38 to –1.401298E45 for negative values and 1.401298E-45 to 3.402823E38 for positive values.

Each floating-point value consists of three parts: the sign, the exponent, and the mantissa. In a single-precision number, the sign takes 1 bit, the exponent takes 8 bits, and the mantissa uses the remaining 23 bits and an additional implied bit.

DOUBLE Data Type

Double-precision floating-point numbers are stored 64-bits (8 bytes) ranging in value from –1.797693134862315D308 to –4.94066D-324 for negative values and 4.94066D-324 to 1.797693134862315D308 for positive values.

Each floating-point value consists of three parts: the sign, the exponent, and the mantissa. In a double-precision number, the sign takes 1 bit, the exponent takes 11 bits, and the mantissa uses the remaining 52 bits and an additional implied bit.

CURRENCY Data Type

A currency number is stored as an 8-byte, two's complement integer, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range from –922337203685477.5808 to 922337203685477.5807.

The CURRENCY data type is extremely useful for calculations involving money, or for any fixed-point calculation where accuracy is particularly important.

STRING Data Type

There are two kinds of strings:

- Variable-length strings. A variable-length string can contain up to approximately 32,768 characters. Some storage overhead is required so a string can never actually be 32K in length.
- Fixed-length strings. A fixed-length string contains a declared number of characters, no more than approximately .

The codes for these characters range from 0-255. String characters used by Visual Basic consist of American Standard Code for I nformation Interchange (ASCII) characters. The first 128 characters (0-127) correspond to the letters and symbols on a standard U.S. keyboard.

CONTROL Data Type

The CONTROL data type can only be declared as a formal parameter of Visual Basic procedure. Within a procedure declaration, CONTROL is used as follows:

```
SUB ChangeText (AnyControl AS CONTROL, NewValue$)
  IF TYPE OF AnyControl IS TextBox THEN
    AnyControl.Text = NewValue$
  END IF
END SUB
```

FORM Data Type

The FORM data type can only be declared as a formal parameter of a Visual Basic procedure. Within a procedure declaration, FORM is used as follows:

```
SUB ChangeCaption( EmployeeInfo AS FORM, NewCaption$)
  EmployeeInfo.Caption = NewCaption$
END SUB
```

Operators

The following table lists the Visual Basic arithmetic, relational, and logical operators according to precedence, from highest to lowest.

Type	Name of operation	Symbol
Arithmetic	Exponentiation	^
	Negation (additive inverse)	–
	Multiplication and division	*, /
	Integer division	\
	Modulo arithmetic	MOD
	Addition and subtraction	+, –
Relational	Equal, greater than, less than, and so on	=, >, <, <>, <=, >=
Logical (bit-wise)	Logical negation	NOT
	And	AND
	Inclusive OR	OR
	Exclusive OR (either or but not both)	XOR
	Logical equivalence	EQV
	Implication (first operand false or second operand true)	IMP

The following sections describe the logical operators. Logical operators can be used with this syntax:

result = numeric-expression1 logicaloperator numeric-expression2

The truth tables in each of the following sections refer to this syntax.

AND Operator

The AND operator is a logical conjunction operator you use to evaluate two or more relational expressions. If, and only if, both expressions are True (–1) is the entire expression true. If one or none is true, the entire expression is False (0).

The AND operator also performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table:

Bit in expression 1	Bit in expression 2	Bit in result
0	0	0
0	1	0
1	0	0
1	1	1

EQV Operator

The EQV operator performs a bit-wise comparison of indentically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table:

Bit in expression 1	Bit in expression 2	Bit in result
0	0	1
0	1	0
1	0	0
1	1	1

IMP Operator

The IMP operator performs a bit-wise comparison of indentically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table:

Bit in expression 1	Bit in expression 2	Bit in result
0	0	1
0	1	1
1	0	0
1	1	1

NOT Operator

The NOT operator evaluates each bit in a numeric expression and sets the corresponding bit in *result* to its bit-wise complement according to the following truth table:

Bit in expression 1	Bit in expression 2
0	1
1	0

The NOT operator inverts the bit values of any variable. If an integer variable has the value 0 (false), the variable becomes –1 (true), and vice versa.

OR Operator

The OR operator is a logical disjunction operator you use to evaluate two or more relational expressions. If either expression is True (–1), or both expressions are true, the entire expression is true. If neither is true, the entire expression is False (0).

The OR operator also performs a bit-wise comparison of indentically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table:

Bit in expression 1	Bit in expression 2	Bit in result
0	0	0
0	1	1
1	0	1
1	1	1

XOR Operator

The XOR operator is a logical exclusive or operator you use to evaluate two or more relational expressions. If either expression is True (–1), but not both expressions, the entire expression is true. If both expressions are true or both expressions are false, the entire expression is False (0).

The XOR operator also performs a bit-wise comparison of indentically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following truth table:

Bit in expression 1	Bit in expression 2	Bit in result
0	0	0
0	1	1
1	0	1
1	1	0

Error Messages

This appendix lists:

- Run-time error messages that can be trapped in your program
- Basic compiler (BC) error messages
- CVPACK Errors
- Microsoft Library Manager (LIB) error messages
- Microsoft Segmented-Executable Linker (LINK) error messages

Trappable Error Messages

Table E.1 lists the error messages you can trap at run time. Error code is the number returned by the ERR function. When programming, you should rely on the error code rather than the message, since the text of the error message may change between versions of Basic.

Table E.1 Trappable Errors

Error code	Message
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of DATA
5	Illegal function call
6	Overflow
7	Out of memory
8	Label not defined
9	Subscript out of range
10	Duplicate definition
11	Division by zero
12	Illegal in direct mode
13	Type mismatch
14	Out of string space
16	String formula too complex
17	Cannot continue
18	Function not defined
19	No RESUME

Table E.1 Trappable Errors (*continued*)

Error code	Message
20	RESUME without error
24	Device timeout
25	Device fault
26	FOR without NEXT
27	Out of paper
29	WHILE without WEND
30	WEND without WHILE
33	Duplicate label
35	Subprogram not defined
37	Argument-count mismatch
38	Array not defined
40	Variable required
50	FIELD overflow
51	Internal error
52	Bad file name or number
53	File not found
54	Bad file mode
55	File already open
56	FIELD statement active
57	Device I/O error
58	File already exists
59	Bad record length
61	Disk full
62	Input past end of file
63	Bad record number
64	Bad file name
67	Too many files
68	Device unavailable
69	Communication-buffer overflow
70	Permission denied
71	Disk not ready
72	Disk-media error
73	Feature unavailable
74	Rename across disks
75	Path/File access error

Table E.1 Trappable Errors (*continued*)

Error code	Message
76	Path not found
80	Feature removed
81	ISAM - Invalid name
82	ISAM - Table not found
83	ISAM - Index not found
84	ISAM - Invalid column
85	ISAM - No current record
86	ISAM - Duplicate value for unique index
87	ISAM - Invalid operation on NULL index
88	ISAM - Database needs repair
89	ISAM - Insufficient ISAM buffers
260	No timer available
271	Invalid screen mode
272	Invalid when forms are showing
340	Control array element does not exist
341	Invalid object array index
342	Not enough room to allocate control array
343	Object not an array
344	Must specify index for object array
345	Reached limit: cannot create any more controls for this form
360	Object already loaded
361	Can't load or unload this object
362	Can't unload controls created at design time
364	Object was unloaded
365	Unable to unload within this context
380	Invalid property value
381	Invalid property array index
382	Property can't be set at run time
383	Property is read-only
384	Property can't be modified when form is minimized or maximized
385	Must specify index when using property array
386	Property not available at run time
387	Property can't be set on this control
400	Form already displayed; can't show form modally
401	Can't show non-modal form when modal form is displayed

Table E.1 Trappable Errors (*continued*)

Error code	Message
402	Must close or hide topmost modal form first
403	MDI form cannot be shown modally
410	Property can't be modified on MDI form
420	Invalid object reference
421	Method not applicable for this object
422	Property not found
423	Property or control not found
424	Object required
425	Invalid object use
430	No currently active control
431	No currently active form
480	Can't create AutoRedraw image

BC Error Messages

Argument-count mismatch (ERR: 37)

An incorrect number of arguments was used in a SUB or FUNCTION procedure call.

Possible solutions:

- In the programming environment, use the Find command from the Search menu to examine the DECLARE statement.
- Compare the DECLARE statement for the SUB or FUNCTION procedure with the CALL statement to make sure the argument list has the same number of items in both cases.

Array already dimensioned

More than one condition can cause this error:

- More than one DIM statement for the same static array.
- An OPTION BASE statement that occurs after an array is dimensioned.
- An attempt to REDIM a static array.

Possible solutions:

- Using more than one DIM statement for the same static array.
- Using OPTION BASE after an array is dimensioned.
- Attempting to redimension a static array using REDIM.

Array not defined (ERR: 38)

Your program is attempting to use an array that is not currently defined.

Make sure the array is defined with a DIM or REDIM statement.

Array not dimensioned (warning)

An array is referred to but its dimensions have not been declared.

If you are compiling with BC, this error is not fatal; that is, the program will execute, although program results may be incorrect.

Array too big

There is not enough user data space to accommodate the array declaration. Possible solutions:

- Reduce the size of the array.
- Use the $DYNAMIC metacommand at the beginning of your program.

This error also may occur if the array size exceeds 64K, the array is not dynamic, and the /AH option was not used on the compiler invocation line. (All three of these conditions must be present to generate the error.) You can also try:

- Reducing the size of the array
- Making the array dynamic and using the /AH command-line option.

AS clause required

A variable declared with an AS clause is referred to without that clause.

If the first declaration of a variable has an AS clause, every subsequent DIM, REDIM, SHARED, or COMMON statement that refers to that variable must have an AS clause.

Remove the AS clause from initial variable declaration or add the AS clause to an invalid reference.

AS clause required on first declaration

A variable that has not been declared using an the AS clause is being referenced with an AS clause.

Change the variable declaration to include an AS clause or remove the invalid reference.

AS missing

The compiler expects an AS keyword, as in the following correct statement:

```
OPEN "FILENAME" FOR INPUT AS #1
```

Asterisk missing

The asterisk is missing from a string definition in a user-defined type.

