

Classic PowerBASIC 9 For Windows

KeyWord Quick Finder



[Introducing Classic PowerBASIC For
Windows](#)

[New Statements and Functions](#)

[Command Summary](#)

[Running PB/Win](#)

[Data Types](#)

[Built-in numeric equates](#)

[Built-in string equates](#)

[Glossary](#)

[Register](#)

[Technical Support](#)

[Downloads](#)

[Peer Support Forums](#)

<http://www.PowerBASIC.com>

Telephone: +1 (941) 473-
7300

Copyright © 1996 - 2012 PowerBASIC,
Inc.

Introducing Classic PowerBASIC 9 for Windows

[Top](#) [Previous](#)
[Next](#)

Classic PowerBASIC for Windows is a native code compiler for Win95/98/ME, WinNT, Windows 2000, Windows XP, Windows Vista, and Windows 7. It creates applications with a Graphical User Interface (GUI), to provide the typical "Look and Feel" of Windows. It creates highly efficient executables and industry-standard DLLs for optimum flexibility. The machine code generated by Classic PowerBASIC is among the most efficient in the industry, both in terms of size and speed. It compares most favorably with leading compilers of any dialect, Pascal, C++, Fortran, and others.

Our favorite slogan is "We put the Power in BASIC", and we sincerely believe you will find this to be true. With compilation speeds of 1 million lines per minute, unrivaled performance, and the smallest executables in the industry, Classic PowerBASIC has become the new standard of comparison in Windows programming. **Thank you for joining us in the War on Bloatware!**

Features

- Create client [COM](#) applications and [COM components](#) using [Dispatch](#), [Direct](#), [Automation](#), or [Dual interfaces](#).
- Fast and Small 32-bit EXEs and [DLLs](#) for Microsoft Windows 95/98/ME/NT/2000/XP/Vista/Windows 7
- 32-bit [protected mode](#) code generation for maximum performance
- [Dynamic Dialog Tools](#) for easy creation of Graphic User Interface applications
- A complete [graphics package](#) for easy development of graphic presentations, splash screens and more
- Support for [Windows only printers](#) with the [XPRINT](#) statement and functions
- Supports existing [Line Printers](#), with Classic PowerBASIC [LPRINT](#) statements and functions
- A complete set of advanced string manipulation functions: [VERIFY](#), [REMOVE](#), [REPLACE](#), [EXTRACT](#), [TALLY](#), [REPEAT](#), [PARSE](#), and [many more](#)
- [REGEXPR](#) and [REGREPL](#) functions for regular expression search and replace
- Array [Sort](#) and [Scan](#), element [Insert](#) and [Delete](#)
- [MIN](#) and [MAX](#) value Functions that work with both numeric and string data types
- [PEEK](#), [POKE](#), [PEEK\\$](#), [POKE\\$](#) for direct memory access
- Pointers and Indexed Pointers for direct memory access
- [Matrix](#) operations: Init, Identity, Transposition, Inversion, scalar, and matrix math
- 80-bit Extended-precision math
- [Register Variables](#) for increased performance: up to six unique register variables:

integer-class (2) or floating-point (4)

- Unsigned Integer types: [BYTE](#) (8-bit), [WORD](#) (16-bit) and [DWORD](#) (32-bit)
- Signed Integer types: [INTEGER](#) (16-bit), [LONG](#) (32-bit) and [QUAD](#) (64-bit)
- Two [Currency](#) variable types
- Native [ASCIIZ](#) (Nul-terminated) strings
- User-Defined [TYPES](#) and [UNIONS](#)
- [FIELD](#) variables for file I/O.
- [Variant](#), [GUID](#), and [Object](#) variables
- Optional parameters in BASIC [Subs](#), [Functions](#), [Methods](#), and [Properties](#)
- Optional parameter passing to CDECL and SDECL procedures.
- Optional requirement that variables be declared before use
- Built-in 32-bit [Inline Assembler](#) with 80486, Pentium, and SIMD opcodes
- Inline Assembler includes Floating-Point and MMX instructions
- Direct export of Subs and Functions
- Import Subs and Functions from the entire Win32 or any 32-bit DLL
- Multi-thread application support: [Create](#), [Suspend](#), [Resume](#), [Status](#), and [Close](#)
- Client/Server [Network Communications](#) - TCP/UDP for E-mail, FTP, etc.
- High-speed [Serial Communications](#) support
- True 32-bit code pointers, great for callbacks
- Easy to use syntax highlighting [Integrated Development Environment](#) (IDE) and [debugger](#)

See Also

[The Integrated Development Environment](#)

[Running PB/Win](#)

[Debugging PB/Win Programs](#)

- [#ALIGN](#) metastatement aligns the next instruction to a boundary.
- [#COM DOC](#) metastatement specifies a help string to be included in a COM Type Library.
- [#COM HELP](#) metastatement specifies the name of the associated help file and the help context code to be included in a COM Type Library.
- [#COM NAME](#) metastatement specifies the name of the server and the version number to be included in a COM Type Library.
- [#COM GUID](#) metastatement specifies the [GUID](#) which identifies the entire application or library (APPID or LIBID) to be included in a COM Type Library.
- [#COM TLIB ON](#) metastatement specifies that the compiler should create a [type library](#) for the compiled EXE or DLL.
- [#COM TLIB OFF](#) metastatement specifies that the compiler should not create a type library for the compiled EXE or DLL.
- [#DEBUG CODE ON](#) metastatement activates generation of [debug](#) code.
- [#DEBUG CODE OFF](#) metastatement suppresses generation of debug code, from that line, until a subsequent [#DEBUG CODE ON](#) (or the end of the [Sub/Function/Method/Property](#)) is reached.
- [#DEBUG DISPLAY ON](#) metastatement enables error display mode when an untrapped [run-time error](#) occurs within a compiled Classic PowerBASIC program.
- [#DEBUG DISPLAY OFF](#) metastatement suppresses displaying of untrapped error messages.
- [#INCLUDE ONCE](#) metastatement includes a file only one time during compilation, regardless of how many times it appears in the program
- [#MESSAGES COMMAND](#) metastatement specifies that only %WM_COMMAND messages are to be sent to a Controls [Callback](#) Function, just as in earlier versions of Classic PowerBASIC.
- [#MESSAGES NOTIFY](#) specifies that %WM_NOTIFY messages (as well as %WM_COMMAND messages) are to be sent to a Controls Callback Function.
- [#OPTIMIZE](#) metastatement chooses between faster execution or smaller code size.
- [#UTILITY](#) metastatement. Compiler directive to allow external utility programs to read text inserted on the [#UTILITY](#) line.
- [BITSE](#) function compares integer-class values for equivalent bits regardless of sign.
- [BUILD\\$](#) function concatenates multiple [strings](#) with high efficiency.
- [CB.CTL](#) function returns the control id for the message being processed in a Callback function.

- [CB.CTLMSG](#) function returns the specific control message describing the event which occurred.
- [CB.HNDL](#) function returns the window handle of the parent dialog in a Callback function.
- [CB.LPARAM](#) function returns the wParam value for the message being processed in a Callback function.
- [CB.MSG](#) function returns the numeric message value of the message being processed in a Callback function.
- [CB.WPARAM](#) function returns the wParam value for the message being processed in a Callback function.
- [CB.NMCODE](#) function returns the specific notification message describing the event which occurred.
- [CB.NMHDR](#) function returns the address (a [pointer](#)) to the NMHDR [UDT](#) for a notification message.
- [CB.NMHDR\\$](#) function returns the contents of the NMHDR UDT as a [dynamic string](#).
- [CB.NMHWND](#) function returns the handle of the control which sent this message.
- [CB.NMID](#) function returns the control id for the message being processed in a Callback function.
- [CLASS / END CLASS](#) block creates the code and data for an object.
- [CLIPBOARD GET BITMAP](#) statement copies a bitmap from the ClipBoard and stores it in this newly created [GRAPHIC BITMAP](#).
- [CLIPBOARD GET OEMTEXT](#) statement copies a text string from the ClipBoard and converts it to OEM format if necessary.
- [CLIPBOARD GET TEXT](#) statement copies a text string from the ClipBoard and converts it to ASCII format if necessary.
- [CLIPBOARD GET UNICODE](#) statement copies a text string from the ClipBoard and converts it to [Unicode](#) format if necessary.
- [CLIPBOARD RESET](#) statement deletes the contents of the ClipBoard.
- [CLIPBOARD SET BITMAP](#) statement copies a GRAPHIC BITMAP, [GRAPHIC CONTROL](#), or [GRAPHIC WINDOW](#) to the ClipBoard.
- [CLIPBOARD SET OEMTEXT](#) statement copies a OEMTEXT formatted string to the ClipBoard.
- [CLIPBOARD SET TEXT](#) statement copies a ASCII formatted string to the ClipBoard.
- [CLIPBOARD SET UNICODE](#) statement copies a UNICODE formatted string to the ClipBoard.
- [COMBOBOX FIND](#) statement. Strings in the [ComboBox](#) are searched to find the first string which begins with the specified characters.

- [COMBOBOX FIND EXACT](#) statement. Strings in the ComboBox are searched to find the first string which exactly matches the specified characters.
- [COMBOBOX GET COUNT](#) statement returns the number of items in the ListBox of a ComboBox.
- [COMBOBOX GET SELCOUNT](#) statement returns the number of selected items in the ListBox of a ComboBox.
- [COMBOBOX GET STATE](#) statement checks to see if a specific item is selected.
- [COMBOBOX GET USER](#) statement retrieves the value in the user data area of the ComboBox.
- [COMBOBOX INSERT](#) statement inserts a new data item at a specific location.
- [COMBOBOX SET TEXT](#) statement replaces a string for a specific data item with a new string.
- [COMBOBOX SET USER](#) statement sets a value in the user data area of the ComboBox.
- [COMBOBOX UNSELECT](#) statement sets all items in a ComboBox control to an unselected state.
- [CONTROL ADD LISTVIEW](#) statement adds a ListView control to a [dialog](#).
- [CONTROL ADD PROGRESSBAR](#) statement adds a ProgressBar control to a dialog.
- [CONTROL ADD STATUSBAR](#) statement adds a StatusBar control to a dialog.
- [CONTROL ADD TAB](#) statement adds a Tab Control to a dialog.
- [CONTROL ADD TOOLBAR](#) statement adds a ToolBar control to a dialog.
- [CONTROL ADD TREEVIEW](#) statement adds a TreeView control to a dialog.
- [CONTROL SET FONT](#) statement. Select a font to be used for a particular Windows Control.
- [DISPLAY BROWSE](#) statement displays a folder selection dialog to return the user's choice.
- [DISPLAY COLOR](#) statement displays a color selection dialog to return the user's choice.
- [DISPLAY FONT](#) statement displays a font selection dialog to return user choices.
- [DISPLAY OPENFILE](#) statement displays an OpenFile selection dialog to return user choices.
- [DISPLAY SAVEFILE](#) statement displays a SaveFile selection dialog to return user choices.
- [ERL\\$](#) function returns the last [label](#), [line number](#), or procedure name executed prior to the most recent error.
- [EVENT SOURCE](#) statement declares an event interface within a Class definition.

- [EVENTS](#) statement attaches or detaches an event handler to/from an event source.
- [EXE.EXTN\\$](#) read-only [user defined type](#) returns the extension (with a leading period) of the program which is currently executing.
- [EXE.FULL\\$](#) read-only user defined type returns the complete drive, path, and file name of the program which is currently executing.
- [EXE.NAME\\$](#) read-only user defined returns just the file name of the program which is currently executing.
- [EXE.NAMEX\\$](#) read-only user defined returns the file name and the extension of the program which is currently executing.
- [EXE.PATH\\$](#) read-only user defined returns the complete drive and path of the program which is currently executing.
- [EXIT METHOD](#) transfers program execution out of a [METHOD](#) structure.
- [EXIT PROPERTY](#) transfers program execution out of a [PROPERTY](#) structure.
- [FONT END](#) statement destroys a font when it is no longer needed.
- [FONT NEW](#) statement creates a new font for use with [GRAPHIC PRINT](#), [XPRINT](#), [CONTROLS](#), etc.
- [GLOBALMEM ALLOC](#) statement allocates a moveable memory block.
- [GLOBALMEM FREE](#) statement de-allocates a memory block.
- [GLOBALMEM LOCK](#) statement lock a moveable memory block at a specific memory location.
- [GLOBALMEM SIZE](#) statement returns the size of memory block.
- [GLOBALMEM UNLOCK](#) statement unlocks a moveable memory block.
- [GRAPHIC DETACH](#) statement detaches a [graphic target](#) ([Window](#), [Control](#), or [bitmap](#)) which may be currently attached to the process.
- [GRAPHIC GET LINES](#) statement retrieves the number of lines that can be [printed](#) on the graphic target.
- [GRAPHIC GET SCALE](#) statement retrieves the current [coordinate limits](#) for the graphic target.
- [GRAPHIC IMAGELIST](#) statement display an image from an [Imagelist](#).
- [GRAPHIC INKEY\\$](#) statement reads a keyboard character if one is ready.
- [GRAPHIC INPUT](#) statement reads data from the keyboard from within a [Graphic Window](#).
- [GRAPHIC INPUT FLUSH](#) statement removes all buffered keyboard data.
- [GRAPHIC INSTAT](#) statement determines whether a keyboard character is ready.
- [GRAPHIC LINE INPUT](#) statement reads an entire line from the keyboard from within a [Graphic Window](#) or a [Graphic Control](#).

- [GRAPHIC SCALE PIXELS](#) statement sets or resets the graphic coordinate system to pixel coordinates.
- [GRAPHIC SET FONT](#) statement selects a font for the [GRAPHIC PRINT](#), [GRAPHIC INPUT](#), and [GRAPHIC LINE INPUT](#) statements.
- [GRAPHIC WAITKEY\\$](#) statement reads a keyboard character, waiting until one is ready.
- [GRAPHIC WINDOW CLICK](#) statement checks whether a GRAPHIC WINDOW has been clicked with the mouse.
- [IDISPINFO.CODE](#) pseudo-object. When [OBJRESULT](#) is %DISP_E_EXCEPTION, this Get Property returns a long integer value which represents a more specific error code.
- [IDISPINFO.CONTEXT](#) pseudo-object. When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a long integer value which is the context of the topic within the help file (IDISPINFO.HELP\$).
- [IDISPINFO.DESC\\$](#) pseudo-object. When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing a textual, human-readable description of the status.
- [IDISPINFO.HELP\\$](#) pseudo-object. When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing drive, path, and filename of a Help File with more information about this particular status code.
- [IDISPINFO.PARAM](#) pseudo-object. When OBJRESULT is either %DISP_E_PARAMNOTFOUND or %DISP_E_TYPEREMISMATCH, this Get Property returns a long integer value which represents the parameter number of the first parameter which failed to match the requirements needed.
- [IDISPINFO.SOURCE\\$](#) pseudo-object. When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing a textual, human-readable description of the source of the exception.
- [IDISPINFO.CLEAR](#) pseudo-object. Clears all properties which may have been set by prior execution of IDISPINFO.SET in this thread.
- [IDISPINFO.SET](#) pseudo-object. Sets the properties which for future execution of IDISPINFO.
- [IMAGELIST ADD BITMAP](#) statement adds a bitmap to an ImageList.
- [IMAGELIST ADD BITMAP](#) statement adds a resource or disk file bitmap to an ImageList.
- [IMAGELIST ADD ICON](#) statement adds a memory icon to an ImageList.
- [IMAGELIST ADD ICON](#) statement adds a resource or disk file icon to an ImageList.
- [IMAGELIST ADD MASKED](#) statement adds a masked memory bitmap to an ImageList.

- [IMAGELIST ADD MASKED](#) statement adds a masked resource or disk file bitmap to an ImageList.
- [IMAGELIST GET COUNT](#) statement retrieves the number of images in an ImageList.
- [IMAGELIST KILL](#) statement destroys a specified ImageList.
- [IMAGELIST NEW BITMAP](#) statement creates a new bitmap ImageList.
- [IMAGELIST NEW ICON](#) statement creates a new icon ImageList.
- [IMAGELIST SET OVERLAY](#) statement declares an image in an ImageList as an overlay image.
- [INSTANCE](#) statement declares an INSTANCE variable which is unique to each object.
- [INTERFACE / END INTERFACE](#) Block (Direct) declares a direct object interface and its member [Methods/Properties](#).
- [ISFILE](#) function determines whether or not a file exists.
- [ISFOLDER](#) function determines whether or not a folder exists.
- [ISINTERFACE](#) function determines whether an object supports a particular interface.
- [ISMISSING](#) function determines whether an optional parameter was passed by the calling code.
- [ISWIN](#) function determines whether a Control/Dialog/Window currently exists.
- [LISTBOX FIND](#) statement. Strings in the [Listbox](#) are searched to find the first string which begins with the specified characters.
- [LISTBOX FIND EXACT](#) statement. Strings in the Listbox are searched to find the first string which exactly matches the specified characters.
- [LISTBOX GET COUNT](#) statement returns the number of items in the Listbox.
- [LISTBOX GET SELCOUNT](#) statement returns the number of selected items in the Listbox.
- [LISTBOX GET SELECT](#) statement searches the Listbox and returns the first selected item.
- [LISTBOX GET STATE](#) statement checks an item in the Listbox to see if it is currently selected.
- [LISTBOX GET USER](#) statement retrieves the value in the user data area of the Listbox.
- [LISTBOX INSERT](#) statement inserts a new data item at a specified location in the Listbox.
- [LISTBOX SET TEXT](#) statement replaces the string for a specific data item with a new string.
- [LISTBOX SET USER](#) statement sets a value in the user data area of the Listbox.
- [LISTBOX UNSELECT](#) statement sets a specific data item in the Listbox control to an

unselected state.

- [LISTVIEW DELETE COLUMN](#) statement deletes a column from a [ListView](#) control.
- [LISTVIEW DELETE ITEM](#) statement deletes a data item from a ListView control.
- [LISTVIEW FIND](#) statement. Strings in the ListView are searched to find the first string which begins with the specified characters.
- [LISTVIEW FIND EXACT](#) statement. Strings in the ListView are searched to find the first string which exactly matches the specified characters
- [LISTVIEW FIT CONTENT](#) statement adjusts the width of the specified column to fit the width of the data items displayed in that column.
- [LISTVIEW FIT HEADER](#) statement adjusts the width of the specified column to fit the width of the data items displayed in that column, and the header text at the top of that column.
- [LISTVIEW GET COLUMN](#) statement returns the width of the specified column.
- [LISTVIEW GET COUNT](#) statement returns the number of data items in the ListView control.
- [LISTVIEW GET HEADER](#) statement returns the specified column header text.
- [LISTVIEW GET MODE](#) statement retrieves the display mode of the specified ListView control.
- [LISTVIEW GET SELCOUNT](#) statement returns the number of selected items in the ListView control.
- [LISTVIEW GET SELECT](#) statement returns the next primary data item which is currently selected.
- [LISTVIEW GET STATE](#) statement tests a data item to see if it is currently selected.
- [LISTVIEW GET STYLEXX](#) statement retrieves the current setting of the ListView controls extended style.
- [LISTVIEW GET TEXT](#) statement returns a string data item from the ListView control.
- [LISTVIEW GET USER](#) statement retrieves the value in the user data area of the ListView control.
- [LISTVIEW INSERT COLUMN](#) statement inserts a new vertical column in a Report Mode ListView control.
- [LISTVIEW INSERT ITEM](#) statement inserts a new data item in the ListView control.
- [LISTVIEW RESET](#) statement deletes all data items from the specified ListView control.
- [LISTVIEW SELECT](#) statement the selects a specified string data item in the ListView control.
- [LISTVIEW SET COLUMN](#) statement changes the width of a specific ListView column.
- [LISTVIEW SET HEADER](#) statement sets the column header text to be displayed

above the specified column on the ListView control.

- [LISTVIEW SET IMAGE](#) statement displays the specified image next to the item specified.
- [LISTVIEW SET IMAGE2](#) statement displays a secondary "status" image next to the primary image.
- [LISTVIEW SET IMAGELIST](#) statement attaches an Imagelist to the ListView control.
- [LISTVIEW SET MODE](#) statement changes the display mode of the specified ListView control.
- [LISTVIEW SET OVERLAY](#) statement displays the specified overlay image on top of the image specified.
- [LISTVIEW SET STYLEXX](#) statement alters the current settings of the ListView controls extended style.
- [LISTVIEW SET TEXT](#) statement replaces the text, if any, for the specified data item with new text.
- [LISTVIEW SET USER](#) statement sets a value in the user data area of the ListView control.
- [LISTVIEW SORT](#) statement sorts all the items in a ListView control.
- [LISTVIEW UNSELECT](#) statement unselects a specified data item in a ListView control.
- [LISTVIEW VISIBLE](#) statement. The specified data item is scrolled, if necessary, to ensure that the data item is visible.
- [ME](#) pseudo-variable. A pseudo object variable to reference the current object.
- [METHOD / END METHOD](#) statements defines a Method procedure within a [class](#).
- [MYBASE](#) pseudo-variable. A pseudo object variable to reference the inherited parent object.
- [OBJRESULT\\$](#) function returns a string which describes an [OBJRESULT](#) (hResult) code.
- [PATHNAME\\$](#) function parses a path/file name to extract its component parts.
- [PATHSCAN\\$](#) function finds a file on disk and returns the path and/or file name parts.
- [PROCESS GET PRIORITY](#) retrieves the Priority Value for the current process.
- [PROCESS SET PRIORITY](#) sets the Priority Value for the current process.
- [PROGRESSBAR GET POS](#) statement returns the current position of the [ProgressBar](#).
- [PROGRESSBAR GET RANGE](#) statement returns the current range of the [ProgressBar](#).
- [PROGRESSBAR SET POS](#) statement sets the current position of the [ProgressBar](#).

- [PROGRESSBAR SET RANGE](#) statement sets the minimum and maximum ranges of the ProgressBar.
- [PROGRESSBAR SET STEP](#) statement specifies the default increment value to be used by the PROGRESSBAR STEP statement.
- [PROGRESSBAR STEP](#) statement advances the current position of the ProgressBar by the default increment value.
- [PROPERTY GET](#) statement retrieves a data value from an object.
- [PROPERTY SET](#) statement assigns a data value to an object.
- [RAISEEVENT](#) statement calls an Event Handler code.
- [SCROLLBAR GET PAGESIZE](#) statement retrieves the current page size.
- [SCROLLBAR GET POS](#) statement returns the current position of the [ScrollBar](#).
- [SCROLLBAR GET RANGE](#) statement Returns the current range of the ScrollBar.
- [SCROLLBAR GET TRACKPOS](#) statement retrieves the current position of the scroll box.
- [SCROLLBAR SET PAGESIZE](#) statement sets the current page size.
- [SCROLLBAR SET POS](#) statement sets the current position of the ScrollBar.
- [SCROLLBAR SET RANGE](#) statement sets the range of the ScrollBar.
- [STATUSBAR SET PARTS](#) statement sets the number of parts to be displayed in the [StatusBar](#).
- [STATUSBAR SET TEXT](#) statement assigns the text to be displayed in the specified part of the StatusBar.
- [TAB DELETE](#) statement deletes a page from the [Tab](#) control.
- [TAB GET COUNT](#) statement returns the number of pages in a Tab control.
- [TAB GET DIALOG](#) statement retrieves the handle of the dialog for a specific page in a Tab control.
- [TAB GET SELECT](#) statement returns the currently selected page in a Tab control.
- [TAB INSERT PAGE](#) statement adds a new page to a Tab control.
- [TAB RESET](#) statement deletes all pages in a Tab control.
- [TAB SELECT](#) statement selects a specific page in a Tab control to be the active page.
- [TAB SET IMAGELIST](#) statement assigns an ImageList to be used in a Tab control.
- [THREAD GET PRIORITY](#) retrieves the Priority Value for a thread.
- [THREAD SET PRIORITY](#) sets the Priority Value for a thread
- [TIX](#) statement measures elapsed CPU cycles.
- [TOOLBAR ADD BUTTON](#) statement adds a button to a [Toolbar](#) control.

- [TOOLBAR ADD SEPARATOR](#) statement adds a separator to a Toolbar control.
- [TOOLBAR DELETE BUTTON](#) statement deletes a button from a Toolbar control.
- [TOOLBAR GET STATE](#) statement retrieves the state of a button on a Toolbar control.
- [TOOLBAR GET COUNT](#) statement retrieves the number of buttons on a Toolbar control.
- [TOOLBAR SET IMAGELIST](#) statement attaches an ImageList to a Toolbar control.
- [TOOLBAR SET STATE](#) statement sets the state of a button on a Toolbar control.
- [TREEVIEW DELETE](#) statement deletes a data item from a [TreeView](#) control.
- [TREEVIEW GET BOLD](#) statement retrieves the bold attribute for a data item.
- [TREEVIEW GET CHECK](#) statement retrieves the checkmark attribute for a data item.
- [TREEVIEW GET CHILD](#) statement returns the handle of the first child item of a specified data item.
- [TREEVIEW GET COUNT](#) statement retrieves the number of data items in the TreeView.
- [TREEVIEW GET EXPANDED](#) statement retrieves the expanded attribute for the data item.
- [TREEVIEW GET NEXT](#) statement returns the handle of the next sibling data item.
- [TREEVIEW GET PARENT](#) statement returns the handle of the parent for a specified data item is returned.
- [TREEVIEW GET PREVIOUS](#) statement returns the handle of the previous sibling data item.
- [TREEVIEW GET ROOT](#) statement returns the handle of the very first data item (topmost) in the TreeView.
- [TREEVIEW GET SELECT](#) statement retrieves the handle of the currently selected data item.
- [TREEVIEW GET TEXT](#) statement retrieves the text of a specific data item .
- [TREEVIEW GET USER](#) statement retrieves the value in the user data area for a specific data item of the TreeView.
- [TREEVIEW INSERT ITEM](#) statement adds a new data item to a TreeView control.
- [TREEVIEW RESET](#) statement deletes all data items from the specified TreeView control.
- [TREEVIEW SELECT](#) statement selects a specific data item in the TreeView control.
- [TREEVIEW SET BOLD](#) statement sets the bold attribute for specific data item.
- [TREEVIEW SET CHECK](#) statement sets the checkmark attribute for a specific data item.
- [TREEVIEW SET EXPANDED](#) statement sets the expanded attribute for a specific

data item.

- [TREEVIEW SET IMAGELIST](#) statement attaches an ImageList to a TreeView control.
- [TREEVIEW SET TEXT](#) statement replaces the text, if any, for the specified data item with new text.
- [TREEVIEW SET USER](#) statement sets the value in the user data area for a specific data item in the TreeView control.
- [TREEVIEW UNSELECT](#) statement sets all items in a TreeView control to an unselected state.
- [UCODEPAGE](#) statement sets the default codepage used for ANSI / UNICODE conversions
- [WINDOW GET ID](#) statement returns the integer ID for a Window (usually a Control).
- [WINDOW GET PARENT](#) statement retrieves the handle of the parent.
- [XPRINT GET COLLATE](#) statement retrieves the XPRINT collate status.
- [XPRINT GET COLORMODE](#) statement retrieves the XPRINT colormode status.
- [XPRINT GET COPIES](#) statement retrieves the XPRINT copy count.
- [XPRINT GET DUPLEX](#) statement retrieves the XPRINT duplex status.
- [XPRINT GET PAPER](#) statement retrieves the current paper size/type.
- [XPRINT GET PAPERS](#) statement retrieves a list of supported paper types.
- [XPRINT GET SCALE](#) statement retrieves the current coordinate limits for the host printer page.
- [XPRINT GET TRAY](#) statement retrieves the active printer tray.
- [XPRINT GET TRAYS](#) statement retrieves a list of supported paper trays.
- [XPRINT IMAGELIST](#) statement prints an image from an ImageList.
- [XPRINT SCALE PIXELS](#) resets the coordinate system to the original default pixel coordinates.
- [XPRINT SET COLLATE](#) statement changes the XPRINT collate status.
- [XPRINT SET COLORMODE](#) statement changes the XPRINT colormode status.
- [XPRINT SET COPIES](#) statement changes the XPRINT copy count.
- [XPRINT SET DUPLEX](#) statement changes the XPRINT duplex status.
- [XPRINT SET FONT](#) statement selects a font for the XPRINT statement.
- [XPRINT SET PAPER](#) statement sets a new paper size/type.
- [XPRINT SET TRAY](#) statement sets a new active printer tray.

See Also

[Changes to existing Statements and Functions](#)

[New in the IDE](#)

[Additional Changes](#)

Changes to existing Statements and Functions

[Top](#) [Previous](#)
[Next](#)

- [#INCLUDE](#) metastatement has been enhanced with an ONCE option. If the optional keyword ONCE is included, the specified file is included only one time during compilation, regardless of how many times it appears in the program. This is particularly useful when including common declaration files like [WIN32API.INC](#) to avoid redundant code, and the resulting errors. #INCLUDE metastatements can now be nested as many as twelve levels deep.
- [%DEF\(%PB_EXE\)](#) now returns the correct value in all cases.
- [ACODE\\$](#) function may now contain an optional code page parameter. The code page parameter represents the code page to be used for the conversion process.
- [ARRAY SORT](#) statement now offers a custom [array](#) sorting option. A custom array may be [user-defined types](#), [fixed-length strings](#), or [ASCIIZ strings](#). With a custom array sort, you can write your own simple function to tell Classic PowerBASIC the correct sequence for any two array elements.
- [ASM](#) statement has been expanded to support the full range of SIMD opcodes. ASM statements may now contain a [label](#) - ASM Label: or ! Label:. Support for returning METHOD and PROPERTY return value assignments have been added.
- [BGR](#) function now accepts individual red, green, and blue values or a single [RGB](#) value.
- [CLIPBOARD](#) syntax and documentation are substantially enhanced.
- [COMM](#) function has been enhanced to retrieve the Clear-To-Send (CTS) and Data-Set-Ready (DSR) states.
- [COMMAND\\$](#) function has been improved with an option to either return the complete trailer, or any one of the arguments.
- [CSNG](#) and [CVS](#) now limit string display to 7 significant digits.
- [DECLARE](#) statements now support declarations of [THREAD FUNCTIONS](#).
- [DIALOG FONT](#) statement may now be used to specify the style and character set used for the [dialog](#) font.
- [DIR\\$](#) function has been expanded with an optional ONLY keyword to return only files that match the specified attribute. For example: DIR\$(mask\$, ONLY %SUBDIR) just the directory entries which match mask\$ are returned. The DIR\$ function may optionally assign the complete directory data structure that receives information about the found file or subdirectory. to an appropriate [UDT](#) variable if you include the TO clause as a parameter.
- [EOF](#) function now supports an optional # symbol preceding the file number parameter.

- [EXIT](#) statement has been expanded to add EXIT METHOD and EXIT PROPERTY options.
- [EXTRACT\\$](#), [INSTR](#), [PARSE\\$](#), [REMAINS\\$](#), [REMOVES\\$](#), [RETAINS\\$](#), and [TALLY](#) have been dramatically improved in performance in most situations.
- [FIELD](#) statement has been updated with two new options. FIELD STRING converts a [field string](#) to a [dynamic string](#), assigns the current sub-section data to it. FIELD RESET converts a field string to a nul (zero-length) dynamic string.
- [FOR/NEXT](#) statements have been optimized. In certain situations, previous versions of Classic PowerBASIC optimized FOR/NEXT loops to count down instead of up for improved execution speed. This optimization could cause the counter variable to contain a value which was not expected when execution of the loop was complete. This optimization has been improved so that the counter variable value is always correct upon loop completion, even if EXIT FOR was used to force an early termination.

Previous version of Classic PowerBASIC supported a single NEXT statement used with multiple nested FOR/NEXT loops, such as NEXT c, b, a. This is no longer supported and you will need to update your code to use multiple NEXT statements.

- [FUNCTION/END FUNCTION](#) statements may now be prepended with the word THREAD for clarity and self-documentation. This is recommended.

```
THREAD FUNCTION FuncName(ByVal x&) AS LONG
...
END FUNCTION
```

- [GRAPHIC BITMAP LOAD](#) statement has been improved with an optional stretch mode parameter to enhance the quality of bitmaps which are changed in size.
- [GRAPHIC FONT](#) statement has been enhanced to allow the points and style attributes to be optional parameters.
- [GRAPHIC INKEY\\$/INSTAT/WAITKEY\\$](#) now recognize the F11 and F12 keys.
- [GRAPHIC SCALE](#) statement has been expanded with the PIXELS option. GRAPHIC SCALE PIXELS sets or resets the coordinate system to pixel coordinates.
- [GRAPHIC STRETCH](#) statement has been improved with an optional stretch mode parameter to enhance the quality of resized bitmaps.
- [GRAPHIC WINDOW CLICK](#) now detects single and double mouse clicks.
- INTERFACE/END INTERFACE Block has been updated to support both Dispatch and Direct Interfaces. Dispatch interfaces are now defined using INTERFACE IDBIND interfacename. Previous versions of Classic PowerBASIC compilers used an older style syntax of "INTERFACE DISPATCH interfacename" for this structure. It was updated to better reflect the nature of the description.
- [ISFILE](#) operation and documentation now match. ISFILE now only supports files and not directories.
- statement has been divided into four different categories:

- [LET](#) statement with standard data type has been improved to support compound assignments (+=, -=, *=, /=, \=, &=, AND=, OR=, XOR=, EQV=, IMP=, and MOD=).
- [LET](#) statement with UDTs support direct assignment of data from one user-defined type variable to another.
- [LET](#) statement with Objects assigns an object reference to an object variable.
- [LET](#) statement with Variants assigns a value or an object reference to a [variant](#) variable.
- [LISTVIEW FIND](#) and [LISTVIEW FIND EXACT](#) starting row to search the entire ListView has been corrected to be 1.
- [METHOD](#) and [PROPERTY](#) statements may now return a [User Defined Type](#).
- [PROPERTY](#) methods can now take ByRef parameters.
- [OBJECT](#) statement has been expanded with the RAISEEVENT keyword. OBJECT RAISEEVENT will call or execute a member Method of an event Interface. The OBJECT statement has also been enhanced to support all COM compatible data types as parameters, return, and assignment values.
- [PATHNAME\\$](#) has been enhanced to support relative paths.
- [PRINT#](#) statement, when used without any parameters outputs a blank line to the file (i.e. a CR/LF only).
- [PROGID\\$](#) function, has been enhanced to accept ProgIDs up to 99 characters, even though COM rules indicate that a ProgID cannot contain more than 39 characters.
- [RGB](#) function now accepts individual red, green, and blue values or single [BGR](#) value.
- [SHELL](#) statement now supports an optional EXIT TO clause. If specified, the exit code of the child process (the value returned by the [WinMain](#) function) is retrieved.
- [TAB](#) control resizing and positioning has been improved.
- [THREAD CREATE](#) statement now supports an optional stack size parameter to specify the requested size of the stack for this newly created thread.
- [UCODE\\$](#) function may now contain an optional code page parameter. The code page parameter represents the code page to be used for the conversion process.
- [WRITE#](#) statement, when used without any parameters outputs a blank line to the file (i.e. a CR/LF only). WRITE# has been extended to allow a trailing comma or semicolon, the final carriage return / line feed is suppressed and replaced with a comma delimiter.
- [XPRINT FONT](#) statement has been enhanced to allow the points and style attributes to be optional parameters.
- [XPRINT SCALE](#) statement has been expanded with the PIXELS option. XPRINT SCALE PIXELS resets the coordinate system to pixel coordinates.

See Also

[New Statements and Functions](#)

[New in the IDE](#)

[Additional Changes](#)

- Dramatic improvement in GRAPHIC execution speed.
- Improved TAB Controls execution and reduced flicker.
- [IUnknown](#), [IAutomation](#), and [IDispatch](#) have been added as new data types for creating an [object reference](#)
- [Instance](#) variables are now supported by a new [variable scope](#) definition named [INSTANCE](#). These variables are only visible within the object they are declared in and are unique for each object of a particular [class](#).
- Control callbacks now receive %WM_NOTIFY messages unless this is overridden with the [#MESSAGES](#) metastatement.
- [Callback functions](#) may now return a second result value through a special Windows data area named DWL_MSGRESULT.
- The compiler now resolves all forward references to internal procedures automatically
- When an optional variant parameter is "missing", Classic PowerBASIC substitutes a variant of type %VT_ERROR with %DISP_E_PARAMNOTFOUND (&H80020004) set as the error value. The [ISMISSING](#) function can be used to determine whether a byref non-variant optional parameter was passed by the calling code.
- Parameter variable names in [DECLARE](#) statements may be reserved words.
- Many new predefined numeric equates and string equates have been built-in to the compiler. See the [Built-in string equates](#), [Built-in numeric equates](#), and the [Built In RGB Color Equates](#) topics for a complete list.
- Any [numeric equate](#) or [string equate](#) which is pre-defined in the compiler may be assigned a revised value if the new assignment is performed before the equate is referenced in the program.
- Named parameters are now supported on all Dispatch objects.
- Corrected compatibility errors with VBScript.
- Errors assigning nul objects from a variant to an object variable have been resolved.
- [Error 409](#) added: "Procedure is too large".
- [Error 464](#) added: "Undefined class reference"
- [Error 469](#) added: "Quad integer variable expected"
- [Error 483](#) added: "Requires Object Procedure (Method/Property)"
- [Error 484](#) altered: "Requires Procedure (Function/Method...)"
- [Error 487](#) added: "Multiple NEXT not allowed"
- [Error 492](#) added: "Invalid SORT function"
- [Error 500](#) added: "Variable name must be unique"

- [Error 502](#) added: "COM interface name expected"
- [Error 507](#) added: "OLE variable expected".
- [Error 508](#) added - "INSTANCE not allowed here"
- [Error 509](#) added: "Interface mismatches class"
- [Error 543](#) altered: "Must be outside Sub/Function/Class..."
- [Error 566](#) added: "CLASS expected"
- [Error 567](#) added: "END CLASS expected"
- [Error 568](#) added: "METHOD expected"
- [Error 569](#) added: "END METHOD expected"
- [Error 570](#) added: "PROPERTY expected"
- [Error 571](#) added: "END PROPERTY expected"
- [Error 572](#) added: "PROPERTY GET expected"
- [Error 573](#) added: "Valid only in a CALLBACK FUNCTION"
- [Error 574](#) added: "Not allowed in an Event Class"
- [Error 575](#) added: "EVENT SOURCE is not declared"
- [Error 576](#) added: "Too many Interfaces"
- [Error 577](#) added: "EVENT INTERFACE expected"
- [Error 578](#) added: "INHERIT of Base Class expected"
- [Error 579](#) added: "BYREF variable or BYVAL/BYREF variant expected"
- [Error 580](#) added: "Duplicate GUID usage"
- [Error 581](#) added: "Type Library creation error"
- [Error 582](#) added: "Duplicate Dispatch interface"
- [Error 583](#) added: "Unpaired PROPERTY definition"
- [Error 584](#) added: "Mismatched PROPERTY pair"
- [Error 585](#) added: "PROPERTY requires BYVAL parameters"
- [Error 586](#) added: "User Defined Type or AS expected"
- [Error 587](#) added: "Invalid Constructor/Destructor"
- [Error 588](#) added: "Indirect operand must be bracketed: [12]"
- [Error 589](#) added: "Dual/IDispatch interface is required"
- [Error 590](#) added: "PROPERTY SET requires at least one parameter"
- [Error 591](#) added: "BYVAL with OUT is not allowed"
- [Error 592](#) added: "Return value required"
- [Error 593](#) added: "Dual or Automation interface is required"

- [Error 594](#) added: "Macro ends with continuation ' _'"
- [Error 595](#) added: "Object return type required"
- [Error 596](#) added: "Inherited interface expected"
- [Error 597](#) added: "Invalid name or sequence in the interface"
- [Error 598](#) added: "CLASS METHOD name expected"
- [Error 599](#) added: "Invalid within an INTERFACE"
- [Error 600](#) added: "Macro phase error, referenced before define"
- [Error 601](#) added: "One INHERIT per interface"
- [Error 602](#) added: "Hidden interface referenced by COM"
- [Error 603](#) added: "Incompatible with a Dual/IDispatch interface"
- [Error 604](#) added: "Incompatible with #COM TLIB generation"
- [Error 605](#) added: "Macro parameter mismatch"
- [Error 606](#) added: "Macro empty parentheses "()" are needed"
- [Error 607](#) added: "Too many macro expansions"
- [Error 613](#) added: "Cannot compile - the program is now running"
- [Error 801 to 815](#) "Internal error"

See Also

[New Statements and Functions](#)

[Changes to existing Statements and Functions](#)

[New in the IDE](#)

- Printing from the IDE has been improved with margins and header with page, file information, date-time stamp and a separating line.
- The [debugger](#) now supports evaluate/modify of [FIELD](#), [BIT](#), and [SBIT](#) variables.
- Multiple files can now be saved and opened in the IDE using a single [Project File](#).
- You can now easily select multiple different source code files from the [Code Finder](#) dialog.
- The Code Finder now lists [Methods](#) and [Properties](#) as well as [Subs](#), [Functions](#), and [Macros](#).
- The IDE Context menu [Select block](#) option now supports [Classes](#), interfaces, Methods, and Properties along with other block statements.
- Improved interaction with [Classic PowerBASIC Forms](#).
- Insert [GUID](#) option, allows insertion of a unique GUID for COM and other operations.
- Improved handling of [bookmarks](#)
- Improved handling of [breakpoints](#)
- The Help | About dialog box now displays the serial number of compiler.
- Further optimized debugging speed with improved reliability.
- Improved [undo](#) interaction.
- Fixed an issue with the code finder dialog box not honoring the previous state of the source window (maximized or normal).
- File | Open Project accelerator changed to J as it was conflicting with the accelerator for Print.
- An issue with the Variable Watch Window not recognizing single letter variable names if the caret was to the left of the name has been resolved.
- Improved interaction with the resource compiler.
- Literals in DATA statements that contained escaped quotes are now property colored.
- Improved interaction with the Classic PowerBASIC and Microsoft SDK help files.
- Fixed an issue in the Evaluate and Variable Watch Window with displaying certain [Quad Integer](#) values.
- Resizing of Register and Variable Watch Windows flashing on Windows Vista has been resolved.
- Fixed an issue with the Code Finder dialog box sort arrows on Windows Vista.
- Fixed an issue with the [resource compiler](#) corrupting certain user-defined resources.
- A program being debugged on 64-bit versions of Vista not always terminating has been resolved.

- The Microsoft Windows SDK help file no longer receives special treatment. It works just like any other custom help file, using the new "Windows SDK.PBKeys" index in the bin folder.
- Fixed an issue with the Bookmark dialog sort-direction icons being inappropriately displayed under Vista.
- The Code Finder does not list [METHOD/PROPERTY OBJRESULT](#) any more.
- Trailing whitespace is trimmed from lines of files at load time.
- Resolved an issue on Windows Vista when the IDE is zoomed.
- The edit window is now constrained to maximum physical line length on horizontal scrolling.
- Improved colorization and capitalization of names that match keywords when used in [TYPE](#) or [UNION](#) blocks.
- Fixed an issue when printing source code with a dark background and light text.

See Also

[New Statements and Functions](#)

[Changes to existing Statements and Functions](#)

[Additional Changes](#)

The Classic PowerBASIC for Windows Compiler (PB/Win) is comprised of two core applications: the [Integrated Development Environment](#) and the compiler itself. This chapter describes launching the compiler directly.

See Also

[Running PB/Win From Windows](#)

[Running PB/Win From DOS](#)

[PB/Win Command Line Switches](#)

[The Integrated Development Environment](#)

Running PB/Win From Windows

[Top](#) [Previous](#) [Next](#)

Double-click the PB/Win Compiler icon (PBWIN.EXE). A dialog box will appear asking for a file name and compile options:



Type the name of the BASIC source file, plus any desired options, and click the OK button. To abort, click the Cancel button. See below for command-line parameters that may be specified in the dialog box.

See Also

[Running PB/Win From Windows](#)

[Running PB/Win From DOS](#)

[PB/Win Command Line Switches](#)

[The Integrated Development Environment](#)

Running PB/Win From The Command Prompt

[Top](#) [Previous](#)
[Next](#)

Run PBWIN.EXE from the command prompt, using a command-line with the following syntax:

```
PBWIN.EXE [/Ipath] [/L] [/Q] [/Cfilename] FileName
```

where *FileName* is the name of the source file to compile. If you just type PBWIN (omitting *FileName*), you'll get a dialog box asking for the name of the file to compile.

Classic PowerBASIC first attempts to open the source file using the *FileName* specified. If the file cannot be opened and *FileName* does not have an explicit .BAS extension, Classic PowerBASIC appends .BAS to the specified file name, and attempts to open that file. If *FileName* is a Long File Name (LFN) or contains spaces, it must be enclosed in quotes.

Classic PowerBASIC also supports Long File Names in all metastatements, for example:

```
#INCLUDE "C:\Program Files\Classic PowerBASIC\LIBRARY.INC".
```

See Also

[Running PB/Win](#)

[Running PB/Win From Windows](#)

[PB/Win Command Line Switches](#)

[The Integrated Development Environment](#)

Include /I

The /I command-line option provides the compiler with a search path list when looking for [#INCLUDE](#) and [#RESOURCE](#) files. Multiple directories can be specified in this path list by separating each path with a semicolon (;).

During compilation, the compiler scans this path list for the necessary files before checking the current (default) directory. To ensure that the current (default) directory is searched ahead of this path list, specify a period followed by a backslash (\) at the beginning of the path list. For example:

```
/I.\;C:\CLASPB9\WINAPI;D:\SOURCE
```

The Include parameter also works with Long File Name (LFN) paths, provided that individual LFNs are enclosed in quotes. For example:

```
/I"C:\Program files\My Applications\";C:\PB;"D:\Source Code\"
```

See [#INCLUDE](#) and [#RESOURCE](#) for additional details.

Log /L

The /L command-line option causes the compiler to generate a log file with all of the compile results, including [error](#) code and error line number, if an error occurs during compile-time.

Quiet /Q

The /Q command-line option causes the compiler not to display a message box when compiling is finished. This should only be used with the /L option.

Command /C

The /C command-line option specifies a filename that contains the complete command-line. This may be used to specify very long command lines to the compiler of up to 1024 bytes, which may otherwise exceed the operating system limits. This may be useful in situations where the /I path is very long, and the full path to the source file is very long. The /C option may not be used in conjunction with any other command-line options.

See Also

[Running Classic PB/Win](#)

[Running Classic PB/Win From Windows](#)

[Running Classic PB/Win From DOS](#)

The PB/Win Integrated Development Environment

[Top](#) [Previous](#)
[Next](#)

This topic will help you learn how to use all the options available in the Classic PowerBASIC Integrated Development Environment (which we will refer to as the IDE). You will learn how to use the editor, move from window to window, menu to menu, and choose menu commands. See [Debugging PB/Win Programs](#) for information on the Integrated Debugger.

To launch the IDE, double-click the PBEDIT.EXE icon, type PBEDIT at the command-line, or use the START menu entry.

The PB/Win editor (PBEDIT.EXE) can also be launched from the command-line, and supports the following command-line options:

```
PBEDIT.EXE [/G:row,col:] [/P:MainFile] [Filename]
```

The command-line options may be prefixed with either a forward-slash (/) or a hyphen (-). Multiple files can be specified for the *Filename* parameter, each separated by space characters. Long file names should be enclosed in double-quote marks (").

Goto /G:

The /G: command-line option causes the IDE to move the caret to the row and column specified. For example, /G:10,20: cause the caret to start at line 10, column 20. The /G option must be terminated by a trailing colon.

```
PBEDIT.EXE /G:10,20: "Project Bluepad.bas"
```

Primary Source File /P:

The /P: command-line option specifies the name of the file that will be set as the Primary Source File. This option is useful when working on large applications that span multiple source code files, especially when loading multiple files at startup. When a compile/execute/debug operation begins, the IDE automatically uses the Primary Source File as the "main" file, regardless of which other files are loaded or have focus in the IDE.

The Primary Source File will be one of the files loaded into the IDE.

```
PBEDIT.EXE /P:Project.bas "Support Library.inc" Project.rc "Data file index.txt"
```

See Also

[The Classic PowerBASIC User Interface](#)

[Toolbar Buttons](#)

[Editor Hot Keys](#)

[IDE Context Menu](#)

[Custom Help Files](#)

[File Templates](#)

[Code Finder Dialog Box](#)

[Command Line Dialog Box](#)

[Debugger Evaluate Dialog Box](#)

[Find Dialog Box](#)

[Go to Line Dialog Box](#)

[Primary Source File Dialog Box](#)

[Replace Dialog Box](#)

[IDE Options](#)

The Classic PowerBASIC User Interface

[Top](#) [Previous](#)
[Next](#)

Insert File	Insert a document (file) at the caret position in the current document.
New File	Create a new empty document in the editor.
New File As	Create a new empty document in the editor, using a specified file template.
Open File	Use the Open dialog box to load an existing document.
Save File	Save the current document using its current name.
Save File As	Save the current document using a new name.
Print	Print the current document.
Close File	Close the current document.
Close All Files	Closes all files open in the editor.
[Recent Files list]	A list of the most recently loaded source code documents.
Command Prompt	Open a command prompt window ("DOS box").
Exit	Close all open documents and exit the IDE.
Undo	Undo the last action or deletion.
Clear	Delete the selected text.
Cut	Remove selected text and place it in the Clipboard.
Copy	Copy the selected text and place it in the Clipboard.
Paste	Copy text from the Clipboard into the current document.
Select All	Select all the text in the current document.
Block selection	Perform the selected operation on every line in the selected block. Operations include comment /uncomment, indent/unindent by a tab level, and indent/unindent by one space.
Insert GUID	Inserts a new unique GUID at the current insertion point.
Find	Search the current document for a selected word or phrase. See Find dialog for more information.
Find Next	Search the current document for the next occurrence of the previous word or phrase used in Find.
Replace	Search the current document for a selected word or phrase and replace it. See Replace dialog for more information.
Go to Line	Move the caret to the selected line number in the current document. See Go to Line dialog for more information.
Go to Bookmark	Move the caret to the selected bookmark in an open document.

Code Finder	The Code Finder dialog presents a list of Subs , Functions , Methods , Properties , and Macros in current document, to quickly jump to a selected section of code.
Compile	Compile the current source document or, if youve chosen one, the Primary Source File.
Compile and Execute	Compile and then Run the current source document or, if youve chosen one, the Primary Source File .
Compile and Debug	Compile and then Debug the current source document or, if youve chosen one, the Primary Source File.
Set Primary Source File	Define which module to compile when the Compile or Debug options are selected. The Primary Source File can also be specified via the /P: IDE command-line switch .
Command Line	Set the command-line to pass to the program when debugging or executing from within the IDE. See COMMAND\$.
Classic PowerBASIC COM Browser	Open the Classic PowerBASIC COM Browser (PBROW.EXE) to generate COM interface code from Object libraries.
Classic PowerBASIC Forms	Open the Classic PowerBASIC Forms (PBFORMS.EXE) visual designer and code generator application (if installed).
Cascade	Stack all open windows and overlap them so that part of each underlying window is visible.
Tile Horizontal	Arrange your open windows from top to bottom without overlapping one another.
Tile Vertical	Arrange your open windows from left to right so that they display next to each other.
Options	Display the Options dialog, for configuring the Classic PowerBASIC IDE.
[current files list]	A list of files currently opened in the Classic PowerBASIC IDE.
Run	In edit mode, this will compile and then debug the current source document or, if youve chosen one, the Primary Source File . In debug mode, this causes your program to run until it hits a breakpoint or ends.
Run to Caret	Begin running the program. It continues to run until the debugger either reaches the current line, or encounters a breakpoint, etc. <i>CTRL+F8</i> is the hot-key for the <i>Run to Caret</i> option.
Animate	The debugger runs the program using an automated Step-Into technique. Execution continues until a breakpoint is reached, the Stop button is pressed, or the program completes. The Animate delay can be set through the IDE's Options Dialog .
Stop	Halt the debugger. If the debugger is already halted, this has no effect.
Step Into	If the current line contains a call to a Sub , Function , Method , or Property , the debugger traces execution into that procedure. You cannot step into

an API call, or into an external module. *F8* is the *Step Into* hot-key.

Step Over

The debugger executes the current line of code. If the line contains a reference to a Sub, Function, Method, or Property, the debugger executes that code without tracing into the procedure. SHIFT+F8 is the Step Over hot-key.

Step Out

The debugger runs the code until the current Sub, Function, Method, or Property exits. If the current function is [PBMAIN](#) or [WINMAIN](#), the code is executed until the program is finished or another breakpoint is encountered. CTRL+SHIFT+F8 is the Step Out hot-key.

Evaluate Variable

Evaluate or modify a [variable](#), or add/remove a variable in the Watch window. It is not possible to use this to change the length of a string. Also see Watch CPU Registers.

Clear all Watches

Remove all variables from the Watch window.

Toggle Breakpoint

Set or release a breakpoint on the current line. F9 is the Toggle Breakpoint hot-key.

Clear all Breakpoints

Release all breakpoints in the program.

Watch CPU Registers

Show or hide the Register Watcher window, which lets you see the state of the [CPU registers](#) and flags when debugging.

Variable watch window

Show or hide the Variable Watcher window, which lets you see the state of the [ERR function](#) and any variables you choose to watch when debugging.

Program Restart

If the current program is halted/stopped, the program will be reset, ready for debugging to commence again. *SHIFT+F5* is the *Reset* hot-key.

Exit Debugging

Halts the current program and terminates the debugger. The variable list in the Watch window is retained between debugging sessions, until the IDE is closed.

Dynamic Help

Displays the help file with information about the closest word to the edit caret.

PB/Win Contents

Displays the PB/Win help files table of contents tab.

PB/Win Index

Displays the PB/Win help files index tab.

PB/Win Search

Displays the PB/Win help files search tab.

Help for Windows SDK

Display the [Win32.hlp](#) help file (if installed).

[other help files]

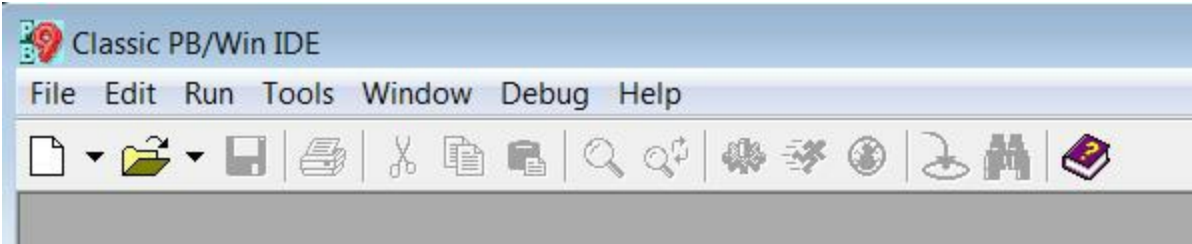
Display an add-on help file.

Classic PowerBASIC Web Site

Launch your default Web Browser and connect to the [PowerBASIC web site](#).

About PBEdit Display version information for the Classic PowerBASIC IDE.

The [Classic PowerBASIC IDE](#) was designed to provide you with the tools you need to quickly and intuitively develop high-performance applications. This section briefly describes each element of the IDE.





Create a new empty document (file) in the [editor](#).



Use the Open File dialog box to load an existing document.



Save the current document if it has been modified and unsaved.



Print the current document to a printer.



Move the selected text from the document to the clipboard.



Copy the selected text from the document to the clipboard.



Copy the text from the clipboard into the current document.



Search the current document for a word or phrase. See [Find dialog](#) for more information.



Search the current document for a word or phrase and replace it. See [Replace dialog](#) for more information.



Compile the current source document (or *Primary Source File* if specified).



Compile and Execute the current (or Primary) source document



Compile and Debug the current (or Primary) source document.



Launch the [Go to Line dialog](#) to jump to a specific line in the current document.



Launch the [Code Finder dialog](#), which presents a list of [Subs](#), [Functions](#), [Methods](#), [Properties](#), and [Macros](#) in current document, to quickly jump to a selected section of code.



Display the Classic PowerBASIC or the WIN32.HLP file.

Editor Hot Keys

[Top](#) [Previous](#) [Next](#)

The following table summarizes the hot-keys available in the [Classic PowerBASIC IDE](#) Editor Window:

Keystroke	Description
F1	Dynamic Help
CTRL+ALT+F1	Display the Table of Contents tab of the Classic PowerBASIC For Windows Help File
CTRL+ALT+F2	Display the Index tab of the Classic PowerBASIC For Windows Help File
CTRL+ALT+F3	Display the Search tab of Classic PowerBASIC For Windows Help File
F2	Code Finder dialog
F3	Find dialog/Find next
SHIFT+F3	Find previous
F4	Duplicate current line
CTRL+F4	Close current document
ALT+F4	Exit PBEDIT
F5	Compile and debug (if in edit mode) or Run program (if in debug mode)
CTRL+F5	Animate program.
F6	Clear to end-of-line
CTRL+F6	Switch to the next document window
F8	Step into next program line (when debugging)
CTRL+SHIFT+F8	Step out of current procedure (when debugging)
SHIFT+F8	Step over next program line (when debugging)
CTRL+F8	Run to caret (when debugging)
F9	Break (stop the program being debugged)
ALT+BACKSPACE	Undo
SHIFT+INSERT	Paste text from clipboard
SHIFT+DELETE	Cut text to clipboard
CTRL+DELETE	Cut text to clipboard
CTRL+INSERT	Copy text to clipboard
CTRL+HOME	Move to start of document
CTRL+END	Move to end of document

CTRL+PAGEUP	Move to start of document, maintaining caret position on screen
CTRL+PAGEDOWN	Move to end of document, maintaining caret position on screen
CTRL+0	Go to bookmark 0
ALT+0	Set bookmark 0
CTRL+1	Go to bookmark 1
ALT+1	Set bookmark 1
CTRL+2	Go to bookmark 2
ALT+2	Set bookmark 2
CTRL+3	Go to bookmark 3
ALT+3	Set bookmark 3
CTRL+4	Go to bookmark 4
ALT+4	Set bookmark 4
CTRL+5	Go to bookmark 5
ALT+5	Set bookmark 5
CTRL+6	Go to bookmark 6
ALT+6	Set bookmark 6
CTRL+7	Go to bookmark 7
ALT+7	Set bookmark 7
CTRL+8	Go to bookmark 8
ALT+8	Set bookmark 8
CTRL+9	Go to bookmark 9
ALT+9	Set bookmark 9
TAB	Indent marked block by one tab level
SHIFT+TAB	Outdent marked block by one tab level
SPACE	Indent marked block by one space
SHIFT+SPACE	Outdent marked block by one space
CTRL+A	Select all
CTRL+B	Go to Bookmark dialog
CTRL+C	Copy text to clipboard
CTRL+D	Duplicate current line
CTRL+E	Build and Execute
CTRL+F	Find dialog
CTRL+G	Go to Line dialog

CTRL+I	Toggle auto-indent mode
CTRL+K	Clear to end-of-line
CTRL+L	Select current line
CTRL+M	Compile the current document (or primary source file , if any)
CTRL+N	Create a new document, using the default file template
CTRL+O	Open an existing document
CTRL+P	Print the current source document
CTRL+Q	Comment -out marked block
CTRL+SHIFT+Q	Uncomment -out marked block
CTRL+R	Find and Replace dialog
CTRL+S	Save the current document
CTRL+T	Delete the word at the caret
CTRL+U	Paste text from clipboard
CTRL+V	Paste text from clipboard
CTRL+X	Cut text to clipboard
CTRL+Y	Cut current line to clipboard
CTRL+Z	Undo last change
CTRL+ALT+G	Insert a GUID

When editing a file in the Classic PowerBASIC [IDE](#), a popup context menu is available by right-clicking the mouse within the edit window. The available content of the menu is automatically determined by position and text located at the point where the context menu is activated. The full context menu looks like this:

Undo	Reverse the most recent edit change from the undo stack. Each successive Undo operation removes the most recent change from the undo stack and reverses the editing in the document. Undo operations can be performed until the undo stack is depleted; however, the undo stack is cleared when a Save or Compile operation is performed.
Clear	Delete the currently selected block of text.
Cut	Copy the currently selected block of text to the clipboard, and delete the highlighted block from the file.
Copy	Copy the currently selected block of text to the clipboard.
Paste	Paste the contents of the clipboard into the current file.
Select Line	Select (highlight) the complete line at the context-menu point.
Select Block	Select (highlight) a complete block of code. This menu item is available when the context menu is activated on the first line of a formal block. Formal blocks include those that begin with the #PBFORMS metastatement (PB/Win only), the FOR/NEXT and SELECT CASE blocks, plus the usual CALLBACK , CLASS , INTERFACE , FUNCTION , METHOD , PROPERTY , SUB , TYPE , and UNION statements.
Insert GUID	Inserts a new unique GUID at the current insertion point.
Select All	Select (highlight) the entire contents of the current file.
Run to Caret	Run the program until execution reaches the current caret position (debug mode only).
Watch Variable	Add the variable at the current caret position to the Variable Watcher window (or remove it, if its already there). The Variable Watcher window is visible only in debug mode.
Evaluate Variable	Evaluate or modify the variable at the current caret position (debug mode only).
Toggle Bookmark	Add or remove a bookmark at the current caret position.

Toggle Breakpoint	Add or remove a breakpoint at the current caret position. Breakpoints only work in debug mode.
Help	Launch context-sensitive help. If the context menu point is on a Classic PowerBASIC keyword, the appropriate topic is displayed in the Classic PowerBASIC help file. If the context point is not a recognized keyword, the WIN32.HLP file is launched instead. This feature is useful for context-sensitive help on both reserved keywords, and API functions and data structures, etc.
Open File	Open the file indicated in the #INCLUDE metastatement at the context menu point. The item is only enabled when the context menu point is targeting an #INCLUDE metastatement. Open File works even if the #INCLUDE metastatement is commented out.

See Also

- [The Integrated Development Environment](#)
- [Debugging PB/Win Programs](#)

A *file template* is the framework for a new file, which you can load into the [IDE](#) with the "New File As" option. While a template can contain anything you like, it is typically used to automate the basic boilerplate needed for a new document. For example, the "Generic PB program" template creates a new file with the following information already filled out:

```
#COMPILE EXE
#DIM ALL

FUNCTION PBMAIN () AS LONG

END FUNCTION
```

What's more, the caret is conveniently placed in the middle of the [FUNCTION](#) block for you, letting you get right to programming!

You can readily build templates of your own, or modify the ones that come with the IDE. A template is simply a text file created according to a few simple rules. Let's look at the default template (you can load it into the IDE, NotePad, or any other text editor). Classic PowerBASIC templates use ".PBTPPL" for their file extension. The default template is "Default.pbtpl", then. You can find it in the Bin subdirectory for your compiler ("C:\ClasPB9\Bin", by default).

The first line starts out with a number:

```
1
```

This is the template version number, 1 (one). There may be other options in the future, but the number must always be 1, for now.

The second line contains the file extension to apply to files that are created with this template:

```
.bas
```

The third line gives the name of the template, which will be used in the "New File As" menu:

```
Generic PB program
```

The following lines give the text to be filled into the file created by the template. There is one special character, the "|" vertical bar or pipe symbol. This indicates where the caret should be placed after the text is filled in.

```
#COMPILE EXE
#DIM ALL

FUNCTION PBMAIN () AS LONG

    |

END FUNCTION
```

That's all there is to it!

After creating a new template, save the .PBTPPL file in the Bin subdirectory for your compiler. The default location for this is, typically, "C:\ClasPB9\Bin\". Now, the next time

you start the Classic PowerBASIC IDE, your custom template will be available on the "New File As" menu.

See Also

[The Integrated Development Environment](#)
[Project Files](#)

A project file is used to speed up the process of loading multiple source code files, especially when the source files are saved in different directories. When you open a project file all the individual source code files are opened in the IDE. There is no limit to the number of files that may make up a project.

A project file is saved with an extension of .PBP or .PRJ extension, unless the list of project file extensions has been modified, see the [Editor Preferences](#) topic for information on modifying the extensions used for project files.

See Also

[The Classic PowerBASIC Integrated Development Environment
File Templates](#)

The [Classic PowerBASIC IDE](#) has built-in context-sensitive help for Classic PowerBASIC keywords. If the caret is placed on a keyword when you invoke help, you will get help for that specific keyword. Now, you can add context-sensitive help for your own help files. Here's how.

For each help file, create a text file with a name of your choice, with a file extension of .PBKeys (using the Classic PowerBASIC IDE, NotePad, or any other text editor). The first line of the text file must contain the name of the help file, as it will be shown in the IDE's help menu, like so:

```
MenuName="PowerTree 1.1"
```

The next line of the PBKeys file specifies the name and location of the help file. If the help file is in the same directory as the .PBKeys file, you can specify just the filename, without the path. Otherwise, you must provide a fully-qualified absolute path:

```
HelpFile="C:\PTreeW11\PwrTree.hlp"
```

Each following line specifies a help keyword. This keyword must be present in the index of the help file, in order for context-sensitive help to work.

```
HelpKey="AccessBlock"  
HelpKey="ptCreateIndex"  
HelpKey="ptAdd"  
and so forth.
```

When you're done, save the .PBKeys file in the Bin subdirectory for your compiler. The default location for this is, typically, "C:\ClasPB9\Bin\". Now, the next time you start the Classic PowerBASIC IDE, your custom keywords will be recognized by the context-sensitive help system. You will also be able to load the help file from the Help menu.

If your help file does not appear in the Help menu when you start the IDE, make sure the HelpFile line of the .PBKeys file specifies the correct location and name for your help file.

The complete PowerTree .PBKeys file, "PowerTree 1.1.PBKeys", is already installed in your compiler's Bin subdirectory. **Please note** that the custom help list is only loaded if you have PowerTree 1.1, and it's installed at the location specified in the HelpFile line of the PBKeys file.

See Also

[The Integrated Development Environment](#)

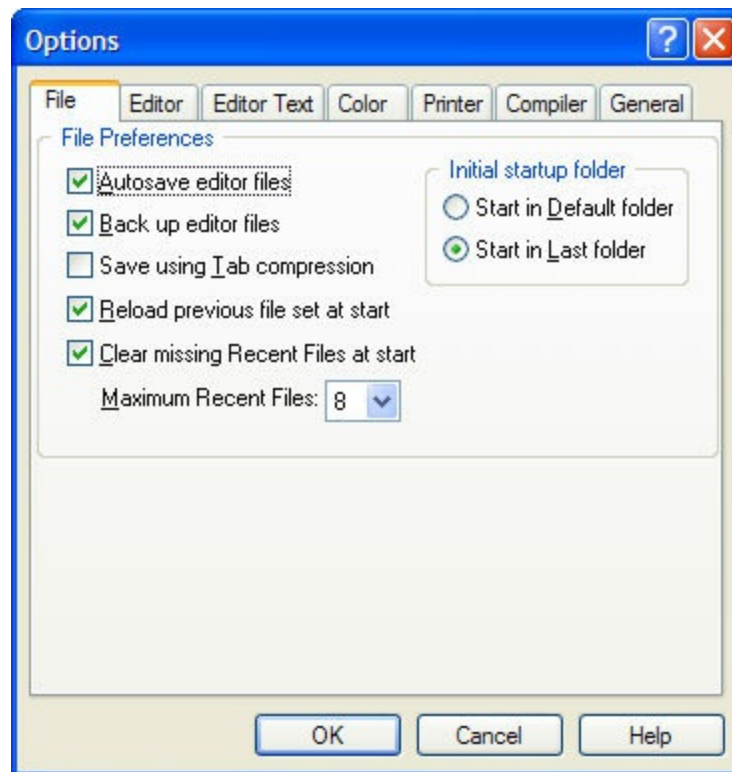
The section describes the options that are available to customize the [IDE](#) environment, file paths, and compiler behavior. These options are divided into seven categories:

Browsing for Include folders	Settings for Compiler search order of include files
Compiler Preferences	Settings for file types, the Compiler, and the Debugger
Editor Preferences	Main settings for the Editor
Editor Text Preferences	Font settings for the Editor
File Preferences	File handling settings for the Editor
General Preferences	General configuration settings and options
Printer Preferences	Font settings for printing source code
Syntax Color Preferences	Syntax color settings for editing and printing

See Also

[The Integrated Development Environment](#)

[Debugging PB/Win Programs](#)



Autosave editor files

If Autosave is "on", the [IDE](#) will save all open files before compiling or [debugging](#) code. Autosave also saves files when switching to Classic PowerBASIC Forms via the F7 hot-key, or via the Tools menu (PB/Win only). If Autosave is "off", only the [primary source file](#) will be saved before compiling or debugging.

Back up editor Files

When saving files, the IDE will rename the previous disk file with the .BAK file extension, and save the latest copy under the original filename. This option provides a simple method of preserving the previously saved version of the source code.

Save using tab compression

When saving files, the IDE can compress leading spaces on every line into tabs, using the tab size specified under Editor Preferences. This helps maintain your preferred indentation levels when working with others who choose different tab sizes. It also reduces your source file size.

Reload previous file set at start

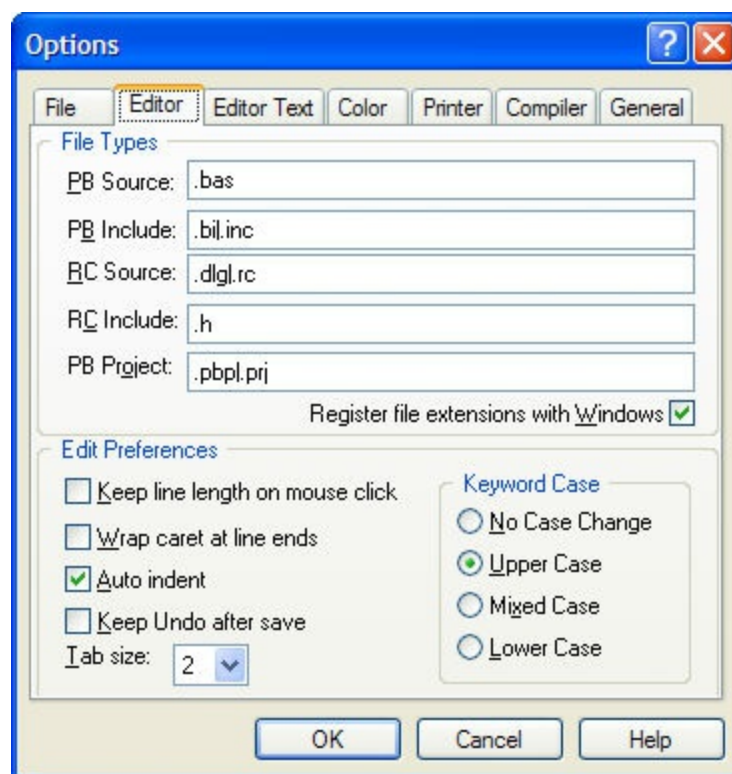
The files that were previously open in the IDE are reloaded automatically the next time the IDE starts up. Files are reloaded in their original order, regardless of the Primary Source File, if any. The caret position(s) are maintained. If a file is specified as a command line argument, then only the specified file is loaded into the IDE.

Clear missing recent files at start

The IDE checks the Recent Files list (located in the *File* menu) at start up. If any file cannot be located and read, the corresponding entry in the Recent Files list is automatically removed. Where files are located

across a network or removable media, this option may need to be unchecked.

Maximum Recent files	Specifies the maximum number of Recent Files tracked in the File menu, in the range of 0 through 9. Select 0 to disable the Recent Files list; otherwise, the selected number of previous files is tracked between sessions. Also see Reload previous file set at start.
Start in default folder	The IDE's initial directory for File Open/Save, etc. This option will retain the default directory assigned to the IDE at start-up. The folder option allows the initial directory to be specified in the "Start In" settings of a desktop Shortcut, etc.
Start in last folder	The IDE uses the directory that was in use the last time the IDE was closed down. This option may useful when working for extended periods on the same project.



PB Source

This is the file extension, or list of extensions, you expect to use for main Classic PowerBASIC source code modules: programs you can compile directly. You may enter multiple extensions by separating each with the vertical bar or "pipe" character, "|". The default setting for PB Source is ".bas".

PB Include

This is the file extension, or list of extensions, you expect to use for Classic PowerBASIC include files: bits of code that you will [#include](#) in a main module before compiling. You may enter multiple extensions by separating each with the vertical bar or "pipe" character, "|". The default setting for PB Include is ".bil.inc".

RC Source

This is the file extension, or list of extensions, you expect to use for [resource scripts](#): programs that are compiled with the RC.EXE [resource compiler](#). You may enter multiple extensions by separating each with the vertical bar or "pipe" character, "|". The default setting for RC Source is ".dlg.rc".

RC Include

This is the file extension, or list of extensions, you expect to use for your Classic PowerBASIC include files: bits of code that you will [#include](#) in a resource script before compiling with the RC.EXE resource compiler. You may enter multiple extensions by separating each with the vertical bar or "pipe" character, "|". The default setting for RC Include is ".h".

PB Project

This is the file extension, or list of extensions, you expect to use for

your Classic PowerBASIC [Project files](#). You may enter multiple extensions by separating each with the vertical bar or "pipe" character, "|". The default setting for a project file Include is ".pbp|.prj".

Register file extensions with Windows

Check this box to register your selected file extensions with Windows. This allows Windows to automatically load files with these extensions into the [Classic PowerBASIC IDE](#) when you click on a file in Explorer, or launch it from the Start menu, for example.

Keep Line Length

Clicking the mouse cursor beyond the right-most character of a line does not extend the line beyond the end of the actual text content.

Wrap Caret at Line Ends

Check this box to have left-arrow wrap to the previous line, and right arrow wrap to the next line, instead of stopping at the start or end of the current line.

Auto Indent

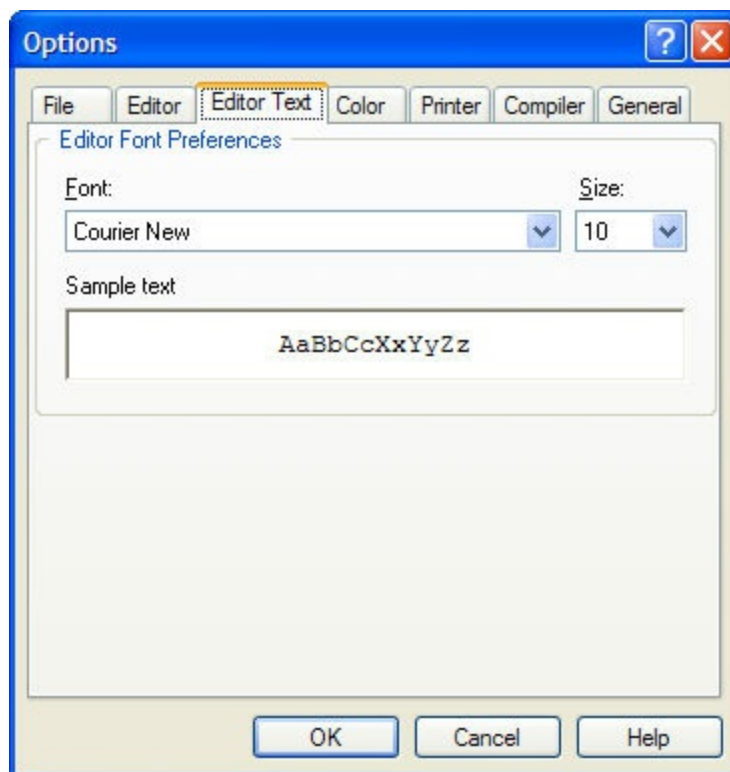
The IDE provides automatic indenting when *ENTER* is pressed, in order to assist with writing visually structured code. The Indent depth depends on the context of the text on the preceding line. For example, if the previous line starts with the word [FUNCTION](#), the following line is automatically indented. Auto-indent can be toggled from within the editor with the *CTRL+I* hot-key combination. See Tab Size.

Tab Size

The number of characters between "tab stops", in the range 1 through 8 inclusive. When the *TAB* key is pressed, the IDE substitutes space characters to move the caret to the next tab stop position. Tab Size also affects the Auto Indent depth.

Keyword Case

The IDE automatically sets the capitalization of reserved keywords as directed by this option. The use of capitalization can help readability of code. By default, the IDE applies keyword capitalization to BASIC source code files only, which are determined by the file extensions set under Compiler Preferences. Use care when applying capitalization to resource files (for example, .RC files, .H, and .DLG files) as these usually contain case-sensitive keywords. Custom keyword colors can be configured in the [Syntax Color Preferences page](#), and the editor font can be configured on the [Editor Text Preferences page](#).



Font

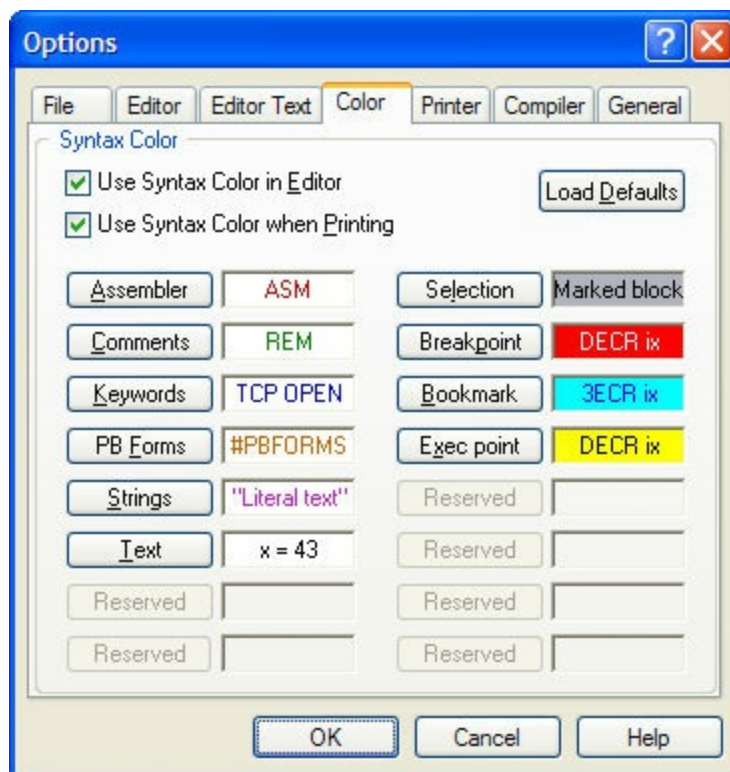
The [IDE](#) editor uses a fixed-width ANSI font. By default, the IDE uses the Courier New font. However, an alternative fixed-width font may be selected from this drop-down list.

Size

The desired point size of the IDE editor font. The default is usually 12 point.

Sample Text

How the editor text will appear with the selected font and at the selected font size.



Use Syntax Color in Editor

The [IDE](#) can show colored reserved keywords and other types of syntax in the source code file. Both the text (foreground) and background colors can be individually customized for each syntax type. The use of highlighting can increase readability of code. Also see *Use Syntax Color when Printing*.

Use Syntax Color when Printing

The IDE can optionally print source code with coloring applied to the reserved keywords and other syntax types. Printing with syntax coloring enabled only affects the text (foreground) - background coloring is not printed. Also see *Use Syntax Color in Editor*.

Load Defaults

Reset the syntax color table back to the default color scheme.