Bad file mode (ERR: 54)

More than one condition can cause this error:

- You attempted to load a file with an invalid format. For example, you tried to load a Visual Basic for Windows binary form file.
- You tried to load a text file as a binary file, or vice versa.
- A PUT or GET statement specified a sequential file.
- A FIELD statement specified a file not opened for random access.
- A PRINT # statement specified a sequential file opened for input.
- An INPUT # statement specified a file opened for output or appending.
- Visual Basic tried to use an include file previously saved in binary format. Include files must be saved in text format.
- An ISAM statement or function file-number argument refers to an open file that is not an ISAM table.
- You are using the CHAIN or RUN statement and errors have occurred. For example, the program you ran using the RUN statement may contain duplicate procedure names.

Possible solutions:

- Resave the file in text format and reload as a text file.
- Resave the file in binary format and reload as a binary file.
- Verify the correct file criteria for your PUT, GET, FIELD, PRINT #, or INPUT # statements.
- Reload the include file, save it in text format, then run the program again.
- Make sure all ISAM file references refer to the correct file type.
- If you are using the CHAIN or RUN statement, loading all files into the programming environment may give you additional information on the type of errors encountered.

If you need additional help, move the cursor to a keyword and press F1 to open the Help window.

BASE missing

Basic expected the keyword BASE here, as in OPTION BASE.

Binary source file (warning)

More than one condition can cause the error:

- The file you have attempted to compile is not compatible with this version of Basic. Try saving the binary file as an ASCII file and then recompile it. (All source files saved by BasicA should be saved with the ,A option.)
- You try to use the /ZI or /ZD CodeView options with binary source files. (This is a warning message only.)

Block IF without END IF

There is no corresponding END IF in a block IF construct.

Possible solutions:

- Insert an END IF statement at the end of the IF...END IF structure.
- Check other control structures within the IF...END IF block and verify that they are correctly matched.

BYVAL allowed only with numeric arguments

You cannot pass string or record arguments with a BYVAL clause.

Cannot generate listing for Basic binary source files (warning)

You are attempting to compile a binary source file with the BC command and the /A option. Recompile without the /A option.

CASE without SELECT

The first part of a SELECT CASE statement is missing or misspelled.

Possible solutions:

- Insert the SELECT keyword at the beginning of the SELECT...END SELECT structure or correct misspelling as necessary.
- Check other control structures within the SELECT statement block and verify that they are correctly matched.

Comma missing

Basic expects a comma.

COMMON name illegal

Basic encountered an illegal block-name specification (for example, a block name that is a Basic reserved word) in a named COMMON block.

CONST/DIM SHARED follows SUB/FUNCTION (warning)

CONST and DIM SHARED statements should appear before any SUB or FUNCTION procedure definitions.

If you are compiling with BC, this error is not fatal; that is, the program will execute, although the results may be incorrect.

Control structure in IF...THEN...ELSE incomplete

An unmatched NEXT, WEND, END IF, END SELECT, or LOOP statement appears in a single-line IF...THEN...ELSE statement.

CURRENCY type illegal in alternate math pack

This error occurs whenever the CURRENCY type is used with an alternate math pack. If you use the /Fpa option while compiling, your program cannot contain:

- The at symbol (@) to denote CURRENCY data type.
- The DEFCUR statement.
- The CURRENCY keyword.

Data-memory overflow

There is too much program data to fit in memory. This error is often caused by too many constants or too much static array data. If you are using the BC command, or the Make EXE File or Make Library command, try turning off any debugging options. If memory is still exhausted, break your program into multiple modules or use the /AH and /FS options.

DECLARE required

An implicit SUB or FUNCTION procedure call appears before the procedure definition. (An implicit call does not use the CALL statement.) All procedures must be defined or declared before they are implicitly called.

DEF without END DEF

There is no corresponding END DEF in a multiline function definition.

Insert and END DEF statement at the end of the DEF FN definition.

DEFxxx character specification illegal

You specified a nonalphabetic character in the letter range of the DEF*type* statement.

Division by zero (ERR: 11)

This error is caused by dividing by zero in an expression.

Possible solutions:

- Check the variable values and intermediate calculations to identify the source of the miscalculation.
- Incorporate error checking into your routine to prevent this error.

DO without LOOP

The terminating LOOP clause is missing from a DO...LOOP statement.

Possible solutions:

- Insert the LOOP keyword at the end of the DO...LOOP structure.
- Check other control structures within the DO...LOOP block to verify that they are correctly matched.

Duplicate definition (ERR: 10)

More than one condition can cause this error:

- You tried to change the dimensions of a static array while your program is running.
- For ISAM, you tried to create an index that already exists.

Possible solutions:

- If you plan to change the dimensions of an array while your program is running, make sure it is a dynamic array and use the REDIM statement to make the change.
- Use REDIM PRESERVE to enlarge or reduce the size of the array without losing data.

Duplicate label (ERR: 33)

Two program lines were assigned the same number or label. Each line number or label in a module must be unique.

Possible solutions:

- Delete the highlighted item and specify a unique line label or line number.
- Check all SUB and FUNCTION procedure definitions in the module, as well as the module-level code, for the duplicate label; when it is found, rename or delete.
- In the programming environment, use the Find command from the Search menu to locate the SUB and FUNCTION procedure definitions.

Dynamic array element illegal

Dynamic array elements are not allowed with VARPTR$.

Dynamic array illegal

Dynamic arrays cannot be user-defined type elements. If you use an array as an element of a user-defined type, it must be a static array.

Element not defined

A user-defined type element is referenced, but not defined.

For example, if the user-defined type MYTYPE contained elements A, B, and C, then an attempt to use the variable D as an element of MYTYPE would cause this message to appear.

Possible solutions:

- Locate the appropriate TYPE statement and define the element referenced.
- Verify that you are referencing the correct user-defined type element.

ELSE without IF

An ELSE clause appears without a corresponding IF. Sometimes this error is caused by incorrectly nested IF statements.

Possible solutions:

- Insert the IF keyword at the beginning of the IF...END IF structure.

Check other control structures within the IF...END IF block to verify that they are correctly matched. For example, a nested IF block without a matching END IF inside the outer IF...END IF block may generate this error.

ELSEIF without IF

An ELSEIF statement appears without a corresponding IF. Sometimes this error is caused by incorrectly nested IF statements.

END DEF without DEF

An END DEF statement has no corresponding DEF FN statement.

Delete the END DEF statement or insert a corresponding DEF FN statement.

END IF without Block IF

The beginning of an IF block is missing.

Possible solutions:

- Insert the IF keyword at the beginning of the IF...ENDIF structure.
- Check other control structures within the IF...ENDIF block to verify that they are correctly matched. For example, a nested IF block without a matching END IF inside the outer IF...END IF block generates this error.

End of file unexpected in TYPE declaration

If, while parsing a TYPE statement, the compiler reaches the end of the source file without seeing the END TYPE statement, the program halts execution.

END SELECT without SELECT

The end of a SELECT CASE statement appears without a beginning SELECT CASE. The beginning of the SELECT CASE statement may be missing or misspelled.

Possible solutions:

- Insert the SELECT CASE keywords at the beginning of the SELECT...END SELECT structure or correct misspelling as necessary.
- Check other control structures within the SELECT statement block to verify that they are correctly matched.

END SUB/FUNCTION without SUB/FUNCTION

The SUB or FUNCTION statement is missing from a procedure. You may have deleted it during editing.

Equal sign missing

Basic expects an equal sign (=).

EVENT ON without /v or /w on command line (warning)

The EVENT ON statement requires a /V or /W option on the command line.

This is a warning message only, and can be generated in two different situations:

- If there are ON *event* statements (such as ON KEY or ON COM) in the program, you must use a /V or /W option on the BC command line for the program to run properly.
- If there are no ON *event* statements in the program, then you must change the EVENT ON statement to an EVENT OFF statement, and must not use a /V or /W option.

EXIT DO not within DO...LOOP

An EXIT DO statement is used outside of a DO...LOOP statement.

Delete the EXIT DO statement or reposition it within a DO...LOOP structure.

EXIT FOR not within FOR...NEXT

An EXIT FOR statement is used outside of a FOR...NEXT statement.

Delete the EXIT FOR statement or reposition it within a FOR...NEXT structure.

Expression too complex

Certain internal limitations of Visual Basic have been exceeded.

For example, during expression evaluation, strings that are not associated with variables are assigned temporary locations. A large number of such strings or a numeric expression with many complicated subexpressions can cause this error.

Possible solution:

Simplify expressions or assign strings to variables.

Feature unavailable (ERR: 73)

You tried to perform an ISAM operation without the correct ISAM TSR loaded (PROISAM or PROISAMD), or FRE(-3) when no expanded memory is available.

You also may have tried to use a feature of another version of Basic that is not available with this version of Microsoft Visual Basic. or a Microsoft Basic that is unavailable under the operating system you are running (for example, using the LOCK or UNLOCK statement while running MS-DOS 2.1).

Fixed-length string illegal

You cannot use a fixed-length string as a formal parameter to a SUB or FUNCTION procedure.

Compare the SUB/FUNCTION call with the applicable DECLARE statement and verify that defined parameters do not conflict.

FOR index variable already in use

This error occurs when an index variable is used more than once in nested FOR loops.

Use a unique variable for each FOR...NEXT structure.

FOR index variable illegal

This error is usually caused when an incorrect variable type is used in a FOR-loop index. A FOR-loop index variable must be a simple numeric variable.

FOR without NEXT (ERR: 26)

Each FOR statement must have a matching NEXT statement. For example:

```
FOR i% = 1 TO 10
  PRINT i%
  FOR x% = 1 TO 10
    PRINT x%
  NEXT x%
NEXT i%
```

Form variable already in use or illegal form name

The form name in a $FORM metacommand has either already been declared as a variable or procedure, or it contains nonalphanumeric characters.

Formal parameter specification illegal

There is an error in a SUB or FUNCTION parameter list.