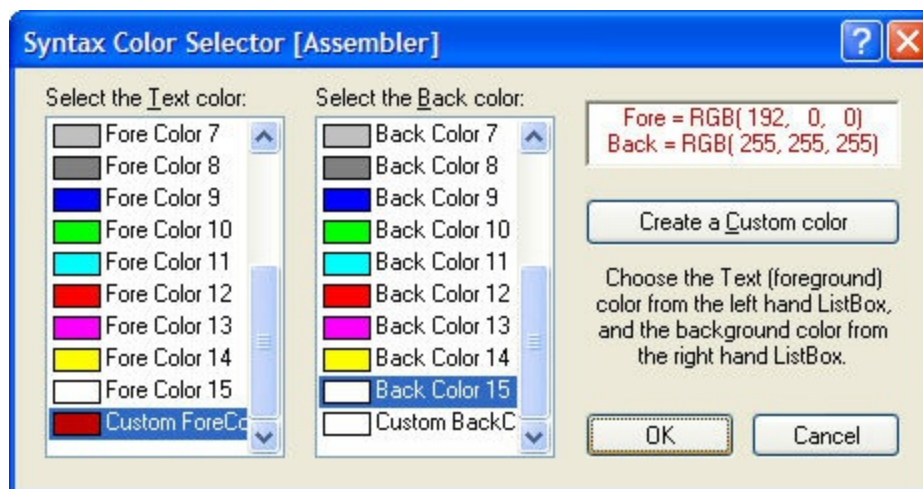
Assembler Launch the color selection dialog to choose the text (foreground) and background colors for inline assembler code.

Comments [Comments and REM](#) statement syntax color.

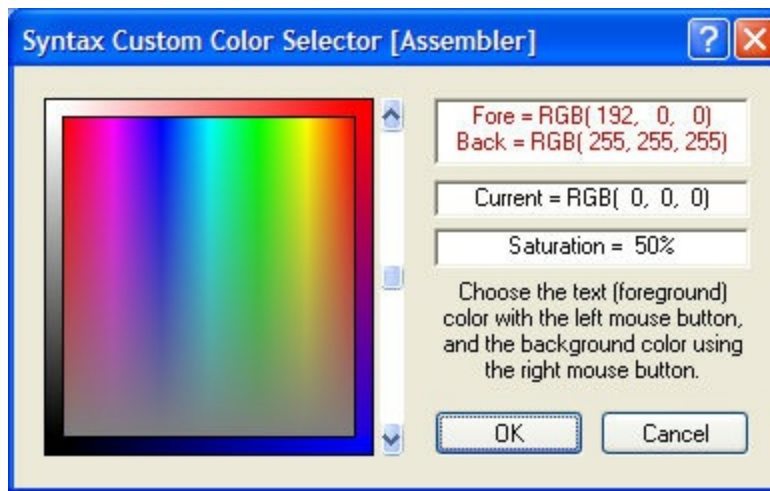
Keywords The syntax coloring applied to reserved keywords.

PB Forms The coloring applied to [Classic PowerBASIC Forms](#) named-block metastatements. Note: Classic PowerBASIC Forms is a GUI visual design tool, and therefore IDE support for it is currently restricted to the Classic PowerBASIC for Windows product line. In the [Classic Console Compiler's](#) IDE, the PB Forms syntax option is disabled, and reserved for future use.

- Strings** The syntax coloring applied to [literal strings](#).
- Text** The remaining types of syntax. Typically, this includes variable names, API function names, etc.
- Selection** The color used when selecting (highlighting) blocks of text, for example, in anticipation of clipboard operations such as Cut/Copy/Paste, etc.
- Breakpoint** The color used to highlight a [breakpoint](#).
- Bookmark** The color used to highlight a bookmark.
- Exec point** The color used to highlight the execution point, which is the next line to be executed in the [debugger](#).



- Select the Text color** The list contains a set of the 16 standard colors, plus the current custom text color. To choose a color in the list, simply select (click) on it. To adjust the custom text color, select the Create a Custom color button. The top-right preview control shows the current color combination as it will appear in the editor window.
- Select the Back color** The Back color list box control functions identically to the Text color list box control, but adjusts the background color rather than the text color.
- [Preview Fore/Back]** Preview of the current color combination and the RGB color parameters required to display the colors.
- Create a Custom color** The launches the [Syntax Custom Color Selector dialog](#).
- OK** Accept the current text and background color selections, and return to the Options dialog.
- Cancel** Abort the Syntax Color Selector dialog without making any changes to the color settings.



[Color map]

A color map based on the current display color-depth, to facilitate easy selection of custom colors. To choose the text color, use the left mouse button and click on the desired point in the color map. Use the right mouse button to choose the background color. The border of the color map provides a convenient set of linear (static) color gradients to assist in choosing colors that are more "pure". The current text and background combination is shown in the top-right preview control, as it will appear in the editor window.

[Scroll bar]

The scrollbar adjusts the "white saturation" level of the color map. Moving the scrollbar upward adds more whiteness to the color map colors, whereas moving the scrollbar downward reduces the whiteness of the color map colors. The default is 50% white saturation. Changing the white saturation level does not affect the gradient border around the color map.

[Preview Fore/Back]

Preview of the current color combination and the RGB color parameters required to display the colors.

[Current RGB]

Indicates the RGB color under the mouse cursor in the color map.

[Saturation]

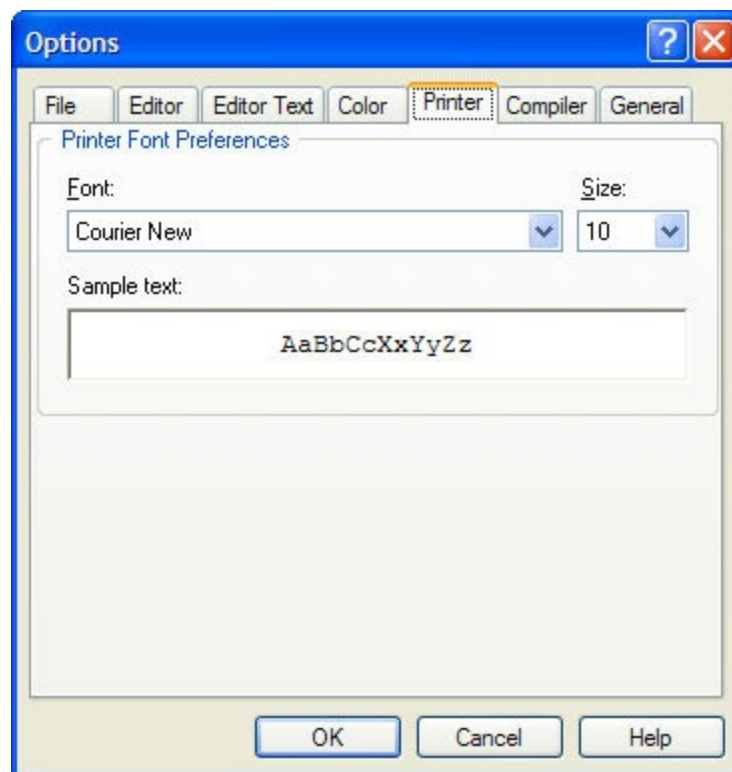
Indicates the "white saturation" level of the color map. This is adjusted with the scroll bar located to the right of the color map

OK

Accept the current custom text and background color selections, and return to the Color Selector dialog.

Cancel

Abort the Syntax Custom Color Selector dialog without making any changes to the custom color settings.



Font

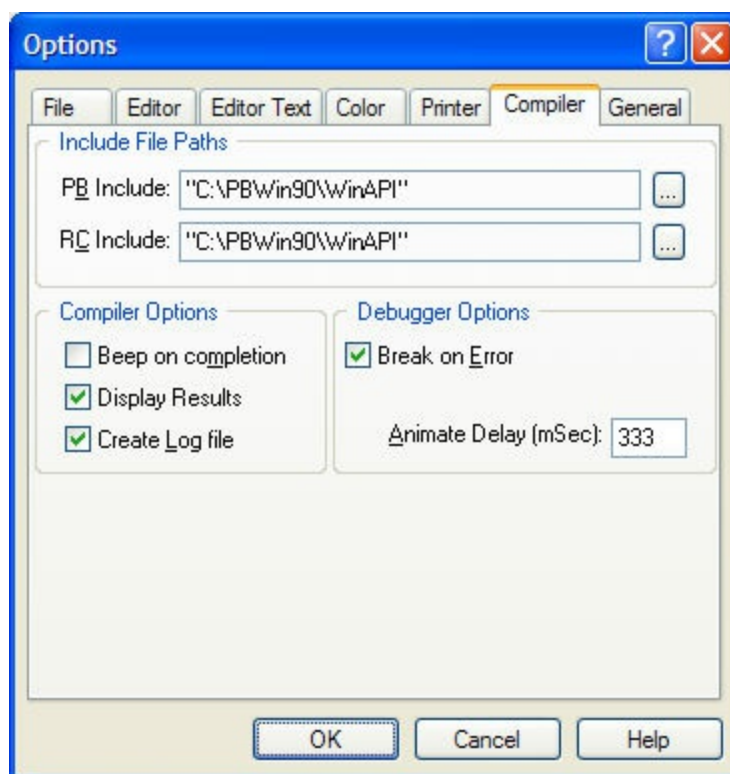
As with the [IDE](#) edit window, printing is performed with a fixed-width ANSI font. By default, this is the Courier New font. However, an alternative font may be selected from this drop-down list.

Size

The desired point size of the printed font.

Sample Text

How the printed text will appear with the selected font and at the selected font size.



Include File Paths

PB Include

The path (or paths) where the Compiler may search for source code files referenced in [#INCLUDE](#) metastatements, and [PBR files](#) referenced with [#RESOURCE](#) metastatements. Multiple paths are automatically separated with semi-colons. Use the Ellipsis button () to adjust the Include path settings - see [Browsing for Include folders](#) for more information. Note that this field behaves identically to the /I command-line compiler parameter.

RC Include

The path where the [Resource Compiler](#) may search for additional resource files, as may be referenced within a RC file. The text entered into this field is prefixed with "/I" and passed as a command-line parameter to the nominated resource compiler. Use the Ellipsis button () to adjust the RC Include path settings - see [Browsing for Include folders](#) for more information.

Compiler Options

Beep on completion

The *default system sound* is played when compilation is completed successfully. The default system sound can be changed in Control Panel.

Display results

After compilation of Classic PowerBASIC source code, a dialog can be displayed to summarize the compilation results, providing details on compiled code size, data and [string literal](#) size, etc. This dialog

must be dismissed before the compiled application can be executed or [debugged](#). If a [compile-time error](#) occurs, this results dialog is always displayed by Classic PowerBASIC.

Create log file

During compilation, a log file is created in the same directory as the [primary source file](#). The log file contains the same information as the Display Results dialog discussed above. The file is assigned the same "base name" as the main source code file, but with the extension .LOG (i.e., PROJECT1.LOG). In case of a compile-time error, this log file will contain details of the nature of the error (in addition to the compile-time error message display produced by the compiler itself).

Debugger Options

Break on Error

Causes the [debugger](#) to stop after every statement to check the error status and then automatically halt program execution when an error occurs (non-zero [ERR](#) value). In debugging with this setting enabled, programmed with larger iteration counts can reduce debugging speed to unacceptable levels. For best results, it is instead recommend to enable [#DEBUG ERROR ON](#) in your program instead of this option.

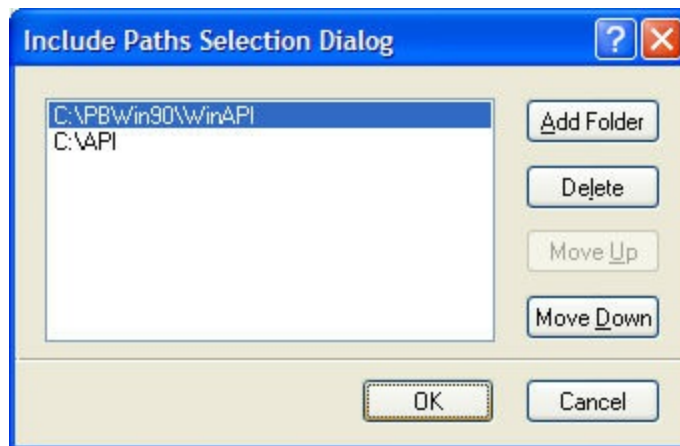
Animate Delay

The debugger's *Animate* debug mode pauses for at least the given amount of time before execution of the next line of code occurs. Animation is very useful for watching the general flow of a program. The delay is specified in milliseconds (*mSec*). The larger the delay value, the greater the delay between execution of lines of code. The default value is set for 333 milliseconds (1/3 of a second).

Browsing for Include folders

[Top](#) [Previous](#) [Next](#)

The Include Paths Selection dialog provides a simple method of creating an Include file list for the Classic PowerBASIC compiler, and the [Resource Compiler](#). The Include folder list specifies the search order that the compilers use to locate [#INCLUDE](#) and [#include](#) files. The Include Paths Selection Dialog box is launched by the Ellipsis buttons on the [Compiler Preferences](#) tab page.



Folder list

The list of folders in a drag list control. The folders appear in the order in which the compiler search for [#INCLUDE](#) (Classic PowerBASIC) or [#include](#) (Resource Compiler) files. There are two ways to rearrange the order of folders:

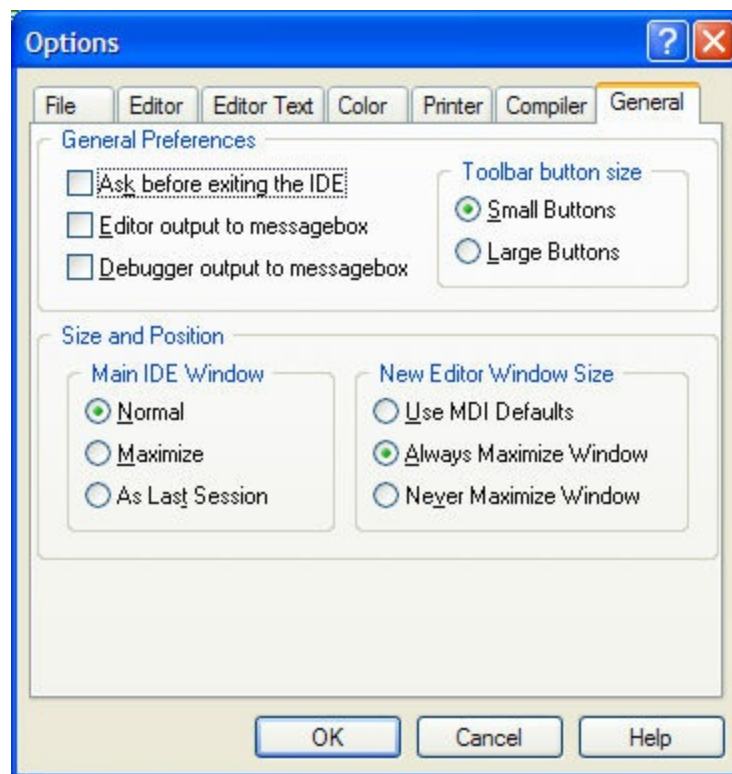
1. Click and drag the individual folder names up and down in the Folders List; or
2. Select (highlight) a folder and use the Move Up and Move Down buttons to reposition the folder in the list.

Add Folder

Launch the standard Windows "Browse for Folder" dialog, where the folder tree can be navigated. The default folder is the currently selected folder in the Folders list to the left of the Add Folder button or the current folder if none are selected. The Browse for Folder dialog looks like this:



- Delete** Delete the currently selected folder. If all folders are deleted, a new entry specifying the current folder is automatically created, ensuring at least one folder appears in the list.
- Move Up** Move the currently selected folder up one position in the Folders List, increasing the search priority of the selected folder. The compilers search the Folders List in the order they appear.
- Move Down** Move the currently selected folder up down position in the Folders List, decreasing the search priority of the selected folder. The compilers search the Folders List in the order they appear.
- OK** Accept all changes to the Folders List, and return to the Compiler Preferences dialog.
- Cancel** Cancel any changes made to the Folder List, and return to the Compiler Preferences dialog.



General Preferences

Ask before exiting When selected, a confirmation dialog will appear when the [IDE](#) is about to be closed. Canceling the dialog will prevent the IDE from closing. The IDE will always prompt to save any files that have not been saved since their last modification, regardless of whether this option is selected.

Editor output to messagebox When selected, editor output (such as error codes and compilation status) is displayed using message boxes as well as the output window.

Debugger output to messagebox When selected, [debugger](#) output (such as errors and [#DEBUG PRINT](#) information) is displayed using message boxes as well as the output window.

Toolbar Button size

Small Buttons The [IDE and debugger toolbars](#) are displayed with small buttons and icons, allowing the maximum amount of screen *real estate* for the editor windows. If changed, this option comes into effect when the IDE is next launched.

Large Buttons Complement of *Small Buttons*. Larger toolbar buttons are easier to "hit" with the mouse, but slightly reduce the amount of space available for editor windows.

Size and Position

Main IDE Window	When the IDE is launched, the initial window size is determined by this setting (Normal, Maximize, or As Last Session).
New File Window Size	When a source code file is opened, or created from scratch, the initial size of the editor window is determined by this setting (Use MDI Defaults, Always Maximize, Never Maximize),

Command Line Dialog Box

[Top](#) [Previous](#) [Next](#)

The Command Line Dialog allows the programmer to specify an arbitrary command-line parameter string that is passed to the application when the *Compile and Execute*, or *Compile and Debug* options are used.

The result can be read with [COMMAND\\$](#) within the program, for the purposes of testing the application.



- | | |
|------------------|--|
| Arguments | An arbitrary string passed to the application in the COMMAND\$ parameter. |
| OK | The text in the <i>Arguments</i> field is accepted and retained for the session. |
| Cancel | The previous command-line text, if any, is retained unaltered. |

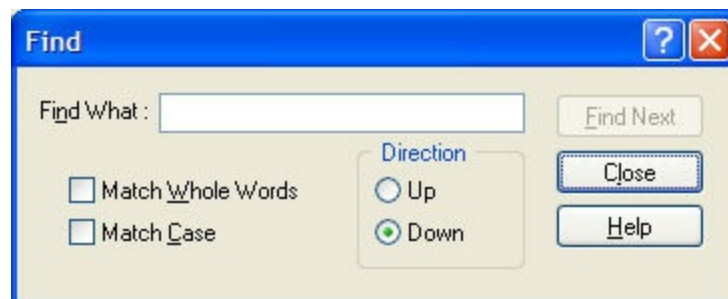
See Also

[The Integrated Development Environment](#)

Find Dialog Box

[Top](#) [Previous](#) [Next](#)

The Find Dialog Box allows you to search the currently displayed source code file for a specific phrase or word. You can limit the number of matches by specifying options such as Match Whole Words or Match Case.



Find What

Enter the phrase or word to search for. For example, searching for PRINT will locate every instance of that word in the current file. The text should be entered as it is anticipated to be formatted in the current file. For example, the number of spaces between words must match the number specified in the Find What field. Do not include quotes unless the anticipated match also includes quotes.

Match Whole

This excludes matches that occur within a word. For example, with Match Whole Words enabled, searching for LOG will not match on DIALOG, but will match on LOG(x).

Match Case

When Match Case is enabled, the Find What text must exactly match the capitalization of the word or phrase in the current file. For example, searching for Print will match Print, but not PRINT or print.

Direction

By default, searching starts at the current caret position, and moves toward the end of the current file. The Direction options allow you to specify whether the search should proceed from the caret position upward toward the top of the file, instead of downward.

Find Next

Instructs the editor to locate the next match in the current file. If no further matches are located, a notification appears. If a match is made, the matching text is highlighted in the file.

Close

Cancel the Find Dialog. After the Find Dialog has been closed, you can repeat the last Find operation by pressing the F3 key, even if you have opened or switched to a new file. However, the Find What text is not preserved between sessions of the [IDE](#).

Help

This help topic.

See Also

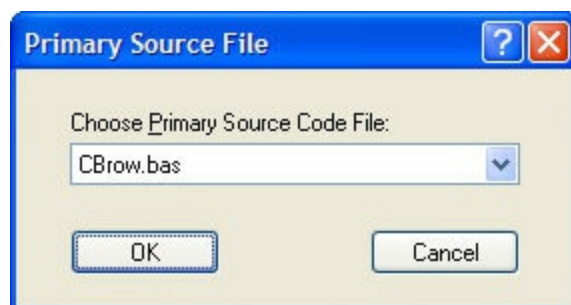
[The Integrated Development Environment](#)

Primary Source File Dialog Box

[Top](#) [Previous](#) [Next](#)

The Primary Source File Dialog allows the programmer to define which source code module is regarded as the "main" program file. That is, when a compile/execute/debug operation begins, the [IDE](#) automatically uses the Primary Source File as the "main" file, regardless of which other files are loaded or have focus in the IDE.

The Primary Source File will be one of the files loaded into the IDE, and this can be via the Recent Files list (if the *Reload previous file set at start IDE* option is enabled).



Primary Source File	The name of the file designated to be the main file to compile and/or debug, even when multiple files are open. Choose None to disable the Primary Source File usage.
OK	The name in the Primary Source File list box is accepted and retained for the session as the "main" source code file.
Cancel	The previous Primary Source File, if any, remains unaltered.

See Also

[The Integrated Development Environment](#)

Line numbers and Labels

[Top](#) [Previous](#) [Next](#)

Line numbers are in the range 1 to 65535, which serve to identify program lines. Classic PowerBASIC takes a relaxed stance toward line numbers. They can be freely interspersed with labels, and used in some parts of a program and not others. In fact, they do not even need to follow in numeric sequence. No two lines can have the same number, and no line can have both a label and a number. Line numbers are essentially labels.

While line numbers and labels serve the same purpose, their usage is slightly different. Line numbers are just a concession to compatibility with Interpretive BASIC. Line numbering can lead to bad programming style. Since the numbers themselves can be in any order, they give a false sense of structure to a program. We recommend that you avoid line numbers, and use labels instead.

Using labels instead of numbers allows you to make the flow of your program much more readable. For example:

```
GOSUB BuildQuarks
```

tells you much more than

```
GOSUB 1723
```

Each label must appear on a line by itself (though a [comment](#) may follow) and it serves to identify the statement immediately following it. Labels must begin with a letter and contain any number of letters, digits, and an underscore. Case is insignificant - *THISLABEL*, *thislabel*, and *ThisLabel* are all the same. A colon must follow a label, however, and statements that refer to the label must not include the colon.

```
MSGBOX "Now Sorting Invoices"
GOSUB SortInvoices
MSGBOX "All Done!"
EXIT FUNCTION
```

```
SortInvoices: ' This is a legal label
{sorting code goes here}
RETURN
```

The following is illegal, however:

```
ExitPoint: a = a + 1 ' a label must be on a line by itself
```

Finally, it should be noted that symbol names must be unique: a label may not share the name of any other symbol ([Sub](#) name, [Function](#) name, [Method](#) name, [Property](#) name, [user-defined type](#) or [union](#) definition, [variable](#) name, etc), and they are local to the Sub, Function, Method, or Property in which they appear.

See Also

[Long lines](#)

[Statement separation](#)

[Structured Programming](#)

[Variables](#)

The underscore character (`_`) can be used to split "logical" lines of source code, across physical lines in the source code file. The underscore character must be preceded by at least one whitespace character.

The effect of using a line continuation character is for "visual" appearance only - the compiler itself treats lines split this way as only one contiguous line of code.

For example, if we take the following line of code:

```
DECLARE FUNCTION Call32& LIB "CALL32.DLL" ALIAS "Call32" (Param1 AS ANY, BYVAL id&)
```

We could rewrite this line to place its component parts on separate lines of code for clarity:

```
DECLARE FUNCTION Call32& _  
    LIB "CALL32.DLL" _  
    ALIAS "Call32" _  
    (Param1 AS ANY, BYVAL id&)
```

The compiler treats text that appears after the line continuation character as a remark. However, we still recommend that such comments are preceded by a [REM](#) or an apostrophe (`'`) symbol to clearly distinguish remarks from the actual code.

```
DECLARE FUNCTION Call32& _      ' The prototype declaration  
    LIB "CALL32.DLL" _         ' The DLL name  
    ALIAS "Call32" _           ' The exported function name  
    (Param1 AS ANY, _         ' 1st parameter  
    BYVAL id&)                ' 2nd parameter
```

See Also

[Line numbers and Labels](#)

[Statement separation](#)

[Structured Programming](#)

[Variables](#)

Statement separation

[Top](#) [Previous](#) [Next](#)

The colon character (:) can be used to separate multiple statements on a single (logical) line of source code. For example:

```
FOR x& = 1 TO 10 : INCR y& : NEXT x&
```

is directly equivalent to:

```
FOR x& = 1 TO 10
  INCR y&
NEXT x&
```

In general, placing only one statement per line leads to more readable and maintainable source code; however, using the colon separator can be useful for combining statements on single-line [IF/THEN](#) statements, etc. For example

```
IF x! < 0 THEN INCR y# : INCR z# : DECR Count& : GOTO LastX
```

See Also

[Line numbers and Labels](#)

[Long lines](#)

[Structured Programming](#)

[Variables](#)

Variables represent numeric or string values. Unlike constants, the value of a variable can change during program execution. Like [labels](#), variable names must begin with a letter and can contain up to 255 letters and digits (although in practical terms you really cannot exceed the length of a line). Be generous in naming important variables. In Classic PowerBASIC, long variable names do not steal run-time memory.

The [Single-precision](#) variables, *EndOfMonthTotals* and *emt*, both require exactly four bytes of run-time storage. A good rule of thumb is to preserve a balance, keeping variable names short enough so that statements can fit on one line. Many programmers use single-letter variables for loop counters (i, j, k, l and x, y, z are favorites). However, you can use names like count, total, index, and so on for greater clarity, especially if you have nested loops.

Classic PowerBASIC has many built-in variable types: [Dynamic string](#); [Fixed-length string](#); [ASCII string](#); [Field](#), [Integer](#); [Long integer](#); [Quad integer](#); [Byte](#), [Word](#); [Double word](#); [Single](#); [Double](#); and [Extended floating point](#); [Currency](#) and [CurrencyX](#); [Variant](#), [Object](#), [Guid](#), plus [Pointer](#), [arrays](#), and [Bit and Sbit bitfield subtypes](#).

Declaring a variable as a specific type:

Use the [DIM](#) statement to declare a variable and use the AS *type* syntax:

```
DIM iVar AS INTEGER
```

Appending a type-specifier to the variable name:

```
bat# = 1.312 ' bat# is a Double-precision variable
hat% = 3      ' hat% is an Integer variable
DEFINT c      ' Variables beginning with c are now Integer
cats = 16     ' cats is an Integer variable by DEFINT
```

Bear in mind that *cat?*, *cat%*, *cat&*, *cat&&*, *cat!*, *cat#*, *cat###*, *cat@*, *cat@@*, and *cat\$* are ten separate variables. Although using *cat* over and over again to create different variables like this is legal, good programming practice suggests that you use somewhat different names for different variables. It is also much better to use descriptive and more easily understood names for your variables rather than single letters. It's extremely difficult to [debug](#) a program in which *x@* has been entered instead of *x!* or *x#*. Imagine the confusion of trying to distinguish *x&&* and *x&*. If you had used variable names like *count!*, *result#*, *remain###*, and *company\$*, you would have had considerably less trouble keeping your variables (and their types) apart.

See Also

[Default Variable Typing](#)

[Variable Scope](#)

THREADED variables

LOCAL, GLOBAL and STATIC considerations

INSTANCE statement

For most applications, good programmers use an organized approach to programming called *structured programming*. The original interpreted BASICs did not really support this kind of programming. However, Classic PowerBASIC, with its control structures and more advanced [functions](#), [subroutines](#), [methods](#), and [properties](#), is very well suited to structured programming style.

Structured programming is based on the theory that modularization makes for better programs. Modularization means grouping statements together (making modules) that have some relation to each other. In other words, you break up your program into logical functional sections. This makes it easier to write, [debug](#), and understand the program.

Ideally, modules should be no more than a page long. This seemingly arbitrary constraint makes it easier to absorb the entire module at a glance. It is easier to understand a series of ten single-page modules than it is a single ten-page program.

For some projects, after this initial breakup, you're ready to write the program. More complicated problems might require you to break the modules into subsidiary pieces. This process continues until you have refined the material enough so that you can write the code that corresponds to your ideas. This entire process is often described in books as "top-down design", since you start with a general description and work toward a more specific one.

Once you have the logical organization, you can start to design the overall structure of your program. For short, simple programs, these steps may only take a few minutes. For complex programs, it could take months.

To summarize the steps of structured programming (also known as 'top-down programming' or 'top-down design'):

1. Plan your program on paper. Ask yourself the following questions:
 - a. What is the overall purpose of the program?
 - b. What kind of input will it need?
 - c. How will it process that input?
 - d. What kind of output will the program produce? To where (screen, printer, disk)?
 - e. How should the input and output look?
 - f. How can the program be broken up into discrete processes (modules?)
 - g. How will those modules fit into the main program, and how will they communicate?
 - h. Can those modules be broken up into even smaller functional segments?
2. Next, write your main program. Don't worry about writing the individual modules that you separated out earlier. Instead, write *stubs*: Dummy statements that allow the main program to continue. This allows you to test the logic of your main program.
3. Finally (and this step will actually be several steps), write the modules *one at a*

time. Test and debug each module thoroughly before proceeding to the next. If you've broken your module into even smaller processes, write the code for those processes *first*, test and debug each process, and then put them together to build your module.

See Also

[Line numbers and Labels](#)

[Long lines](#)

[Statement separation](#)

[Variables](#)

[Debugging PB/Win Programs](#)

What is a DLL?

[Top](#) [Previous](#) [Next](#)

A Dynamic Link Library (DLL) is a Windows executable library module containing one or more [Subs](#), [Functions](#), or [classes](#) that can be called by executables or other DLLs. Unlike executables, DLLs do not have a single entry point. Instead, like libraries, DLLs have multiple entry points, one for each exported Sub, Function, or classes.

To get a better idea of how a DLL works, it helps to understand the difference between static and dynamic linking. Static linking is the process of writing one or more modules, and then linking them, along with whatever other run-time, third-party, etc., libraries that may be needed to create a complete, stand-alone executable program. When a program uses a Sub or Function from a static-link library, a copy of that Sub or Functions code is statically linked into the programs executable file.

If two programs that are running concurrently use the same routine from a library, they would each have their own copy of that routine. It would be more efficient if the two programs could share a single copy of the routine. DLLs provide that capability by resolving your application's references to external procedures at run-time.

In contrast to a static-link library, the code in a DLL is not linked into a program that uses the DLL. Instead, a DLLs code and resources are in a separate executable file, usually with a .DLL extension. This file must be present when the application runs. You will still have to write one or more modules to implement the functions that are specific to your application.

However, the linking process is divided into two stages. You first place [DECLARE](#) statements into your application to temporarily satisfy the references your program makes to the DLL services, in order to create an EXE (or DLL) file. The second stage happens at run-time, when your program calls one of the DLLs services.

At that time, the Function calls in the program are dynamically linked to their entry points in the DLL(s). The operating system resolves external references by establishing a link between the application calls and the code, in the DLL, that implement the required functions. The Windows environment supports both static and dynamic linking.

See Also

[Why use Dlls?](#)

[Creating a Dynamic Link Library](#)

[Private and Exported Procedures](#)

[Dll example](#)

[LibMain](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

There are a number of mitigating reasons to create a DLL. Among them are:

- **Performance**

Parts of your code, while functional, might not execute as fast as you would like. Once you've isolated the bottleneck area(s), a machine code DLL is an obvious choice for optimizing just those areas of your application that are running too slowly.

- **Resources**

Unlike conventional libraries, when a DLL is loaded into memory by the operating system, its Subs and Functions are accessible by all other programs (or DLLs). Only one copy of the DLL needs to be present in memory. This is possible because the library is not linked into any one of the programs permanently. It is present, in memory, making its services available to any program (or other DLL) which may need them.

- **Code re-use**

You might have a set of procedures that are common to a number of different applications. Instead of having those procedures appear in every application that needs them, it is better to put them in a DLL where they can be accessed by all the applications. This reduces the size of your executables while giving you the flexibility of updating the DLL itself, without having to re-compile every application that uses its services.

- **Maintenance**

A DLL can be updated and redistributed without having to re-compile any of the applications (or other DLLs) that use its services.

See Also

[What is a DLL?](#)

[Creating a Dynamic Link Library](#)

[Private and Exported Procedures](#)

[Dll example](#)

[LibMain](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

A [DLL](#) contains one or more exported [Classes](#), [Subs](#), or [Functions](#) that may be called by applications or other DLLs. A DLL may also contain any number of private Subs or Functions that can only be called from within the library. Creating a DLL with Classic PowerBASIC is straightforward. Below are the steps to follow to convert parts of a Visual Basic program to a DLL.

- Step 1:** The first step is to identify the sections of your application that are used in multiple programs, or in the case of Visual Basic, Subs and Functions that you need to execute faster.
- Step 2:** Save those Subs and Functions as text, and change the file extension to .BAS. This will become the source module that will be compiled into a DLL with Classic PowerBASIC. You could also create the source file from scratch, if you so wish.
- Step 3:** Launch PBEDIT.EXE (the Classic PowerBASIC IDE) and add the EXPORT keyword to any Sub or Function in the DLL source code (that you wish to be made accessible to external applications). Add [#COMPILE DLL](#) to the top of the source code file, and make any other changes to your .BAS source module. See the [SUB/END SUB](#) and [FUNCTION/END FUNCTION](#) topics for more information on the exact syntax.
- Step 4:** Click the [compile button](#) on the Classic PowerBASIC IDE toolbar.

Any [compile-time errors](#) will be flagged at this point. Repeat steps 3 to 4 above until no more errors are reported. You are then ready to start testing and debugging your DLL. Debugging is done using the Classic PowerBASIC symbolic Debugger built into the Classic PowerBASIC IDE (PBEDIT.EXE). See the section on [Debugging](#) for more information.

See Also

[What is a DLL?](#)

[Private and Exported Procedures](#)

[Dll example](#)

[LibMain](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

Private and Exported Procedures

[Top](#) [Previous](#) [Next](#)

There are two basic types of procedures in a [DLL](#): private and exported. Exported procedures are those which are made available to applications and other DLLs. Private, or local, procedures are support-type routines, accessible only from within the DLL.

In the following example, the first procedure defines an exported [Sub](#) that accepts two arguments: a [string](#) and an [Integer](#). The second procedure defines an exported function that accepts a single string argument, and returns an Integer. Finally, the third procedure defines a private Sub that accepts a single Integer argument. The first two routines are callable from an external .EXE or another DLL. The third one is not.

```
#COMPILE DLL
SUB MySub (sArg AS STRING, BYVAL iArg AS INTEGER) EXPORT
    ' Body goes in here
END SUB
FUNCTION MyFunc (sArg AS STRING) EXPORT AS INTEGER
    ' Body goes in here
END FUNCTION
SUB MyPrivateSub(BYVAL iArg AS INTEGER)
    ' Body goes in here
END SUB
```

Alternatively, you may specifically declare Subs and Functions as private, by using the PRIVATE keyword:

```
SUB MyPrivateSub(BYVAL iArg AS INTEGER) PRIVATE
    ' Body goes in here
END SUB
```

See Also

[What is a Dll?](#)

[Creating a Dynamic Link Library](#)

[Dll example](#)

[LibMain](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

A very simple example is a [DLL](#) with a [function](#) that will add one to any [Long-integer](#) passed to it as a parameter:

```
#COMPILE DLL
FUNCTION AddOne ALIAS "AddOne" (BYVAL x AS LONG) EXPORT AS LONG
    AddOne = x + 1
END FUNCTION
```

The ALIAS keyword is used to indicate the capitalization that Classic PowerBASIC will assign the function. In Win32, all exported (and imported) [Sub](#) and Function names are case-sensitive. If the ALIAS keyword was omitted, Classic PowerBASIC will capitalize the exported name and this could cause "Missing DLL entry point" errors if the calling code did not match the capitalization exactly.

By default, all Subs and Functions in Classic PowerBASIC are private, which means they cannot be seen outside of the DLL. The EXPORT keyword is used on the Sub or Function definition line to indicate that the routine is to be exported, i.e., made accessible to applications and other DLLs.

When compiled into a DLL, *AddOne* is visible to outside applications. A Visual Basic program needs only include a prototype, or a [DECLARE](#) statement for the function, in order to call it as if it were a VB function:

```
DECLARE FUNCTION AddOne LIB "ADDONE.DLL" ALIAS "ADDONE" (BYVAL x&) AS LONG
```

AddOne is then accessible from within your Visual Basic code:

```
a& = 4
b& = AddOne( a& ) ' returns 5
```

If *AddOne* were not exported, Visual Basic would generate a [run-time error](#) when the example code attempts to call it.

If the EXPORT keyword is not used in the Sub or Function definition, the procedure will not be visible to outside applications. See the Visual Basic documentation for more information on calling DLLs from within Visual Basic code.

By using the ALIAS keyword in the DLL source code, you can have Classic PowerBASIC export the Sub or Function using any capitalization you want. You can use the ALIAS clause to export the Sub or Function with a completely different name, in order to enhance or disguise the internal Sub or Function name:

```
' Exported as "ADDONE1"
FUNCTION AddOne1 (BYVAL x&) EXPORT AS LONG

' Exported as "AddOne2"
FUNCTION AddOne2 ALIAS "AddOne2" (BYVAL x&) EXPORT AS LONG

' Exported as "ExprtFnctn1"
FUNCTION AddOne3 ALIAS "ExprtFnctn1" (BYVAL x&) EXPORT AS LONG
```

Because the name after the ALIAS keyword is in quotes, the compiler will not convert it to upper case. Note that the name in the ALIAS clause is the name that you would use to access the Sub or Function from Visual Basic. Likewise, when importing Subs and

Functions from external DLLs into Classic PowerBASIC, the ALIAS clause must exactly match the capitalization of the exported name in the DLL.

See Also

[What is a Dll?](#)

[Creating a Dynamic Link Library](#)

[Private and Exported Procedures](#)

[LibMain](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

In addition to the functions you want to export (plus any supporting private routines), a [DLL](#) can contain an optional function called [LIBMAIN](#) (or its synonyms [DLLMAIN](#) and [PBLIBMAIN](#)). Windows calls LIBMAIN when a DLL is loaded into and unloaded from memory by an application. The use of LIBMAIN in your code is optional.

See Also

[What is a Dll?](#)

[Creating a Dynamic Link Library](#)

[Private and Exported Procedures](#)

[Dll example](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

Debugging Classic PowerBASIC Programs

[Top](#) [Previous](#)
[Next](#)

Once your code is written, the next step is to test it to make sure it performs according to specifications. Regardless of the computer language used, certain programming [errors](#) are common: misspelled or misused [variables](#), inverted logical tests, mistakes in syntax, and "reasonable" tests that cause disastrous failures when unreasonable data is supplied. Each language also has its own common errors, unique because of the peculiarities of its language.

Some of BASIC's unique problems include the free conversion of most numeric data-types, side effects of [global](#) variables, default data types, and overuse of [GOTO](#) causing problems with incorrect branching. These are well known to the experienced BASIC programmer but are not generally found in other languages.

The [Classic PowerBASIC Integrated Development Environment](#) (PBEDIT.EXE) can be used to find, and correct, both general programming errors and errors specific to BASIC. Nearly every program has bugs at least at first. To find them, you may need to check any statement in the program, display the value of any variable, and observe the program flow from line to line. PBEDIT has all these capabilities and more.

This section explains how to use PBEDIT to find and fix errors in a sample program, by providing a list of the debugging commands, a description of each, and then showing how each is invoked. If you follow certain guidelines when creating your program, you will find debugging easier (and less necessary). The procedures we describe here will help you form your own set of guidelines that will make your programs easier to write and maintain.

See Also

[How the integrated debugger works](#)

[The DEBUG Menu](#)

[Debugging a simple program](#)

[The Integrated Development Environment](#)

How the integrated debugger works

[Top](#) [Previous](#) [Next](#)

The integrated debugger works in conjunction with the Classic PowerBASIC editor and is a part of the Classic PowerBASIC environment. The debugger allows you to debug at the Classic PowerBASIC level rather than at the machine level. That makes it a source-level debugger.

To debug a Classic PowerBASIC program using the integrated debugger, first load the program into the editor and choose *Compile and Debug* from the [toolbar](#) or menu. Your program will be compiled, and if there are no compile-time errors, it will begin executing.

Breakpoints are places where the program will stop. In most cases, you will want to set one or more breakpoints in your program. The program executes up to (but not including) the line containing the breakpoint and then passes control of the debugger over to you. Breakpoints that you set remain in place until you clear them or exit the [IDE](#).

Once at a breakpoint you can:

- Display the value of a [variable](#) (with the Evaluate Variable button or [menu item](#))
- Set up a list of variables (in the Variable Watch window) and see how their values change as the program executes
- Clear breakpoints, set new ones, or both
- Single-step the program (run it one line at a time)
- Run the program to the next breakpoint

See Also

[Debugging PB/Win Programs](#)

[The DEBUG Menu](#)

[Debugging a simple program](#)

The DEBUG Menu

[Top](#) [Previous](#) [Next](#)

The Debug Menu provides the essential tools for [debugging](#) a Classic PowerBASIC program. We will run through these in their order of appearance:

Run	F5
Run to Caret	Ctrl+F8
Animate	Ctrl+F5
Stop	
Step Into	F8
Step Over	Shift+F8
Step Out	Shift+Ctrl+F8
Evaluate Variable	
Clear all Watches	
Toggle Breakpoint	F9
Clear all Breakpoints	
Watch CPU Registers	
✓ Variable watch window	
Program Restart	Shift+F5
Exit Debugging	

- Run** Begin running the program. It will continue to run until the [debugger](#) either encounters a breakpoint, or runs out of code to execute. F5 is the hot-key for the Run option.
- Run to Caret** Begin running the program. It continues to run until the debugger either reaches the current line, or encounters a [breakpoint](#), etc. CTRL+F8 is the hot-key for the Run to Caret option.
- Animate** The debugger runs the program using an automated Step-Into technique. Execution continues until a breakpoint is reached, the Stop button is pressed, or the program completes. The Animate delay can be set through the [IDE's Options Dialog](#).
- Stop** Halt the debugger. If the debugger is already halted, this has no effect.
- Step Into** If the current line contains a call to a [Sub](#), [Function](#), [Method](#), or [Property](#), the debugger traces execution into that procedure. You cannot step into an API call, or into an external module. F8 is the Step Into hot-key.
- Step Over** The debugger executes the current line of code. If the line contains a reference to a Sub, Function, Method, or Property, the debugger executes that code without tracing into the procedure. SHIFT+F8 is the Step Over hot-key.
- Step Out** The debugger runs the code until the current Sub, Function, Method, or Property exits. If the current function is [PBMAIN](#) or [WINMAIN](#), the code is executed until the program is finished or another breakpoint is encountered. CTRL+SHIFT+F8 is the Step Out hot-key.

Evaluate Variable	Evaluate or modify a variable , or add/remove a variable in the Watch window. It is not possible to use this to change the length of a string. Also see Watch CPU Registers.
Clear all Watches	Remove all variables from the Watch window.
Toggle Breakpoint	Set or release a breakpoint on the current line. F9 is the Toggle Breakpoint hot-key.
Clear all Breakpoints	Release all breakpoints in the program.
Watch CPU registers	Show or hide the Register Watcher window, which lets you see the state of the CPU registers and flags when debugging.
Variable watch window	Show or hide the Variable Watcher window, which lets you see the state of the ERR function and any variables you choose to watch when debugging.
Program Reset	If the current program is halted/stopped, the program will be reset, ready for debugging to commence again. SHIFT+F5 is the Reset hot-key.
Exit Debugger	Halts the current program and terminates the debugger. The variable list in the Watch window is retained between debugging sessions, until the IDE is closed.

See Also

[Debugging PB/Win Programs](#)

[How the integrated debugger works](#)

[Debugging a simple program](#)

Debugging a simple program

[Top](#) [Previous](#) [Next](#)

For our first example, we'll use a simple program designed to read a text file and display it. Along the way, the program counts the number of words and tabulates the lengths of all words found - how many words are one character long, how many are exactly two characters long, and so on. The sample program, TWORD.BAS (\PBWin90\Samples\TWord\TWORD.BAS), contains a number of bugs; you will be using the [Classic PowerBASIC integrated debugger](#) to find each of them.

Be sure to make copies of the TWORD.DAT data file; TWORD.BAS reads that file and makes specific errors because of the data. While another data file may work as well, it is possible that one or more of the bugs will not occur if you use a different data file.

Here is a [listing of the TWORD.BAS program](#).

When you have loaded TWORD.BAS into the editor, click on the [Debugger button](#) on the [toolbar](#), or select [Run from the menu](#), then [Compile and Debug](#).

At this point, the debugger will have scrolled the program and highlighted the line containing the definition of the [variable](#) *MaxWordLen*, since that will be the first line executed when the program begins to run. The highlight is called the *execution bar* and marks the line of code at the execution position. In other words, that line will be executed next.

To make the program run, click on the Run button in the toolbar or press [F5](#). The program's output appears in the User screen, which allows you to see how the program would look if you weren't using the debugger. If the User screen is not visible you may have to select it by using the Windows Taskbar, ALT+TAB, or by re-sizing the [Classic PowerBASIC IDE](#) to a smaller size and different location until the User screen is visible. TWORD prompts you for the name of the file to read. Enter TWORD.DAT and press ENTER. TWORD displays the first line of the file then locks up because of one of the bugs in the program. To regain control, click on the [Stop button](#). You can choose [Program Reset](#) (or press the [SHIFT+F5](#) hot-key) to quit running the flawed program. Clicking the Run button lets you restart the program.

Next See: [Setting and using breakpoints](#)

See Also

[Debugging PB/Win Programs](#)

[How the integrated debugger works](#)

[The DEBUG Menu](#)

[The Integrated Development Environment](#)


```
'=====
'
'  Test Word example (TWORD.BAS)
'  Copyright (c) 1998-2006 Classic PowerBASIC, Inc.
'  All Rights Reserved.
'
'  Read a text file and count the number of words of length
'  1, 2, 3, and so on. This program contains intentional
'  bugs. Use it in conjunction with the Classic PowerBASIC Users
'  Guide to learn the Classic PowerBASIC integrated debugger.
'
'=====

#IF NOT %DEF(%WINAPI)
  DECLARE FUNCTION GetModuleFileName LIB "KERNEL32.DLL" _
    ALIAS "GetModuleFileNameA" (BYVAL hModule AS DWORD, _
      lpFileName AS ASCIIZ, BYVAL nSize AS LONG) AS LONG
#ENDIF

DEFLng A-Z

FUNCTION AppPath() AS STRING

  LOCAL p  AS ASCIIZ * 256
  LOCAL ix AS LONG

  GetModuleFileName 0, p, SIZEOF(p)

  FOR ix = LEN(p) TO 1 STEP -1
    IF MID$(p, ix, 1) = "\" OR MID$(p, ix, 1) = "/" THEN
      FUNCTION = LEFT$(p, ix)
      EXIT FUNCTION
    END IF
  NEXT

  FUNCTION = ""

END FUNCTION

FUNCTION PBMAIN() AS LONG

  MaxWordLen = 16          'count words up to 16 chars
                           'longer words go into Overlong
  DIM WordLength(MaxWordLen) 'the array used to hold counts
  Blank$ = CHR$(32)        'a space marks end of a word

  FilePath$ = AppPath
  IF LEN(FilePath$) THEN
    CHDRIVE FilePath$
    CHDIR FilePath$
  END IF

  PRINT "Warning: This is a program intended for use in a"
  PRINT "practice session of the Classic PowerBASIC debugger."
  PRINT "If you are NOT running this program in the "
  PRINT "integrated debugging environment, press CTRL+BREAK"
  PRINT "now!"
  PRINT "See the Debugging chapter in the Classic PowerBASIC Users"
```

```
PRINT "Guide for details."
PRINT
```

```
WHILE InFile$ = ""
    LINE INPUT "Enter the name of the input file: ";InFile$
    IF InFile$ <= SPACE$(LEN(InFile$)) THEN InFile$=""
    IF InFile$ = "" THEN
        BEEP
        PRINT "TWORD canceled!"
        EXIT FUNCTION
    END IF
WEND
```

```
OPEN InFile$ FOR INPUT AS #1
```

```
'If the file can't be opened, give user an error message.
IF ERR THEN
    PRINT "Unable to open file "; InFile$
    EXIT FUNCTION
END IF
```

```
WHILE NOT (EOF(1))          'read file until nothing left
    LINE INPUT #1,FirstString$ 'get a line
    PRINT FirstString$       'display it
    WHILE FirstString$<>""
        GOSUB GetAWord       'pull word for FirstString$
                                'put it in SecondString$

        Test = LEN(SecondString$)
        IF Test <= 16 THEN
            WordLength(Test) = WordLength(Test) + 1
        ELSE
            Overlong = Overlong + 1
        END IF
    WEND
WEND
```

```
CLOSE 1
```

```
PRINT "Length Count"
FOR Count% = 1 TO 16
    PRINT Count%, WordLength(Count%)
NEXT Count%
PRINT "Greater"; OverLong
```

```
WAITKEY$
```

```
EXIT FUNCTION
```

```
GetAWord:
```

```
position = INSTR(FirstString$, Blank$) 'a word is a
    ' sequence of characters ended by a blank or
    ' the end of the line
IF position = 0 THEN
    'the word is the remainder of the line
    SecondString$ = FirstString$
    FirstString$ = ""
ELSE
    'pull the word from the line
    SecondString$ = LEFT$(FirstString$, position - 1)
END IF
RETURN
```

END FUNCTION

We know the program did not fail within the first few lines; it requested the name of the input file, and it successfully opened that file. Therefore, the problem must have been caused by something further along in the code.

The first suspicious line concerns the *GetAWord* [subroutine](#). Set a breakpoint at the line reading:

```
position = INSTR(FirstString$, Blank$)
```

Use the arrow keys to move to that line. As you do, you'll notice that the execution bar doesn't move. That is because you are not executing the program; you are just using the source browser to move within the program source.

To set a breakpoint at the line to which you moved, double-click on it or press the F9 key. The line is highlighted, indicating that the breakpoint has been set. If you wanted to remove the breakpoint at that line, double-click on it or press the F9 key again. The breakpoint highlighting differs from the execution highlighting, and this difference helps you to avoid confusion over highlighted breakpoints and the current program position.

Once more, click on the [Run button](#) (or press F5). TWORD starts running again, and things happen just as before, with one exception: after the first line from the data file has been displayed on the User screen, TWORD halts and waits for further commands. Classic PowerBASIC has reached the breakpoint. The caret and the execution bar are on the line containing the breakpoint.

The breakpoint line cannot be doubly highlighted, so the execution bar obscures the breakpoint highlighting until the program executes further. The program stops each time it reaches the breakpoint line.

You can also stop a program running within the debugger by clicking the Stop button. When you do this, the program executes the current line and stops at the beginning of the next. Control is then returned to the Classic PowerBASIC debugger, and the execution bar highlights the next line to be executed. You may now use debugger commands to step through the program or resume execution.

Next See: [Tracing execution](#)

See Also

[Debugging PB/Win Programs](#)

[How the integrated debugger works](#)

[The DEBUG Menu](#)

[Debugging a simple program](#)

Now that you have executed [TWORD](#) to the first breakpoint, you can trace the execution one line at a time by pressing F8 or by clicking on the Step-Into button. When you press F8, the [debugger](#) runs the execution line and stops at the beginning of the next line.

Perhaps there is something wrong with that [INSTR](#) function call? You could not check the value of the variable *position* while the [breakpoint](#) line was highlighted, because the breakpoint line had not yet executed. After pressing F8 and executing the breakpoint line, the value of the [variable](#) *position* should be known. The value of *position* is critically important, so let's check it.

Next See: [Evaluating a variable](#)

See Also

[Debugging PB/Win Programs](#)

[How the integrated debugger works](#)

[The DEBUG Menu](#)

[Debugging a simple program](#)

Evaluating a variable

[Top](#) [Previous](#) [Next](#)

To see the value of *position*, click on the [Evaluate toolbar button](#), or right-click on the variable and choose the Evaluate variable item from the context-menu - since this is a context-sensitive menu, it will actually read "Evaluate position". In either case, the opens the Evaluate dialog containing two data entry fields. The [variable](#) name *position* should be automatically filled in the Variable Name field, and the content of the variable shown in the *Value* field. In this case, *position* is expected to contain 3, and it does. There is no error so far. To return from the pop-up window to the main part of the debugger, click on the Close button or press ESCAPE.

The next few lines are supposed to remove the word from the beginning of the line and put the word into *SecondString\$*. To check if that routine functions correctly, you should examine the values of *FirstString\$* and *SecondString\$* before and after the routine alters them. The debugger should display:

```
To be or not to be; that is the question.
```

From the appearance of the string, you can see that the first blank should appear in position 3, and that the program has correctly determined that position. The variable *SecondString\$* ought to contain the last word processed and should have no value yet. You can check that by entering *SecondString\$* in the *Variable Name* field; if you do, you will find no error.

Everything seems normal so far. Press ESCAPE to return to the main part of the debugger, then press F8 to step the program one more line. Since you already know *position* is not 0, you'll find out what you need to know by pressing F8 several more times, stopping when the execution bar is over the final [END IF](#) of that routine. At that point, both *FirstString\$* and *SecondString\$* have been processed.

Once more, evaluate *SecondString\$*. This time, *SecondString\$* does contain data: the word "To". This seems correct. When you ask to see the value of *FirstString\$*, though, you get a surprise: *FirstString\$* has not been changed at all! This explains the lockup for the first line; subroutine *GetAWord* was correctly supplying the first word, but was not removing that first word from the entry string. Therefore, the first line never actually became shorter, so it was being processed and reprocessed endlessly.

To correct the bug, exit the debugger and insert a routine in the editor that shortens *FirstString\$* by the length of *SecondString\$*. Insert a line reading something like:

```
FirstString$ = MID$(FirstString$, position + 1)
```

immediately before the END IF in *GetAWord*. Make this correction, then save it. Click on the *Compile and Debug* icon in the toolbar and run the program through the debugger again.

Next See: [Summary](#)

See Also

[Debugging PB/Win Programs](#)

[Debugging a simple program](#)

[Setting and using breakpoints](#)

[Tracing execution](#)

Debugging TWORD.BAS Summary

[Top](#) [Previous](#) [Next](#)

TWORD fails because the program goes into an infinite loop. The infinite loop was caused by the fact that the number of characters removed was not shortening the input .

While tracking down this bug, you learned to:

- Set and use [breakpoints](#)
- Run a program without stopping at each line
- Step through your source one line at a time
- [Evaluate](#) the values of variables

See Also

[Debugging PB/Win Programs](#)

[How the integrated debugger works](#)

[The DEBUG Menu](#)

[Debugging a simple program](#)

The care of numbers constitutes an important part of every programming system. Fortunately, Classic PowerBASIC allows you to ignore most technical considerations about internal number handling. If you never give a thought to such matters as calculation speed, precision, and memory requirements, your programs will usually continue to work as you expect. However, an understanding of the underlying issues will help when you need to write programs that are faster, more accurate, and require less memory.

For efficiency, Classic PowerBASIC stores and processes data in different forms. It supports eleven unique numeric types, three string types, and also pointers. The following tables summarize the most important features and distinctions of these data types. The rest of this section explains these features in detail.

Numeric Data storage requirements and ranges

Data Type	Size	Decimal Range	Binary Range
Integer	16 bits (2 bytes), signed	-32,768 to 32,767	-2 ¹⁵ to 2 ¹⁵ -1
Long-integer	32 bits (4 bytes), signed	-2,147,483,648 to 2,147,483,647	-2 ³¹ to 2 ³¹ -1
Quad-integer	64 bits (8 bytes), signed	-9.22*10 ¹⁸ to +9.22*10 ¹⁸	-2 ⁶³ to 2 ⁶³ -1
Byte	8 bits (1 byte), unsigned	0 to 255	0 to 2 ⁸ -1
Word	16 bits (2 bytes), unsigned	0 to 65,535	0 to 2 ¹⁶ -1
Double-word	32 bits (4 bytes), unsigned	0 to 4,294,967,295	0 to 2 ³² -1
Single-precision	32 bits (4 bytes)	8.43*10 ⁻³⁷ to 3.40*10 ³⁸	
Double-precision	64 bits (8 bytes)	4.19*10 ⁻³⁰⁷ to 1.79*10 ³⁰⁸	
Extended-precision	80 bits (10 bytes)	3.4*10 ⁻⁴⁹³² to 1.2*10 ⁴⁹³²	
Currency	64 bits (8 bytes)	-9.22*10 ¹⁴ to +9.22*10 ¹⁴	
Extended-currency	64 bits (8 bytes)	-9.22*10 ¹⁶ to +9.22*10 ¹⁶	
Variant	128 bits (16 bytes)	{data-dependent}	{data-dependent}

Variable type-specifiers and keywords

Variable type	Type specifier	Element size	DEF type	Type keyword
Pointer	N/A	4	N/A	PTR/POINTER
Integer	%	2	DEFINT	INTEGER
Long-integer	&	4	DEFLNG	LONG
Quad-integer	&&	8	DEFQUD	QUAD
Byte	?	1	DEFBYT	BYTE
Word	??	2	DEFWRD	WORD

Double-word	???	4	DEFDWD	DWORD
Single-precision	!	4	DEFSNG	SINGLE
Double-precision	#	8	DEFDBL	DOUBLE
Extended-precision	##	10	DEFEXT	EXT/EXTENDED
Currency	@	8	DEFCUR	CUR/CURRENCY
Extended-currency	@@	8	DEFCUX	CUX/CURRENCYX
String	\$	4	DEFSTR	STRING
Fixed-length string	N/A	N/A	N/A	STRING * x
ASCIIZ string	N/A	N/A	N/A	ASCIIZ, ASCIZ
FIELD string	\$	16	N/A	FIELD
Variant	N/A	16	N/A	VARIANT
GUID	N/A	16	N/A	GUID
IAUTOMATION	N/A	4	N/A	IAUTOMATION
IDISPATCH	N/A	4	N/A	IDISPATCH
IUNKNOWN	N/A	4	N/A	IUNKNOWN

Byte (?)

[Top](#) [Previous](#) [Next](#)

Bytes are 8-bit (1 byte) unsigned integers ranging in value from 0 to 255 (0 to 2^8-1). The type-specifier character for a Byte is: ?.

Byte variables are identified by following the variable name with a question mark (i.e., *var?*), or by using the [DEFBYT](#) statement as described in the previous discussion of Integers. You can also declare Byte variables using the BYTE keyword with the [DIM](#) statement. For example:

```
DIM I AS BYTE
```

Byte variables are particularly useful for storing small, unsigned integer quantities like the number of days in a month. You should not use Byte variables in [FOR/NEXT](#) loops, as they are highly inefficient.

A Classic PowerBASIC Byte variable is equivalent to a bool data type (in lowercase) used by most modern C compilers. A bool is a non-traditional 8-bit unsigned data type, whereas a BOOL data type (in capital letters) is equivalent to a [Long-integer](#) in Classic PowerBASIC. Be aware that some older C compilers may freely interchange bool and BOOL keywords.

A Delphi byte is equivalent to a Classic PowerBASIC Byte.

See Also

[Double-word \(???\)](#)

[Integers \(%\)](#)

[Long integers \(&\)](#)

[Quad integers \(&&\)](#)

[Word \(??\)](#)

Word (??)

[Top](#) [Previous](#) [Next](#)

Words are 16-bit (two [byte](#)) unsigned integers with a range of 0 to 65535 (0 to $2^{16}-1$). The [type-specifier](#) character for a Word is: ??.

Word [variables](#) are identified by following the variable name with two question marks (i.e., *var??*), or by using the [DEFWRD](#) statement as described in the previous discussion of Integers. You can also declare word variables using the WORD keyword with the [DIM](#) statement. For example:

```
DIM I AS WORD
```

Word values effectively extend the positive range for Integer, but still only require two bytes for storage.

A C/C++ UINT16 and a Delphi word are equivalent to a Classic PowerBASIC Word.

See Also

[Byte \(?\)](#)

[Double-word \(???\)](#)

[Integers \(%\)](#)

[Long integers \(&\)](#)

[Quad integers \(&&\)](#)

Integers (%)

[Top](#) [Previous](#) [Next](#)

To Classic PowerBASIC, an Integer is a number with no decimal point (what mathematicians would call whole numbers) with a range of -32,768 to +32,767 (-2^{15} to $2^{15} - 1$). These values stem from the underlying 16-bit representation of an Integer: 32,768 is 2^{15} , and are therefore 2 bytes (16-bits) wide. The type-specifier character for Integer is: %.

Integers are identified by following the variable name with a percent sign (eg: *var%*), or by using the [DEFINT](#) statement. For example, if you use this declaration in your program code:

```
DEFINT I, J, K
```

then all [variables](#) following this declaration that start with the letter I, J, or K will be an Integer by default. You can also declare an Integer variable using the INTEGER keyword with the [DIM](#) statement. For example:

```
DIM I AS INTEGER
```

A C/C++ short variable and a Delphi smallint are both equivalent to a Classic PowerBASIC Integer.

See Also

[Byte \(?\)](#)

[Double-word \(???\)](#)

[Long integers \(&\)](#)

[Quad integers \(&&\)](#)

[Word \(??\)](#)

Double-word (???)

[Top](#) [Previous](#) [Next](#)

Double-words are 32-bit (four [byte](#)) unsigned integers with a range of 0 to 4,294,967,295 (0 to $2^{32}-1$). The [type-specifier](#) character for a Double-word is: ???.

Double-word [variables](#) are identified by following the [variable](#) name with three question marks (i.e., *var???*), or by using the [DEFDWD](#) statement as described in the previous discussion of Integers. You can also declare Double word variables using the DWORD keyword with the [DIM](#) statement. For example:

```
DIM I AS DWORD
```

As for [Word](#) values and Integers, Double-word values have a larger positive range than a [Long-integer](#), and still require only four bytes. Double-word values are useful for indicating absolute memory addresses, such as may be used to store [pointer](#) values.

A Classic PowerBASIC Double-word is equivalent to a UINT32 in C/C++. In 32-bit C/C++ code, a UINT is also equivalent to a Classic PowerBASIC Double-word variable. Note that 16-bit C/C++ code uses UINT to describe a 16-bit Word variable.

A C++ unsigned int and a Delphi longword are equivalent to a Classic PowerBASIC Double-word.

See Also

[Array Data Types](#)

[Bit Data Types](#)

[Constants and Literals](#)

[GUID Data Types](#)

[Object Data Types](#)

[Pointers](#)

[User Defined Types](#)

[Unions](#)

[Variant Data Types](#)

Long integers (&)

[Top](#) [Previous](#) [Next](#)

Like regular [Integers](#), Long integers cannot contain decimal points. However, they span a much greater range, from -2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$) yet occupy just 4 [bytes](#) (32-bits). The [type-specifier](#) character for a Long integer is: &.

Long integers are identified by following the [variable](#) name with an ampersand (i.e., *var&*) or by using the [DEFLNG](#) statement as described in the previous discussion of Integers. You can also declare Long-integer variables using the LONG keyword with the [DIM](#) statement. For example:

```
DIM I AS LONG
```

Long integers are the most efficient numeric data type in Classic PowerBASIC and should be used in all cases where speed is important and a greater numeric range is not required. (Using [Byte](#) and Integer variables in [FOR/NEXT](#) loops is actually slower than using a Long integer.)

A Classic PowerBASIC Long-integer variable is equivalent to the BOOL data type (in capital letters) commonly used by C/C++ compilers. Note that a bool (lowercase) is a non-traditional data type, equivalent to a Byte in Classic PowerBASIC. Be aware that some older C compilers may freely interchange bool and BOOL keywords.

A C/C++ int and a Delphi longint variable are also equivalent to a Classic PowerBASIC Long integer.

See Also

[Byte \(?\)](#)

[Double-word \(???\)](#)

[Integers \(%\)](#)

[Quad integers \(&&\)](#)

[Word \(??\)](#)

Quad integers (&&)

[Top](#) [Previous](#) [Next](#)

Quad-integers are 64-bit (8 [byte](#)) signed integers (twice as many bits as [Long integers](#)) with a range of -9.22×10^{18} to 9.22×10^{18} (-2^{63} to $2^{63} - 1$). The [type-specifier](#) character for a Quad integer is: &&.

Quad-integer [variables](#) are identified by following the variable name with two ampersands (i.e., *var&&*), or by using the [DEFQUD](#) statement as described in the previous discussion of Integers. You can also declare Quad-integer variables using the QUAD keyword with the [DIM](#) statement. For example:

```
DIM I AS QUAD
```

Although a Quad integer actually has 19 digits of precision, only 18 digits of accuracy can be "displayed" with [STR\\$](#). A 19-digit value will be rounded to 18 digits in scientific notation when used with STR\$. STR\$ works with up to 16 significant digits by default, so the enhanced form of STR\$ (eg: STR\$(*var*,18)), must be used to generate the 17th and 18th digits of a Quad integer for display purposes.

A C/C++ LARGE_INTEGER and a Delphi int64 are both equivalent to a Classic PowerBASIC Quad integer.

See Also

[Byte \(?\)](#)

[Double-word \(???\)](#)

[Integers \(%\)](#)

[Long integers \(&\)](#)

[Word \(??\)](#)

Single-precision floating-point (!)

[Top](#) [Previous](#) [Next](#)

Single-precision floating-point numbers (or more simply, Single-precision) may be the most versatile numeric type supported by Classic PowerBASIC. Single-precision values can contain decimal points and have a range of $\pm 8.43 \times 10^{-37}$ to 3.40×10^{38} . The [type-specifier](#) character for a Single-precision floating-point is: !.

Single-precision variables are identified by following the variable name with an exclamation point (i.e., *var!*) or by using the [DEFSNG](#) statement as described in the previous discussion of Integers. You can also declare Single-precision variables using the SINGLE keyword with the [DIM](#) statement. For example:

```
DIM I AS SINGLE
```

While Single-precision numbers can represent both enormous and microscopic values, they are limited to six digits of precision. In other words, Single-precision does a good job with figures like \$451.21 and \$6,411.92, but \$671,421.22 cannot be represented exactly because it contains too many digits. Neither can 234.56789 or 0.00123456789. A Single-precision representation will come as close as it can in six digits: \$671,421, or 234.568, or 0.00123457. Depending on your application, this rounding off can be a trivial or crippling deficiency. Like most modern compilers, Classic PowerBASIC uses the IEEE standard for all floating-point arithmetic.

C/C++, Delphi, and Visual Basic all offer a single data type that is identical to the Classic PowerBASIC Single-precision variable.

See Also

[Currency \(@\) and Extended-currency \(@@\)](#)

[Double-precision floating-point \(#\)](#)

[Extended-precision floating-point \(##\)](#)

Double-precision floating-point (#)

[Top](#) [Previous](#) [Next](#)

Double-precision floating-point numbers are to [Single-precision](#) numbers what [Long-integers](#) are to [Integers](#). They take twice as much space in memory (8 [bytes](#) versus 4 bytes), but have a greater range ($\pm 4.19 \times 10^{-307}$ to 1.79×10^{308}) and a greater accuracy (15 to 16 digits of precision versus the 6 digits of Single-precision). A Double-precision, 5,000-element [array](#) requires 40,000 bytes. An Integer array with the same number of elements occupies only 10,000 bytes. The [type-specifier](#) character for a Double-precision floating-point is: #.

Double-precision [variables](#) are identified by following the variable name with a Number symbol (i.e., *var#*) or by using the [DEFDBL](#) statement as described in the previous discussion of Integers. You can also declare Double-precision variables using the DOUBLE keyword with the [DIM](#) statement. For example:

```
DIM I AS DOUBLE
```

C/C++, Delphi, and Visual Basic all offer a double data type that is identical to the Classic PowerBASIC Double-precision variable.

See Also

[Array Data Types](#)

[Bit Data Types](#)

[Constants and Literals](#)

[GUID Data Types](#)

[Object Data Types](#)

[Pointers](#)

[User Defined Types](#)

[Unions](#)

[Variant Data Types](#)

Extended-precision floating-point

[Top](#) [Previous](#) [Next](#)

(##)

Extended-precision floating-point numbers are the basis of computation in Classic PowerBASIC. The [type-specifier](#) character for an Extended-precision floating-point is: ##. In Classic PowerBASIC, all floating point calculations are performed in extended precision for maximum accuracy. Extended-precision has also been provided as a declarable variable type, so you can take advantage of its extra exponent range and precision.

Extended-precision variables require 10 [bytes](#) of storage each. They have a range of approximately $\pm 3.4 \times 10^{-4932}$ to 1.2×10^{4932} , and offer 18 digits of precision. All 18 digits can be "displayed" using the extended [STR\\$](#) format (eg, `STR$(var##,18)`).

Extended-precision [variables](#) are identified by adding two Number symbols following a variable name (i.e., `var##`) or by using the [DEFEXT](#) statement.. You can also declare Extended-precision variables using the EXT or EXTENDED keywords with the [DIM](#) statement. For example:

```
DIM I AS EXT
DIM J AS EXTENDED
```

See Also

[Array Data Types](#)

[Bit Data Types](#)

[Constants and Literals](#)

[GUID Data Types](#)

[Object Data Types](#)

[Pointers](#)

[User Defined Types](#)

[Unions](#)

[Variant Data Types](#)

Currency (@) and Extended-currency (@@)

[Top](#) [Previous](#)
[Next](#)

Currency [variables](#) are 8 [byte](#) binary representations of floating-point numbers, which are considered to always have a fixed number of digits to the right of the decimal point.

Currency numbers have a range of approximately -9.22×10^{14} to $+9.22 \times 10^{14}$, and Extended-currency have a range of -9.22×10^{16} to $+9.22 \times 10^{16}$.

The type-specifier character for Currency and Extended-currency floating-point is: @ and @@ respectively.

You can also use the [DEFCUR](#) or [DEFCUX](#) statement as described under [Integers](#). They can also be declared using the CUR/CURRENCY or CUX/CURRENCYX keywords with the [DIM](#) statement. For example:

```
DIM I AS CUR
DIM J AS CURRENCYX
```

Currency variables (@) have up to 4 digits of precision after the decimal point, and are useful for prices and quantities where fractions of a cent are desired. Extended-currency variables (@@) have two digits of precision after the decimal point. They are optimized for financial calculations where fractions of a cent are *not* required.

The currency data types are especially useful for financial calculations, as they avoid the round-off errors associated with Single, Double, and Extended-precision numbers (which must be an exact power of two in order to be represented exactly). While many numbers can be represented exactly as a power of two, there are also many that cannot. For example, 1.10000002384185791 is the closest power of two to 1.1, in single precision.

So, when assigning [numeric literal](#) values to a Currency or Extended-currency variable, we recommend using a type specifier to ensure the value is given the intended precision. For example:

```
DIM x1 AS CUR
x1 = 1.0001@
```

```
DIM x2 AS CUX
x2 = 1.01@@
```

Internally, Currency and Extended-currency numbers are stored as [Quad-integers](#) with an implied decimal point (at 4 places for Currency, and at 2 places for Extended-currency). This approach ensures that all of the digits of the variables can be represented exactly.

Currency and Extended Currency perform a similar role as BCD variables in some BASIC dialects to ensure monetary values can be represented exactly; however, the internal storage of BCD variables and CUR/CUX differs substantially.

Delphi and Visual Basic both offer a *currency* data type that is identical to the PowerBASIC Currency variable.

See Also

[Array Data Types](#)

[Bit Data Types](#)

[GUID Data Types](#)

[Object Data Types](#)

[Pointers](#)

[User Defined Types](#)

[Unions](#)

[Variant Data Types](#)

You can think of ASCIIZ strings (or its synonym ASCIZ) as [fixed-length strings](#) where the last character is always a nul ([CHR\\$\(0\)](#) or [\\$NUL](#)) terminator. Like fixed-length strings, ASCIIZ strings contain character data of fixed length and any attempt to assign a string longer than the defined length will result in truncation.

If you assign a string (a [string literal](#), or a [dynamic](#) or fixed-length string variable) to an ASCIIZ string that is shorter than the defined length, the string will not be padded on the right. Instead, the nul-terminator goes right after the last string character. The contents of the remainder of the string buffer are undetermined. Because an ASCIIZ string requires the nul-terminator byte, ASCIIZ strings are usually defined with a length of at least two bytes.

You declare ASCIIZ string [variables](#) using the ASCIIZ or ASCIZ keywords with the [DIM](#) statement. For example:

```
DIM MyStr1 AS ASCIIZ * 40
DIM MyStr2 AS ASCIZ * 40
```

This creates a 40 [byte](#) ASCIIZ string named *MyStr*. The number of characters you can store in an ASCIIZ string is always *one less* than the defined length of the string. The last character is used to hold the nul-terminator. Therefore, a 40 byte ASCIIZ string can hold up to 39 characters. If you need 40 usable bytes, DIM the string to have a 41 byte length. It is on this basis that ASCIIZ strings should be defined with a length of at least two bytes.

When assigning string data to an ASCIIZ string, the assignment will stop if an embedded \$NUL byte is encountered. For example:

```
DIM a AS STRING
DIM b AS ASCIIZ * 10
a = CHR$("ABC", 0, "DEF")
b = a ' B will contain "ABC"
```

Unlike dynamic strings, the length of ASCIIZ strings is determined at compile-time, not run-time. In addition, unlike dynamic strings, ASCIIZ strings do not use handles. When you pass an ASCIIZ string to a routine, you are actually passing a [pointer](#) to the string's data.

The address of the contents of an ASCIIZ string can always be obtained with the [VARPTR](#) function. [LOCAL](#) ASCIIZ string memory is released when the associated [Sub](#), [Function](#), [Method](#), or [Property](#) ends. Subsequent calls to the procedure will result in new storage locations for the ASCIIZ string data; however, the location of a [LOCAL](#) fixed-length string does not move until the string memory is released when the procedure terminates.

LOCAL ASCIIZ strings are created on the [stack](#) frame, so LOCAL ASCIIZ strings will be limited to the amount of available stack space available. Typically this is less than 1 MB unless a larger stack frame has been allocated with [#STACK](#) metastatement. If larger ASCIIZ (or fixed-length) strings are required, it is advisable to make them [INSTANCE](#), [STATIC](#) or [GLOBAL](#) since those are not created within the stack frame.

The address of the contents of STATIC and GLOBAL ASCIIZ strings stays constant for the duration of the module. STATIC and GLOBAL Scalar (non-array) ASCIIZ strings may be up to 16,777,216 bytes each.

See Also

[Dynamic \(Variable-length\) strings \(\\$\)](#)

[FIELD strings](#)

[Fixed-length strings](#)

[String expressions](#)

Dynamic (Variable-length) strings (\$) [Top](#) [Previous](#) [Next](#)

Dynamic string [variables](#) contain character data of arbitrary length. Internally, each string variable uses four [bytes](#) that contain a handle number, which is used to identify and locate information about a string. The [type-specifier](#) character for a dynamic string is: \$.

String variables are designated by following the variable name with a dollar sign (\$) or the [DEFSTR](#) type definition. You can also declare dynamic string variables using the STRING keyword with the [DIM](#) statement. For example:

```
DIM MyStr AS STRING
```

Classic PowerBASIC allocates strings using the Win32 OLE string engine. This allows you to pass strings from your program to [DLLs](#), or API calls that support OLE strings. Note, however, that Visual Basic and Visual C++ store data in OLE strings using 16-bits per character (Unicode format) while Classic PowerBASIC stores them in 8-bit format. In Classic PowerBASIC, strings may contain either ASCII or ANSI string data.

The distinction between ASCII and ANSI only becomes important when using the strings for specific tasks. For example, when dealing with API calls, string data is usually interpreted as ANSI data by the API functions, whereas Classic PowerBASIC statements such as [UCASE\\$](#) treat the string data as ASCII.

Most standard DLLs designed to work with Visual Basic should still work with Classic PowerBASIC, because VB converts OLE strings from Unicode to ANSI before passing them to a DLL, and Classic PowerBASIC will accept and work with the ANSI string data.

The address of the contents of a non-empty string can be obtained with the [STRPTR](#) function. The address of the string handle can be obtained with [VARPTR](#) function. An empty (null) string may not return a valid STRPTR value.

Dynamic strings move in memory with each assignment statement: that is, STRPTR will return a different address when the content of the string is changed. However, the associated string handle obtained by VARPTR stays constant for the duration of the life (scope) of the string variable.

LOCAL dynamic string memory and handles are released when the associated [Sub](#), [Function](#), [Method](#), or [Property](#) ends. Subsequent calls to a routine will result in new storage locations for both the handle and the string data. The address of the handle of a STATIC or [GLOBAL](#) dynamic string stays constant for the duration of the module.

Dynamic strings and field strings cannot be part of [UDT](#) (User-Defined Type) or [UNION](#) structures.

In C/C++, a dynamic string is referred to as a BSTR data type.

See Also

[ASCIIZ strings](#)

[FIELD strings](#)

[Fixed-length strings](#)

String expressions

Field strings are a special form of [dynamic string](#), which have all the capabilities of a dynamic string, but may also represent a defined part of a [random file](#) buffer or a defined part of a dynamic string.

Field strings must always be declared using [DIM](#), [INSTANCE](#), [LOCAL](#), [STATIC](#), [GLOBAL](#), or [THREADED](#). They may be used in the same manner as a dynamic string [variable](#), or they can be bound to a file buffer for an *open* random-access file or a dynamic string using a corresponding [FIELD](#) statement. Each field string occupies sixteen [bytes](#) of memory, and requires slightly more general overhead than a regular dynamic string variable.

When used with a file

A random-access file buffer is automatically created for use when [GET](#) or [PUT](#) statements are used without a target variable. In this case, the file data is read or written using this file buffer, and the buffer is accessed with one or more field strings.

If a field is defined by a single field (*nSize*) parameter, it represents the length of the field, with the start position implied by the preceding field within the statement. If two parameters are used, they represent the start (*nStart*) and end (*nEnd*) positions, indexed to one.

If a string value shorter than the declared size is assigned to a field string, it is padded with blank spaces and placed into the file buffer. There is no requirement to use [LSET](#) for assignment. When used with a file buffer, the field string is only valid when the nominated file is open. Once the file has been closed, field strings bound to the file buffer will be empty (zero length), rather than a string of the length defined in the FIELD statement. For example:

```
LOCAL fld1, fld2, fld3 AS FIELD
OPEN "test.dat" FOR RANDOM AS #1 LEN = 30
FIELD #1, 5 AS fld1, 10 AS fld2, 15 AS fld3
fld1 = "Bob"           ' Stores "Bob  "
CSET fld2 = "Zale"     ' Stores "    Zale  "
RSET fld3 = "#1"       ' Stores "                #1"
? STR$(LEN(fld1))      ' Displays 5
? STR$(LEN(fld2))      ' Displays 10
? STR$(LEN(fld3))      ' Displays 15
CLOSE #1
? STR$(LEN(fld1))      ' Displays 0
```

When used with a dynamic string

A field variable bound to a dynamic string works very much like a pointer, so the programmer must use care in field variable selection. For example, if you bind a GLOBAL FIELD variable to a LOCAL string variable, then attempt to reference the global string after the local is destroyed (i.e., released when the owning [Sub/Function/Method/Property](#) exits), a fatal exception error (GPF) is likely to occur. The same could happen after an

array has been [erased](#), or a [REDIM](#) is used to change the memory allocation. To avoid problems with [scope](#), it is suggested that field variables be bound only with strings within the same scope (LOCAL, GLOBAL, etc.).