Formal parameters not unique

A SUB or FUNCTION declaration contains duplicate parameters, as in this example:

```
SUB GetName(A,B,C,A) STATIC
```

Function name illegal

The name of a DEF FN function must begin with FN.

Function not defined (ERR: 18)

You must define a function before using it.

Possible solutions:

- Verify that the name of the function you are calling is spelled correctly.
- Use the DECLARE statement to declare the function procedure and invoke argument data-type checking.
- Use the DEF FN statement to define the function.

GOSUB missing

GOSUB is missing from an ON *event* statement.

GOTO missing

GOTO is missing from an ON ERROR statement.

GOTO or GOSUB expected

Basic expects a GOTO or GOSUB statement.

Identifier cannot include period

User-defined type identifier and record element names cannot contain periods.

A variable name cannot contain a period if the part of the name preceding the period has been used in an AS usertype clause anywhere in the program.

Although variable names can contain periods, it's a good idea to use a period only as a record variable separator.

If you have programs that use a period in variable names, you could change them to use mixed case instead. For example, variable ALPHA.BETA would become AlphaBeta.

Identifier expected

You are attempting to use a number or a Basic reserved word where an identifier is expected.

Illegal type character in numeric constant

A numeric constant contains an inappropriate type-declaration character.

$INCLUDE-file access error

The include file named in the $INCLUDE metacommand cannot be located.

Input file not found

The source file you named on the BC command line was not found, so the compilation was not carried out.

Check the spelling of the source file name and the subdirectory location of the source file. Then retype the BC command line with the correct path specification for the source file.

INPUT missing

Basic expects the reserved word INPUT.

Integer between 1 and 32767 required

The statement requires an integer argument in the specified range.

Invalid character

Basic found an invalid character, such as a control character, in the source file.

Invalid .MAK file

You have attempted to load a Make (.MAK) file which is not valid in Visual Basic for MS-DOS format.

Possible solutions:

- Check the file name and verify that it is correct.
- Individually load the forms and modules contained in the project.

Invalid in SUB, FUNCTION, or DEF FN

The statement is not allowed in a SUB or FUNCTION procedure or DEF FN function.

Move the highlighted statement to the module level of a non-form module.

Invalid outside SUB, FUNCTION, or DEF FN

Visual Basic has encountered a statement that is not allowed in module-level code. For example:

- EXIT SUB, EXIT FUNCTION, or EXIT DEF statements are not allowed in module-level code.
- Executable statements are not allowed at the module level of a form module.

A possible solution is to move the highlighted statement to a SUB, FUNCTION, or DEF FN function.

Invalid outside SUB/FUNCTION

Visual Basic has encountered a statement that is not allowed in module-level code or DEF FN functions.

Possible solution:

Move highlighted statement to a SUB or FUNCTION procedure.

IS missing

There is a missing IS keyword in the CASE part of a SELECT CASE statement.

Label not defined (warning)

A line label is referenced (in a GOTO statement, for example), but does not occur in the program.

Possible solutions:

- Insert the appropriate line label as referenced in a GOTO or other statement.
- If you are using the RUN statement, make sure the label is defined at the module level in a non-form code module. No executable statements are allowed at the module level of a form module.
- ON...GOSUB and ON ERROR must refer to module-level code. Use ON LOCAL ERROR to refer to an error handler at the subroutine level.

Left parenthesis missing

Basic expected a left parenthesis, or a REDIM statement tried to reallocate space for a scalar.

Line number or label missing

A line number or label is missing from a statement that requires one. For example, a GOTO statement requires a line number or label argument.

Line too long

Lines are limited to 255 characters.

LOOP without DO

The DO starting a DO...LOOP statement is missing or misspelled.

Possible solutions:

- Insert the DO keyword at the beginning of the DO...LOOP structure or correct misspelling as necessary.
- Check other control structures within the DO...LOOP block and verify that they are correctly matched.

Lower bound exceeds upper bound

The lower bound exceeds the upper bound defined in a DIM statement.

Math overflow

The result of a calculation is too large to be represented in Basic number format.

$Metacommand error (warning)

The syntax of a metacommand is incorrect. Metacommands must be preceded by a comment. For example, both of the $INCLUDE metacommands below are correct:

```
REM $INCLUDE: 'filespec'
' $INCLUDE: 'filespec'
```

If you are compiling the program with the Visual Basic Compiler (BC.EXE), this error is not fatal. The program will execute, but the results may be incorrect.

Compare the valid metacomand syntax above to your statement and make changes as necessary.

Method not applicable for this object (ERR: 421)

You tried to use a method that is not available for use with the specified object. For example, Command1.ADDITEM produces this error because the ADDITEM method is used only with a list box or combo box control.

Possible solutions:

- Check the method's topic in online Help for a list of applicable controls; change the reference as required.
- Check the object's topic in online Help for a list of applicable methods; change the reference as required.

Minus sign missing

Basic expects a minus sign.

Module-level code illegal in form modules

You cannot have executable statements at the module level of a form.

All Basic statements are executable except the following:

- COMMON
- CONST
- DATA
- DECLARE
- DEF*type*
- DIM (for static arrays)

- OPTION BASE
- OPTION EXPLICIT
- REM
- SHARED
- STATIC
- TYPE
- All metacommands

Must be first statement on the line

In block IF...THEN...ELSE constructs, IF, ELSE, ELSEIF, and END IF can only be preceded by a line number or label.

In SELECT...END SELECT constructs, CASE and END SELECT can be preceded only by a line number or label.

Remove the statement proceding the SELECT or IF...THEN statement.

Must precede executable statements

A $FORM metacommand, COMMON statement, DECLARE, or an OPTION EXPLICIT statement is misplaced. $FORM, COMMON, DECLARE, and OPTION EXPLICIT must appear before any executable statements. All Basic statements are executable except the following:

- COMMON
- CONST
- DATA
- DECLARE
- DEF*type*
- DIM (for static arrays)
- OPTION BASE
- OPTION EXPLICIT
- REM
- SHARED
- STATIC
- TYPE
- All metacommands

Name of subprogram illegal

A SUB or FUNCTION procedure name is a Basic reserved word, or the same name appears as a variable or SUB or FUNCTION name earlier in the module.

Nested function definition

A SUB or FUNCTION procedure or a DEF FN function appears inside another SUB or FUNCTION procedure or DEF FN function.

NEXT without FOR (ERR: 1)

Each NEXT statement must have a matching FOR statement. For example:

```
FOR i% = 1 TO 10
  PRINT i%
  FOR f% = 1 to 10
    PRINT f%
  NEXT f%
NEXT i%
```

Match up all FOR...NEXT pairs within each control structure; insert FOR or NEXT items as required.

ON ERROR without /E on command line

When using the BC command, programs containing ON [LOCAL] ERROR statements must be compiled with the On Error (/E) option.

ON event without /v or /w on command line

When using the BC command, programs containing ON *event* statements (such as ON KEY, ON TIMER, or ON COM) must be compiled with one of these options. The /V option will cause the program to check between each program statement for occurrence of the event. The /W option will, in most cases, cause this check to be done only at each labeled statement in the program.

Only simple variables allowed

Names of user-defined types and arrays are not permitted as arguments in READ and INPUT statements. Array elements that are not of a user-defined type are permitted.

Overflow in numeric constant

The numeric constant is too large to be represented within the range Basic allows for that type of number.

Parameter type mismatch

A SUB or FUNCTION procedure parameter type does not match the DECLARE statement's argument or the calling argument.

Possible solutions:

- Press F on the variable to display the variable's type; redefine type as necessary.
- Compare the SUB/FUNCTION call with the applicable DECLARE statement and correct conflicts with defined parameter types.

Path/File access error (warning)

More than one condition can cause this error:

- During an OPEN, MKDIR, CHDIR, or RMDIR operation, the operating system was unable to make a correct connection between the path and file name.
- You tried to save a file that replaces an existing read-only file. If your project (.MAK) file is read-only, the entire program cannot be saved.

Possible solutions:

- In the programming environment, make sure the file name you entered in the text box is formatted correctly. Any file name can contain a full or partial path.
- From MS-DOS, use the attrib command to change the read-only status of the specified file.

Property name too long

The property name specified is not a valid property name and exceeds 40 characters.

Property not applicable to object

This property does not exist for the object it is used with. See the documentation for the object to see what properties it has.

Record/string assignment required

You did not use an appropriate data type for a string or record variable assignment in an LSET statement. Numeric types are not allowed.

Record variable required

Before you can use ISAM INSERT, UPDATE, or RETRIEVE in statements, you must define a record variable by use of a TYPE...END TYPE statement.

RESUME without /X on command line

When using the BC command, programs containing RESUME, RESUME NEXT, and RESUME 0 statements must be compiled with the Resume Next (/X) option.

Right parenthesis missing

Basic expects a right (closing) parenthesis.

SEG or BYVAL not allowed in CALLS

BYVAL and SEG are permitted only in a CALL statement.

SELECT without END SELECT

The end of a SELECT CASE statement is missing or misspelled.

Possible solutions:

- Insert an END SELECT statement at the end of the SELECT...END SELECT structure or correct misspelling as necessary.
- Check other control structures within the SELECT statement block and verify that they are correctly matched.

Semicolon missing

Basic expects a semicolon (;).

Separator illegal

There is an illegal delimiting character in a PRINT USING or WRITE statement. Use a semicolon or a comma as a delimiter.

Simple or array variable expected

The compiler expects a variable argument.

Skipping forward to END TYPE statement

An error in the TYPE statement has caused Basic to ignore everything between the TYPE and END TYPE statements.

Statement ignored (warning)

You are using the BC command to compile a program that contains TRON and TROFF statements without using the /D option. This error is not fatal; that is, the program will execute, although the results may be incorrect.

Statement unrecognizable

You have probably typed a Basic statement incorrectly.

Statements/labels illegal between SELECT CASE and CASE

Statements and line labels are not permitted between SELECT CASE and the first CASE statement.

Insert REM statement to comment the line. Comments and statement separators are permitted between SELECT CASE and the first CASE statement

String assignment required

The string assignment is missing from an RSET statement.

String constant required for ALIAS

The DECLARE statement ALIAS keyword requires a string-constant argument.