In addition, the dynamic string must contain data for the bound field strings to reference the data. For example:

```
LOCAL x$, sFirst AS FIELD, sSecond AS FIELD
FIELD x$, 3 AS sFirst, 3 AS sSecond
x$ = ""
? STR$(LEN(sFirst))    ' Displays 0 since x$ is empty
x$ = SPACE$(6)         ' Allocate data to the string
sFirst = "111"
sSecond = "222"
? STR$(LEN(sFirst))    ' Displays 3 as x$ now contains data
```

Field strings and dynamic strings cannot be part of [UDT](#) (User-Defined Type) or [UNION](#) structures.

See Also

[ASCIIZ strings](#)

[Dynamic \(Variable-length\) strings \(\\$\)](#)

[Fixed-length strings](#)

[String expressions](#)

Fixed-length strings

[Top](#) [Previous](#) [Next](#)

As their name implies, fixed-length strings have a pre-defined length, and any attempt to assign a string longer than the defined length will result in truncation. If you assign a string (a [string constant](#), or a [dynamic](#) or [ASCIIZ](#) string [variable](#)) to a fixed-length string that is shorter than the defined length, the string will be padded on the right with spaces. The major difference between dynamic strings and fixed-length strings is that once defined, the length of a fixed-length string cannot be changed. It is "fixed" for the duration of program execution.

You declare fixed-length string variables using the `STRING * x` format with the [DIM](#) statement. For example:

```
DIM MyStr AS STRING * 10
```

Although fixed-length strings were provided mainly to allow you to define string fields within a [User-Defined Type](#) or [Union](#), they can also be used wherever other string values are valid.

Unlike dynamic strings, the length of fixed-length strings is determined at compile-time, not run-time. In addition, unlike dynamic (variable-length) strings, fixed-length strings do not use handles. When you pass a fixed-length string to a routine, you are actually passing a [pointer](#) to the string's data.

A declaration of a fixed-length string or fixed-length string pointer must explicitly state the length of the variable, because the compiler must know it to allocate memory, and to pad the variable with spaces upon assignment.

The address of the contents of a fixed length string can always be obtained with the [VARPTR](#) function. [LOCAL](#) fixed-length string memory is released when the associated [Sub](#), [Function](#), [Method](#), or [Property](#) ends. Subsequent calls to the routine will result in new storage locations for the fixed-length string data; however, the location of a `LOCAL` fixed-length string does not move until the string memory is released when the routine terminates.

`LOCAL` fixed-length strings are created on the stack frame, so `LOCAL` fixed-length strings will be limited to the amount of available stack space available. Typically this is less than 1 MB unless a larger stack frame has been allocated with [#STACK](#) metastatement. If larger fixed-length (or `ASCIIZ`) strings are required, it is advisable to make them [INSTANCE](#), [STATIC](#), or [GLOBAL](#) since those are not created within the [stack](#) frame.

The address of the contents of `STATIC` and `GLOBAL` fixed-length strings stays constant for the duration of the module. `STATIC` and `GLOBAL` Scalar (non-[array](#)) fixed-length strings may be up to 16,777,216 bytes each.

See Also

[ASCIIZ strings](#)

[Dynamic \(Variable-length\) strings \(\\$\)](#)

FIELD strings

String expressions

String expressions

[Top](#) [Previous](#) [Next](#)

A string expression consists of [string literals](#), string variables, and [string functions](#), optionally combined with the concatenation operators (+ or &). String expressions always produce strings as their result. Note that when the ampersand (&) is used as a string concatenation operator, it must be surrounded by white space, to differentiate it from the Long-integer type-specifier (i.e., LongVar&) and the number base prefix (i.e. &H0FF, &O77). Examples of string expressions include:

```
"Cats and dogs"           ' string constant
firstname$                ' string variable
firstname$ + lastname$    ' string concatenation
a$ = "Cats " & "and " & "dogs" ' string concatenation
LEFT$(a$ + z$,7)          ' string function
a$ + MID$("Cats and dogs",5,3)
RIGHT$(MID$(a$ + z$,1,6),3)
```

Note that [fixed-length strings](#) are always a fixed length (defined in the corresponding [DIM](#) statement), string concatenation involving these strings works differently than you might expect. For instance, the following program fragment:

```
DIM Greeting AS STRING * 40
Greeting = "hello"
Greeting = greeting + "there"
```

This appends (adds) the five-character string *"there"* to the 40-character fixed-length string (*"hello"*, followed by 35 spaces), but the result is truncated to 40 characters (the predefined length of the string variable Greeting), which causes the newly appended string to be lost. One solution to this problem is to use the [RTRIM\\$](#) function to remove the trailing spaces from *"hello"* before appending *"there"*:

```
DIM Greeting AS STRING * 40
Greeting = "hello"
Greeting = RTRIM$(Greeting) + " there"
```

Variables of [user-defined types](#) may be used as string operands without any need to specify the individual UDT members:

```
TYPE MyType
  ItemOne AS STRING * 10
  ItemTwo AS STRING * 10
END TYPE
DIM SomeData AS MyType
SomeData.ItemOne = "hello"
SomeData.ItemTwo = "world!"
X$ = "Look at this!" + $CRLF + SomeData
```

See Also

[ASCIIZ strings](#)

Dynamic (Variable-length) strings (\$)

FIELD strings

Fixed-length strings

String Operations

String Operations

[Top](#) [Previous](#) [Next](#)

The following functions manipulate and manage string data:

ACODE\$	Translate a Unicode string into an ANSI string
ARRAY ASSIGN	Assign a number of values to successive elements of an array
ARRAY DELETE	Delete a single item from a given array
ARRAY INSERT	Insert a single item into a given array
ARRAY SCAN	Scan all or part of an array for a given value
ARRAY SORT	Sort all or part of a given array
BIN\$	Return a string with the binary (base 2) representation of a value
BUILDS	Concatenate multiple strings with high efficiency
CHOOSE\$	Return one of several values, based upon the value of an index
CHR\$	Convert one or more ASCII codes into ASCII character(s)
CLSID\$	Return a 16-byte (128-bit) GUID string containing a CLSID
COMM LINE	Receive a CR/LF terminated "line" of data from a serial port
COMM PRINT	Send a "line" of binary data through a serial port
COMM RECV	Receive binary data from a serial port
COMM SEND	Send a string of binary data through a serial port
COMMAND\$	Return the command-line used to start the program
CSET	Center a string within the space of another string or UDT
CSET\$	Return a string containing a centered (padded) string
CURDIR\$	Return the current directory for a given drive
DATA	Declare an array of constants to be read by READ\$
DATACOUNT	Return the total count of the number of local data items
DATE\$	Set and retrieve the system date
DIM	Declare and dimension arrays, scalar variables , and pointers
DIR\$	Return a filename that matches the given mask
DIR\$ CLOSE	Force the release the operating system FindNext handle
ENVIRON	Modify the current program's environment table.
ENVIRON\$	Retrieve strings from the operating system's environment table
ERASE	Deallocate array memory
ERL\$	Return the last label, line number, or procedure name executed prior to the most recent error.
ERROR\$	Return a string containing the descriptive name of an error

<u>EXTRACT\$</u>	Return up to the first occurrence of a specified character
<u>EXE</u>	Return the path and/or name of the executing program.
<u>FIELD</u>	Bind a field string variable to a particular sub-section of a random file buffer or a dynamic string variable
<u>FIELD RESET</u>	Reset the FIELD string to a nul (zero-length) dynamic string
<u>FIELD STRING</u>	Change the FIELD string to a dynamic string, but first assigns the current sub-section data to it
<u>FILENAMES\$</u>	Return the file-system name of an open file
<u>FORMAT\$</u>	Return a string containing formatted numeric data
<u>FUNCNAMES\$</u>	Return the name of the current <u>Sub/Function/Method/Property</u>
<u>GET</u>	Read a record from a <u>random-access file</u>
<u>GET\$</u>	Read a string from a file opened in <u>binary mode</u>
<u>GUID\$</u>	Return a 16-byte (128-bit) Globally Unique Identifier GUID
<u>GUIDTXT\$</u>	Return a 38-byte human-readable GUID/UUID string
<u>HEX\$</u>	Hexadecimal (base 16) string representation of an argument
<u>IIF\$</u>	Return one of two values based upon a True/False evaluation
<u>INPUT#</u>	Load variables with data from a <u>sequential file</u>
<u>INPUTBOX\$</u>	INPUTBOX\$ displays a dialog box containing a prompt
<u>INSTR</u>	Search a string for the first occurrence of a character or string
<u>JOIN\$</u>	Return a string consisting of all of the strings in a <u>string array</u>
<u>LCASE\$</u>	Return a lowercase version of a string argument
<u>LEFT\$</u>	Return the left-most <i>n</i> characters of a string
<u>LEN</u>	Return the logical length of a variable, UDT, or <u>Union</u>
<u>LET</u>	Assign a value to a variable.
<u>LET (with Types)</u>	Assign data to a <u>user-defined type</u> variable.
<u>LET (with Variants)</u>	Assign a value or an object reference to a <u>variant</u> variable
<u>LINE INPUT#</u>	Read line(s) from a sequential file into a string variable or array
<u>LPRINT</u>	Output text and data to a <u>printer</u> device
<u>LPRINT\$</u>	Return the current printer device used for LPRINT operations
<u>LSET</u>	Left-align a string within the space of another string or <u>UDT</u>
<u>LSET\$</u>	Return a string containing a left-justified (padded) string
<u>LTRIM\$</u>	Return a string with leading characters or strings removed
<u>MAX\$</u>	Return the argument with the largest (maximum) value

MCASE\$	Return a mixed case version of a string argument
MID\$	Return a portion of a string
MID\$	Replace characters in a string with characters from another string
MIN\$	Return the argument with the smallest (minimum) value
MKBYT\$	Convert a Byte value into a binary encoded string
MKCUR\$	Convert a Currency value into a binary encoded string
MKCUX\$	Convert an Extended Currency value into a binary encoded string
MKD\$	Convert a Double-precision value into a binary encoded string
MKDWD\$	Convert a Double-word value into a binary encoded string
MKE\$	Convert an Extended-precision value into a binary encoded string
MKI\$	Convert a Integer value into a binary encoded string
MKL\$	Convert a Long-integer value into a binary encoded string
MKQ\$	Convert a Quad-integer value into a binary encoded string
MKS\$	Convert a Single-precision value into a binary encoded string
MKWRD\$	Convert a Word value into a binary encoded string
MKDIR	Create a subdirectory/folder (like the DOS MKDIR command)
NUL\$	Return a string containing a specified number of \$NUL characters
OBJRESULT\$	Returns a string which describes an OBJRESULT (hResult) code.
OCT\$	Return a string that is a octal (base 8) representation of a value
PARSE	Parse a string and extract all delimited fields into an array
PARSE\$	Return a delimited field from a string expression
PARSECOUNT	Return the count of delimited fields in a string expression
PATHNAME\$	Parse a path/file name to extract component parts
PATHSCAN\$	Find a file on disk and return the path and/or file name parts.
PEEK\$	Return a sequence of bytes starting at a specific memory location
POKE\$	Store a sequence of bytes starting at a specific memory location
PRINT#	Write a complete array to a sequential file
PROGID\$	Return the alphanumeric PROGID string (text) of a given CLSID
PUT	Write a record to a random-access file or variable to a binary file
PUT\$	Write a string to a file opened in binary mode
READ\$	Retrieve string data from a local DATA list
REGEXPR	Scan a string for a matching "wildcard" or regular expression
REGREPL	Scan a "wildcard" match in a string with a new string
REMAINS	Return all characters beyond occurrence of a specified character

<u>REMOVE\$</u>	Return a copy of a string with characters or strings removed
<u>REPEAT\$</u>	Return a string consisting of multiple copies of a specified string
<u>REPLACE</u>	Replace all occurrences of one string with another string
<u>RESET</u>	Clear a string, string <u>array subscript</u> , or an entire array
<u>RETAIN\$</u>	Return a string with all non-specified characters removed
<u>RIGHT\$</u>	Return the rightmost <i>n</i> characters of a string
<u>RSET</u>	Right justify a string into the space of a string variable or UDT
<u>RSET\$</u>	Return a string containing a right-justified (padded) string
<u>RTRIM\$</u>	Return a copy of a string with trailing characters/strings removed
<u>SIZEOF</u>	Return the total or physical length of any Classic PowerBASIC variable
<u>SPACE\$</u>	Return a string consisting of a specified number of spaces
<u>STR\$</u>	Return the string representation of a number in printable form
<u>STRDELETE\$</u>	Delete a specified number of characters from a string expression
<u>STRING\$</u>	Return a string with multiple copies of the specified character
<u>STRINSERT\$</u>	Insert a string at a specified position within another string
<u>STRPTR</u>	Return the address of the data held by a <u>variable length string</u>
<u>STRREVERSE\$</u>	Reverse the contents of a string expression
<u>SWAP</u>	Exchange the values of two strings, pointers, or pointer targets
<u>SWITCH\$</u>	Return one item of a series based upon a True/False evaluation
<u>TAB\$</u>	Return a string with TAB characters expanded with spaces
<u>TALLY</u>	Count the number of occurrences of specified characters/strings
<u>TIME\$</u>	Read and/or set the system time
<u>TRIM\$</u>	Return a string with leading and trailing characters removed
<u>TYPE SET</u>	Assign the value of a UDT or string expression to a UDT
<u>UCASE\$</u>	Return an all-uppercase (capitalized) version of a string
<u>UCODE\$</u>	Translate an ANSI string into a Unicode string
<u>UCODEPAGE</u>	Set the default codepage used for ANSI / UNICODE conversions
<u>USING\$</u>	Format string/numeric expressions using a mask string
<u>VAL</u>	Return the numeric equivalent of a string argument
<u>VARIANT\$</u>	Return the <u>dynamic string</u> contained in a Variant variable
<u>VARPTR</u>	Return the 32-bit address of a string handle
<u>VERIFY</u>	Determine if each character of a string is in another string

It is often useful to treat a set of [variables](#) as a group. This lets you perform repetitive operations more easily. An array is a group of string or numeric data sharing the same variable name. The individual values that make up an array are called elements. An element of an array can be used in a statement or expression wherever you would use a regular string or numeric variable. In other words, each element of an array is itself a variable.

Classic PowerBASIC provides several statements that perform operations on an array as a whole, allowing you to sort its contents, scan its contents for data that matches a certain condition, and insert data into or delete data from the existing structure.

You can think of an array as a row of boxes, numbered from zero to a predetermined number: four, in the example figure below. Each box holds a distinct value, which may or may not differ from the values in the other boxes. The boxes and their numbers are represented by parentheses surrounding a number; for example, *item%(3)* represents box number three of the array *item%*. Thus, if the *value* held within box number 3 is 1952, the statement *total%=item%(3)* would place the value 1952 into the variable *total%*.

<i>item%(n)</i>				
<i>item%(0)</i>	<i>item%(1)</i>	<i>item%(2)</i>	<i>item%(3)</i>	<i>item%(4)</i>
50	10	-5	1952	104

Dimensioning a dynamic array with [DIM](#) or [REDIM](#) also clears each element, unless the [PRESERVE](#) option is present. Each element of each numeric array is set to zero; [string arrays](#) are set to the null string ("", length zero). Declaring the name and type of an array, as well as the number and organization of its elements, is performed by the DIM statement. For example:

```
DIM payments(55) AS CURRENCYX
```

creates an array variable *payments*, consisting of 56 [Extended-currency](#) elements, numbered 0 through 55. Array *payments* and an Extended-currency variable also named *payments* are separate variables. If this is confusing, you'll understand why we suggest that you use different variable names.

See Also

- [Subscripts](#)
- [String arrays](#)
- [Multidimensional arrays](#)
- [Array storage requirements](#)
- [Internal representations of arrays](#)
- [Arrays within User-Defined Types](#)
- [Array operations](#)

Individual [array](#) elements are selected with subscripts or index numbers, which are [Long-integer](#) expressions within parentheses to the right of an array [variable's](#) name. For example, *payments*(3) and *payments*(44) are two of *payments* 56 elements. Normally, the first element of an array has a subscript value of zero, although this can be changed with the [DIM](#) statement. Some examples follow:

```
' This DIM statement declares a 56-element array
' with subscript bounds of 0 TO 55.
DIM payments1(55) AS CURRENCY
```

```
' This DIM statement declares a 56-element array
' with subscript bounds of 1 TO 56
DIM payments2(1 TO 56) AS CURRENCYX
```

You must DIM all arrays before you can use them. This is a different approach than that used by some BASIC dialects, which assume that an array contains 10 elements (0 to 9) if the array is not explicitly dimensioned.

Classic PowerBASIC allows you to define a range of subscript values rather than just setting an upper limit. In addition to the TO syntax shown above, Classic PowerBASIC supports the classic BASIC syntax that uses a colon (:) character to separate range values. Use the TO syntax in preference to the colon syntax, since the latter may not be supported in future versions of Classic PowerBASIC. For example, the following two DIM statements are equivalent:

```
DIM fog(50:60) AS LONG      ' Classic syntax
```

or

```
DIM fog(50 TO 60) AS LONG   ' Preferred syntax
```

and both work to create an array *fog*, consisting of eleven Long-integer elements numbered 50 through 60. The statement:

```
DIM clouds(50 TO 60, 25 TO 45) AS LONG
```

creates the two-dimensional Long-integer array named *clouds*, containing 231 (11 * 21) elements. Classic PowerBASIC's subscript range declaration capability allows you to model a program's data structures more closely to the problem at hand.

For example, consider a program tracking 19th-century birth statistics. This program's central data structure is a Long-Integer array of 100 elements that contain the number of babies born in each year of the last century. Ideally, you would create an array that used subscript values equal to the year in which the births occurred (for example, *births*(1851) represents how many babies came into the world in 1851), so that a code passage like:

```
DIM births(1899) AS LONG
FOR year& = 1800 TO 1899
    INCR Total&, births(year&)
NEXT year&
```

would be as straightforward as possible. Unfortunately, DIM births(1899) AS LONG creates a 1900-element array (from 0 to 1899), of which the first 1800 are wasted. Traditionally, BASIC programmers have tackled this problem by declaring the array as:

```
DIM births$(99)
```

and by playing games with subscripts:

```
FOR year% = 1800 TO 1899
  INCR Total%, births$(year%-1800)
NEXT year%
```

While this sort of thing works, it complicates things and slows programs down because suddenly there are 100 subtractions that weren't there before. It's better to declare a range, like this:

```
DIM births$(1800 TO 1899) ' array births has subscripts
                           ' ranging from 1800 to 1899
FOR year% = 1800 TO 1899
  Total% = Total% + births$(year%)
NEXT year%
```

```
DIM birth1$(99)           ' Array has 100 elements from 0 TO 99
DIM birth2$(1 TO 99)      ' Array has  99 elements from 1 TO 99
DIM birth3$(3 TO 99)      ' Array has  97 elements from 3 TO 99
```

See Also

[Array Data Types](#)

[String arrays](#)

[Multidimensional arrays](#)

[Array storage requirements](#)

[Internal representations of arrays](#)

[Arrays within User-Defined Types](#)

[Array operations](#)

String arrays

[Top](#) [Previous](#) [Next](#)

The elements of string arrays hold strings instead of . Each string can be a different length. For example [DIM](#) words\$(50) creates a sequence of 51 independent string variables:

```
DIM words$(50)
words$(0) = "Daniel likes cats."      ' 18-character string
words$(1) = ""                       ' a null string
words$(2) = "Nicki is a sweet child." ' 23-character string
' assign more array values here
words$(50) = SPACE$(200)             ' 200-character string
```

See Also

[Array Data Types](#)

[Multidimensional arrays](#)

[Array storage requirements](#)

[Internal representations of arrays](#)

[Arrays within User-Defined Types](#)

[Array operations](#)

[Arrays](#) can have one or more dimensions, up to a maximum of eight. A one-dimensional array such as payments is a simple list of values. A two-dimensional array represents a table of numbers with rows and columns of information. Some examples of multidimensional arrays are:

```
DIM one!(15)           ' a one-dimensional list
DIM two!(15,20)        ' a two-dimensional table
DIM three!(7,9,1)      ' an 8 by 10 game board with room in the third
                        ' dimension for 2 items: piece type and color
```

Arrays of four to eight dimensions are possible, but they become more difficult to conceptualize and keep straight. You can define:

```
DIM five%(5,5,10,20,3) ' a five-dimensional array
```

but it's probably better to redesign this array into several smaller ones with fewer dimensions, or use an array of [User-Defined Types](#).

See Also

[Array Data Types](#)

[Subscripts](#)

[String arrays](#)

[Array storage requirements](#)

[Internal representations of arrays](#)

[Arrays within User-Defined Types](#)

[Array operations](#)

Array storage requirements

[Top](#) [Previous](#) [Next](#)

A Classic PowerBASIC [array](#) may contain up to 4,294,967,295 elements, and the data may occupy as much as all available memory. However, all individual index numbers must fall within the range of a Long-integer variable (-2,147,483,648 to +2,147,483,647).

Classic PowerBASIC stores array data in main memory for all array types (including [LOCAL](#) arrays). Therefore, there is no arbitrary array size limit imposed by the amount of free [stack](#) space, such as can be experienced with large LOCAL [ASCIIZ](#), and [Fixed-length string](#) variables. The availability of main memory is the prime consideration (typically up to 2 Gb can be used). However, LOCAL arrays do require the storage of around 128 bytes on the stack for the array descriptor table.

See Also

[Array Data Types](#)

[Subscripts](#)

[String arrays](#)

[Multidimensional arrays](#)

[Internal representations of arrays](#)

[Arrays within User-Defined Types](#)

[Array operations](#)

Internal representations of arrays

[Top](#) [Previous](#) [Next](#)

Classic PowerBASIC stores [arrays](#) in column-major order: *Array*(0,0) is first (lowest) in memory, then *Array*(1,0), then *Array*(2,0), and so on through all the rows of the array. After the rows are taken care of, the next column is stored.

While Classic PowerBASIC supports lower boundary values that are non-zero, Classic PowerBASIC generates the most efficient code if the lower boundary parameter is omitted (i.e., the array uses the default lower boundary of zero).

Array boundary values can be obtained at run-time via the [LBOUND](#) and [UBOUND](#) functions. Descriptive attributes of an array can be retrieved with the [ARRAYATTR](#) function. These attributes include such information as the data type and the number of [dimensions](#), etc.

See Also

[Array Data Types](#)

[Subscripts](#)

[String arrays](#)

[Multidimensional arrays](#)

[Array storage requirements](#)

[Arrays within User-Defined Types](#)

[Array operations](#)

Arrays within User-Defined Types

[Top](#) [Previous](#) [Next](#)

In prior versions of this compiler, arrays could not be part of a [UDT](#) structure. However, we now support both [one](#) and [two-dimensional arrays](#) of variables that have a fixed-length (for each element) - this includes [ASCIIZ strings](#), [fixed-length strings](#), and all numeric variable classes. Individual arrays within a UDT may be up to 16 Megabytes each (although UDTs themselves are limited to 16 Megabytes).

Two-dimensional arrays within [Types](#) work exactly as do any other array in Classic PowerBASIC, except that their dimensions are specified by positive numeric constant values, and are therefore not dynamically alterable. That is, the dimension sizes must be specified with [numeric equates](#) or [numeric literal](#) values, and these cannot be altered at run-time.

Like conventional arrays, the default lower array boundary is zero, but positive non-zero values may be used to specify a specific range of [subscript](#) index values for the array, separated from the upper array boundary subscript with the TO keyword. Additionally, both the lower and upper subscript index values must be zero or greater (ie, negative subscript values are not permitted). Examples of valid syntax follow:

```
TYPE MYTYPE
  id AS INTEGER           ' Scalar UDT member
  Styles(6)              AS DWORD ' 7 elements (0 TO 6)
  Yrs(1980 TO 2010) AS LONG  ' 31 elements
  Team(100 TO 101) AS BYTE  ' 2 elements
  Rating(1 TO 10) AS DWORD  ' 10 elements
  X(1 TO 5, 0 TO 5) AS EXT   ' 30 elements (5x6)
  Y(4,3) AS QUAD           ' 20 elements (5x4)
END TYPE
```

See Also

[Array Data Types](#)

[Subscripts](#)

[String arrays](#)

[Multidimensional arrays](#)

[Array storage requirements](#)

[Internal representations of arrays](#)

[Array operations](#)

Array Operations

[Top](#) [Previous](#) [Next](#)

The following functions can be used to manipulate and manage arrays:

#DEBUG ERROR	Control generation of error checking code
#DIM	Specify if variables must be declared before use
ARRAY ASSIGN	Assign a number of values to successive elements of an array
ARRAY DELETE	Delete a single item from a given array
ARRAY INSERT	Insert a single item into a given array
ARRAY SCAN	Scan all or part of an array for a given value
ARRAY SORT	Sort all or part of a given array
ARRAYATTR	Return descriptive attributes of a given array
BIT CALC	Set or reset a bit in an implied bit-array
BIT	Return the value of a particular bit in an implied bit-array
BIT	Manipulate individual bits of an implied bit-array
DATA	Declare an array of constants to be read by READ\$
DATACOUNT	Return the total count of the number of local data items
DIM	Declare and dimension arrays, scalar variables, and pointers
ERASE	Deallocate array memory
FILESCAN	Rapidly scan an open file , before loading into an array with GET
GET	Read a complete array from a binary file
JOIN\$	Return a string consisting of all of the strings in a string array
LBOUND	Return the lowest subscript of an array's specific dimension
LET	Assign a Variant to an array or an array to a Variant
LINE INPUT#	Read line(s) from a sequential file into a string variable or array
MAT	Matrix calculations on numeric arrays
PARSE	Parse a string and extract all delimited fields into an array
PRINT#	Write a complete array to a sequential file
PUT	Write a complete array to a binary file
READ\$	Retrieve string data from a local DATA list
REDIM	Declare dynamic arrays, allocate, reallocate, deallocate memory
RESET	Set an array subscript or an entire array to zero or null/empty
UBOUND	Return the highest subscript of an array's specific dimension

User-Defined Types (UDTs)

[Top](#) [Previous](#) [Next](#)

[Arrays](#) are useful when you need to treat a set of similar [variables](#) as a unit. For instance, ten test scores or ten student names. But what if you need to store several unrelated data types and be able to treat *them* as a unit? That is where User-Defined Types come in. When you define a User-Defined Type (UDT), you are actually defining a template for a new variable Type.

Once created, you can define as many variables of your new Type as you please. Moreover, since User-Defined Types can be associated with a [random file's](#) buffer, this provides you with a whole new way to access your random files.

Classic PowerBASIC's User-Defined Type is similar to a C struct or Pascal record. The elements of a User-Defined Type may include any of Classic PowerBASIC's data types, with the exception of [dynamic \(variable-length\) strings](#), [field strings](#), and *arrays* of dynamic strings.

To get an idea of the power of the User-Defined Type, imagine you are a teacher who needs a program to keep track of student grades. Since your school is on a very tight budget (and what schools aren't these days?), you decide to write the program yourself in Classic PowerBASIC. For each student in the class you need to track the following information:

- The student's name
- A student number
- A mailing address
- The name and phone of the person to contact in case of an emergency
- The relationship of the contact person to the student

Currently these records are being kept in a small file box. The information about each student is contained on a single file card. How do you transfer this information to the computer? Simple. Define a *Student Record* Type that will contain all the information about a single student.

The variables you create as User-Defined Types are often called *records* or *record variables*, since each variable of that Type contains one record, or *one set of related information*. The individual elements are referred to as *fields* or *members*. In the example above, each set of student information is a record, and each piece of information within that record (the last name for example) is a field.

See Also

[Accessing the fields of a User-Defined Type](#)

[Nesting User-Defined Types](#)

[Arrays within User-Defined Types](#)

[Using arrays of User-Defined Types](#)

[Using User-Defined Types with procedures and functions](#)

[Storage requirements and restrictions](#)

[Unions](#)

Defining User-Defined Types

[Top](#) [Previous](#) [Next](#)

The definition of a [User-Defined Type](#) begins with the reserved word [TYPE](#) and ends with the keywords `END TYPE`. In between, you define the names and data types of the member elements (fields) that are to be part of the new Type. For example:

```
TYPE StudentRecord
  LastName      AS STRING * 20 ' A 20-character string
  FirstName     AS STRING * 15 ' A 15-character string
  IDnum         AS LONG       ' Student ID, a Long-integer
  Contact       AS STRING * 30 ' Emergency contact person
  ContactPhone  AS STRING * 14 ' Their phone number
  ContactRel    AS STRING * 8  ' Relationship to student.
  AverageGrade  AS SINGLE     ' Single-precision % grade
END TYPE
```

Remember that the definition of a User-Defined Type does not set aside memory for storing data of that Type. Rather, it defines a template for the new Type *StudentRecord*. Then when the compiler encounters a statement declaring (or creating) a [variable](#) of the new Type, it will "know" how many bytes of storage to set aside for the variable. In order to use this new Type, you must declare variables of that Type with the [DIM](#) statement:

```
DIM Student AS StudentRecord
```

See Also

[User-Defined Types \(UDTs\)](#)

[Accessing the fields of a User-Defined Type](#)

[Nesting User-Defined Types](#)

[Arrays within User-Defined Types](#)

[Using arrays of User-Defined Types](#)

[Using User-Defined Types with procedures](#)

[Storage requirements and restrictions](#)

[Unions](#)

Accessing the fields of a User-Defined Type

[Top](#) [Previous](#)
[Next](#)

To work with the individual fields within a record variable, separate the field name from the variable name with a period. Here are some examples using the *Student* variable in the above [DIM](#) statement:

```
Last$      = Student.LastName
Message$   = "Id number is: " + STR$(Student.IdNum)
Student.FirstName = "Bob"
Student.LastName  = "Smith"
Fullname$ = Student.LastName + " " + Student.FirstName
Fullname$ = RTRIM$(Student.LastName) + ", " + RTRIM$(Student.FirstName)
```

Note that the last two statements above produce slightly differing results. The former produces a string that contains the text plus any [\\$SPC](#) (space) characters that pad the text in each of the Student.LastName and Student.FirstName members. Comparatively, the latter statement returns a string with these padding characters removed. In many cases, it can be easier to use [ASCIIZ](#) string members to alleviate the need to frequently trim such [fixed-length strings](#), but allowance must be made for the additional [\\$NUL](#) terminator byte required by ASCIIZ strings.

See Also

- [User-Defined Types \(UDTs\)](#)
- [Defining User-Defined Types](#)
- [Accessing the fields of a User-Defined Type](#)
- [Nesting User-Defined Types](#)
- [Arrays within User-Defined Types](#)
- [Using arrays of User-Defined Types](#)
- [Using User-Defined Types with procedures](#)
- [Storage requirements and restrictions](#)
- [Unions](#)

Nesting User-Defined Types

[Top](#) [Previous](#) [Next](#)

The fields within a [User-Defined Type](#) can be made up of other User-Defined Types. Just like a set of Chinese boxes, with each box containing a smaller box, you can nest one User-Defined Type within another. The result is that you create data structures that have a hierarchy similar to the directory tree structure of your hard drive.

Instead of storing the student names as two separate fields, we could instead define a Type called *NameRec* as follows:

```
TYPE NameRec
  Last      AS STRING * 20
  First     AS STRING * 15
  Initial   AS STRING * 1
END TYPE
```

Then, when we define our *Student Record* Type, we can define the field containing the individual student's name as *NameRec*:

```
TYPE StudentRecord
  FullName   AS NameRec
  IdNum      AS LONG
  Contact    AS NameRec
  ContactPhone AS STRING * 14
  ContactRel AS STRING * 8
  AverageGrade AS SINGLE
END TYPE
```

You could, of course carry this idea a step further, and define other components of the student record as nested records. For instance, a *ContactRecord* or even a *PhoneRec* but we'll leave that refinement up to you. To access the fields of a nested record, simply extend the dot notation. Just as the backslash (\) is used to separate the individual subdirectory names in a path (i.e., C:\PROJECTS\PROGRAM), the period is used within record variable names to separate the member elements from the base Type. For instance:

```
StudentRecord.FullName
```

refers to the *FullName* field (which happens to be of Type *NameRec*) within *Student Record*, and:

```
StudentRecord.FullName.First
```

refers to the sub-field *First* within the *FullName* field.

You can nest User-Defined Types as deeply as you want to, as long as the entire name used to refer to a field is within the maximum identifier length of 255 characters. In practical terms however, you probably would not want to carry nesting beyond two or, at most, three levels. Beyond that, it becomes clumsy, difficult to remember, and you are more likely to make typing errors. Note that User-Defined Types cannot contain circular references - for example, a UDT called *StudentRecord* cannot contain a field of Type *StudentRecord*.

See Also

[User-Defined Types \(UDTs\)](#)

[Defining User-Defined Types](#)

[Accessing the fields of a User-Defined Type](#)

[Arrays within User-Defined Types](#)

[Using arrays of User-Defined Types](#)

[Using User-Defined Types with procedures](#)

[Storage requirements and restrictions](#)

[Unions](#)

Using arrays of User-Defined Types

[Top](#) [Previous](#) [Next](#)

You can create arrays of [User-Defined Types](#) just as you can create [arrays](#) of integers or string or any of Classic PowerBASIC's other data types. For example:

```
DIM Class(1 TO 30) AS StudentRecord
```

To access the individual elements of the *Class* array, you use [subscript](#) index values just as you do with any other array. The third student record is *Class(3)*, for instance. The period separator and the field name follows the array subscript:

```
Class(3).FullName.First
```

This would access the first name of the third student in the class array. Think of it this way: the array is made up of elements of the Type *Student Record*, so the subscript belongs with the name of the [variable](#) as a whole.

You can create [multidimensional arrays](#) of User-Defined Types just as you can with any other Classic PowerBASIC data type. The limit on the number of elements and dimensions in such arrays is governed by the same rules as well: The limits are defined by the amount of data storage required for each element. Additionally, arrays within structures must contain a static subscript list, defined at compile-time. Therefore, arrays within structures cannot be [redimensioned](#) at run-time.

See Also

[User-Defined Types \(UDTs\)](#)

[Defining User-Defined Types](#)

[Accessing the fields of a User-Defined Type](#)

[Arrays within User-Defined Types](#)

[Nesting User-Defined Types](#)

[Using User-Defined Types with procedures](#)

[Storage requirements and restrictions](#)

[Unions](#)

Using User-Defined Types with procedures

[Top](#) [Previous](#)
[Next](#)

[Subroutines](#), [functions](#), [Methods](#), and [Properties](#) can process [User-Defined Types](#) as well as any other data type. This topic covers the following topics:

- Passing fields as arguments
- Passing records as arguments
- Passing record arrays as arguments

Passing fields as arguments

Members in User-Defined Types that are one of the built-in Classic PowerBASIC types ([INTEGER](#), [WORD](#), [STRING](#), and so on) can be passed to procedures and functions as if they were simple variables. For example, given the User-Defined Type *PatientRecord*, as follows:

```
TYPE PatientRecord
  FullName AS STRING * 32
  AmountDue AS DOUBLE
  IdNum AS LONG
END TYPE
DIM Patient AS PatientRecord
```

you could use a procedure *PrintStatement*:

```
SUB PrintStatement(Id AS LONG, AmountPastDue AS DOUBLE)
  ' access Id and AmountPastDue
END SUB
```

like this:

```
CALL PrintStatement(Patient.IdNum, Patient.AmountDue)
```

Passing records as arguments

You can also write your procedures to accept arguments of User-Defined Types. This is especially useful if you want to pass many arguments; rather than have a long argument list, you can pass a single User-Defined Type. For example, given the *PatientRecord* User-Defined Type discussed in the previous section, you could write your *PrintStatement* procedure as follows:

```
SUB PrintStatement(Patient AS PatientRecord)
  ' access Patient.IdNum and Patient.AmountDue
END SUB
```

You'd call *PrintStatement* like this:

```
CALL PrintStatement(Patient)
```

Passing record arrays as arguments

Procedures can accept [arrays](#) of records as easily as they can accept arrays of other Types. For example, if you had an array of *PatientRecords*, each containing a patient record with an amount due, you could write a function that returns the total amount due for all the patient records in the array:

```
FUNCTION TotalAmountDue(Patients() AS PatientRecord)
    DIM total AS DOUBLE
    RESET total
    FOR ix = LBOUND(Patients) TO UBOUND(Patients)
        total = total + Patients(ix).AmountDue
    NEXT
    TotalAmountDue = total
END FUNCTION
```

You might call the function like this:

```
DIM Patients(1 TO 100) AS PatientRecord
' more code here
x$ = "Total amount due:" + STR$(TotalAmountDue(Patients()))
```

See Also

[User-Defined Types \(UDTs\)](#)

[Storage requirements and restrictions](#)

[Unions](#)

Storage requirements and restrictions

[Top](#) [Previous](#) [Next](#)

You can determine the amount of storage required for a variable of a [User-Defined Type](#) using the [LEN](#) function. To determine the requirements for a student record, for example, use:

```
RecordSize = LEN(Student)
```

The address of a record [variable](#), as returned by the [VARPTR](#) function, is the address in memory of the first byte of data in the record. You can also obtain the starting address of the fields within the record by passing the full name of the field (*Student.IdNum*, for example) to the [VARPTR](#) function.

A single UDT structure is limited to 16 MB (16,777,216 bytes). [Locally](#) dimensioned [UDT](#) structures are limited to the amount of free stack space available, typically less than 1 MB. If larger UDT structures are required, use a [STATIC](#) or [GLOBAL](#) declaration instead (since these are not stored on the stack). The same rules apply to [Unions](#) (and LOCAL [fixed-length](#) and [ASCIIZ](#) strings).

Note that the array statements cannot be directly used on [arrays](#) within UDTs. However, you can use [DIM..AT](#) to define an array (of the same data type) at the address of the UDT array, and employ [ARRAY](#) statements on that array. The [ARRAY](#) statements can be used on arrays of UDT structures. An individual array within a UDT may occupy as much as the full 16 MB UDT size limit.

See Also

[User-Defined Types \(UDTs\)](#)

[Unions](#)

Built-in User-Defined Types

[Top](#) [Previous](#) [Next](#)

The compiler provides a set of built-in User-Defined Types, including:

```
TYPE DispParams
  VariantArgs AS VARIANT
  NamedDispID AS VARIANT
  CountArgs   AS DWORD
  CountNamed  AS DWORD
END TYPE
```

DispParams is used internally by the compiler to send parameters to Dispatch methods and properties.

```
TYPE DirData
  FileAttributes AS DWORD
  CreationTime   AS QUAD
  LastAccessTime AS QUAD
  LastWriteTime  AS QUAD
  FileSizeHigh   AS DWORD
  FileSizeLow    AS DWORD
  Reserved0      AS DWORD
  Reserved1      AS DWORD
  FileName       AS ASCIIZ * 260
  ShortName      AS ASCIIZ * 14
END TYPE
```

DirData is used with the DIR\$ function to retrieve file or directory information.

```
TYPE Point
  x AS LONG
  y AS LONG
END TYPE
```

Used with the NMHDR User-Defined type (see below).

```
TYPE NMHDR
  HwndFrom AS DWORD
  IdFrom   AS DWORD
  Code     AS LONG
END TYPE
```

NMHDR is used with CB.NMHDR and contains information about notification messages.

```
TYPE NMCHAR
  Hdr      AS NMHDR
  Ch       AS DWORD
  dwItemPrev AS DWORD
  dwItemNext AS DWORD
END TYPE
```

NMCHAR is used with CB.NMHDR and contains information about a character notification messages.

```
TYPE NMKEY
  Hdr AS NMHDR
```



```
nVKey    AS DWORD
uFlags   AS DWORD
END TYPE
```

NMKEY is used with CB.NMHDR and contains information about key notification messages.

```
TYPE NMMOUSE
  Hdr          AS NMHDR
  dwItemSpec   AS DWORD
  dwItemData    AS DWORD
  Pt           AS POINT
  dwHitInfo    AS LONG
END TYPE
```

NM_MOUSE is used with CB.NMHDR and contains information about key notification messages.

```
TYPE NMTOOLTIPS_CREATED
  Hdr          AS NMHDR
  hwndToolTips AS DWORD
END TYPE
```

NMTOOLTIPS_CREATED is used with CB.NMHDR and contains information about %NM_TOOLTIPSCREATED messages.

See Also

[Built-in numeric equates](#)

[Built-in string equates](#)

[Built-in RGB Color Equates](#)

If you have ever programmed in Pascal or C, you may be familiar with the concept of a Union. A Union is similar in some ways to a [User-Defined Type](#). Both have data fields that can be made up of any of Classic PowerBASIC's data types, including records and other Unions, and except for the UNION keyword, they are defined the same way. The major difference between User-Defined Types and Unions, is that each field within a Union occupies the same memory location as all the others.

While the concept may appear abstract, Unions provide an avenue to freely convert data from one format to another, simply by writing the data into the Union as one data format, and reading the data back as another. Combining the versatility of a UDT with the flexibility of a Union can extend this functionality dramatically, such as splitting data into its component parts.

For example, the following definition would create a Union called *WordFld* and a *WordFld* variable called *MyVar*:

```
TYPE HiLo
  Lo AS BYTE
  Hi AS BYTE
END TYPE

UNION WordFld
  Whole AS WORD
  Part  AS HiLo
END UNION

DIM MyVar AS WordFld

MyVar.Whole = &HBC1F      'assign a value to the entire word
a$ = HEX$(MyVar.Part.Hi)  'returns Hi byte of the word
b$ = HEX$(MyVar.Part.Lo)  'returns Lo byte of the word
```

When you access the field *MyVar.Whole*, you are reading the entire contents of the Union as a word. On the other hand, when you refer to *MyVar.Part.Hi*, you are referring to the high byte of *MyVar*.

See Also

[User-Defined Types \(UDTs\)](#)

[Union Storage requirements and restrictions](#)

Storage requirements and restrictions

[Top](#) [Previous](#) [Next](#)

A single [Union](#) structure is limited to 16 MB (16,777,216 bytes). [Locally](#) dimensioned Union structures are limited to the amount of free [stack](#) space available, typically less than 1 MB. If larger [UDT](#) structures are required, use a [STATIC](#), [GLOBAL](#), or [INSTANCE](#) declaration instead, since these are not created on the stack. The same rules apply to User-Defined Types (and LOCAL [fixed-length](#) and [ASCIIZ](#) strings). An individual [array](#) within a Union may occupy as much as the full 16 MB Union size limit.

See Also

[User-Defined Types \(UDTs\)](#)

[Unions](#)

A pointer is a variable that holds the 32-bit (4 [byte](#)) address of code or data located elsewhere in memory. It is called a pointer because it literally *points* to that location. The location pointed to is known as the *target* of the pointer.

Pointers represent a powerful addition to the BASIC programmer's arsenal. The address is defined at run-time, so your program can reference any memory location as if it were a standard [variable](#). When a pointer is used to access a memory location, it is called "indirect addressing".

Pointers are declared using the [DIM](#) statement, and the type of the target must be specified. The keywords PTR and POINTER are synonymous.

```
DIM i AS INTEGER PTR 'declares i as a pointer to an Integer
```

or:

```
DIM i AS INTEGER POINTER
```

The above example declares *i* as an [Integer](#) pointer. Before it can be used, *i* must be initialized with an actual address of a [variable](#) (easily done with the [VARPTR](#) function; or [STRPTR](#) for string). When you assign a value to a pointer variable, you are giving it an address to use later when you wish to reference the actual target. A pointer's name alone references the pointer variable. A pointer's name with an at sign (@) prefix, references the pointer's target:

```
DIM Ptr1 AS BYTE PTR ' declares Ptr1 as a byte pointer
DIM Ptr2 AS BYTE PTR ' declares Ptr2 as a byte pointer
DIM Byte1 AS BYTE    ' Declares Byte1 as a byte variable
DIM Byte2 AS BYTE    ' Declares Byte2 as a byte variable
Ptr1 = VARPTR(Byte1) ' Ptr1 points to Byte1
@Ptr1 = 36           ' Sets Byte1 to the value 36
Ptr2 = VARPTR(Byte2) ' Ptr2 points to Byte2
@Ptr2 = @Ptr1 + 4    ' Sets Byte2 to 40 (36 + 4)
```

In summary, when you reference a pointer variable *without* an at-sign, you are referencing the 32-bit address contained in it. When you precede the name *with* an at-sign, you are referencing the target data located at the address "pointed to" by the pointer.

By assigning the address of another pointer to a pointer, we can set up another level of indirection. *Pointers to pointers* are useful when setting up linked lists in memory. You can then access the target by adding a second at-sign in front of the pointer's name:

```
DIM y AS STRING POINTER
DIM z AS STRING POINTER
DIM TmpStr AS STRING
y = VARPTR(TmpStr) ' y points to TmpStr
z = VARPTR(y)      ' z points to y
@y = "A"           ' put an "A" in TmpStr
@@z = "B"          ' overwrite it with a "B"
Display @y         ' display the target value of y
```

PowerBASIC supports up to 200 levels of indirection. For each level, you add another preceding at-sign to the pointer name. You can only use the (@) prefix with pointer

variables.

A pointer with a value of zero (0) is considered a null-pointer by PowerBASIC.

Windows will generate a General Protection Fault (GPF) if you attempt to access data at an invalid pointer address. See the section on [assembler programming](#) for more information.

The true power of pointers resides in their speed and flexibility. Traditionally, to access memory, a BASIC programmer had to use combinations of [PEEK](#) and [POKE](#). This allowed the programmer to address memory as bytes. If the target data took any other form, conversion was necessary. Pointers allow you to address the target data in any fashion you desire, even as a [user-defined structure](#). Moreover, because the setup of calling PEEK and POKE is no longer necessary, access is much faster.

Let's say that we want to scan all the characters in a buffer, replacing all upper case "A"s with lower case "a"s. The code might look something like this:

```
SUB Lower(zStr AS STRING)
  DIM s AS BYTE PTR, ix AS INTEGER
  s = STRPTR(zStr) ' Access the dynamic string directly
  FOR ix = 1 TO LEN(zStr)
    IF @s = 65 THEN @s = 97 ' "A" -> "a"
    INCR s
  NEXT
END SUB
```

When using a pointer to a structure, the prefix is placed before the structure name when you wish to access an element of the structure. The structure name by itself refers to its address. This distinction is extremely important when treating structures as a whole. The following example shows two ways of doing a simple bubble sort of an [array](#) of [User-Defined Types](#). The first uses conventional BASIC methods, the second uses pointers to illustrate their speed and efficiency.

```
'-- Example 1 -----
#COMPILE EXE
#DIM ALL
TYPE NameRec
  Last   AS STRING * 20   ' Last name
  First  AS STRING * 20   ' First name
END TYPE
FUNCTION PBMAIN () AS LONG
  DIM Rec(1 TO 10) AS NameRec
  DIM RP AS NameRec POINTER
  DIM ix AS LONG, ij AS LONG
  DIM hFile AS DWORD
  '-- Put some data in the records --
  FOR ix = 1 TO 10
    Rec(ix).First = CHOOSE$(ix,"Jacob","Michael","Joshua","Matthew","Ethan", _
                             "Emily","Emma","Madison","Abigail","Olivia")
    Rec(ix).Last  = CHOOSE$(ix,"SMITH","JOHNSON","WILLIAMS","JONES","BROWN", _
                             "DAVIS","MILLER","WILSON","MOORE","TAYLOR")
  NEXT ix
  '-- Sort UDT array in ascending order using a bubble sort
  '-- ARRAY SORT Rec(),FROM 1 TO 20,ASCEND will do this as well
  FOR ix = 9 TO 1 STEP -1
```

```

        FOR ij = 1 TO ix
            IF Rec(ij-1).Last > Rec(ij).Last THEN
                SWAP Rec(ij-1), rec(ij)
            END IF
        NEXT ij
    NEXT ix
#IF %DEF(%PB_CC32)
    FOR ix = 1 TO 10
        PRINT TRIM$(Rec(ix).Last) + ", " + TRIM$(Rec(ix).First)
    NEXT ix
    PRINT
    PRINT "Press any key to quit ... "
    WAITKEY$
#ELSE
    DIM msg AS STRING
    FOR ix = 1 TO 10
        msg = msg + TRIM$(Rec(ix).Last) + ", " + TRIM$(Rec(ix).First) + $CRLF
    NEXT ix
    MSGBOX msg
#ENDIF
END FUNCTION

```

'-- Example 2 -----

' The difference between example 1 and this example is
' that we're manipulating pointers (4 bytes) instead
' of whole records (40 bytes).

#COMPILE EXE

#DIM ALL

TYPE NameRec

```

    Last   AS STRING * 20   ' Last name
    First  AS STRING * 20   ' First name

```

END TYPE

FUNCTION PBMAIN () AS LONG

```

    DIM Rec(1 TO 10) AS NameRec
    DIM RP AS NameRec POINTER
    DIM ix AS LONG, ij AS LONG
    DIM hFile AS DWORD

```

'-- Put some data in the records --

```

FOR ix = 1 TO 10
    Rec(ix).First = CHOOSE$(ix,"Jacob","Michael","Joshua","Matthew","Ethan", _
                             "Emily","Emma","Madison","Abigail","Olivia")
    Rec(ix).Last  = CHOOSE$(ix,"SMITH","JOHNSON","WILLIAMS","JONES","BROWN",
_
                             "DAVIS","MILLER","WILSON","MOORE","TAYLOR")

```

NEXT ix

'-- Sort UDT array in ascending order using a bubble sort with pointers
'-- note a bubble sort is not recommended for large collections
'-- and note ARRAY SORT Rec(),FROM 1 TO 20,ASCEND will do this as well
'-- so this is only to show pointers to UDT arrays in action!

RP = VARPTR(Rec(1))

FOR ix = 9 TO 1 STEP -1

FOR ij = 1 TO ix

'note pointers to array elements use zero based subscripts in brackets!

IF @RP[ij-1].Last > @RP[ij].Last THEN

SWAP @RP[ij-1], @RP[ij]

END IF

NEXT ij

NEXT ix

#IF %DEF(%PB_CC32)

FOR ix = 1 TO 10

```

        PRINT TRIM$(Rec(ix).Last)+ ", " +TRIM$(Rec(ix).First)
    NEXT ix
    PRINT
    PRINT "Press any key to quit ... "
    WAITKEY$
#ELSE
    DIM msg AS STRING
    FOR ix = 1 TO 10
        msg = msg + TRIM$(Rec(ix).Last)+ ", " +TRIM$(Rec(ix).First) + $CRLF
        MSGBOX msg
    NEXT ix
#ENDIF
END FUNCTION

```

If you declare a member of a structure as a pointer, the @ prefix is used with the member name, not the structure name. The previous example could be improved by adding a couple of pointers to the structure to point to the previous and next record, respectively. This lets you allocate memory for a record only when needed, instead of pre-allocating a fixed-size array of records. The modified structure would look something like this:

```

TYPE NameRec
    Last AS STRING * 20      ' Last name
    First AS STRING * 20     ' First name
    Nxt AS NameRec PTR       ' Pointer to next record
    Prv AS NameRec PTR       ' Pointer to previous record
END TYPE
DIM Rec AS NameRec

```

The pointer members are then accessed like this:

```

Rec.@Nxt      ' next record
Rec.@Prv      ' previous record

```

Putting the @ prefix in front of the structure name (i.e., @Rec) would cause a [compile-time error](#), as Rec itself is not a pointer.

When calculating the length of the Type, all pointers are internally stored as Double-word ([DWORD](#)) variables, so NameRec is 48 bytes long (20 + 20 + 4 + 4). If you need to know the length of a Type, it is easier to let PowerBASIC calculate it for you using the [LEN](#) function than doing it yourself:

```
length = LEN(structure)
```

See Also

[Pointers to ASCIIZ and fixed-length strings](#)

[Pointers to arrays](#)

[Pointers to arrays with dual indexes](#)

Pointers to ASCIIZ and fixed-length strings

[Top](#) [Previous](#)
[Next](#)

A declaration of an [ASCIIZ](#) or [fixed-length](#) string must explicitly state the maximum length of the string, in order for the compiler to allocate memory accordingly. When declaring [pointers](#) to fixed-length strings, you may also state the maximum length of the string. This will allow [INCR](#) and [DECR](#) to move the pointer to the next or previous string, respectively. If you do not supply the length for a fixed-length string pointer, INCR and DECR will move the pointer by one [byte](#).

However, with an ASCIIZ string pointer, the length limit may be explicitly stated, or it may be left as an ambiguous value, by skipping the length clause entirely. For example, the following lines are valid:

```
DIM x AS ASCIIZ PTR * 41
DIM y AS ASCIIZ PTR * 2
DIM z AS ASCIIZ PTR
```

This rule applies to scalar pointers, [arrays](#) of pointers, pointers as function parameters, and pointers as members of a [User-Defined Type](#) or [Union](#). If the optional length limit is specified, Classic PowerBASIC will always truncate a string assignment to fit correctly in the memory allocated to the variable.

If the length is ambiguous, it becomes the programmer's responsibility to ensure the target buffer is not overflowed leading to memory corruption or General Protection Faults (GPF). Use caution in this case.

See Also

[Pointers \(@\)](#)

[Pointers to arrays](#)

[Pointers to arrays with dual indexes](#)

Pointers to arrays

[Top](#) [Previous](#) [Next](#)

In order to work with [arrays](#) created by other languages, such as VB arrays, Classic PowerBASIC supports an extension of the [pointer](#) syntax, called "Pointer Indexing". As noted above, a pointer allows you access a data element at specific address in memory. An index pointer allows you to access data elements beyond the memory address in the base pointer. Consider an array with 6 elements:

```
DIM x%(0 TO 5)
```

The address of the first element, `x%(0)`, is the base with the remaining elements stored in memory, one after the other. To access the array using an index pointer, you simply assign the address of the first element to your base pointer:

```
DIM xPtr AS INTEGER POINTER
xPtr = VARPTR(x%(0))

@xPtr[0] = 0      ' same as x%(0)
@xPtr[1] = 1      ' same as x%(1)
@xPtr[2] = 2      ' same as x%(2)
@xPtr[3] = 3      ' same as x%(3)
@xPtr[4] = 4      ' same as x%(4)
@xPtr[5] = 5      ' same as x%(5)
```

Note the syntax used to access the elements of the array. It consists of the pointer's name, with the `@` prefix, and followed by the array index in square brackets. The number used inside of the brackets is a multiplier. The number inside the brackets is multiplied by the size of the target data (a two byte [Integer](#) in this case) to calculate the target address.

The primary differences between arrays and index pointers are that index pointers do not allocate any memory of their own - they use memory which has already been allocated elsewhere. Their [lower bound](#) is always zero. For example, you can [dimension](#) your six-element array from 1990 to 1995. However, to access the array data using an index pointer, you will still need to use 0 through 5:

```
x%(1990 TO 1995)
DIM xPtr AS INTEGER PTR
xPtr = VARPTR(x%(1990))

@xPtr[0] = 0      ' same as x%(1990)
@xPtr[1] = 1      ' same as x%(1991)
@xPtr[2] = 2      ' same as x%(1992)
@xPtr[3] = 3      ' same as x%(1993)
@xPtr[4] = 4      ' same as x%(1994)
@xPtr[5] = 5      ' same as x%(1995)
```

Consider the following VB code:

```
Sub Sum_Click()
  ReDim PriceData!(1 TO TotalElements%)
  Call FillSumArray(PriceData!())
  Total! = GetSum(PriceData!(1), TotalElements%)
End Sub
```

When the "Sum" button is pressed, a dynamic array is created and filled. *FillSumArray()* is VB code to read the price data from a database file and place it into the array. *GetSum()* is Classic PowerBASIC code to add up all of the prices and return the total, since Classic

PowerBASIC handles calculations faster than VB does.

```
FUNCTION GetSum!(Price!, BYVAL TotalElements%) EXPORT
    DIM PriceData AS SINGLE PTR
    DIM Total!
    DIM k%
    PriceData = VARPTR(Price!)
    FOR k% = 0 TO TotalElements% - 1
        Total! = Total! + @PriceData[k%]
    NEXT
    GetSum! = Total!
END FUNCTION
```

In the above example, *GetSum!* takes the Visual Basic array, adds up all the values, and returns the total as a result. Since a pointer is a memory address, we need the memory address of the first element in the array. In VB, you can pass the memory address of a variable by passing it "by reference", or BYREF. This tells Visual Basic not to pass the value of a variable to a [Sub](#), [Function](#), [Method](#), or [Property](#), but to pass the address in memory where the variable is located. This is handled through the DECLARE statement in Visual Basic.

```
DECLARE FUNCTION GetSum! LIB "SUMS.DLL" (Prices!, BYVAL Elements%)
```

By not using the BYVAL keyword before the variable Prices!, we've told Visual Basic to pass a memory address to the variable. You'll notice in the DECLARE statement that the variable Prices! does not include any parentheses to indicate that it is an array. If we were to change it to Prices!(), VB would pass a handle to an array descriptor, not an address to the array data. The Classic PowerBASIC code also needs to know how many elements there are in the array, so that is passed as the second parameter.

Since only the first element of the array is passed to *GetSum!*, we'll need to use a pointer to access the remainder of the elements.

```
DIM PriceData AS SINGLE PTR
```

Remember that all pointers are initialized to null (zero). To access the array, we need to assign the memory address for the element passed. [VARPTR](#) is used to get the address of the passed element.

```
PriceData = VARPTR(Price!)
```

An indexed pointer can then be used to access all of the elements in the array. The VB array was dimensioned from *1 to TotalElements*; however indexed pointers in Classic PowerBASIC all start with a subscript of zero. So to reconcile the difference, we subtract the lower bound (1) from *TotalElements* in our [FOR/NEXT](#) loop. A [DIM](#) statement is not required to access an array using this method.

```
FOR k% = 0 TO Elems% - 1
    Total! = Total! + @PriceData[k%]
NEXT
```

It is also possible to use indexed-pointers with [dynamic string](#) arrays. For example:

```
DIM Arr1(1 TO 3) AS STRING
DIM pArr1 AS STRING POINTER

Arr1(1) = "a1"
Arr1(2) = "a2"
Arr1(3) = "a3"
```

```
PArr1 = VARPTR(Arr1(1)) ' The 1st array element  
DisplayText @pArr1[2]  ' This references Arr1(3)
```

Indexed pointers make it easy to manipulate arrays created by other languages such as VB, Delphi, C/C++, etc.

See Also

[Pointers \(@\)](#)

[Pointers to ASCIIZ and fixed-length strings](#)

[Pointers to arrays with dual indexes](#)

Pointers to arrays with dual indexes

[Top](#) [Previous](#) [Next](#)

Indexed [pointers](#) with [dual indexes](#) require an "OF *limit*" clause on both indexes. While simple [arrays](#) (arrays with one index) store data sequentially, dual indexes interleave each row of data. The *OF* clause is used by the compiler to calculate the size of each row and column. *limit* is the [upper bound](#) of the index (zero-based):

```
DIM DataPtr AS INTEGER PTR
DIM z%(0 TO 8, 0 TO 3)
DataPtr = VARPTR(z%(0, 0))
FOR y = 0 TO 3
  FOR x = 0 TO 8
    Value% = @DataPtr[x OF 8, y OF 3]
  NEXT x
NEXT y
```

The following example uses a [lower bound](#) other than zero:

```
DIM DataPtr AS INTEGER PTR
DIM z%(1990 TO 1998, -1 TO 3)
DataPtr = VARPTR(z%(1990, -1))
FOR y = 0 TO 4
  FOR x = 0 TO 8
    Value% = @DataPtr[x OF 8, y OF 4]
  NEXT x
NEXT y
```

If you subtract the lower bound from itself and the upper bound (to get a lower bound of zero), you get 8 for the upper bound, which is then used for *limit* after the *OF* keyword.

See Also

[Pointers \(@\)](#)

[Pointers to ASCIIZ and fixed-length strings](#)

[Pointers to arrays](#)

Constants and Literals

A string literal is simply a group of characters surrounded by quotation marks. For example:

```
"This is a string"
```

```
"3.14159"
```

```
"John Jones, Attorney at Law"
```

A string literal can include the double-quote character, simply by doubling the character within the string. For example:

```
A$ = "This is a ""string"""
```

Numeric literals represent numeric values. They consist primarily of the digits 0 through 9 and a decimal point. Negative values need a leading minus sign (-); a plus sign (+) is optional for positive values. The amount of precision you supply determines the internal representation ([Integer](#), [Long-integer](#), [Quad-integer](#), [Byte](#), [Word](#), [Double-word](#), [Single-precision](#), [Double-precision](#), [Extended-precision](#), and [Currency](#)) which Classic PowerBASIC will use in processing that literal value.

You can also force a literal value to be stored with a given precision by following the constant with one of the variable type-specifiers (% , & , && , ? , ?? , ??? , ! , # , ## , @ , @@). This ability becomes very important when working with Currency and other floating-point numbers.

For example, the statement `eVar## = 1.1` stores the Single-precision representation of 1.1 (which is 1.10000002384185791) in the Extended-precision variable `eVar##`. In order to store the exact quantity 1.1 in `eVar##`, you must follow 1.1 with the Extended-precision type-specifier (`##`). For example

```
DIM x1 AS EXT, x2 AS EXT
x1 = 1.1          ' Single-precision literal
x2 = 1.1##       ' Extended-precision literal
a$ = STR$(x1,18)  ' 1.10000002384185791
b$ = STR$(x1)     ' 1.10000002384186
c$ = STR$(x2,18)  ' 1.1
d$ = STR$(x2)     ' 1.1
```

If a type-specifier does not follow a numeric constant, the following rules are used to determine the precision the value will be stored in:

1. If the value contains no decimal point and is in the range 0 to 255, Classic PowerBASIC stores the value as a Byte.
2. If the value is an integer in the range -32,768 to 32,767, yet outside the range for Byte constants, Classic PowerBASIC stores the value as an Integer.
3. If the value is an integer in the range 32,768 to 65,535, Classic PowerBASIC will store the value as a Word.
4. If the value is an integer in the range -2^{31} to $2^{31}-1$ inclusive (about -2 billion to +2 billion), yet outside the range for Word constants, Classic PowerBASIC stores the value as a Long-integer.
5. If an integer value is positive, exceeds the maximum value for a Long-integer, and still falls within the range for a Double-word, Classic PowerBASIC will store the value as a

Double-word.

6. If the value is an integer too large to fit in a Long or Double-word, but small enough to fit in a Quad integer, it will be stored as a Quad integer.
7. If the value contains a decimal point and has up to six significant digits, Classic PowerBASIC stores it as a Single-precision floating-point.
8. A numeric constant with a decimal point and more than six significant digits, but less than 17, or a whole number too large to be a Quad-integer but small enough to fall within the range of Double-precision floating-point, is stored in Double-precision floating-point format. Larger values (with up to 18 significant digits) are stored in Extended-precision format.

For example:

```
345.1           ' A Single-precision constant
1.10321         ' A Single-precision constant
1.103213        ' A Double-precision constant
3453212.1234     ' A Double-precision constant
1112223.4445556667 ' An Extended-precision constant
```

When the sign of an Integer constant is not apparent (there is no type-specifier), Classic PowerBASIC uses the following rules to determine whether to store the value as signed or unsigned:

If the number includes a type-specifier, the value will be signed or unsigned according to the type (specifically: Byte, Word and Double-word are unsigned; Integer, Long-integer, Quad-integer are signed).

If there is no type-specifier, and the number is a 16-bit quantity expressed as exactly 4 hexadecimal digits (or the number is a 32-bit quantity expressed as exactly 8 hexadecimal digits) and the most significant bit is set, the value is considered to be signed. All other hexadecimal constants are evaluated as unsigned. The same rules apply to 16 and 32 bit binary and octal literals.

Such signed hexadecimal values can be forced to evaluate as unsigned by adding an additional leading zero digit, or by adding a type-specifier suffix. This behavior is designed for compatibility with other BASIC dialects.

Some examples of these rules follow:

```
32767??        ' A Word constant           (unsigned)
-40000         ' A Long-integer constant    (signed)
32             ' A Byte constant            (unsigned)
-32            ' An Integer constant         (signed)
&H08000        ' A Word constant            (unsigned)
&H8000         ' An Integer constant         (signed)
&H8000&        ' An Integer constant         (signed)
&H08000&       ' An Integer constant         (signed)
&H80000000     ' Long-integer constant         (signed)
&H80000000&&   ' Long-integer constant         (signed)
&H080000000    ' Double-word constant            (unsigned)
&H800000000??? ' Double-word constant            (unsigned)
```

This sequence of events allows Classic PowerBASIC to make an intelligent decision about

the constants in your program. Rather than arbitrarily making all ambiguous constant references signed (or unsigned), it tries to determine what type of constant you intended to use. You do not necessarily have to make a decision about the size of an integer constant, only whether it should be signed or unsigned. All you need to do to guarantee that a constant will be treated as an unsigned value is to place a leading zero in the number. You need not consider whether the number is a Byte, Word, or Double-word (as long as you do not exceed the largest Double-word value).

It is sometimes convenient to express integers in number systems (bases) other than decimal (which is base 10). This is particularly true when expressing information that is binary in nature; for example, machine addresses. Classic PowerBASIC allows you to specify integer data in Hexadecimal (base 16), Octal (base 8), and Binary (base 2) notation.

Hexadecimal constants consist of up to 16 characters, where each character is from the set 0 through 9 and A through F (and a through f), and must be preceded by &H. An additional (leading) zero can also be included to force the hexadecimal value to be treated as an unsigned value, or a suitable type-specifier can be added instead. The following are equivalent ways of specifying an unsigned hexadecimal value in the Double-word (DWORD) range:

```
A??? = &H0FFFFFFFFF
A??? = &HFFFFFFFF???
```

Octal constants contain only the characters 0 through 7, can be up to 22 digits long, and must be preceded by &O, &Q, or simply &.

```
B = &Q7777
```

Binary constants contain only 0s and 1s, can be up to 64 digits long, and must be preceded by &B.

Each of the following constants represents the integer value 256 (decimal):

```
256 ~ &H100 ~ &O400 ~ &Q400 ~ &400 ~ &B100000000
```

Just as with decimal constants, a hexadecimal, octal or binary constant may have a type-specifier. You can use ?, ??, ???, %, &, && (Byte, Word, Double-word, Integer, Long-integer, Quad-integer) type-specifiers with constants in any number base.

If you do not use a type-specifier with these constants, the compiler will select the smallest integer type that will contain the number. When there is no type-specifier, Classic PowerBASIC stores the value as unsigned, unless the most significant bit is a sign bit and the leading digit is not a zero. For example, &H8000 is signed, because its most significant bit is a sign bit (1000 0000 0000 0000), and its leading digit is non-zero. On the other hand, &H08000 is unsigned; although its most significant bit is a sign-bit (1000 0000 0000 0000), its leading digit is a zero.

You can use the VAL function to convert strings to numeric values. Such strings can contain decimal, hexadecimal, binary and octal numbers in string format. See VAL function for more info.

Classic PowerBASIC programs process two distinct classes of data: *variables* and *constants*. A variable is allowed to change its value as a program runs. A constant's value is fixed at compile-time, and cannot change during program execution (hence, it remains constant). Classic PowerBASIC supports four types of constants: string literals, numeric literals, string equates and numeric equates.

- [Numeric Equates](#)
- [String Equates](#)

See Also

[Defining Constants](#)

[Array Data Types](#)

[Bit Data Types](#)

[GUID Data Types](#)

[Object Data Types](#)

[Pointers](#)

[User Defined Types](#)

[Unions](#)

[Variant Data Types](#)

PB/Win [constants](#) (also known as equates) are defined by prefixing the name of the constant with a "%" character. MSBASIC and VB define constants with the CONST keyword. The MSBASIC/VB compiler then does type conversions at the point of use, if the constant's type was not specified. That overhead does not happen (and is not necessary) with PB/Win. [String equates](#) are specified with a leading "\$" character.

However, the [MACRO](#) facilities in PB/Win offer a way to retain the CONST syntax in your code, while maintaining the low overhead advantage of Classic PowerBASIC. For example:

```
MACRO CONST = MACRO
[statements]
CONST Something = 1&
CONST Something_Else = 2???
CONST AppTitle = "My Application"
[statements]
MSGBOX FORMAT$(Something), ,AppTitle
```

During compilation, the CONST keyword is replaced by the MACRO word, which dynamically creates a new macro that, in turn, defines a constant.

See Also

[Constants and Literals](#)

[Numeric Equates](#)

[Built-in numeric equates](#)

[Built In RGB Color Equates](#)

[String Equates](#)

[Built-in string equates](#)

Classic PowerBASIC allows you to refer to numeric constants by name. Be aware that equates have [global scope](#); that is, they are visible throughout your program. Unlike [variables](#), you can use an equate on the left side of an assignment statement only once, and only a constant value (or a simple variable expression) may be assigned to it. If an expression is used, all parts of the expression must consist of constants, numeric equates; bitwise operators like [AND](#), [OR](#); and [NOT](#); the [arithmetic operators](#) +, -, *, /, and \, and the [relational operators](#) >, <, >=, <=, <>, =; and the [CVQ](#) function. For example, the following are all legal equate definitions:

```
%X = 1
%Y = 1 + 1
%Z = %X * %Y
%Q = (1& OR 2&) + (NOT 0)
%R = (%Q <> 100&)
%S = CVQ("DemoOnly")
```

If you append the words AS COM to a numeric equate assignment, the equate definition will be included in any [TYPE LIBRARY](#) which may be generated from this program.

```
%SCROLL_FLAG = 99 AS COM
```

A value must be assigned to each equate before it is referenced, even if that value is zero. If you fail to define an equate, an error will be generated during compilation. When assigning a value to an equate, the line of code is never executed. Numeric equates must be created outside of any [SUB](#), [FUNCTION](#), [METHOD](#), or [PROPERTY](#). Numeric equates are global, and may be referenced anywhere in the module. For readability, we suggest placing equates at the top of your code.

You can also use equates to reduce the incidence of "magic numbers" in your programs. Magic numbers are mysterious values that mean something to you when you first write a program, but not when you come back to it six months later. Equates are particularly well suited for making programs more readable. For example, consider an array to track chess pieces. If we define:

```
%MAXPIECES = 32
%NPARAM    = 3
%NTYPE     = 1
%RANK      = 2
%FILE      = 3
%KING      = 1
%PAWN      = 2
```

we can then define an [array](#) of pieces and make statements like the following:

```
DIM piece(1:%MAXPIECES, 1:%NPARAM)
piece(1, %NTYPE) = %KING
piece(1, %RANK)  = 4
piece(1, %FILE)  = 1
```

This sets up a 32 x 3 array for piece information. The first element is the type of unit, the second and third give its current position on the board. Note how much more readable this is than:

```
DIM piece(1:32, 1:3)
```

```

piece(1, 1) = 1
piece(1, 2) = 4
piece(1, 3) = 1

```

We could achieve a similar effect by using [comments](#), but there is no way to ensure that when the program changes, the comments will be updated. Using equates reduces the need for comments.

Besides being more readable, equates allow us to easily change a program by changing only the definition of a single equate, rather than changing every occurrence of a particular value. For example: say you run a preschool, and you want to keep track of some data that depends on how many kids you have. Furthermore, you have to print out reports each week. Rather than type the number in several places, only to have to change it every week, you can assign the number to a constant.

```
%NUMKIDS = 28
```

Then, you can use the constant, `%NUMKIDS`, throughout your program.

```

' Calculate income; the enrollment fee is $85 a week;
' Parents pay whether their kids miss days or not
income% = %NUMKIDS * 85
' Calculate actual attendance
attend% = %NUMKIDS - absent%
' Calculate how much the lunches cost per kid; note the
' use of another constant for cost; it may vary too!
perkid% = %LUNCHCOST / attend%
' Calculate net profit per kid after paying for lunches (you'd
' actually have far more overhead than this, but we'll keep it simple)
net% = (income% - perkid%) / %NUMKIDS
' and so on

```

If your enrollment stays stable, you still have a program that is much easier to follow. Moreover, if your enrollment changes, you only need to change the constant assignment statements to run a revised program. Think of the time you will save - enough to take the kids on an extra field trip.

You might also want to assign the value of an equate conditionally, using the [#IF](#) metastatement. For example:

```

%BIGCLASS = 1
#IF %BIGCLASS
  %NUMKIDS = 40
#ELSE
  %NUMKIDS = 20
#ENDIF

```

Equates make [SELECT](#) statements more readable too:

```

SELECT CASE piece(x, %NTYPE)
CASE %KING
  ' process king moves
CASE %PAWN
  ' process pawn moves
CASE %QUEEN
  ' process queen moves
END SELECT

```

This code will continue to make sense when you return to it after a long absence.

Numeric equates may be assigned a specific integer class if the literal value has a [type-](#)

[specifier](#) appended. For example:

```
%MAX_BYTE      = 255?  
%MAXIMUM_INT    = 32767%  
%MAXIMUM_DWORD  = &HFFFFFFFF???  
%MAXIMUM_LONG   = &H7FFFFFFFF&  
%MINIMUM_LONG   = &H80000000&
```

Numeric equates which are derived from an equation are pre-calculated by Classic PowerBASIC during the compilation process, to ensure that unnecessary calculations are eliminated from the executable code. If this optimization was not performed, Classic PowerBASIC code would need to perform the same calculation every time the equate was used in the code. Examples of numeric equates derived from expressions follows:

```
%WHATEVER1 = 10  
%WHATEVER2 = (%WHATEVER1 * 3) + 1  
%DEBUG      = -1&  
%RELEASE    = NOT %DEBUG  
%DEMO       = %RELEASE AND (NOT %DEBUG)
```

During compilation the actual numeric value of %WHATEVER2 is pre-calculated as 31, and the values of %RELEASE and %DEMO are calculated from the value of %DEBUG. Note that operators like AND and OR work as bitwise operators, rather than logical operators, in numeric equate assignments.

Duplicate definitions of both numeric and string equates are permitted by Classic PowerBASIC, provided the actual equate content is identical. If the content is not identical, a compile-time Error 468 ("Duplicate Equate") will occur.

See Also

[Constants and Literals](#)

[Defining Constants](#)

[Built-in numeric equates](#)

[String Equates](#)

[Built-in string equates](#)

Built-in numeric equates

[Top](#) [Previous](#) [Next](#)

The compiler provides a convenient set of built-in [numeric equates](#).