String expression required

The statement requires a string-expression argument.

String variable required

The statement requires a string-variable argument.

SUB or FUNCTION missing

A DECLARE statement has no corresponding procedure.

SUB/FUNCTION without END SUB/FUNCTION

The terminating statement is missing from a procedure.

Insert the appropriate END SUB or END FUNCTION statement into your procedure.

Subprogram error

More than one condition can cause this error:

- You have a label defined within a SUB or FUNCTION procedure or DEF FN function and you try to branch to the label from outside that procedure or function.
- You have a RETURN statement, with a target line number or line label, within a SUB or FUNCTION procedure.
- You use the STATIC statement outside a SUB or FUNCTION procedure or DEF FN function, or you use the SHARED statement outside a SUB or FUNCTION procedure.
- You place a RUN statement, with no file name, within a SUB or FUNCTION procedure or DEF FN function.

Subprograms not allowed in control statements

SUB, FUNCTION, or DEF FN definitions are not permitted inside control constructs such as IF...THEN...ELSE and SELECT CASE.

Subscript syntax illegal

An array subscript contains a syntax error; for example, an array subscript contains both string and integer data types.

Syntax error (ERR: 2)

More than one condition can cause this error:

- At compile time, the most common cause is an incorrectly typed Basic keyword or argument.
- At run time, this error can be caused by an incorrectly formatted DATA statement.
- In form code, this error can occur when the object specified is not found or explicit reference to the object is required.

- In an ISAM program, this error can occur if the number of *keyvalues* in a SEEKGT, SEEKGE, or SEEKEQ operation is greater than the number of values that makes up the current index.

Possible solutions:

- Check to verify that you have not misspelled or incorrectly formatted the Basic keyword or argument.
- Explicitly reference the object in form code.
- Compare the number of *keyvalues* in the ISAM SEEKxx operation with the number of values in the current index; reconcile as necessary.

Syntax error in numeric constant

A numeric constant is not properly formed.

THEN missing

Basic expects a THEN keyword.

TO missing

Basic expects a TO keyword.

Too many arguments in function call

Calls to SUB or FUNCTION procedures or DEF FN functions are limited to 60 arguments.

Too many dimensions

Arrays are limited to 60 dimensions.

Too many labels

The number of lines in the line list following an ON...GOTO or ON...GOSUB statement exceeds 255.

Too many local strings in a SUB/FUNCTION

The maximum number of local strings in a procedure is 255.

Too many named COMMON blocks

The maximum number of named COMMON blocks permitted in a program is 126.

Too many TYPE definitions

The maximum number of user-defined types permitted in a program is 240.

Too many variables for INPUT

An INPUT statement is limited to 60 variables.

Too many variables for LINE INPUT

Only one variable is allowed in a LINE INPUT statement.

Type mismatch (ERR: 13)

More than one condition can cause this error:

- The variable specified is not of the required type. For example, you tried to use the SWAP statement with a string variable and a numeric variable, or with two numeric variables of different types.
- You tried to specify an alphanumeric label with the RUN statement.
- In form code, you tried to specify an object that cannot be found, or explicit reference to the object is required.
- In a multiple-module program, you have mismatched COMMON statements.
- During an ISAM OPEN FOR operation, the data type of a record element does not match the data type of a table column.
- During an ISAM INSERT operation, the number of elements in the record does not match the number of columns in table, or the data type of a record element does not match the data type of a table column.
- During an ISAM RETRIEVE operation, the data type of a record element does not match the data type of a table column.
- In an ISAM SEEK*operand* statement, there is a data type mismatch between the *keyvalue* argument and the current index.
- During an ISAM UPDATE operation, the record variable is not the same user-defined type as the table you are attempting to update.

TYPE missing

The TYPE keyword is missing from an END TYPE statement.

Type more than 65535 bytes

A user-defined type cannot exceed 64K.

Possible solution:

Split the user-defined type into to record variables or redefine the user-defined type so that it does not exceed 64K.

Type not defined

A variable or parameter was declared with an undefined data type or ISAM table type.

Use the TYPE statement to define the data or table type.

Type not valid for ISAM

The OPEN statement in ISAM uses a user-defined type as the table-type under the following rules:

- The names of the fields must be less than or equal to 30 characters long.
- You cannot have an element of type SINGLE.

Verify that your OPEN statement specifies a user-defined type that adheres to the rules above; change reference as required.

TYPE statement improperly nested

User-defined type definitions are not allowed in procedures.

Typed variable not allowed in expression

Variables that are of user-defined types are not permitted in expressions such as this, where X is a user-defined type:

```
CALL ALPHA((X))
```

Pass elements of user-defined types as arguments. For example:

```
CALL Alpha((X.FirstEl))
```

Variable name not unique

This error may be generated by one of the following conditions:

- You are attempting to define a variable when that variable name already exists at the same program level.
- You are attempting use a dotted variable (for example, `x.a`) in a SHARED statement where the name `x` to already being used as a record variable.
- You are attempting use a dotted variable (for example, `x.a`) in a SUB or FUNCTION procedure where `x` to already being used as a record variable and is not shared.

Variable not declared

You are using OPTION EXPLICIT and did not declare a variable before using it.

Variable required (ERR: 40)

At compile time, Visual Basic encountered an INPUT, LET, READ, or SHARED statement without a variable argument.

At run time, a GET or PUT statement did not specify a variable when an operation was performed on a file opened in binary mode.

Variable-length string required

You cannot use a fixed-length string as a formal parameter to a procedure or within a FIELD statement.

WEND without WHILE (ERR: 30)

Each WEND statement must have a matching WHILE statement.

Verify that other control structures within the WHILE...WEND structure are correctly matched. For example, an IF without a matching ENDIF inside the WHILE...WEND structure generates this error.

The DO...LOOP statement provides a more powerful and flexible loop control structure than WHILE...WEND.

WHILE without WEND (ERR: 29)

Each WHILE statement must have a matching WEND statement.

Verify that other control structures within the WHILE...WEND structure are correctly matched. For example, an IF without a matching ENDIF inside the WHILE...WEND structure generates this error.

Wrong number of dimensions

An array reference contains the wrong number of dimensions.

Possible solution:

Compare the array declaration and correct the invalid reference.

CVPACK Errors

Number	Message
CK1000	**unknown error; contact Microsoft Product Support Services**

CVPACK detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK1001 **out of memory**

The executable file is too big for the available memory. This error can occur under DOS when there is little extra memory. Even though CVPACK uses virtual memory, which involves swapping to disk, some information can be stored only in real memory.

One of the following may be a solution:

- Recompile one or more of the object files without debugging information. If the file was compiled using the /Zi option, use either /Zd or no option.
- Add more memory to your computer.

CK1002 **out of virtual memory**

There was not enough virtual memory for CVPACK to pack the executable file. Virtual memory can be any of the following:

- Conventional memory. Remove TSR (terminate-and-stay-resident) programs or run CVPACK outside of a shell or a makefile.
- Extended or expanded memory. Run CVPACK under a DPMI server, such as Qualitas' 386-Max or the server provided in a DOS session under Windows enhanced mode.
- Disk space. Free some disk storage.

CK1003 **cannot open file**

CVPACK could not open the specified executable file.

More than one condition may cause this error:

- The specified file does not exist. Check the spelling of the file name and path.
- The executable file was opened or deleted by another process.
- There was not enough space on disk.

CK1004 **file is read-only**

CVPACK cannot pack a read-only file. Change the read attribute on the executable file and run CVPACK again.

CK1005 **invalid executable file**

CVPACK could not process the executable file. One of the following may be a cause:

- The debugging information in the executable file is corrupt.
- The executable file is a zero-length file.

CK1006 **invalid module** *module*

The given object file did not have a valid format.

Check the linker version.

CK1007 **invalid** *table* **table in module** module

The given table in the given object file was not valid.

Check the compiler and linker versions.

CK1008 **cannot write packed information**

There was not enough space on disk for CVPACK to write the packed executable file. This leaves a corrupt file on disk.

Make more space available on disk and relink the program.

CK1009 **module *module* unknown type index *number*; contact Microsoft Product Support Services**

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK1010 **symbol error in module *module*; contact Microsoft Product Support Services**

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK1011 **error in type *number* for module *module*; contact Microsoft Product Support Services**

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK1012 **no Symbol and Type Information**

The executable file does not contain debugging information.

Link the program using the /CO option to put at least minimal debugging information in the executable file. To include full debugging information in an object file, compile or assemble using the /Zi option. To include minimal information and line numbers, compile or assemble using the /Zd option.

CK1013 **debugging information missing or unknown format**

One of the following has occurred:

- The program did not contain debugging information. Recompile using /Zi or /Zd, then link using /CO.
- The executable file was linked using an obsolete or unsupported linker. Use Microsoft LINK version 5.3x or later.
- The executable file was already packed using a previous version of CVPACK.

CK1014 **module *module* type *number* refers to skipped type index; contact Microsoft Product Support Services**

The debugging information in the executable file is corrupt. This is due to an internal error in the compiler. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK1015 **too many segments in module** *module*

The alloc_text pragma was used more than 20 times in an object file that was compiled with Microsoft C version 6.x or earlier.

One of the following may be a solution:

- Recompile using Microsoft C/C++ version 7.0 or later.
- Split the object file into multiple files.
- Group the pragma statements according to segment.

CK1016 **unable to execute MPC for CVPACK /PCODE**

CVPACK could not find MPC.EXE on the path.

CK1017 **precompiled types file** *filename* **not found**

The program used a precompiled header, but the program was linked without the object file that was created when the header was precompiled.

CK1018 **precompiled types object file** *filename* **inconsistent with precompiled header used to compile object file** *filename*

The program used a precompiled header, but the object file linked to the program was not the object file that was created when the header was precompiled. Either the user or the creator changed since the last compilation.

Recompile and relink. If a makefile is used, check the makefile dependencies.

CK1020 **packed type index exceeds 65535 in module** *module*

The debugging information exceeded a CVPACK limit.

This error may occur when precompiled headers are used.

One of the following may be a solution:

- Eliminate unused type strings.
- Compile some object files without debugging information.