For determining the compiler version (see [%DEF](#) for more information):

`%PB_CC32, %PB_DLL32, %PB_EXE, %PB_REVISION, %PB_REVLETTER, %PB_WIN32`

For use with [ARRAYATTR](#):

`%VARCLASS_BYT, %VARCLASS_WRD, %VARCLASS_DWD, %VARCLASS_INT, %VARCLASS_LNG,
%VARCLASS_QUD, %VARCLASS_SNG, %VARCLASS_DBL, %VARCLASS_EXT, %VARCLASS_CUR,
%VARCLASS_CUX, %VARCLASS_VRNT, %VARCLASS_IFAC, %VARCLASS_TYPE, %VARCLASS_ASC,
%VARCLASS_FIX, %VARCLASS_STR, %VARCLASS_FLD`

For use with [BUTTONS](#):

`%BN_CLICKED, %BN_DBLCLK, %BN_DISABLE, %BN_DOUBLECLICKED, %BN_HILITE, %BN_KILLFOCUS,
%BN_PAINT, %BN_SETFOCUS, %BN_UNHILITE, %IDOK, %IDCANCEL, %IDABORT, %IDRETRY,
%IDIGNORE, %IDYES, %IDNO, %IDCLOSE, %IDHELP, %IDTRYAGAIN, %IDCONTINUE, %BS_TEXT,
%BS_PUSHBUTTON, %BS_DEFPUSHBUTTON, %BS_DEFAULT, %BS_CHECKBOX, %BS_AUTOCHECKBOX,
%BS_RADIOBUTTON, %BS_3STATE, %BS_AUTO3STATE, %BS_GROUPBOX, %BS_USERBUTTON,
%BS_AUTORADIOBUTTON, %BS_OWNERDRAW, %BS_LEFTTEXT, %BS_ICON, %BS_BITMAP, %BS_LEFT,
%BS_RIGHT, %BS_CENTER, %BS_TOP, %BS_BOTTOM, %BS_VCENTER, %BS_PUSHLIKE, %BS_MULTILINE,
%BS_NOTIFY, %BS_FLAT, %BS_RIGHTBUTTON`

For use with [Callback](#) functions:

`%NM_OUTOFMEMORY, %NM_CLICK, %NM_DBLCLK, %NM_RETURN, %NM_RCLICK, %NM_RDBLCLK,
%NM_SETFOCUS, %NM_KILLFOCUS, %NM_CUSTOMDRAW, %NM_HOVER, %NM_NCHITTEST, %NM_KEYDOWN,
%NM_RELEASEDCAPTURE, %NM_SETCURSOR, %NM_CHAR, %NM_TOOLTIPS_CREATED, %NM_LDOWN,
%NM_RDOWN, %NM_THEMECHANGED, %SC_SIZE, %SC_MOVE, %SC_MINIMIZE, %SC_MAXIMIZE,
%SC_NEXTWINDOW, %SC_PREVWINDOW, %SC_CLOSE, %SC_VSCROLL, %SC_HSCROLL, %SC_MOUSEMENU,
%SC_KEYMENU, %SC_ARRANGE, %SC_RESTORE, %SC_TASKLIST, %SC_SCREENSAVE, %SC_HOTKEY,
%SC_DEFAULT, %SC_MONITORPOWER, %SC_CONTEXTHELP, %WM_ACTIVATE, %WM_ACTIVATEAPP,
%WM_CANCELMODE, %WM_CAPTURECHANGED, %WM_CHAR, %WM_CLOSE, %WM_COMMAND, %WM_CREATE,
%WM_DESTROY, %WM_DRAWITEM, %WM_HELP, %WM_HSCROLL, %WM_INITDIALOG, %WM_KEYDOWN,
%WM_KEYUP, %WM_KILLFOCUS, %WM_LBUTTONDOWN, %WM_LBUTTONDOWN, %WM_LBUTTONUP,
%WM_MBUTTONDOWN, %WM_MBUTTONDOWN, %WM_MBUTTONUP, %WM_MOUSEACTIVATE, %WM_MOUSEFIRST,
%WM_MOUSEHOVER, %WM_MOUSELAST, %WM_MOUSELEAVE, %WM_MOUSEMOVE, %WM_MOUSEWHEEL,
%WM_MOVE, %WM_NCACTIVATE, %WM_NCCALCSIZE, %WM_NCCREATE, %WM_NCDESTROY, %WM_NCHITTEST,
%WM_NCLBUTTONDOWN, %WM_NCLBUTTONDOWN, %WM_NCLBUTTONUP, %WM_NCMBUTTONDOWN, %WM_NCMBUTTONDOWN,
%WM_NCMBUTTONUP, %WM_NCMBUTTONUP, %WM_NCMOUSEMOVE, %WM_NCPAINT, %WM_NCRBUTTONDOWN,
%WM_NCRBUTTONDOWN, %WM_NCRBUTTONDOWN, %WM_NCRBUTTONDOWN, %WM_NCXBUTTONDOWN, %WM_NCXBUTTONDOWN,
%WM_NCXBUTTONDOWN, %WM_NCXBUTTONDOWN, %WM_NOTIFY, %WM_NULL, %WM_PAINT, %WM_QUIT,
%WM_RBUTTONDOWN, %WM_RBUTTONDOWN, %WM_RBUTTONDOWN, %WM_RBUTTONDOWN, %WM_SIZE, %WM_SYSKEYDOWN,
%WM_SYSKEYUP, %WM_TIMER, %WM_VSCROLL, %WM_USER`

For use with [CONTROL SHOW STATE](#) and [DIALOG SHOW STATE](#):

`%SW_HIDE, %SW_SHOWNORMAL, %SW_NORMAL, %SW_SHOWMINIMIZED, %SW_SHOWMAXIMIZED,
%SW_MAXIMIZE, %SW_SHOWNOACTIVATE, %SW_SHOW, %SW_MINIMIZE, %SW_SHOWMINNOACTIVE,
%SW_SHOWNA, %SW_RESTORE, %SW_SHOWDEFAULT, %SW_FORCEMINIMIZE, %SW_MAX`

For use with [COMBOBOXES](#):

`%CBS_SIMPLE, %CBS_DROPDOWN, %CBS_DROPDOWNLIST, %CBS_OWNERDRAWFIXED,
%CBS_OWNERDRAWVARIABLE, %CBS_AUTOHSCROLL, %CBS_OEMCONVERT, %CBS_SORT,
%CBS_HASSTRINGS, %CBS_NOINTEGRALHEIGHT, %CBS_DISABLENOSCROLL, %CBS_UPPERCASE,
%CBS_LOWERCASE, %CBN_CLOSEUP, %CBN_DBLCLK, %CBN_DROPDOWN, %CBN_EDITCHANGE,
%CBN_EDITUPDATE, %CBN_ERRSPACE, %CBN_KILLFOCUS, %CBN_SELCANCEL, %CBN_SELCHANGE,
%CBN_SELENDOK, %CBN_SETFOCUS`

For use with [DIALOG](#) and/or [CONTROL](#) styles:

`%DLGC_WANTARROWS, %DLGC_WANTTAB, %DLGC_WANTALLKEYS, %DLGC_WANTMESSAGE,`

%DLGC_HASSETSEL, %DLGC_DEFPUSHBUTTON, %DLGC_UNDEFPUSHBUTTON, %DLGC_RADIOBUTTON,
%DLGC_WANTCHARS, %DLGC_STATIC, %DLGC_BUTTON, %DS_ABSALIGN, %DS_SYSMODAL, %DS_3DLOOK,
%DS_FIXEDSYS, %DS_NOFAILCREATE, %DS_LOCALEDIT, %DS_SETFONT, %DS_MODALFRAME,
%DS_NOIDLEMSG, %DS_SETFOREGROUND, %DS_CONTROL, %DS_CENTER, %DS_CENTERMOUSE,
%DS_CONTEXTHELP, %DS_SETFOREGROUND, %WS_OVERLAPPED, %WS_POPUP, %WS_CHILD,
%WS_MINIMIZE, %WS_VISIBLE, %WS_DISABLED, %WS_CLIPSIBLINGS, %WS_CLIPCHILDREN,
%WS_MAXIMIZE, %WS_CAPTION, %WS_BORDER, %WS_DLGFRAME, %WS_VSCROLL, %WS_HSCROLL,
%WS_SYSMENU, %WS_THICKFRAME, %WS_GROUP, %WS_TABSTOP, %WS_MINIMIZEBOX,
%WS_MAXIMIZEBOX, %WS_TILED, %WS_ICONIC, %WS_SIZEBOX, %WS_OVERLAPPEDWIN,
%WS_OVERLAPPEDWINDOW, %WS_TILEDWINDOW, %WS_POPUPWINDOW, %WS_CHILDWINDOW,
%WS_EX_DLGMODALFRAME, %WS_EX_NOPARENTNOTIFY, %WS_EX_TOPMOST, %WS_EX_ACCEPTFILES,
%WS_EX_TRANSPARENT, %WS_EX_TOOLWINDOW, %WS_EX_SMCAPTION, %WS_EX_WINDOWEDGE,
%WS_EX_CLIENTEDGE, %WS_EX_CONTEXTHELP, %WS_EX_RIGHT, %WS_EX_LEFT, %WS_EX_RTLREADING,
%WS_EX_LTRREADING, %WS_EX_LEFTSCROLLBAR, %WS_EX_RIGHTSCROLLBAR, %WS_EX_CONTROLPARENT,
%WS_EX_STATICEDGE, %WS_EX_APPWINDOW, %WS_EX_OVERLAPPEDWINDOW, %WS_EX_PALETTEWINDOW,
%WS_EX_LAYERED, %WS_EX_NOINHERITLAYOUT, %WS_EX_LAYOUTRTL, %WS_EX_COMPOSITED,
%WS_EX_NOACTIVATE

For use with the [DIALOG NEW](#) statement:

%HWNDDESKTOP

For use with the [DIR\\$](#) function:

%NORMAL, %HIDDEN, %SYSTEM, %VLABEL, %SUBDIR

For use with the [DISPLAY BROWSE](#) statement:

%BIF_RETURNONLYFSDIRS, %BIF_DONTGOBELOWDOMAIN, %BIF_RETURNFSANCESTORS, %BIF_EDITBOX,
%BIF_NEWDIALOGSTYLE, %BIF_USENEWUI, %BIF_BROWSEINCLUDEURLS, %BIF_UAHINT,
%BIF_NONNEWFOLDERBUTTON, %BIF_NOTRANSLATETARGETS, %BIF_BROWSEINCLUDEFILES,
%BIF_SHAREABLE

For use with the [DISPLAY COLOR](#) statement:

%CC_FULLOPEN, %CC_PREVENTFULLOPEN, %CC_SHOWHELP

For use with the [DISPLAY FONT](#) statement:

%CF_SCREENFONTS, %CF_PRINTERFONTS, %CF_BOTH, %CF_SHOWHELP, %CF_INITTOLOGFONTSTRUCT,
%CF_USESTYLE, %CF_EFFECTS, %CF_APPLY, %CF_ANSIONLY, %CF_SCRIPTSONLY,
%CF_NOVECTORFONTS, %CF_NOSIMULATIONS, %CF_LIMITSIZE, %CF_FIXEDPITCHONLY, %CF_WYSIWYG,
%CF_FORCEFONTEXIST, %CF_SCALABLEONLY, %CF_TTONLY, %CF_NOFACESEL, %CF_NOSTYLESEL,
%CF_NOSIZESEL, %CF_SELECTSCRIPT, %CF_NOSCRIPTSEL, %CF_NOVERTFONTS

For use with the [DISPLAY OPENFILE](#) and [DISPLAY SAVEFILE](#) statements:

%OFN_ALLOWMULTISELECT, %OFN_CREATEPROMPT, %OFN_DONTADDTORECENT, %OFN_ENABLESIZING,
%OFN_EXPLORER, %OFN_EXTENSIONDIFFERENT, %OFN_FILEMUSTEXIST, %OFN_FORCESHOWHIDDEN,
%OFN_HIDEREADONLY, %OFN_LONGNAMES, %OFN_NODEREFERENCELINKS, %OFN_NOLONGNAMES,
%OFN_NONETWORKBUTTON, %OFN_NOREADONLYRETURN, %OFN_NOTESTFILECREATE, %OFN_NOVALIDATE,
%OFN_OVERWRITEPROMPT, %OFN_PATHMUSTEXIST, %OFN_READONLY, %OFN_SHAREAWARE,
%OFN_SHOWHELP

For use with [ERR](#) and [ERRCLEAR](#):

%ERR_NOERROR, %ERR_ILLEGALFUNCTIONCALL, %ERR_OVERFLOW (reserved), %ERR_OUTOFMEMORY,
%ERR_SUBSCRIPTPOINTEROUTOFRANGE, %ERR_DIVISIONBYZERO (reserved), %ERR_DEVICETIMEOUT,
%ERR_INTERNALERROR, %ERR_BADFILENAMEORNUMBER, %ERR_FILENOTFOUND, %ERR_BADFILEMODE,
%ERR_FILEISOPEN, %ERR_DEVICEIOERROR, %ERR_FILEALREADYEXISTS, %ERR_DISKFULL,
%ERR_INPUTPASTEND, %ERR_BADRECORDNUMBER, %ERR_BADFILENAME, %ERR_TOOMANYFILES,
%ERR_DEVICEUNAVAILABLE, %ERR_COMMERROR, %ERR_PERMISSIONDENIED, %ERR_DISKNOTREADY,
%ERR_DISKMEDIAERROR, %ERR_RENAMEACROSSDISKS, %ERR_PATHFILEACCESSERROR,
%ERR_PATHNOTFOUND, %ERR_OBJECTERROR, %ERR_GLOBALMEMORYCORRUPT (formerly
%ERR_FARHEAPCORRUPT), %ERR_STRINGSPACECORRUPT

For use with [GRAPHIC COPY](#), [GRAPHIC GET MIX](#), [GRAPHIC SET MIX](#), [GRAPHIC STRETCH](#), [XPRINT COPY](#), [XPRINT GET MIX](#), [XPRINT SET MIX](#), and [XPRINT STRETCH](#)

(some statements may accept only a subset of these equates):

%MIX_BLACKNESS, %MIX_NOTMERGESRC, %MIX_MASKNOTSRC, %MIX_NOTCOPYSRC, %MIX_MASKSRCNOT,
%MIX_NOT, %MIX_XORSRC, %MIX_NOTMASKSRC, %MIX_MASKSRC, %MIX_NOTXORSRC, %MIX_NOP,
%MIX_MERGENOTSRC, %MIX_COPYSRC, %MIX_MERGESRCNOT, %MIX_MERGESRC, %MIX_WHITENESS,
%BLACKONWHITE, %WHITEONBLACK, %COLORONCOLOR, %HALFTONE

For use with [GRAPHIC IMAGELIST](#) and [XPRINT IMAGELIST](#):

%ILD_NORMAL, %ILD_TRANSPARENT, %ILD_MASK, %ILD_BLEND25, %ILD_BLEND50, %ILD_IMAGE,
%ILD_ROP, %ILD_OVERLAYMASK

For use with [LABELS](#) and [GRAPHIC CONTROLS](#):

%SS_LEFT, %SS_CENTER, %SS_RIGHT, %SS_ICON, %SS_BLACKRECT, %SS_GRAYRECT,
%SS_WHITERECT, %SS_BLACKFRAME, %SS_GRAYFRAME, %SS_WHITEFRAME, %SS_USERITEM,
%SS_SIMPLE, %SS_LEFTNOWORDWRAP, %SS_NOWORDWRAP, %SS_OWNERDRAW, %SS_BITMAP,
%SS_ENHMETAFILE, %SS_ETCHEDHORZ, %SS_ETCHEDVERT, %SS_ETCHEDFRAME,
%SS_REALSIZECONTROL, %SS_NOPREFIX, %SS_NOTIFY, %SS_CENTERIMAGE, %SS_RIGHTJUST,
%SS_REALSIZEIMAGE, %SS_REALSIZE, %SS_SUNKEN, %SS_ENDELLIPSIS, %SS_PATHELLIPSIS,
%SS_WORDELLIPSIS, %SS_ELLIPSISMASK

For use with [LISTBOXES](#):

%LBN_DBLCLK, %LBN_ERRSPACE, %LBN_KILLFOCUS, %LBN_SELCANCEL, %LBN_SELCHANGE,
%LBN_SETFOCUS, %LBS_NOTIFY, %LBS_SORT, %LBS_NOREDRA, %LBS_MULTIPLESEL,
%LBS_OWNERDRAWFIXED, %LBS_OWNERDRAWVARIABLE, %LBS_HASSTRINGS, %LBS_USETABSTOPS,
%LBS_NOINTEGRALHEIGHT, %LBS_MULTICOLUMN, %LBS_WANTKEYBOARDINPUT, %LBS_EXTENDEDSEL,
%LBS_DISABLENOSCROLL, %LBS_NODATA, %LBS_NOSEL, %LBS_STANDARD

For use with [LISTVIEWS](#):

%LVN_BEGINDRAG, %LVN_BEGINLABELEDIT, %LVN_BEGINRDRAG, %LVN_COLUMNCLICK,
%LVN_DELETEALLITEMS, %LVN_DELETEITEM, %LVN_ENDLABELEDIT, %LVN_GETDISPINFO,
%LVN_INSERTITEM, %LVN_ITEMCHANGED, %LVN_ITEMCHANGING, %LVN_KEYDOWN, %LVN_SETDISPINFO,
%LVS_ALIGNLEFT, %LVS_ALIGNTOP, %LVS_ALIGNMASK, %LVS_AUTOARRANGE, %LVS_EDITLABELS,
%LVS_OWNERDRAWFIXED, %LVS_NOCOLUMNHEADER, %LVS NOSORTHEADER, %LVS_ICON, %LVS_REPORT,
%LVS_SMALLICON, %LVS_LIST, %LVS_TYPMASK, %LVS_SINGLESEL, %LVS_SORTASCENDING,
%LVS_SORTDESCENDING, %LVS_SHAREIMAGELISTS, %LVS_NOLABELWRAP, %LVS_EDITLABELS,
%LVS_OWNERDATA, %LVS_NOSCROLL, %LVS_OWNERDRAWFIXED, %LVS_SHOWSELALWAYS,
%LVS_EX_GRIDLINES, %LVS_EX_SUBITEMIMAGES, %LVS_EX_CHECKBOXES, %LVS_EX_TRACKSELECT,
%LVS_EX_HEADERDRAGDROP, %LVS_EX_FULLROWSELECT, %LVS_EX_ONECLICKACTIVATE,
%LVS_EX_TWOCLICKACTIVATE, %LVS_EX_FLATSB, %LVS_EX_REGIONAL, %LVS_EX_INFOTIP,
%LVS_EX_UNDERLINEHOT, %LVS_EX_UNDERLINECOLD, %LVS_EX_MULTIWORKAREAS,
%LVS_EX_LABELTIP, %LVS_EX_BORDERSELECT, %LVS_EX_DOUBLEBUFFER, %LVS_EX_HIDELABELS,
%LVS_EX_SINGLEROW, %LVS_EX_SNAPTOGRID, %LVS_EX_SIMPLESELECT, %LVNI_ALL,
%LVNI_FOCUSED, %LVNI_SELECTED, %LVNI_CUT, %LVNI_DROPHILITED, %LVNI_ABOVE,
%LVNI_BELOW, %LVNI_TOLEFT, %LVNI_TORIGHT, %LVM_GETSELECTEDCOLUMN,
%LVM_ISGROUPVIEWENABLED, %LVM_GETOUTLINECOLOR, %LVM_SETOUTLINECOLOR,
%LVM_CANCELEDITLABEL, %LVM_MAPINDEXTOID, %LVM_MAPIDTOINDEX, %LVM SETTILEVIEWINFO,
%LVM_GETTILEVIEWINFO, %LVM SETTILEINFO, %LVM_GETTILEINFO, %LVM SETINSERTMARK,
%LVM_GETINSERTMARK, %LVM_INSERTMARKHITTEST, %LVM_GETINSERTMARKRECT,
%LVM SETINSERTMARKCOLOR, %LVM_GETINSERTMARKCOLOR, %LVM SETINFOTIP, %LVM GETHOVERTIME,
%LVM SETTOOLTIPS, %LVM_GETTOOLTIPS, %LVM_SORTITEMSEX, %LVM SETSELECTEDCOLUMN,
%LVM SETTILEWIDTH, %LVM SETVIEW, %LVM_GETVIEW, %LVM GETSUBITEMRECT,
%LVM SUBITEMHITTEST, %LVM SETCOLUMNORDERARRAY, %LVM GETCOLUMNORDERARRAY,
%LVM SETHOTITEM, %LVM GETHOTITEM, %LVM SETHOTCURSOR, %LVM GETHOTCURSOR,
%LVM APPROXIMATEVIEWRECT, %LVM GETSELECTIONMARK, %LVM SETSELECTIONMARK.
%LVM SETBKIMAGE, %LVM GETBKIMAGE, %LVM SETHOVERTIME, %LVM GETTOPINDEX,
%LVM GETCOUNTPERPAGE, %LVM GETORIGIN, %LVM UPDATE, %LVM SETITEMSTATE,
%LVM GETITEMSTATE, %LVM SETITEMTEXT, %LVM GETITEMTEXT, %LVM SETITEMCOUNT,
%LVM SORTITEMS, %LVM SETITEMPOSITION32, %LVM GETSELECTEDCOUNT, %LVM GETITEMSPACING,
%LVM GETSEARCHSTRING, %LVM SETICONSPACING, %LVM SETEXTENDEDLISTVIEWSTYLE,
%LVM GETEXTENDEDLISTVIEWSTYLE, %LVM_ARRANGE, %LVM_EDITLABEL, %LVM_GETEDITCONTROL,
%LVM_GETCOLUMN, %LVM_SETCOLUMN, %LVM_INSERTCOLUMN, %LVM_DELETECOLUMN,

%LVM_GETCOLUMNWIDTH, %LVM_SETCOLUMNWIDTH, %LVM_GETHEADER, %LVM_CREATEDRAGIMAGE,
%LVM_GETVIEWRECT, %LVM_GETTEXTCOLOR, %LVM_SETTEXTCOLOR, %LVM_GETTEXTBKCOLOR,
%LVM_SETTEXTBKCOLOR, %LVM_GETITEM, %LVM_SETITEM, %LVM_INSERTITEM, %LVM_DELETEITEM,
%LVM_DELETEALLITEMS, %LVM_GETCALLBACKMASK, %LVM_SETCALLBACKMASK, %LVM_GETNEXTITEM,
%LVM_FINDITEM, %LVM_GETITEMRECT, %LVM_SETITEMPOSITION, %LVM_GETITEMPOSITION,
%LVM_GETSTRINGWIDTH, %LVM_HITTEST, %LVM_ENSUREVISIBLE, %LVM_SCROLL, %LVM_REDRAWITEMS,
%LVM_GETBKCOLOR, %LVM_SETBKCOLOR, %LVM_GETIMAGELIST, %LVM_SETIMAGELIST,
%LVM_GETITEMCOUNT

For use with [MENU GET STATE](#) and [MENU SET STATE](#):

%MF_CHECKED, %MF_ENABLED, %MF_GRAYED, %MF_DISABLED, %MF_UNHILITE, %MF_HILITE,
%MF_UNCHECKED

For use with MSGBOX:

%MB_OK, %MB_OKCANCEL, %MB_ABORTRETRYIGNORE, %MB_YESNOCANCEL, %MB_YESNO,
%MB_RETRYCANCEL, %MB_CANCELTRYCONTINUE, %MB_ICONHAND, %MB_ICONQUESTION,
%MB_ICONEXCLAMATION, %MB_ICONASTERISK, %MB_USERICON, %MB_ICONWARNING, %MB_ICONERROR,
%MB_ICONINFORMATION, %MB_ICONSTOP, %MB_DEFBUTTON1, %MB_DEFBUTTON2, %MB_DEFBUTTON3,
%MB_DEFBUTTON4, %MB_APPLMODAL, %MB_SYSTEMMODAL, %MB_TASKMODAL, %MB_HELP, %MB_NOFOCUS,
%MB_SETFOREGROUND, %MB_DEFAULT_DESKTOP_ONLY, %MB_TOPMOST, %MB_RIGHT, %MB_RTLREADING,
%MB_SERVICE_NOTIFICATION, %MB_SERVICE_NOTIFICATION_NT3X, %MB_TYPEMASK, %MB_ICONMASK,
%MB_DEFMASK, %MB_MODEMASK, %MB_MISCMASK

For use with [OBJRESULT](#) and [IDISPINFO](#):

%S_OK, %S_FALSE, %E_UNEXPECTED, %E_NOTIMPL, %E_NOINTERFACE, %E_POINTER, %E_ABORT,
%E_FAIL, %E_ACCESSDENIED, %E_HANDLE, %E_OUTOFMEMORY, %E_INVALIDARG,
%DISP_E_ARRAYISLOCKED, %DISP_E_BADINDEX, %DISP_E_BADPARAMCOUNT, %DISP_E_BADVARTYPE,
%DISP_E_EXCEPTION, %DISP_E_MEMBERNOTFOUND, %DISP_E_NONAMEDARGS, %DISP_E_OVERFLOW,
%DISP_E_PARAMNOTFOUND, %DISP_E_TYPERISMATCH, %DISP_E_UNKNOWNINTERFACE,
%DISP_E_UNKNOWNLCID, %DISP_E_UNKNOWNNAME, %DISP_E_PARAMNOTOPTIONAL

For use with [PROCESS GET PRIORITY](#) and [PROCESS SET PRIORITY](#):

%HIGH_PRIORITY_CLASS, %IDLE_PRIORITY_CLASS, %NORMAL_PRIORITY_CLASS,
%REALTIME_PRIORITY_CLASS

For use with [PROGRESSBARS](#):

%PBS_SMOOTH, %PBS_VERTICAL

For use with [SCROLLBARS](#)::

%SB_HORZ, %SB_VERT, %SB_CTL, %SB_BOTH, %SB_LINEUP, %SB_LINELEFT, %SB_LINEDOWN,
%SB_LINERIGHT, %SB_PAGEUP, %SB_PAGELEFT, %SB_PAGEDOWN, %SB_PAGERIGHT,
%SB_THUMBPOSITION, %SB_THUMBTRACK, %SB_TOP, %SB_LEFT, %SB_BOTTOM, %SB_RIGHT,
%SB_ENDSCROLL, %SBS_HORZ, %SBS_VERT, %SBS_TOPALIGN, %SBS_LEFTALIGN, %SBS_BOTTOMALIGN,
%SBS_RIGHTALIGN, %SBS_SIZEBOXTOPLEFTALIGN, %SBS_SIZEBOXTOPRIGHTALIGN,
%SBS_SIZEBOX, %SBS_SIZEGRIP, %SIF_RANGE, %SIF_PAGE, %SIF_POS, %SIF_DISABLENOSCROLL,
%SIF_TRACKPOS, %SIF_ALL, %SBARS_SIZEGRIP, %SBARS_TOOLTIPS

For use with [TAB](#) Controls:

%TCHT_NOWHERE, %TCHT_ONITEMICON, %TCHT_ONITEMLABEL, %TCHT_ONITEM, %TCIF_TEXT,
%TCIF_IMAGE, %TCIF_RTLREADING, %TCIF_PARAM, %TCIF_STATE, %TCIS_BUTTONPRESSED,
%TCIS_HIGHLIGHTED, %TCN_KEYDOWN, %TCN_SELCHANGE, %TCN_SELCHANGING, %TCN_GETOBJECT,
%TCN_FOCUSCHANGE, %TCS_SCROLLLOPPPOSITE, %TCS_FLATBUTTONS, %TCS_FORCEICONLEFT,
%TCS_FORCELABELLEFT, %TCS_HOTTRACK, %TCS_TABS, %TCS_BUTTONS, %TCS_FIXEDWIDTH,
%TCS_RAGGEDRIGHT, %TCS_FOCUSONBUTTONDOWN, %TCS_OWNERDRAWFIXED, %TCS_TOOLTIPS,
%TCS_FOCUSNEVER, %TCS_EX_FLATSEPARATORS, %TCS_EX_REGISTERDROP

For use with [TEXTBOXES](#):

%EN_CHANGE, %EN_ERRSPACE, %EN_HSCROLL, %EN_KILLFOCUS, %EN_MAXTEXT, %EN_SETFOCUS,
%EN_UPDATE, %EN_VSCROLL, %ES_LEFT, %ES_CENTER, %ES_RIGHT, %ES_MULTILINE,
%ES_UPPERCASE, %ES_LOWERCASE, %ES_PASSWORD, %ES_AUTOVSCROLL, %ES_AUTOHSCROLL,
%ES_NOHIDESEL, %ES_OEMCONVERT, %ES_READONLY, %ES_WANTRETURN, %ES_NUMBER

For use with [THREAD GET PRIORITY](#) and [THREAD SET PRIORITY](#):

```
%THREAD_PRIORITY_ABOVE_NORMAL, %THREAD_PRIORITY_BELOW_NORMAL,  
%THREAD_PRIORITY_HIGHEST, %THREAD_PRIORITY_IDLE, %THREAD_PRIORITY_LOWEST,  
%THREAD_PRIORITY_NORMAL, %THREAD_PRIORITY_TIME_CRITICAL
```

For use with [TOOLBARS](#):

```
%CCS_ADJUSTABLE, %CCS_BOTTOM, %CCS_LEFT, %CCS_NODIVIDER, %CCS_NOMOVEX, %CCS_NOMOVEY,  
%CCS_NOPARENTALIGN, %CCS_NORESIZE, %CCS_RIGHT, %CCS_TOP, %CCS_VERT,  
%TBSTYLE_AUTOSIZE, %TBSTYLE_BUTTON, %TBSTYLE_CHECK, %TBSTYLE_GROUP,  
%TBSTYLE_CHECKGROUP, %TBSTYLE_DROPDOWN, %TBSTYLE_SEP, %TBSTYLE_TOOLTIPS,  
%TBSTYLE_FLAT, %TBSTYLE_LIST, %TBSTYLE_TRANSPARENT, %TBSTYLE_WRAPABLE,  
%TBSTATE_CHECKED, %TBSTATE_DISABLED, %TBSTATE_ELLIPSES, %TBSTATE_ENABLED,  
%TBSTATE_HIDDEN, %TBSTATE_INDETERMINATE, %TBSTATE_MARKED, %TBSTATE_PRESSED,  
%TBSTATE_WRAP, %TBN_BEGINADJUST, %TBN_BEGINDRAG, %TBN_CUSTHELP, %TBN_ENDADJUST,  
%TBN_ENDDRAG, %TBN_GETBUTTONINFO, %TBN_QUERYDELETE, %TBN_QUERYINSERT, %TBN_RESET,  
%TBN_TOOLBARCHANGE
```

For use with [TREEVIEWS](#):

```
%TVS_HASBUTTONS, %TVS_HASLINES, %TVS_LINESATROOT, %TVS_EDITLABELS,  
%TVS_DISABLEDRAHDROP, %TVS_SHOWSELALWAYS, %TVS_RTLDREADING, %TVS_NOTOOLTIPS,  
%TVS_CHECKBOXES, %TVS_TRACKSELECT, %TVS_SINGLEEXPAND, %TVS_INFOTIP,  
%TVS_FULLROWSELECT, %TVS_NOSCROLL, %TVS_NONEVENHEIGHT, %TVS_NOHSCROLL, %TVI_ROOT,  
%TVI_FIRST, %TVI_LAST, %TVI_SORT, %TVE_COLLAPSE, %TVE_EXPAND, %TVE_TOGGLE,  
%TVE_EXPANDPARTIAL, %TVE_COLLAPSERESET, %TVN_BEGINDRAG, %TVN_BEGINLABELEDIT,  
%TVN_BEGINRDRAG, %TVN_DELETEITEM, %TVN_ENDLABELEDIT, %TVN_GETDISPINFO,  
%TVN_ITEMEXPANDED, %TVN_ITEMEXPANDING, %TVN_KEYDOWN, %TVN_SELCHANGED,  
%TVN_SELCHANGING, %TVN_SETDISPINFO
```

For use with [VARIANTVT](#):

```
%VT_EMPTY, %VT_NULL, %VT_I2, %VT_I4, %VT_R4, %VT_R8, %VT_CY, %VT_DATE, %VT_BSTR,  
%VT_DISPATCH, %VT_ERROR, %VT_BOOL, %VT_VARIANT, %VT_DECIMAL, %VT_UNKNOWN, %VT_I1,  
%VT_UI1, %VT_UI2, %VT_UI4, %VT_I8, %VT_UI8, %VT_INT, %VT_UINT, %VT_VOID, %VT_HRESULT,  
%VT_PTR, %VT_SAFEARRAY, %VT_CARRAY, %VT_USERDEFINED, %VT_LPSTR, %VT_LPWSTR,  
%VT_RECORD, %VT_FILETIME, %VT_BLOB, %VT_STREAM, %VT_STORAGE, %VT_STREAMED_OBJECT,  
%VT_STORED_OBJECT, %VT_BLOB_OBJECT, %VT_CF, %VT_CLSID, %VT_VECTOR, %VT_ARRAY,  
%VT_BYREF
```

For use with the [XPRINT GET COLLATE](#) and [XPRINT SET COLLATE](#) statements:

```
%DMCOLLATE_FALSE, %DMCOLLATE_TRUE
```

For use with the [XPRINT GET COLORMODE](#) and [XPRINT SET COLORMODE](#) statements:

```
%DMCOLOR_MONOCHROME, %DMCOLOR_COLOR
```

For use with the [XPRINT GET DUPLEX](#) and [XPRINT SET DUPLEX](#) statements:

```
%DMDUP_SIMPLEX, %DMDUP_VERTICAL, %DMDUP_HORIZONTAL
```

For use with the [XPRINT GET PAPER](#), [XPRINT GET PAPERS](#), and [XPRINT SET PAPER](#) statements:

```
%DMPAPER_LETTER, %DMPAPER_TABLOID, %DMPAPER_LEDGER, %DMPAPER_LEGAL,  
%DMPAPER_STATEMENT, %DMPAPER_EXECUTIVE, %DMPAPER_A3, %DMPAPER_A4, %DMPAPER_A5,  
%DMPAPER_B4, %DMPAPER_B5, %DMPAPER_FOLIO, %DMPAPER_QUARTO, %DMPAPER_10X14,  
%DMPAPER_11X17, %DMPAPER_NOTE, %DMPAPER_ENV_9, %DMPAPER_ENV_10
```

For use with the [XPRINT GET TRAY](#), [XPRINT GET TRAYS](#), and [XPRINT SET TRAY](#) statements:

```
%DMBIN_UPPER, %DMBIN_LOWER, %DMBIN_MIDDLE, %DMBIN_MANUAL, %DMBIN_ENVELOPE,  
%DMBIN_ENVMANUAL, %DMBIN_AUTO, %DMBIN_TRACTOR, %DMBIN_SMALLFMT, %DMBIN_LARGE FMT,  
%DMBIN_LARGE CAPACITY, %DMBIN_CASSETTE, %DMBIN_FORMSOURCE
```

See Also

[Built-in RGB Color Equates](#)

[Constants and Literals](#)

[Numeric Equates](#)

[String Equates](#)










[Built-in string equates](#)

[Built-in Interfaces](#)







[Built-in User Defined Types](#)

The following is a list of [RGB](#) color equates built into the compiler, which can be used with routines that accept RGB color values.







Red Colors

%RGB_INDIANRED	=	&H5C5CCD	
%RGB_LIGHTCORAL	=	&H8080F0	
%RGB_SALMON	=	&H7280FA	
%RGB_DARKSALMON	=	&H7A96E9	
%RGB_LIGHTSALMON	=	&H7AA0FF	
%RGB_CRIMSON	=	&H3C14DC	
%RGB_RED	=	&H0000FF	
%RGB_FIREBRICK	=	&H2222B2	
%RGB_DARKRED	=	&H00008B	







Pink Colors

%RGB_PINK	=	&HCBC0FF	
%RGB_LIGHTPINK	=	&HC1B6FF	
%RGB_HOTPINK	=	&HB469FF	
%RGB_DEEPPINK	=	&H9314FF	
%RGB_MEDIUMVIOLETRED	=	&H8515C7	
%RGB_PALEVIOLETRED	=	&H9370DB	

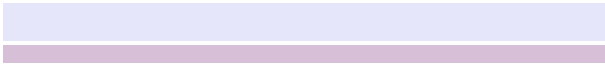
Orange Colors

%RGB_LIGHTSALMON	=	&H7AA0FF	
%RGB_CORAL	=	&H507FFF	
%RGB_TOMATO	=	&H4763FF	
%RGB_ORANGERED	=	&H0045FF	
%RGB_DARKORANGE	=	&H008CFF	
%RGB_ORANGE	=	&H00A5FF	

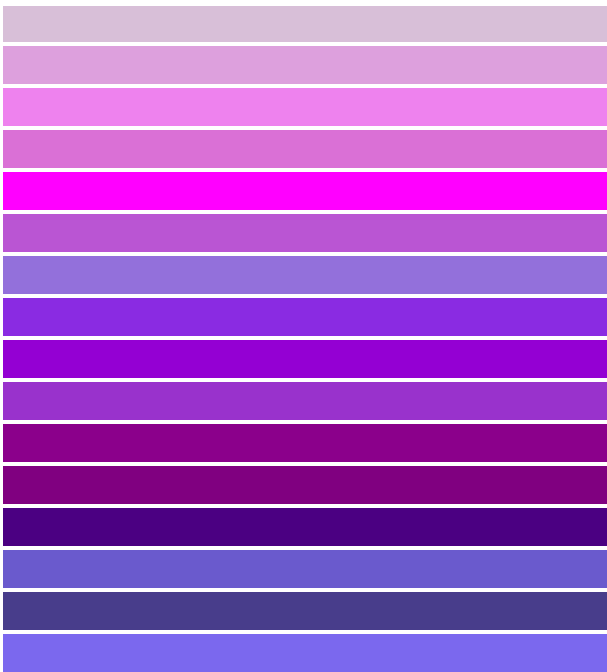
Yellow Colors

%RGB_GOLD	=	&H00D7FF	
%RGB_YELLOW	=	&H00FFFF	
%RGB_LIGHTYELLOW	=	&HE0FFFF	
%RGB_LEMONCHIFFON	=	&HCDFAFF	
%RGB_LIGHTGOLDENRODYELLOW	=	&HD2FAFA	
%RGB_PAPAYAWHIP	=	&HD5EFFF	
%RGB_MOCCASIN	=	&HB5E4FF	
%RGB_PEAHPUFF	=	&HB9DAFF	
%RGB_PALEGOLDENROD	=	&HAAE8EE	
%RGB_KHAKI	=	&H8CE6F0	
%RGB_DARKKHAKI	=	&H6BB7BD	

Purple Colors

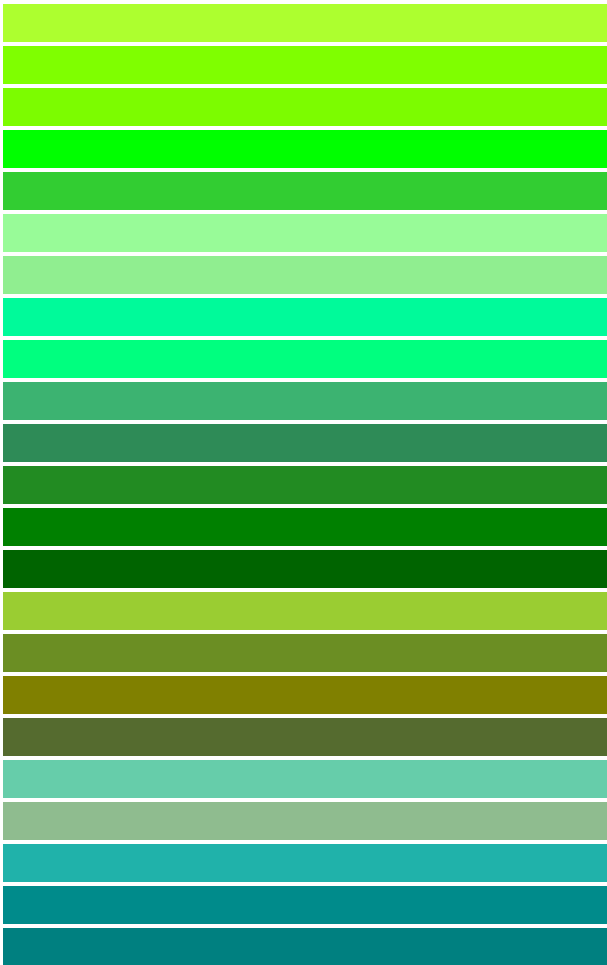
%RGB_LAVENDER	=	&HFAE6E6	
---------------	---	----------	--

<code>%RGB_THISTLE</code>	<code>= &HD8BFD8</code>
<code>%RGB_PLUM</code>	<code>= &HDDA0DD</code>
<code>%RGB_VIOLET</code>	<code>= &HEE82EE</code>
<code>%RGB_ORCHID</code>	<code>= &HD670DA</code>
<code>%RGB_MAGENTA</code>	<code>= &HFF00FF</code>
<code>%RGB_MEDIUMORCHID</code>	<code>= &HD355BA</code>
<code>%RGB_MEDIUMPURPLE</code>	<code>= &HDB7093</code>
<code>%RGB_BLUEVIOLET</code>	<code>= &HE22B8A</code>
<code>%RGB_DARKVIOLET</code>	<code>= &HD30094</code>
<code>%RGB_DARKORCHID</code>	<code>= &HCC3299</code>
<code>%RGB_DARKMAGENTA</code>	<code>= &H8B008B</code>
<code>%RGB_PURPLE</code>	<code>= &H800080</code>
<code>%RGB_INDIGO</code>	<code>= &H82004B</code>
<code>%RGB_SLATEBLUE</code>	<code>= &HCD5A6A</code>
<code>%RGB_DARKSLATEBLUE</code>	<code>= &H8B3D48</code>
<code>%RGB_MEDIUMSLATEBLUE</code>	<code>= &HEE687B</code>



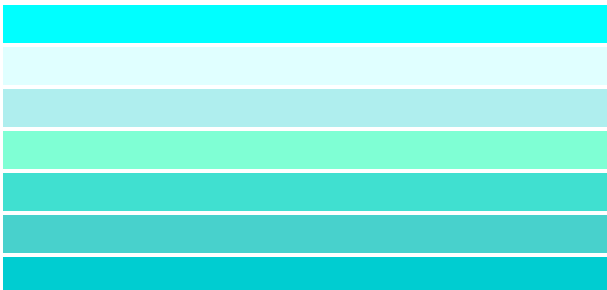
Green Colors

<code>%RGB_GREENYELLOW</code>	<code>= &H2FFFAD</code>
<code>%RGB_CHARTREUSE</code>	<code>= &H00FF7F</code>
<code>%RGB_LAWNGREEN</code>	<code>= &H00FC7C</code>
<code>%RGB_LIME</code>	<code>= &H00FF00</code>
<code>%RGB_LIMEGREEN</code>	<code>= &H32CD32</code>
<code>%RGB_PALEGREEN</code>	<code>= &H98FB98</code>
<code>%RGB_LIGHTGREEN</code>	<code>= &H90EE90</code>
<code>%RGB_MEDIUMSPRINGGREEN</code>	<code>= &H9AFA00</code>
<code>%RGB_SPRINGGREEN</code>	<code>= &H7FFF00</code>
<code>%RGB_MEDIUMSEAGREEN</code>	<code>= &H71B33C</code>
<code>%RGB_SEAGREEN</code>	<code>= &H578B2E</code>
<code>%RGB_FORESTGREEN</code>	<code>= &H228B22</code>
<code>%RGB_GREEN</code>	<code>= &H008000</code>
<code>%RGB_DARKGREEN</code>	<code>= &H006400</code>
<code>%RGB_YELLOWGREEN</code>	<code>= &H32CD9A</code>
<code>%RGB_OLIVEDRAB</code>	<code>= &H238E6B</code>
<code>%RGB_OLIVE</code>	<code>= &H008080</code>
<code>%RGB_DARKOLIVEGREEN</code>	<code>= &H2F6B55</code>
<code>%RGB_MEDIUMAQUAMARINE</code>	<code>= &HAACD66</code>
<code>%RGB_DARKSEAGREEN</code>	<code>= &H8FBC8F</code>
<code>%RGB_LIGHTSEAGREEN</code>	<code>= &HAAB220</code>
<code>%RGB_DARKCYAN</code>	<code>= &H8B8B00</code>
<code>%RGB_TEAL</code>	<code>= &H808000</code>