CK1021 **error in precompiled types signature in module** *module*

The program was compiled with an out-of-date precompiled header.

Delete the object file and recompile.

CK1022 **symbol table for** *filename* **too large**

The size of the debugging information exceeded CVPACK's capacity.

Reduce the number of symbols in the program, or compile some .OBJ files without debugging information.

CK4000 **unknown warning; contact Microsoft Product Support Services**

CVPACK detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

CK4001 **file already packed**

CVPACK took no action because the executable file has already been processed by CVPACK 4.00.

CK4002 **duplicate public symbol** *symbol* **in module** *module*

The given symbol was redefined in the given module. CVPACK deleted the second occurrence of the symbol.

Probably an earlier version of the linker was used. Use LINK 5.30 or later.

CK4003 **error in lexical scopes for module** *module*, **symbols deleted**

The scoping of symbols in the given object module was corrupted. CVPACK deleted the symbols in the module.

This is probably a compiler error. Recompile and relink the object file.

LIB Error Messages

Number **Message**

U1150 **page size too small; use option /PAGE:n to increase it**

The page size of an input library was too small, indicating an invalid input .LIB file.

U1151 **syntax error : illegal file specification**

A command operator was not followed by a module name or file name.

One possible cause of this error is an option specified with a dash (-) instead of a forward slash (/).

U1152 **syntax error : option name missing**

A forward slash (/) appeared on the command line without an option name after it.

U1153 **syntax error : option value missing**

The /PAGE option was given without a value following it.

U1154 **unrecognized option**

An unrecognized name followed the option indicator (/).

An option is specified by a forward slash (/) and a name. The name can be specified by a legal abbreviation of the full name.

U1155 **syntax error : illegal input**

A specified command did not follow correct LIB syntax.

U1156 **syntax error**

A specified command did not follow correct LIB syntax.

U1157 **comma or newline missing**

A comma or newline character was expected in the command line but did not appear.

One cause of this error is an incorrectly placed comma, as in the following command line:

LIB *math.lib*, *-mod1 +mod2*;

The line must be entered as follows:

LIB *math.lib* - mod1 *+mod2*;

U1158 **terminator missing**

The last line of the response file supplied to LIB did not end with a newline character.

U1161 **cannot rename old library**

LIB could not rename the old library with a .BAK extension because the .BAK version already existed with read-only protection.

Change the protection attribute on the .BAK file.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed with a .BAK extension.

One of the following conditions may have this error:

- Another process deleted the file or changed it to read-only.
- The floppy disk containing the file was removed.
- A hard-disk error occurred.

U1163 **error writing to cross-reference file**

The disk or root directory was full.

Delete or move files to make space.

U1164 **name length exceeds 255 characters**

A file name specified on the command line exceeded the LIB limit of 255 characters. Reduce the number of characters in the name.

U1170 **too many symbols**

The number of symbols in all object files and libraries exceeded the capacity of the dictionary created by LIB.

Create two or more smaller libraries.

U1171 **insufficient memory**

LIB did not have enough memory to run.

Remove any shells or resident programs, or add more memory.

U1172 **no more virtual memory**

The LIB session required more memory than the 1-megabyte limit imposed by LIB.

Try using the /NOE option or reducing the number of object modules.

U1173 **internal failure**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1174 **mark : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1175 **free : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1180 **write to extract file failed**

The disk or root directory was full.

Delete or move files to make space.

U1181 **write to library file failed**

The disk or root directory was full.

Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory was full, or the given extract file already existed with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 **cannot open response file**

The response file was not found.

U1184 **unexpected end-of-file on command input**

An end-of-file character was received prematurely in response to a prompt.

U1185 **cannot create new library**

The disk or root directory was full, or the library file already existed with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 **error writing to new library**

The disk or root directory was full.

Delete or move files to make space.

U1187 **cannot open temporary file VM.TMP**

The disk or root directory was full.

Delete or move files to make space.

U1188 **insufficient disk space for temporary file**

The library manager cannot write to the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1189 **cannot read from temporary file**

The library manager cannot read the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1190 **interrupted by user**

LIB was interrupted with either CTRL+C or CTRL+BREAK.

U1191 *libraryname* **: cannot write to read-only file**

Operations cannot be performed on the given library because it is marked as a read-only file.

Change the protection attribute on the library.

U1200 *filename* **: invalid library header**

The input library file had an invalid format.

Either it was not a library file or it was corrupted.

U1203 *filename* **: invalid object file near** *location*

The given file was not a valid object file or was corrupted at the given location.

U2152 *filename* **: cannot create listing**

One of the following may have occurred:

- The directory or disk was full.
- The cross-reference-listing file already existed with read-only protection.

U2155 *module* **: module not in library; ignored**

The specified module was not found in the input library.

One cause of this error is a file name or directory containing a a hyphen or dash (-). LIB interprets the dash as the operator for the delete command. This error occurs if you install a Microsoft language product in a directory that has a dash in its pathname, such as C:\MS-C. The SETUP program for a language calls LIB to create combined libraries, but the dash in the command line passed to LIB causes the library-building session to fail.

Another possible cause of this error is an option specified with a dash (-) instead of a forward slash (/).

U2157 *filename* **: cannot access file**

LIB was unable to open the specified file, probably because the file did not exist.

Check the path and file name.

U2158 *library* **: invalid library header; file ignored**

The given library had an incorrect format and was not combined.

U2159 *filename* **: invalid format (***number***); file ignored**

The given file was not recognized as a XENIX archive and was not combined.

U4150 *module* : **module redefinition ignored**

A module was specified with the add operator (+) to be added to a library, but a module having that name was already in the library.

One cause of this error is an incorrect specification of the replace operator (-+).

U4151 *symbol* : **symbol defined in module** *module*; **redefinition ignored**

The given symbol was defined in more than one module.

U4153 *option* : *value* : **page size invalid; ignored**

The argument specified with the /PAGE option was not valid for that option. The value must be an integer power of 2 between 16 and 32,768. LIB assumed an existing page size from a library that is being combined.

U4155 *modulename* : **module not in library**

The given module specified with a command operator does not exist in the library.

If the replacement command (-+) was specified, LIB added the file anyway. If the delete (-), copy (*), or move (-*) command was specified, LIB ignored the command.

U4156 *library* : **output-library specification ignored**

A new library was created because the file name specified in the <*oldlibrary*> field did not exist. However, a filename was also specified in the <*newlibrary*> field. LIB ignored the <*newlibrary*> specification.

For example, both of the following command lines cause this error if PROJECT.LIB does not already exist:

LIB *project.lib +one.obj, new.lst, project.lib* LIB *project.lib +one.obj, new.lst, new.lib*

U4157 **insufficient memory, extended dictionary not created**

Insufficient memory prevented LIB from creating an extended dictionary.

The library is still valid, but the linker cannot take advantage of the extended dictionary to speed linking.

U4158 **internal error, extended dictionary not created**

An internal error prevented LIB from creating an extended dictionary.

The library is still valid, but the linker cannot take advantage of the extended dictionary to speed linking.

LINK Error Messages

Number	Message

L1001 *option* **: option name ambiguous**

A unique option name did not appear after the option indicator.

An option is specified by a forward slash (/) and a name. The name can be specified by an abbreviation of the full name, but the abbreviation must be unambiguous.

For example, since many options begin with the letter N, the following command causes this error:

```
LINK /N main;
```

This error can also occur if the wrong version of the linker is being used. Check the directories in the PATH environment variable for other versions of LINK.EXE.

L1003 **/Q and /EXEPACK incompatible**

LINK cannot be given both the /Q option and the /EXEPACK option.

L1004 *value* **: invalid numeric value**

An incorrect value was specified with a LINK option.

For example, this error occurs if a nonnumeric string is specified with an option that requires a number.

L1005 *option* **: packing limit exceeds 64K**

The value specified with the /PACKC or /PACKD option exceeded the limit of 65,536 bytes.

L1006 *number* **: stack size exceeds 64K-2**

One of the following may have occurred:

- The given value specified with the /STACK option exceeded the limit of 65,534 bytes.
- A space appeared before or after the colon (:) between /STACK and the argument specified with it.

L1007 **/OVERLAYINTERRUPT : interrupt number exceeds 255**

An overlay interrupt number greater than 255 was specified with the /OV option value.

Check the *DOS Technical Reference* or other DOS technical manual for information about interrupts.

L1008 **/SEGMENTS : segment limit set too high**

The value specified with the /SEG option exceeded 16,375.

L1009 *value* **: /CPARM : illegal value**

The value specified with the /CPARM option was not in the range 1-65,535.

L1020 **no object files specified**

The object-files field was empty.

LINK requires the name of at least one object file.

L1021 **cannot nest response files**

A response file was specified in a response file.

L1022 **response line too long**

A line in a response file was longer than 255 characters.

To extend a field to another line, put a plus sign (+) at the end of the current line.

L1023 **terminated by user**

The LINK session was halted by CTRL+C or CTRL+BREAK.

L1024 **nested right parentheses**

The parentheses for assigning overlays were specified incorrectly.

L1025 **nested left parentheses**

The parentheses for assigning overlays were specified incorrectly.

L1026 **unmatched right parenthesis**

The parentheses for assigning overlays were specified incorrectly.

L1027 **unmatched left parenthesis**

The parentheses for assigning overlays were specified incorrectly.

L1030 **missing internal name**

An IMPORTS statement specified an ordinal value but not an internal name for the routine or data item being imported.

An item imported by ordinal must be given an internal name.

L1031 **module description redefined**

The module-definition (.DEF) file contained more than one DESCRIPTION statement.

L1032 **module name redefined**

The module-definition (.DEF) file contained more than one NAME or LIBRARY statement.

L1033 **input line too long; *number* characters allowed**

The LINK command line cannot exceed the given number of characters.

L1034 **name truncated to *string***

A name specified either on the LINK command line or in a module- definition (.DEF) file exceeded 255 characters. The name was truncated to the given string.

This is a warning, not a fatal error. However, it indicates a serious problem. This message may be followed by another error as LINK tries to use the specified name. For example, if the string is a file name, LINK issues an error when it cannot open the file.

L1035 **syntax error in module-definition file**

A statement in the module-definition (.DEF) file was incorrect.

L1040 **too many exported entries**

The program exceeded the limit of 65,535 exported names.

L1041 **resident names table overflow**

The size of the resident names table exceeded 65,535 bytes.

An entry in the resident names table is made for each exported routine designated RESIDENTNAME and consists of the name plus three bytes of information. The first entry is the module name.

Reduce the number of exported routines or change some to nonresident status.

L1042 **nonresident names table overflow**

The size of the nonresident names table exceeded 65,535 bytes.

An entry in the nonresident names table is made for each exported routine not designated RESIDENTNAME and consists of the name plus three bytes of information. The first entry is the DESCRIPTION statement.

Reduce the number of exported routines or change some to resident status.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Replace long references with short references wherever possible.

L1044 **imported names table overflow**

The size of the imported names table exceeds 65,535 bytes.

An entry in the imported names table is made for each new name given in the IMPORTS section, including the module names, and consists of the name plus one byte.

Reduce the number of imports.

L1045 **too many TYPDEF records**

An object file contained more than 255 TYPDEF records.

TYPDEF records describe communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

This error appears only with programs created by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1046 **too many external symbols in one module**

An object file specified more than 1,023 external symbols.

Break the object file into smaller files.

L1047 **too many group, segment, and class names in one module**

An object file contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes in the object file, or break the object file into smaller files.

L1048 **too many segments in one module**

An object file had more than 255 segments.

Either create fewer segments or break the object file into smaller files.

L1049 **too many segments**

The program contained more than the maximum number of segments.

The maximum number of segments is set with the /SEG option (in the range 1-16,375). If /SEG is not specified, the default is 128.

If this error occurs when linking a p-code program, recompile and use CL's /NQ option to combine the temporary p-code segments.

L1050 **too many groups in one module**

An object file contained more than 21 group definitions (GRPDEF).

Reduce the number of group definitions or split the module.

(Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1051 **too many groups**

The program defined more than 20 groups, not counting DGROUP.

Reduce the number of groups.

L1052 **too many libraries**

An attempt was made to link with more than 32 libraries.

Combine libraries, or use modules that require fewer libraries.

L1053 **out of memory for symbol table**

The program had more symbolic information than could fit in available memory. Symbolic information includes public, external, segment, group, class, and file names.

One of the following may be a solution:

- Eliminate as many public symbols as possible.
- Combine object files or segments.
- Link from the command line instead of from a makefile or PWB.
- Remove TSRs or otherwise free some memory.

L1054 **requested segment limit too high**

LINK did not have enough memory to allocate tables describing the requested number of segments. The number of segments is the value specified with the /SEG option or the default of 128.

One of the following may be a solution:

- Link again using the /SEG option to set fewer segments.
- Remove terminate-and-stay-resident programs or otherwise free some memory.

L1056 **too many overlays**

The program defined more than 127 overlays.

L1057 **data record too large**

An LEDATA record in an object module contained more than 1,024 bytes of data. This is a translator error. (LEDATA is a DOS term explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module. Please report the circumstances of the error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1063 **out of memory for debugging information**

LINK ran out of memory for processing debugging information.

Reduce the amount of debugging information by compiling some object files with /Zd instead of /Zi or with neither option.

L1064 **out of memory - *near/far* heap exhausted**

LINK was not able to allocate enough memory for the given heap.

One of the following may be a solution:

- Reduce the size of code, data, and symbols in the program.
- If the the program is a segmented executable file, put some code into a dynamic-link library.

L1065 **too many interoverlay calls use /DYNAMIC:<nnn>; current limit is** *number*

The program had more than the given limit of interoverlay calls.

The maximum number of interoverlay calls is set with the /DYNAMIC option (in the range 1-10,922). If /DYNAMIC is not specified, the default is 256.

To determine the setting needed by the program, run LINK with the /INFO option. The output gives the number of interoverlay calls that are generated and the current limit.

L1066 **size of** *overlaynumber* **overlay exceeds 64K**

The overlay represented by the given number exceeded the MOVE size limit of 65,535 bytes.

L1070 *segment* **: segment size exceeds 64K**

A single segment contained more than 65,536 bytes of code or data.

Try changing the memory model to use far code or data as appropriate. If the program is in C, use CL's /NT option or the _based keyword (or its predecessor, the alloc_text pragma) to build smaller segments.

L1071 **segment _TEXT exceeds 64K-16**

The segment named _TEXT grew larger than 65,520 bytes. This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the LINK /DOSSEG option.

Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.

Try compiling and linking using the medium or large model. If the program is in C, use CL's /NT option or the _based keyword (or its predecessor, the alloc_text pragma) to build smaller segments.

L1072 **common area exceeds 64K**

The program had more than 65,536 bytes of communal variables. This error occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1073 **file-segment limit exceeded**

The number of physical or file segments exceeded the limit of 255 imposed by Windows for each application or dynamic-link library.

A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.

Reduce the number of segments, or put more information into each segment. Use the /PACKC option or the /PACKD option or both.

L1074 *group* **: group exceeds 64K**

The given group exceeds the limit of 65,536 bytes.

Reduce the size of the group, or remove any unneeded segments from the group. Refer to the map file for a listing of segments.

L1075 **entry table exceeds 64K-1**

The entry table exceeded the limit of 65,535 bytes.

The table contains an entry for each exported routine and for each address that is the target of a far relocation, when PROTMODE is not enabled and the target segment is designated MOVABLE.

Declare PROTMODE if applicable, reduce the number of exported routines, or make some segments FIXED if possible.

L1078 **file-segment alignment too small**

The segment-alignment size specified with the /ALIGN option was too small.

L1080 **cannot open list file**

The disk or the root directory was full.

Delete or move files to make space.

L1081 **out of space for run file**

The disk or the root directory was full.

Delete or move files to make space.

L1082 *filename* **: stub file not found**

LINK could not open the file given in the STUB statement in the module-definition (.DEF) file.

The file must be in the current directory or in a directory specified by the PATH environment variable.

L1083 **cannot open run file**

One of the following may have occurred:

- The disk or the root directory was full.
- Another process opened or deleted the file.
- A read-only file existed with the same name.
- The floppy disk containing the file was removed.
- A hard-disk error occurred.

L1084 **cannot create temporary file**

One of the following may have occurred:

- The disk or the root directory was full.
- The directory specified in the TMP environment variable did not exist.

L1085 **cannot open temporary file** - *message*

LINK could not open a temporary file for the given reason.

One of the following may have occurred:

- The disk or the root directory was full.
- The directory specified in the TMP environment variable did not exist.

L1086 **temporary file missing**

An internal error has occurred.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1087 **unexpected end-of-file on temporary file**

A problem occurred with the temporary linker-output file.

One of the following may have occurred:

- The disk that holds the temporary file was removed.
- The disk or directory specified in the TMP environment variable was full.

L1088 **out of space for list file**

The disk or the root directory was full.

Delete or move files to make space.

L1089 *filename* **: cannot open response file**

LINK could not find the given response file.

One of the following may have occurred:

- The response file does not exist.
- The name of the response file was incorrectly specified.
- An old version of LINK was used. Check your path. To see the version number of LINK, run LINK with the /? option.

L1090 **cannot reopen list file**

The original floppy disk was not replaced at the prompt.

Restart the LINK session.

L1091 **unexpected end-of-file on library**

The floppy disk containing the library was probably removed.

Replace the disk containing the library and run LINK again.

L1092 **cannot open module-definition file**

LINK could not find the specified module-definition (.DEF) file.

Check that the name of the .DEF file is spelled correctly.

L1093 *filename* **: object file not found**

LINK could not find the given object file.

Check that the name of the object file is spelled correctly.

L1094 *filename* **: cannot open file for writing**

LINK was unable to open the given file with write permission.

Check the attributes for the file.

L1095 *filename* **: out of space for file**

LINK ran out of disk space for the specified output file.

Delete or move files to make space.

L1096 **unexpected end-of-file in response file**

LINK encountered a problem while reading the response file.

One of the following may be a cause:

- The response file is corrupt.
- The file was deleted between reads.

L1097 **I/O error -** *message*

LINK encountered the given input or output error.

L1098 **cannot open include file** *filename* **-** *message*

LINK could not open the given include file for the given reason.

An include file is specified in an INCLUDE statement in the module-definition (.DEF) file.

L1100 **stub .EXE file invalid**

The file specified in the STUB statement in the module-definition (.DEF) file is not a valid DOS executable file.

L1101 **invalid object module**

LINK could not link one of the object files.

Check that the correct version of LINK is being used.

If the error persists after recompiling, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1102 **unexpected end-of-file**

The given library or object file had an invalid format.

L1103 **attempt to access data outside segment bounds**

A data record in an object file specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1104 *filename* **: invalid library**

The given file had an invalid format for a library.

L1105 **invalid object due to interrupted incremental compile**

Delete the object file, recompile the program, and relink.

L1106 **unknown COMDAT allocation type for** *symbol*; **record ignored**

This is a translator error. The given symbol is either a routine or a data item.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1107 **unknown COMDAT selection type for *symbol*; record ignored**

This is a translator error. The given symbol is either a routine or a data item.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1108 **invalid format of debugging information**

This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1113 **unresolved COMDEF; internal error**

This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1114 **unresolved COMDAT *symbol*; internal error**

This is a translator error. The given symbol is either a routine or a data item.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1115 *option* **: option incompatible with overlays**

The given option cannot be used when linking an overlaid program.

L1117 **unallocated COMDAT *symbol*; internal error**

This is a translator error. The given symbol is either a routine or a data item.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1123 *segment* **: segment defined both 16-bit and 32-bit**

Define the segment as either 16-bit or 32-bit.

L1127 **far segment references not allowed with /TINY**

The /TINY option for producing a .COM file was used in a program that has a far segment reference.

Far segment references are not compatible with the .COM file format. High-level-language programs cause this error unless the language supports the tiny memory model. An assembly-language program that references a segment address also causes this error.

For example, the following causes this error:

```
mov ax, seg mydata
```

L1128 **too many nested include files in module-definition file**

Nesting of INCLUDE statements in a module-definition (.DEF) file is limited to 10 levels.

L1129 **missing or invalid include file name**

The file specification in an INCLUDE statement in the module- definition (.DEF) file was missing or was not a valid file name.

L2000 **imported starting address**

The program starting address as specified in the END statement in an assembly-language file is an imported routine. This is not supported by Windows.

L2002 **fixup overflow at** *number* **in segment** *segment*

This error message is followed by one of these strings:

- target external *symbol*
- frm seg *name1*, tgt seg *name2*, tgt offset *number*

A fixup overflow is an attempted reference to code or data that is impossible because the source location (where the reference is made "from") and the target address (where the reference is made "to") are too far apart. Usually the problem is corrected by examining the source location.

For information about frame and target segments, see the *Microsoft MS-DOS Programmer's Reference*

L2003 **near reference to far target at** *offset* **in segment** *segment* **pos:** *offset* **target external** *name*

The program issued a near call or jump to a label in a different segment.

This error occurs most often when specifically declaring an external procedure as near that should be declared as far.

This error can be caused by compiling a small-model C program with CL's /NT option.

L2005 **fixup type unsupported at** *number* **in segment** *segment*

A fixup type occurred that is not supported by LINK. This is probably a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2010 **too many fixups in LIDATA record**

The number of far relocations (pointer- or base-type) in an LIDATA record exceeds the limit imposed by LINK.

The cause is usually a DUP statement in an assembly-language program. The limit is dynamic: a 1,024-byte buffer is shared by relocations and the contents of the LIDATA record. There are 8 bytes per relocation.

Reduce the number of far relocations in the DUP statement.

L2011 *identifier* **: NEAR/HUGE conflict**

Conflicting NEAR and HUGE attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 *arrayname* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was declared once as an array of characters and once as an array of real numbers). This error occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.

L2013 **LIDATA record too large**

An LIDATA record contained more than 512 bytes. This is probably a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2022 *entry* **(alias** *internalname***) : export undefined**

The internal name of the given exported routine or data item is undefined.

Possible causes of this error include the following:

- A C function defined as _pascal was specified with mixed case in an EXPORTS statement in the .DEF file, and the /NOI option was used when linking. The _pascal modifier converts names to uppercase, while the /NOI option turns on case sensitivity, so the names did not match.

- A C function defined as __cdecl was specified without a leading underscore (_) in an EXPORTS statement in the .DEF file. The C naming convention adds a leading underscore to the beginning of identifiers. The names did not match.

L2023 *entry* **(alias** *internalname***) : export imported**

The internal name of the given exported routine or data item conflicts with the internal name of a previously imported routine or data item.

L2024 *symbol* **: special symbol already defined**

The program defined a symbol name already used by LINK for one of its own low-level symbols. For example, LINK generates special symbols used in overlay support and other operations.

Choose another name for the symbol to avoid conflict.

L2025 *symbol* **: symbol defined more than once**

The same symbol has been found in two different object files.

L2026 **entry ordinal** *number*, **name** *name* **: multiple definitions for same ordinal**

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 *name* **: ordinal too large for export**

The given exported name was assigned an ordinal that exceeded the limit of 65,535 (64K-1).

L2028 **automatic data segment plus heap exceed 64K**

The size of the sum of the following exceeds 64K:

- Data declared in DGROUP
- The size of the heap specified in the HEAPSIZE statement in the module-definition (.DEF) file
- The size of the stack specified in either the /STACK option or the STACKSIZE statement in the .DEF file

Reduce near-data allocation, HEAPSIZE, or stack.

L2029 *symbol* **: unresolved external**

A symbol was declared to be external in one or more modules, but it was not publicly defined in any module or library.

The name of the unresolved external symbol is given, followed by a list of object modules that contain references to this symbol. This message and the list of object modules are written to the map file, if one exists.

One cause of this error is using the /NOI option for files that use case inconsistently in identifiers. For example, the __pascal modifier in C programs converts names to uppercase. A name defined as __pascal must elsewhere be specified using uppercase.

Another cause of this error is specifying non-ANSI symbols in the Microsoft C run-time libraries without the single underscore (_) that is required for ANSI compatibility. In Microsoft C versions 7.0 and later, the default library OLDNAMES.LIB provides this compatibility. If /NOD is used when linking, OLDNAMES.LIB is ignored, and the non-ANSI names do not match. Also, .OBJ files compiled with Microsoft C versions earlier than 7.0 do not tell LINK to search OLDNAMES.LIB. If a program consists entirely of old .OBJ files, the non-ANSI names do not match. To avoid this error, specify OLDNAMES.LIB in the *<libraries>* field, or omit a specific default library by specifying /NOD:*<library>*.

This error can also occur when multiple libraries with the same name but different paths are specified. LINK searches only the first library with a given file name. Symbols in other libraries that have the same file name are ignored. To avoid this error, change the name of the other libraries.

L2030 **starting address not code (use class *CODE*)**

The program starting address, as specified in the END statement of an .ASM file, should be in a code segment. Code segments are recognized if their class name ends in *CODE*. This is an error in a segmented executable file.

The error message can be disabled by including the REALMODE statement in the module-definition (.DEF) file.

L2041 **stack plus data exceed 64K**

If the total of near data and requested stack size exceeds 64K, the program will not run correctly. LINK checks for this condition only when /DOSSEG is enabled, which is the case in the library startup module for Microsoft language libraries.

For object modules compiled with the Microsoft C or FORTRAN optimizing compilers, recompile with the /Gt command-line option to set the data-size threshold to a smaller number.

This is a fatal LINK error.

L2043 **Quick library support module missing**

The required file QUICKLIB.OBJ was missing. QUICKLIB.OBJ must be linked in when creating a Quick library.

L2044 *symbol* **: symbol multiply defined, use /NOE**

LINK found what it interprets as a public-symbol redefinition, probably because a symbol defined in a library was redefined.

Relink with the /NOE option. If error L2025 results for the same symbol, then this is a genuine symbol-redefinition error.

L2046 **share attribute conflict - segment** *segment* **in group** *group*

The given segment has a different sharing attribute than other segments that are assigned to the given group.

All segments assigned to a group must have the same attribute, either SHARED or NONSHARED. The attributes cannot be mixed.

L2047 **IOPL attribute conflict - segment** *segment* **in group** *group*

The specified segment is a member of the specified group but has an IOPL attribute that is different from other segments in the group.

L2048 **Microsoft Overlay Manager module not found**

Overlays were designated, but an overlay manager was missing.

By default, the overlay manager is the Microsoft Overlay Virtual Environment (MOVE). This is provided in MOVE.LIB, which is a component library of the default combined libraries provided with Microsoft C/C++ version 7.0. The error occurs when LINK cannot find the _moveinit routine.

If the /OLDOVERLAY option is specified, the overlay manager is the Microsoft Static Overlay Manager, which is also provided in the default combined libraries.

L2050 **USE16/USE32 attribute conflict - segment** *segment* **in group** *group*

You cannot group 16-bit segments with 32-bit segments.

L2052 *symbol* **: unresolved external; possible calling convention mismatch**

A symbol was declared to be external in one or more modules, but LINK could not find it publicly defined in any module or library.

The name of the unresolved external symbol is given, followed by a list of object modules that contain references to this symbol. The error message and the list of object modules are written to the map file, if one exists.

This error occurs in a C-language program when a prototype for an externally defined function is omitted and the program is compiled with CL's /Gr option. The calling convention for __fastcall does not match the assumptions that are made when a prototype is not included for an external function.

Either include a prototype for the function, or compile without the /Gr option.

L2057 **duplicate of** *function* **with different size found; record ignored**

An inconsistent class definition was found.

Check the include files and recompile.

L2058 **different duplicate of** *function* **found; record ignored**

An inconsistent class definition was found.

Check the include files and recompile.

L2060 **size of data block associated with** *symbol* **(16-bit segment) exceeds 64K**

A class had too many virtual functions. The given symbol is the v-table for the class, in the form of a decorated name.

L2061 **no space for data block associated with** *function* **inside** *segment* **segment**

The given function was allocated to the given segment, but the segment was full.

L2062 **continuation of COMDAT** *function* **has conflicting attributes; record ignored**

This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L2063 *function* **is allocated in undefined segment**

The given function was allocated to a nonexistent segment.

L2064 **starting address not in the root overlay**

The segment or object file that contains the starting address for the program was placed into an overlay.

The starting address in a C-language program is provided by the main function.

L4000 **segment displacement included near** *offset* **in segment** *segment*

This is the warning generated by the /W option.

L4001 **frame-relative fixup, frame ignored near** *offset* **in segment** *segment*

A reference was made relative to a segment or group that is different from the target segment of the reference.

For example, if _id1 is defined in segment _TEXT, the instruction call DGROUP:_id1 produces this warning. The frame DGROUP is ignored, so LINK treats the call as if it were call _TEXT:_id1.

L4002 **frame-relative absolute fixup near** *offset* **in segment** *segment*

A reference was made relative to a segment or group that was different from the target segment of the reference, and both segments are absolute (defined with AT).

LINK assumed that the executable file will be run only under DOS.

L4004 **possible fixup overflow at** *offset* **in segment** *segment*

A near call or jump was made to another segment that was not a member of the same group as the segment from which the call or jump was made.

This can cause an incorrect real-mode address calculation when the distance between the paragraph address (frame number) of the segment group and the target segment is greater than 64K, even though the distance between the segment where the call or jump was actually made and the target segment is less than 64K.

L4010 **invalid alignment specification**

The number specified in the /ALIGN option must be a power of 2 in the range 2-32,768.

L4011 **/PACKC value exceeding 64K-36 unreliable**

The packing limit specified with the /PACKC option was in the range 65,501-65,536 bytes. Code segments with a size in this range are unreliable on some versions of the 80286 processor.

L4012 **/HIGH disables /EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4013 *option* **: option ignored for segmented executable file**

The given option is not allowed for segmented executable programs.

L4014 *option* **: option ignored for DOS executable file**

The given option is not allowed for DOS programs.

L4015 **/CO disables /DSALLOC**

The /CO and /DSALLOC options cannot be used at the same time.

L4016 **/CO disables /EXEPACK**

The /CO and /EXEPACK options cannot be used at the same time.

L4017 *option* **: unrecognized option name; option ignored**

The given option was not a valid LINK option. LINK ignored the option specification.

One of the following may be a cause:

- An obsolete option was specified to the current version of LINK. For example, the /INCR option is obsolete in LINK version 5.30. The current options are described in the manual and in online help. To see a list of options, run LINK with the /? option.
- An old version of LINK was used. Check your path. To see the version number of LINK, run LINK with the /? option.
- The name was incorrectly specified. For example, the option specification /NODEFAULTLIBSEARCH is an invalid abbreviation of the /NODEFAULTLIBRARYSEARCH option. Option names can be shortened by removing letters only from the end of the name.

L4018 **missing or unrecognized application type; option** *option* **ignored**

The /PM option accepts only the keywords PM, VIO, and NOVIO.

L4020 *segment* **: code-segment size exceeds 64K-36**

Code segments that are 65,501 through 65,536 bytes in length may be unreliable on some versions of the 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with the STACK combine type.

Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type. Linking with versions of LINK earlier than version 2.40 might cause this message since these linkers search libraries only once.

L4022 *group1, group2* **: groups overlap**

The given groups overlap. Since a group is assigned to a physical segment, groups cannot overlap in segmented executable files.

Reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *entry*(*internalname*) **: export internal name conflict**

The internal name of the given exported function or data item conflicted with the internal name of a previous import definition or export definition.

L4024 *name* **: multiple definitions for export name**

The given name was exported more than once, an action that is not allowed.

L4025 *modulename.entry*(*internalname*) **: import internal name conflict**

The internal name of the given imported function or data item conflicted with the internal name of a previous export or import. (The given entry is either a name or an ordinal number.)

L4026 *modulename.entry*(*internalname*) **: self-imported**

The given function or data item was imported from the module being linked. This error can occur if a module tries to import a function or data item from itself or from another source (such as a DLL) that has the same name.

L4027 *name* **: multiple definitions for import internal name**

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 *segment* **: segment already defined**

The given segment was defined more than once in a SEGMENTS statement of the module-definition (.DEF) file.

L4029 *segment* **: DGROUP segment converted to type DATA**

The given logical segment in the group DGROUP was defined as a code segment.

DGROUP cannot contain code segments because LINK always considers DGROUP to be a data segment. The name DGROUP is predefined as the automatic (or default) data segment.

LINK converted the named segment to type DATA.

L4030 *segment* **: segment attributes changed to conform with automatic data segment**

The given logical segment in the group DGROUP was given sharing attributes (SHARED/NONSHARED) that differed from the automatic data attributes as declared by the DATA instance specification (SINGLE/MULTIPLE). The attributes are converted to conform to those of DGROUP.

The name DGROUP is predefined as the automatic (or default) data segment. DGROUP cannot contain code segments because LINK always considers DGROUP to be a data segment.

L4031 *segment* **: segment declared in more than one group**

A segment was declared to be a member of two different groups.

L4032 *segment* **: code-group size exceeds 64K-36**

The given code group has a size in the range 65,501-65,536 bytes, a size that is unreliable on some versions of the 80286 processor.

L4033 **first segment in mixed group** *group* **is a USE32 segment**

A 16-bit segment must be first in a group created with both USE16 and USE32 segments.

LINK continued to build the executable file, but the resulting file may not run correctly.

L4034 **more than 1024 overlay segments; extra put in root**

The limit on the number of segments that can go into overlays is 1024. Segments starting with the 1025th segment are assigned to the permanently resident portion of the program (the root).

L4036 **no automatic data segment**

The application did not define a group named DGROUP.

DGROUP has special meaning to LINK, which uses it to identify the automatic (or default) data segment used by the operating system. Most segmented executable applications require DGROUP.

This warning is not issued if DATA NONE is declared or if the executable file is a dynamic-link library.

L4037 *group* **: both USE16 and USE32 segments in group; assuming USE32**

The given group was allocated contributions from both 16-bit segments and 32-bit segments.

L4038 **program has no starting address**

The segmented executable application had no starting address. A missing starting address will usually cause the program to fail.

High-level languages automatically specify a starting address. In a C-language program, this is provided by the main function.

If you are writing an assembly-language program, specify a starting address with the END statement.

DOS programs and dynamic-link libraries should never receive this message, regardless of whether they have starting addresses.

L4040 **stack size ignored for /TINY**

LINK ignores stack size if the /TINY option is used and if the stack segment has been defined in front of the code segment.

L4042 **cannot open old version**

The file specified in the OLD statement in the module-definition (.DEF) file could not be opened.

L4043 **old version not segmented executable format**

The file specified in the OLD statement in the module-definition (.DEF) file was not a valid segmented executable file.

L4045 **name of output file is** *filename*

LINK used the given filename for the output file.

If the output filename is specified without an extension, LINK assumes the default extension .EXE. Creating a Quick library, DLL, or .COM file forces LINK to use a different extension. In the following cases, if either .EXE or no extension is specified, LINK assumes the appropriate extension:

/TINY option .COM /Q option .QLB LIBRARY statement .DLL

This warning also occurs if the name specified in the LIBRARY statement in the module-definition (.DEF) file does not match the name specified in the <exefile> field.

L4050 **file not suitable for /EXEPACK; relink without**

The size of the packed load image plus packing overhead was larger than it would be for the unpacked load image. There is no advantage to packing this program.

Remove /EXEPACK from the LINK command line. In PWB, turn off the Pack EXE File check box in the Additional Debug/Release Options dialog box under Link Options.

L4051 *filename* **: cannot find library**

LINK could not find the given library file.

One of the following may be a cause:

- The specified file does not exist. Enter the name or full path specification of a library file.
- The LIB environment variable is not set correctly. Check for incorrect directory specifications, mistyping, or a space, semicolon, or hidden character at the end of the line.
- An earlier version of LINK is being run. Check the path environment variable and delete or rename earlier linkers.

L4053 **VM.TMP : illegal filename; ignored**

VM.TMP appeared as an object-file name.

Rename the file and rerun LINK.

L4054 *filename* **: cannot find file**

LINK could not find the specified file.

Enter a new file name, a new path specification, or both.

L4055 **start address not equal to 0x100 for /TINY**

The starting address for a .COM file must be 100 hexadecimal.

Put the following line of assembly source code in front of the code segment:

ORG 100h

L4056 **/EXEPACK valid only for OS/2 and real-mode DOS; ignored**

The /EXEPACK option is incompatible with Windows programs.

L4057 **stack specified for DLL; ignored**

A stack was specified for a dynamic-link library (DLL). Either the /STACK option was used on the command line or the STACKSIZE statement was used in the module-definition (.DEF) file. LINK ignored the specification and did not create a stack.

A DLL does not have a stack.

L4058 **ignoring alias for already defined symbol** *symbol*

The specified symbol was redefined in the program. However, it is an identifier from a C run-time library that has an alias to a new name in OLDNAMES.LIB. LINK ignored the alias for the symbol.

This warning appears only when the /INFO option is specified.

L4067 **changing default resolution for weak external** *symbol* **from** *oldresolution* **to** *newresolution*

LINK found conflicting default resolutions for a weak external. It ignored the first resolution and used the second.

L4068 **ignoring stack size greater than 64K**

A stack was defined with an invalid size. LINK assumed 64K.

L4069 **filename truncated to** *filename*

A file-name specification exceeded the length allowed. LINK assumed the given file name.

L4070 **too many public symbols for sorting**

LINK uses the stack and all available memory in the near heap to sort public symbols for the /MAP option. This warning is issued if the number of public symbols exceeds the space available for them. In addition, the symbols are not sorted in the map file but are listed in an arbitrary order.

L4076 **no segments defined**

There was no code in the program.

This warning can occur if the file contains only resources.

L4077 **symbol** *function* **not defined; ordered allocation ignored**

The given function was specified in a FUNCTIONS statement in the module-definition (.DEF) file, but the function was not defined.

L4079 **symbol** *function* **already defined for ordered allocation; duplicate ignored**

The given function was specified twice in FUNCTIONS statements in the module-definition (.DEF) file.

L4080 **changing substitute name for alias** *symbol* **from** *oldalias* **to** *newalias*

LINK found conflicting alias names. It ignored the first alias and used the second.

L4081 **cannot execute** *program arguments* **-** *message*

LINK could not run the given program (with the given arguments) for the given reason.

L4082 **changing overlay assigment for segment** *segment* **from** *oldnumber* **to** *newnumber*

The given segment was assigned to two overlays, represented by <*oldnumber*> and <*newnumber*>. LINK assumed the <*newnumber*> overlay.

Probably a command-line overlay specification with parentheses conflicted with an overlay specification in the module-definition (.DEF) file.

L4083 **changing overlay assigment for symbol** *symbol* **from** *oldnumber* **to** *newnumber*

The given symbol was assigned to two overlays, represented by <*oldnumber*> and <*newnumber*>. LINK assumed the <*newnumber*> overlay.

Probably a command-line overlay specification with parentheses conflicted with an overlay specification in the module-definition (.DEF) file.

L4084 *option* **: argument missing; option ignored**

The given option requires an argument, but none was specified.

For example, the following option specification causes this error:

```
/ONERROR
```

L4085 *option* **: argument invalid; assuming** *argument*

The given option was specified with a numeric argument that was out of range for the option. LINK assumed the given argument.

For example, the option specification /DYNAMIC:11000 causes the following error:

/DYNAMIC:11000 : argument invalid; assuming 10922

L4086 **/r not first on command line; ignored**

The /r option was specified, but it was not the first argument after LINK on the command line.

The /r option prevents LINK from using extended memory under DOS and forces LINK to use only conventional memory. This option must appear first on the command line. It cannot appear in a response file or the LINK environment variable.

R6907 **DOSX16: not enough memory on exec**

There was not enough EMS or XMS memory to run LINK. Either reconfigure your system to make more EMS or XMS available, or use the /r option when linking. This error only occurs while linking from the command line.