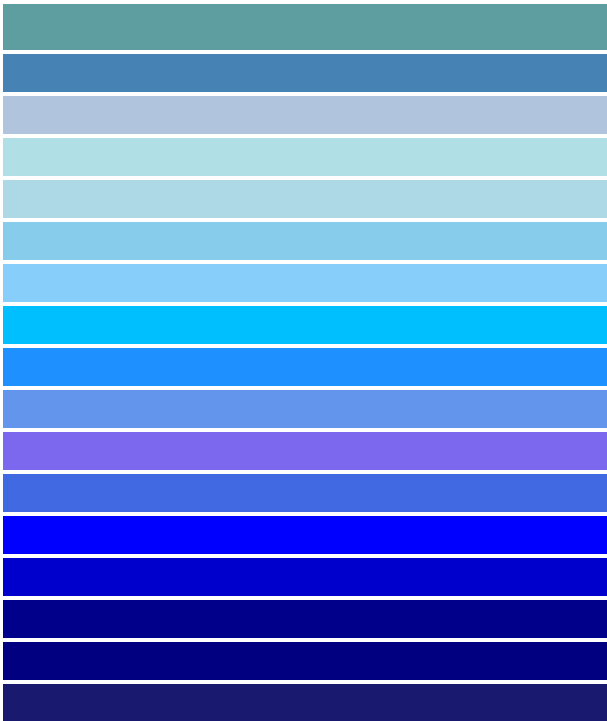


Blue Colors

<code>%RGB_CYAN</code>	<code>= &HFFFF00</code>
<code>%RGB_LIGHTCYAN</code>	<code>= &HFFFFE0</code>
<code>%RGB_PALETURQUOISE</code>	<code>= &HEEEEEAF</code>
<code>%RGB_AQUAMARINE</code>	<code>= &HD4FF7F</code>
<code>%RGB_TURQUOISE</code>	<code>= &HD0E040</code>
<code>%RGB_MEDIUMTURQUOISE</code>	<code>= &HCCD148</code>
<code>%RGB_DARKTURQUOISE</code>	<code>= &HD1CE00</code>

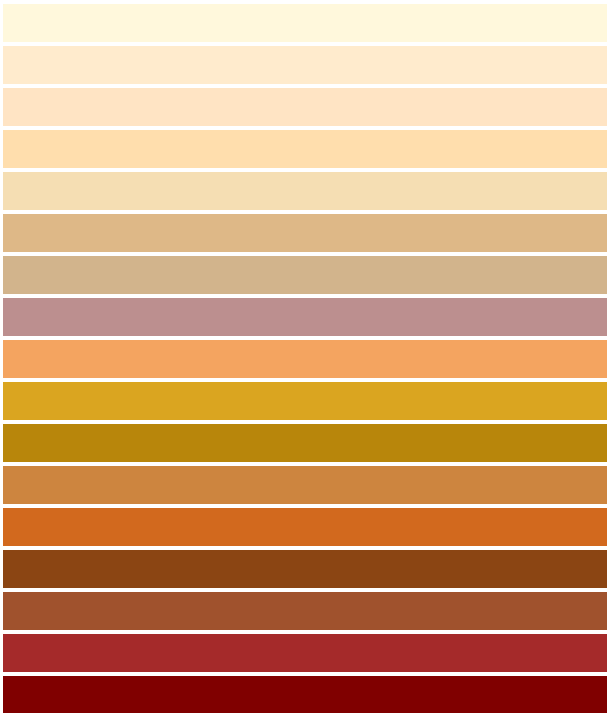


<code>%RGB_CADETBLUE</code>	<code>= &HA09E5F</code>
<code>%RGB_STEELBLUE</code>	<code>= &HB48246</code>
<code>%RGB_LIGHTSTEELBLUE</code>	<code>= &HDEC4B0</code>
<code>%RGB_POWDERBLUE</code>	<code>= &HE6E0B0</code>
<code>%RGB_LIGHTBLUE</code>	<code>= &HE6D8AD</code>
<code>%RGB_SKYBLUE</code>	<code>= &HEBCE87</code>
<code>%RGB_LIGHTSKYBLUE</code>	<code>= &HFACE87</code>
<code>%RGB_DEEPSKYBLUE</code>	<code>= &HFFBF00</code>
<code>%RGB_DODGERBLUE</code>	<code>= &HFF901E</code>
<code>%RGB_CORNFLOWERBLUE</code>	<code>= &HED9564</code>
<code>%RGB_MEDIUMSLATEBLUE</code>	<code>= &HEE687B</code>
<code>%RGB_ROYALBLUE</code>	<code>= &HE16941</code>
<code>%RGB_BLUE</code>	<code>= &HFF0000</code>
<code>%RGB_MEDIUMBLUE</code>	<code>= &HCD0000</code>
<code>%RGB_DARKBLUE</code>	<code>= &H8B0000</code>
<code>%RGB_NAVY</code>	<code>= &H800000</code>
<code>%RGB_MIDNIGHTBLUE</code>	<code>= &H701919</code>



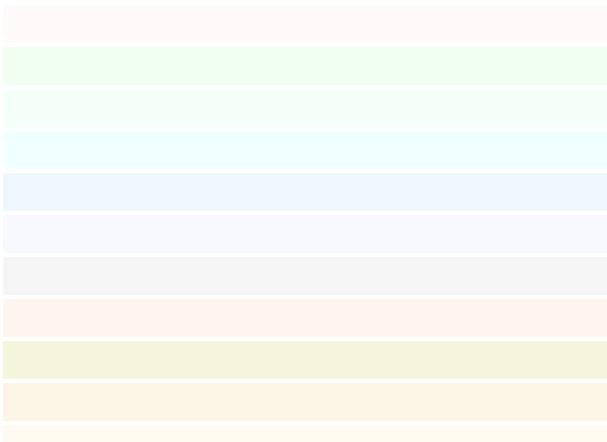
Brown Colors

<code>%RGB_CORNSILK</code>	<code>= &HDCF8FF</code>
<code>%RGB_BLANCHEDALMOND</code>	<code>= &HCDEBFF</code>
<code>%RGB_BISQUE</code>	<code>= &HC4E4FF</code>
<code>%RGB_NAVAJOWHITE</code>	<code>= &HADDEFF</code>
<code>%RGB_WHEAT</code>	<code>= &HB3DEF5</code>
<code>%RGB_BURLYWOOD</code>	<code>= &H87B8DE</code>
<code>%RGB_TAN</code>	<code>= &H8CB4D2</code>
<code>%RGB_ROSYBROWN</code>	<code>= &H8F8FBC</code>
<code>%RGB_SANDYBROWN</code>	<code>= &H60A4F4</code>
<code>%RGB_GOLDENROD</code>	<code>= &H20A5DA</code>
<code>%RGB_DARKGOLDENROD</code>	<code>= &H0B86B8</code>
<code>%RGB_PERU</code>	<code>= &H3F85CD</code>
<code>%RGB_CHOCOLATE</code>	<code>= &H1E69D2</code>
<code>%RGB_SADDLEBROWN</code>	<code>= &H13458B</code>
<code>%RGB_SIENNA</code>	<code>= &H2D52A0</code>
<code>%RGB_BROWN</code>	<code>= &H2A2AA5</code>
<code>%RGB_MAROON</code>	<code>= &H000080</code>

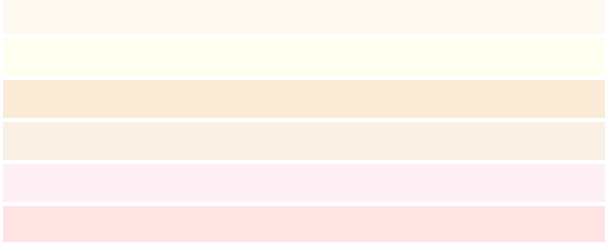


White Colors

<code>%RGB_WHITE</code>	<code>= &HFFFFFF</code>
<code>%RGB_SNOW</code>	<code>= &HFAFAFF</code>
<code>%RGB_HONEYDEW</code>	<code>= &HF0FFF0</code>
<code>%RGB_MINTCREAM</code>	<code>= &HFAFFF5</code>
<code>%RGB_AZURE</code>	<code>= &HFFFFFF0</code>
<code>%RGB_ALICEBLUE</code>	<code>= &HFFF8F0</code>
<code>%RGB_GHOSTWHITE</code>	<code>= &HFFF8F8</code>
<code>%RGB_WHITESMOKE</code>	<code>= &HF5F5F5</code>
<code>%RGB_SEASHELL</code>	<code>= &HEEF5FF</code>
<code>%RGB_BEIGE</code>	<code>= &HDCF5F5</code>
<code>%RGB_OLDLACE</code>	<code>= &HE6F5FD</code>



%RGB_FLORALWHITE	=	&HF0FAFF
%RGB_IVORY	=	&HF0FFFF
%RGB_ANTIQUWHITE	=	&HD7EBFA
%RGB_LINEN	=	&HE6F0FA
%RGB_LAVENDERBLUSH	=	&HF5F0FF
%RGB_MISTYROSE	=	&HE1E4FF



Gray Colors

%RGB_GAINSBORO	=	&HDCDCDC
%RGB_LIGHTGRAY	=	&HD3D3D3
%RGB_SILVER	=	&HC0C0C0
%RGB_DARKGRAY	=	&HA9A9A9
%RGB_GRAY	=	&H808080
%RGB_DIMGRAY	=	&H696969
%RGB_LIGHTSLATEGRAY	=	&H998877
%RGB_SLATEGRAY	=	&H908070
%RGB_DARKSLATEGRAY	=	&H4F4F2F
%RGB_BLACK	=	&H000000



See Also

- [Built-in numeric equates](#)
- [Built-in string equates](#)
- [Built-in Interfaces](#)
- [Built-in User Defined Types](#)
- [Constants and Literals](#)
- [Numeric Equates](#)
- [String Equates](#)

You can define string equates by prefixing a dollar-sign (\$) to the equate data name. The value on the right side of the string equate must be a [string literal](#), or an expression made of string literals. A string literal can also be constructed with combinations of the expanded [CHR\\$](#) function, the [STRING\\$](#) function, the [SPACE\\$](#) function, the [GUID\\$](#) function, and string constants. For example:

```
$Name      = "John Smith"
$Fullname  = "John" & " Smith"
$UserName  = $First & $Last
$PrintCode = CHR$(27, 34, "E") + SPACE$(10) + CHR$(65 TO 90)
$AppGuid   = GUID$("{01234567-89AB-CDEF-FEDC-BA9876543210}")
```

A string equate can include the double-quote character, simply by doubling the character within the string. For example:

```
$ABC = "This is a ""string"""
```

String equates are individually limited to 255 characters. Attempting to create a longer string equate will trigger a compile-time [Error 489](#) ("Invalid string length").

As with [numeric equates](#), Classic PowerBASIC pre-calculates the string equate content during compilation to avoid unnecessary concatenation operations at run-time. Duplicate definitions of both numeric and string equates are permitted by Classic PowerBASIC, provided the actual content is identical. If the content is not identical, a compile-time [Error 468](#) ("Duplicate Equate") will occur.

String equates must be created outside of any [SUB](#), [FUNCTION](#), [METHOD](#), or [PROPERTY](#). String equates are global, and may be referenced anywhere in the module. For readability, we suggest placing equates at the top of your code.

See Also

[Constants and Literals](#)

[Defining Constants](#)

[Numeric Equates](#)

[Built-in numeric equates](#)

[Built In RGB Color Equates](#)

[Built-in string equates](#)

Built-in string equates

[Top](#) [Previous](#) [Next](#)

The compiler also provides a set of built-in [string equates](#), including:

Equate	Character(s)	Definition
\$NUL	CHR\$(0)	Null
\$BEL	CHR\$(7)	Bell
\$BS	CHR\$(8)	Back Space
\$TAB	CHR\$(9)	Horizontal Tab
\$LF	CHR\$(10)	Line Feed
\$VT	CHR\$(11)	Vertical Tab
\$FF	CHR\$(12)	Form Feed
\$CR	CHR\$(13)	Carriage Return
\$CRLF	CHR\$(13,10)	CR and LF
\$EOF	CHR\$(26)	End-of-File
\$ESC	CHR\$(27)	Escape
\$SPC	CHR\$(32)	Space
\$DQ	CHR\$(34)	Double-Quote
\$SQ	CHR\$(39)	Single-Quote
\$QCQ	CHR\$(34, 44, 34)	Double-Quote, Comma, Double-Quote

See Also

[Constants and Literals](#)

[Numeric Equates](#)

[Built-in numeric equates](#)

[Built-in string equates](#)

[String Equates](#)

[Built-in Interfaces](#)

[Built-in User Defined Types](#)

[Built-in RGB Color Equates](#)

[TYPE](#) and [UNION](#) structures may contain bit variables, which are named [BIT](#) (unsigned values) or [SBIT](#) (signed values). Each bit variable may occupy from 1 to 31 bits. When used in a TYPE, bit variables are packed one after another up to a total of 32 bits per bit field. When used in a UNION, all bit variables overlay each other, starting at the first bit position.

Bit variables may only be used as TYPE or UNION members, not as scalar, array, or pointer variables. The size of a bit variable is defined as:

```
var AS BIT * nlit [IN BYTE|WORD|DWORD]
```

where the term "`* nlit`" defines the number of bits (1 to 31), and the optional term "`IN BYTE|WORD|DWORD`", if present, defines the start of a new bit field of 1, 2, or 4 bytes.

```
TYPE abcd
  valu AS BIT * 31 IN DWORD
  sign AS SBIT * 1
  nybl2 AS BIT * 4 IN BYTE
  nybl1 AS BIT * 4
END TYPE
```

The example type above is 5 bytes in size, containing a 4-byte bit field and a 1-byte bit field. In this case, each contain 2 bit variables of varying size. The range of values which may be stored depends upon the number of bits available. For example, "`BIT * 4`" has a range of 0 to 15, "`SBIT * 1`" has a range of -1 to 0, and "`SBIT * 5`" has a range of -16 to +15. BIT and SBIT variables may not be used with SHIFT or ROTATE statements.

```
UNION abcde
  Part1 AS BIT * 8 IN DWORD
  Part2 AS BIT * 16
END UNION
```

The example union above is 2 bytes in size, containing an 8-bit field and an overlapping 16-bit field.

See Also

[TYPE/END TYPE block](#)

[UNION/END UNION statements](#)

Classic PowerBASIC introduces another new [variable](#) class: GUID variables. These are a special form of 16-[byte](#) string that are used to contain a 128-bit Globally Unique Identifier (GUID), primarily for use with [COM Objects](#).

Generally speaking, a GUID variable is assigned a value with the [GUID\\$](#) function, or with a [string equate](#), and that value usually remains constant throughout the program. The GUID variable is typically used only as a parameter, rather than as a term in an expression.

GUID variables must be explicitly declared with [DIM](#), [LOCAL](#), etc, and are used in much the same way as a 16-byte [fixed-length string](#) or a [user-defined type](#) of that size. A GUID variable is only valid as a parameter when its 16 bytes of data are in an appropriate format. For example:

```
$idNull = STRING$(16,0)
' code here
DIM abc AS LOCAL GUID
DIM def AS LOCAL STRING
DIM xyz AS GLOBAL GUID
abc = $idNull
abc = GUID$("{00000000-0000-0000-C000-000000000046}")
xyz = abc
def = GUIDTXT$(xyz)
' def contains "{00000000-0000-0000-C000-000000000046}"
```

See Also

[GUID\\$ function](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[How are GUID's used with objects?](#)

[Object](#) variables are used to access an object. Object variables contain a pointer to the desired object, so they are considered to contain an "[Object Reference](#)". They contain no other value of any kind. Object variables are declared by the name of the [interface](#) they represent. This could be the generic [IDISPATCH](#), [IUNKNOWN](#), and [IAUTOMATION](#) interfaces, or one that is explicitly defined with an INTERFACE structure. For example:

```
' Generic IDispatch Object variable
DIM oWord AS IDISPATCH
LET oWord = NEWCOM "Word.Application"
```

```
' Generic IUnknown Object variable
DIM MyObj as IUNKNOWN
DIM oWord as Int_Application
LET MyObj = NEWCOM "Word.Application"
LET oWord = MyObj
```

```
' Interface-specific Object variable
DIM oWord AS Int_Application
LET oWord = NEWCOM "Word.Application"
```

An object variable may only be used in specific situations, such as execution of an Object Method. It is never legal to reference Object variables in normal numeric or [string expressions](#), nor is it possible to even output their value without the use of the special new functions like [OBJPTR](#). [Methods](#) are executed by using an object variable with a Method name. For example, to call the Method ABC in an interface represented by the object variable MyObject, you would write:

```
CALL MyObject.abc()
```

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you create an object?](#)

[How do you call a Direct Method?](#)

[How do you call a DISPATCH METHOD?](#)

[Late Binding](#)

[ID Binding](#)

Variant [variables](#) are now supported by Classic PowerBASIC, but their use is limited to that of a parameter assignment for conversion of data, for compatibility with other languages and applications, especially [COM Objects](#).

Although notoriously lacking in efficiency, Variants are commonly used as [COM](#) Object parameters due to their flexibility. You can think of a Variant as a kind of container, which can hold a variable of most any data type, numeric, string, or even an entire [array](#). This simplifies the process of calling procedures in a COM [Object](#) Server, as there is little need to worry about the myriad of possible data types for each parameter.

This flexibility comes at a great price in performance, so Classic PowerBASIC limits their use to data storage and parameters only. You may assign a numeric value, a string value, or even an entire array to a Variant with the [LET](#) statement, or its implied equivalent. In the same way, you may assign one Variant value to another Variant variable, or even assign an array contained in a Variant to a compatible Classic PowerBASIC array, or the reverse.

You may extract a simple scalar value from a Variant with the [VARIANT#](#) function for numeric values (regardless of the internal numeric data type), or with the [VARIANT\\$](#) function for string values. You may determine the type of data a Variant variable contains with the [VARIANTVT](#) function. The following table summarizes the predefined (built-in) equates that can be used to examine a Variant:

Result	Equate	Content Type
0	%VT_EMPTY	An Empty Variant
1	%VT_NULL	Null value
2	%VT_I2	Integer
3	%VT_I4	Long-Integer
4	%VT_R4	Single
5	%VT_R8	Double
6	%VT_CY	Currency
7	%VT_DATE	Date
8	%VT_BSTR	Dynamic String
9	%VT_DISPATCH	IDispatch
10	%VT_ERROR	Error Code
11	%VT_BOOL	Boolean

12	%VT_VARIANT	Variant
13	%VT_UNKNOWN	IUnknown
14	%VT_DECIMAL	Decimal
16	%VT_I1	Byte (signed)
17	%VT_UI1	Byte (unsigned)
18	%VT_UI2	Word
19	%VT_UI4	DWORD
20	%VT_I8	Quad (signed)
21	%VT_UI8	Quad (unsigned)
22	%VT_INT	Long-Integer
23	%VT_UINT	DWord
24	%VT_VOID	A C-style void type
25	%VT_HRESULT	COM result code
26	%VT_PTR	Pointer
27	%VT_SAFEARRAY	VB Array
28	%VT_CARRAY	A C-style array
29	%VT_USERDEFINED	User Defined Type
30	%VT_LPSTR	ANSI string
31	%VT_LPWSTR	Unicode string
64	%VT_FILETIME	A FILETIME value
65	%VT_BLOB	An arbitrary block of memory
66	%VT_STREAM	A stream of bytes
67	%VT_STORAGE	Name of the storage
68	%VT_STREAMED_OBJECT	A stream that contains an object
69	%VT_STORED_OBJECT	A storage object
70	%VT_BLOB_OBJECT	A block of memory that represents an object
71	%VT_CF	Clipboard format
72	%VT_CLSID	Class ID
&H1000	%VT_VECTOR	An array with a leading count
&H2000	%VT_ARRAY	Array
&H4000	%VT_BYREF	A reference value

--	--	--

Variants may not be used in an expression, be directly output ([PRINT#](#), etc), or used as a member of a structure such as a [User-Defined Type](#) (UDT) or [UNION](#), etc. Instead, you must first extract the value with one of the above conversion functions, and use that acquired value for calculations.

Internally, a Variant is always 16-bytes in size, and may be passed as either a [BYVAL](#) or a [BYREF](#) parameter, at the programmer's discretion. However, when a BYREF Variant is required as a parameter, only an explicit Variant variable may be passed by the calling code - a [BYCOPY](#) expression is not allowed.

All dynamic strings contained in a Variant must be Wide/Unicode, and Classic PowerBASIC handles these conversions automatically through the [LET](#) statement and its implied equivalent.

There may be some cases where you wish to manipulate the internal structure of a Variant directly. Though possible, you must exercise caution or a serious memory leak could occur. Since a Variant could be the owner of a string, array, etc., you must always reset a Variant ([[LET](#)] *VrntName* = EMPTY) prior to manipulation with [POKE](#), or pointers, etc.

When you use the standard Classic PowerBASIC assignment syntax, for example: [[LET](#)] *VrntName* = 21, all this "housekeeping" is completely automatic and handled by Classic PowerBASIC for you.

Every Variant variable must be explicitly declared with an appropriate statement such as:

```
DIM xyz AS VARIANT or LOCAL xyz AS VARIANT
```

See Also

- [What is an object, anyway?](#)
- [Just what is COM?](#)
- [What is a COM component?](#)

Comparative Data Types C/C++

[Top](#) [Previous](#) [Next](#)

When dealing with C, intrinsic types are in lowercase. Defined types are in all caps by convention. C data types are case-sensitive. Integer-class types can take a modifier of "signed" or "unsigned", and are signed by default.

C arrays are defined by the number of elements and are indexed from zero:

"char foo[32]" translates to [DIM](#) foo(0 TO 31) AS [BYTE](#), or DIM foo AS [STRING * 32](#), depending on the context of the code.

C [arrays](#) are stored in row-major order whereas Classic PowerBASIC arrays are stored in column-major order. Bear in mind that when accessing C arrays the following C code:

```
k = arr[i,j]
```

would translate to Classic PowerBASIC as:

```
k = arr(j,i)
```

C arrays are accessed as follows:

```
(0,0), (0,1), (0,2), ...  
(1,0), (1,1), (1,2), ...
```

whereas Classic PowerBASIC arrays are accessed:

```
(0,0), (1,0), (2,0), ...  
(0,1), (1,1), (2,1), ...
```

Commonly, C/C++ code prefixes data types with "LP" which indicates a [pointer](#). Therefore, items with the LP prefix usually correspond to a pointer in Classic PowerBASIC; however, the size of the pointer's target will depend on the data type.

More information on C/C++ syntax can be found on the Internet, such as at <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf> and <http://www.open-std.org/JTC1/SC22/WG21/>

C/C++ Data Types

Type	Language	Format	Classic PowerBASIC
bool	C++	unsigned 8-bit	BYTE (2)
char	C/C++	signed 8-bit	BYTE (2)
char*	C/C++	char pointer	ASCIIZ (1)
double	C/C++	8-byte float	DOUBLE
float	C/C++	4-byte float	SINGLE
int	C/C++	signed 32-bit	LONG (3)
long	C/C++	signed 32-bit	LONG
short	C/C++	signed 16-bit	INTEGER
void	C/C++	(no return value)	SUB
void *	C/C++	pointer	(ANY) [PTR] (1)

Defined types (SDK types)

Type	Format	Classic PowerBASIC
ATOM	unsigned 16-bit	WORD
BOOL	signed 32-bit	LONG
boolean	8-bit integer	BYTE
Boolean	signed 16-bit	INTEGER
BOOLEAN	8-bit integer	BYTE
BSTR	dynamic string	STRING {unicode}
BYTE	unsigned 8-bit	BYTE
COLORREF	unsigned 32-bit	DWORD
DWORD	unsigned 32-bit	DWORD
HANDLE	unsigned 32-bit	DWORD
HWND/HDC/	unsigned 32-bit	DWORD
INT32	signed 32-bit	LONG
INT64	signed 64-bit	QUAD
LARGE_INTEGER	signed 64-bit	QUAD
LPARAM	signed 32-bit	LONG
LP	pointer	(ANY) [PTR] (4)
LPCSTR	ASCIIZ pointer	ASCIIZ [PTR]
LPDWORD	DWORD pointer	DWORD [PTR]
LPINT	LONG pointer	LONG [PTR]
LPSTR	ASCIIZ pointer	ASCIIZ [PTR]
LPUINT	DWORD pointer	DWORD [PTR]
LPVOID	32-bit pointer	(ANY) [PTR]
LRESULT	signed 32-bit	LONG
NULL	32-bit	0 or %NULL
PASCAL	{calling convention}	/STDCALL
QWORD	unsigned 64-bit	QUAD (2)
STDCALL	{calling convention}	SDECL/STDCALL
UCHAR	unsigned 8-bit	BYTE
UINT	unsigned 32-bit	DWORD (3)
UINT16	unsigned 16-bit	WORD
UINT32	unsigned 32-bit	DWORD

UINT64	unsigned 64-bit	QUAD (<u>2</u>)
VOID	SUB	{no return value}
VOID *	pointer	(<u>ANY</u>) [<u>PTR</u>] (<u>1</u>)
WINAPI	{calling convention}	SDECL/STDCALL
WORD	unsigned 16-bit	WORD
WPARAM	signed 32-bit	LONG

Comparative Data Types Delphi

[Top](#) [Previous](#) [Next](#)

Delphi uses integer conventions similar to C, although the names are case-insensitive, as with BASIC. That is, a Delphi INTEGER value may be either a Classic PowerBASIC [INTEGER](#) or [LONG](#), depending on whether the Delphi code is 16-bit or 32-bit.

The elements of multi-dimensional arrays, in Delphi, are not necessarily stored in a straightforward order in memory. Such arrays are not compatible with other languages.

Delphi Data Types

Type	Format	Classic PowerBASIC
boolean	unsigned 8-bit	BYTE
byte	unsigned 8-bit	BYTE
bytebool	unsigned 8-bit	BYTE
cardinal	unsigned 16/32-bit	WORD/DWORD (5)
comp	signed 64-bit	QUAD
currency	8-byte fixed point	CURRENCY
double	8-byte floating point	DOUBLE
extended	10-byte floating point	EXT
int64	signed 64-bit	QUAD
integer	signed 16/32-bit	INTEGER/LONG (5)
longbool	signed 32-bit	LONG
longint	signed 32-bit	LONG
longword	unsigned 32-bit	DWORD
pchar	ASCIIZ string	ASCIIZ
shortint	signed 8-bit	BYTE (2)
single	4-byte float	SINGLE
smallint	signed 16-bit	INTEGER
variant	data-dependent	VARIANT
word	unsigned 16-bit	WORD
wordbool	unsigned 16-bit	WORD

Visual Basic Data Types

Type	Format	Classic PowerBASIC
Boolean	signed 16-bit	INTEGER
Byte	unsigned 8-bit	BYTE
Const	numeric constant	{ Equate } (2)
Currency	8-byte fixed point	CURRENCY
Double	8-byte float	DOUBLE
Integer	signed 16-bit	INTEGER
Long	signed 32-bit	LONG
Single	4-byte float	SINGLE
String	dynamic string	STRING
String * <i>n</i>	fixed-length string	STRING * <i>n</i>
Variant	data-dependent	VARIANT

Default Variable Typing

[Top](#) [Previous](#) [Next](#)

In [Classic PowerBASIC/DOS](#) and MSBASIC, all [variables](#) without a [type-specifier](#) (%, !, &, etc.) default to [Single-precision](#). In Visual Basic, all untyped variables default to [Variants](#).

In PB/Win, numeric variables without a type specifier are not allowed unless you specifically tell the compiler the default type using the [DEF](#) statement. For example, to mimic the Single-precision default of PB/DOS, simply add a [DEFSNG](#) statement to the top of your code:

```
DEFSNG A-Z
```

See Also

[Variables](#)

[Variable Scope](#)

[THREADED variables](#)

[LOCAL, GLOBAL and STATIC considerations](#)

Classic PowerBASIC offers a [THREADED](#) variable type for use in multi-threaded programs. While multi-threaded programming is an advanced topic and is beyond the scope of this documentation, we'll briefly describe thread [variables](#) in Classic PowerBASIC.

Thread variables (also known as thread-local storage or TLS) can be declared with either a THREADED statement or a [DIM](#) statement (with the THREADED scope clause). In either case, thread variables are visible to all [Subs](#), [Functions](#), [Methods](#), and [Properties](#) in a program, but each thread will have its own unique copy of each of the variables. This allows data to be shared between procedures on a thread by thread basis. For example:

```
THREADED t1 AS LONG
' more code here
FUNCTION MyThread(BYVAL x AS LONG) AS DWORD
    t1 = RND(1,x)
    ' Here t1 is unique to each thread
    CALL MySub
END FUNCTION

SUB MySub
    ? t1
    ' Here t1 is unique to the thread calling MySub
END SUB
```

Thread variables have a distinct advantage over [global](#) variables as they avoid the need to use synchronization techniques (such as a Critical Section object or Mutex) when referencing (reading or writing) from two or more threads at the same time. In addition, all allocation and deallocation of THREADED memory is handled automatically by Classic PowerBASIC without any need for intervention by the programmer. However, by their nature, thread variables impose a slight, yet measurable, performance penalty as compared to other variable types.

See Also

[Variables](#)

[Default Variable Typing](#)

[Variable scope](#)

[LOCAL, GLOBAL and STATIC considerations](#)

The scope of a [variable](#) is defined as its visibility and its lifetime. Visibility means what parts of your program can access it. Lifetime defines when it is created and when it is destroyed. In Classic PowerBASIC, there are many choices of scope to afford the maximum flexibility. You may choose any scope which best fits the needs of your program. When any variable is created in Classic PowerBASIC, it is automatically initialized. Numeric variables are initialized to zero (0). [Dynamic strings](#), [Field strings](#), and [ASCIIZ strings](#) are initialized to a length of zero (no characters). Fixed-length strings and [UDTs](#) are filled with [CHR\\$\(0\)](#). Classic PowerBASIC automatically destroys every variable when at the appropriate time, so you never need worry about this type of memory leak.

- [LOCAL](#) Local variables are only accessible within a single [SUB](#), [FUNCTION](#), [METHOD](#), or [PROPERTY](#). They are automatically created and initialized each time you enter the procedure. They are automatically destroyed when you exit. This is the default variable scope unless you declare otherwise.
- [STATIC](#) Static variables are only accessible within a single SUB, FUNCTION, METHOD, or PROPERTY. They are initialized when your program starts, but retain their value regardless of how many times the procedure is entered and exited. They are destroyed only when the program ends.
- [GLOBAL](#) Global variables are accessible from anywhere in your program. They are initialized when your program starts and are destroyed when the program ends.
- [THREADED](#) Threaded variables are accessible from anywhere in your program, but each thread within your program will have its own unique copy of them. They are created and initialized when a thread is created. They are destroyed when the thread ends. Threaded variables are commonly called Thread Local Storage (TLS).
- [INSTANCE](#) Instance variables are accessible from any method or property in a class. Each object will have its own unique copy of them. They are created and initialized when an object is created. They are destroyed when an object is destroyed.

See Also

[Variables](#)

[Default Variable Typing](#)

[THREADED variables](#)

[LOCAL, GLOBAL and STATIC considerations](#)

LOCAL, GLOBAL and STATIC considerations

[Top](#) [Previous](#)
[Next](#)

All [LOCAL variables](#) are stored on the [stack](#) frame of the [Sub/Function/Method/Property](#) in which they are defined. Therefore, the address of local variables may vary with each invocation of the procedure in which they are defined (because they are created each time the procedure is executed, and destroyed upon exit). It is unsafe to return a [pointer](#) to a local variable, since the storage for that variable is released when the procedure ends.

Conversely, [GLOBAL](#) and [STATIC](#) variables are stored in the main data memory area, so their address stays constant for the duration of the program module, so returning a pointer to a GLOBAL or STATIC variable is quite safe.

Typically, the stack frame of a Classic PowerBASIC application can hold close to 1 MB of LOCAL data (also see [#STACK](#)). Therefore, care must be exercised when large local [ASCIIZ](#) and/or [fixed-length](#) strings are used, to ensure that these strings do not exceed the available stack space when the procedure executes. This means that very large ASCIIZ and fixed length strings may need to be changed to a STATIC, GLOBAL, or [INSTANCE](#) (for an object) type instead, since those types are not created within the stack frame and will not overflow the stack.

Local [dynamic](#) (variable length) strings are not stored in the same manner as ASCIIZ and fixed-length strings. Every dynamic string has an associated 4 byte "string handle", and this handle is stored in the stack frame, but the actual string data is stored in the main data memory area. Therefore, local dynamic strings occupy just 4 bytes of the stack frame regardless of whether the string data itself comprises just one byte or 100 Megabytes.

Similarly, local [arrays](#) have an associated "array descriptor" table. This small table is stored in the stack frame, and the actual array data is stored in the main data memory area. Therefore, arrays also have only a very small impact on the stack frame.

It should also be noted that when a [DIM](#) statement is used (*without an explicit scope clause*), to declare a variable (or array) in a procedure, *and* an identical variable (or array) has already been declared as GLOBAL, the variable in the procedure will be given GLOBAL scope. For example:

```
GLOBAL xyz AS LONG
[statements]
SUB MySub
    DIM xyz AS LONG
    ' Here, xyz is a GLOBAL variable
END SUB
```

To ensure that the variable scope is LOCAL to the procedure, use a LOCAL statement rather than a DIM statement. Alternatively, add an explicit scope clause to the DIM statement. For example:

```
GLOBAL xyz AS LONG
[statements]
SUB MySub
    DIM xyz AS LOCAL LONG
```

```
' Here, xyz is a LOCAL variable  
END SUB
```

Finally, it should be noted that the DIM statement (without an explicit scope clause) does not affect identically named THREADED variables as is it does with GLOBAL variables. For example:

```
THREADED abc AS LONG  
[statements]  
SUB MySub  
    DIM abc AS LONG  
    ' Here abc is a LOCAL variable, not a THREADED variable  
END SUB
```

See Also

[Variables](#)

[Default Variable Typing](#)

[Variable Scope](#)

Arithmetic operators perform normal mathematical operations. Several of these operators merit a word of explanation. The backslash (\) represents integer division. Integer division rounds its operands to integers, to produce a truncated quotient with no remainder. For example, 5 \ 2 evaluates to 2, and 9 \ 10 evaluates to 0. Integer division is also faster than floating-point division when using integer-class variables or expressions.

The remainder of an integer division can be determined with the [MOD](#) (modulo) operator (MOD is valid for all numeric types). MOD is similar to integer division except that it returns the remainder of the division rather than the quotient. For example, 5 MOD 2 returns the value 1, and 9 MOD 10 returns the value 9.

The [ISTRUE](#) operator returns TRUE only if its operand is TRUE (non-zero). ISTRUE is guaranteed to return -1 as its TRUE value, whereas the operators can return any non-zero value.

The [ISFALSE](#) operator returns TRUE only if its operand is FALSE (zero). ISFALSE is guaranteed to return -1 as its TRUE value, where the operators can return any non-zero value.

Classic PowerBASIC arithmetic operators

Operator	Action	Example
^	Exponentiation	10^4
-	Negation	-16
*	Multiplication	45 * 19
/	Floating-point division	45 / 19
\	Integer division	45 \ 19
+	Add	45 + 19
-	Subtract	45 - 19
MOD	Modulo	45 MOD 19
ISFALSE	Boolean False	ISFALSE 45
ISTRUE	Boolean True	ISTRUE 19
NOT , AND , OR , XOR , EQV , IMP	Bit manipulation operations	NOT 0, 45 AND 19 45 OR 19, 45 XOR 19 45 EQV 19, 45 IMP 19

Note: Classic PowerBASIC does not trap numeric overflow or underflow errors in equation and expression evaluation. Please refer to the topics [Errors](#) and [Error](#)

[Trapping](#) for more information.

It is recommended that this table be read in conjunction with the Mathematical Order of Operator Precedence table, and the effects that operator precedence has on the evaluation of numeric expressions.

See Also

[Relational Operators](#)

[Operator Precedence](#)

[LET statement](#)

Relational operators allow you to compare the values of two expressions, to obtain a Boolean result of [TRUE](#) or [FALSE](#). Although they can be used in any boolean expression (for example, `a = (b > c) / 13`), the numeric results returned by relational operators are generally used in an `if` or other decision statements, to make a judgment regarding program flow.

Classic PowerBASIC relational operators

Operator	Relation	Example
=	Equality	5 = 5
<>, ><	Inequality	5 <> 6
<	Less than	5 < 6
>	Greater than	6 > 5
<=, =<	Less than or equal to	5 <= 6
>=, =>	Greater than or equal to	6 >= 5

When [arithmetic](#) and relational operators are combined in an expression, arithmetic operations are always evaluated first. For example, `4 + 5 < 4 * 3` evaluates to TRUE (non-zero), because the arithmetic operations (addition and multiplication) are carried out before the relational operation. This then tests the truth of the assertion `9 < 12`.

Strings and relational operators

Classic PowerBASIC lets you compare string data. [String expressions](#) can be tested for equality, as well as for "greater than" and "less than" alphanumeric ordering.

Two string expressions are equal if *and only if* they contain exactly the same characters in exactly the same order. For example:

```
a$ = "CAT"
x1% = (a$ = "CAT") : x2% = (a$ = "CATS") : x3% = (a$ = "cat")
```

String ordering is based on two criteria: first, the ASCII values of the characters they contain, and second, the length of the strings.

For example, the letter *A* is less than the letter *B* because the ASCII code for *A*, 65, is less than the code for *B*, 66. Note, however, that *B* is less than *a* because the ASCII code for each lowercase letter is *greater* than the corresponding uppercase character (exactly 32 greater). When comparing mixed uppercase and lowercase information, use the [UCASE\\$](#) or [LCASE\\$](#) functions to keep case differences from interfering with the test.

```
city1$ = "Seattle"
city2$ = "Tucson"
IF UCASE$(city1$) > UCASE$(city2$) THEN
    city$ = city1$
ELSE
    city$ = city2$
END IF
```

```
city1$ = UCASE$(city1$)
city2$ = UCASE$(city2$)
IF city1$ > city2$ THEN
    city$ = city1$
ELSE
    city$ = city2$
END IF
```

Note the difference between the two sets of statements. In the first case, the string variables *city1\$* and *city2\$* are converted to uppercase for the comparison only, so the first [IF/THEN](#) returns Tucson. In the second case, the conversion is performed on the variables themselves, so the result will be TUCSON.

Length is important only if both strings are identical up to the length of the shorter string, in which case the shorter one evaluates as less than the longer one; for example, *CAT* is less than *CATS*.

The [ARRAY SORT](#) and [ARRAY SCAN](#) statements allow you to specify whether lower case characters are to be treated as uppercase for comparison purposes. You can also specify a string that explicitly determines the sorting order for all 256 ASCII characters.

See Also

[Arithmetic Operators](#)

[Operator Precedence](#)

Mathematical order of Operator Precedence

[Top](#) [Previous](#)
[Next](#)

- parentheses ()
- exponentiation (^)
- negation (-)
- multiplication (*), floating point division (/)
- integral division (\)
- modulo ([MOD](#))
- addition (+), subtraction (-)
- [relational operators](#) (<, <=, =, >=, >, <>)
- [NOT](#), [ISFALSE](#), [ISTRUE](#)
- [AND](#)
- [OR](#), [XOR](#) (exclusive OR)
- [EQV](#) (equivalence)
- [IMP](#) (implication)

For example, the expression $3+6 / 3$ evaluates to 5, not 3. Division has a higher priority than addition, so the division operation ($6 / 3$) is performed first. Even though the compiler will not get confused, people still could, so a better programming style might be to use $3 + (6 / 3)$ or $3 + 6/3$, either using parentheses or spacing to make the intent clear. Otherwise it is easy to misread the statement as $(3 + 6) / 3$.

To handle operations of the same priority, Classic PowerBASIC proceeds from left to right. For example, in the expression $4 - 3 + 6$, the subtraction ($4 - 3$) is performed before the addition ($3 + 6$), producing the intermediate expression $1 + 6$.

Operations inside parentheses are of the highest priority and are always evaluated first. Within parentheses, standard precedence is used. Use parentheses like garlic: generously, but not to excess.

Another example of the effect of Order of Precedence on an expression follows:

```
x = -1^2
```

At first glance, the result of 1 may be the expected result (since $-1 * -1 = 1$); however, the unary negation operator has a lower precedence than exponentiation, so the expression is evaluated as $x = -(1^2)$ which gives a result value of -1. As noted above, the use of parentheses can clarify the intended expression:

```
x = (-1)^2
```

See Also

[Arithmetic Operators](#)

Relational Operators

Unlike the [DOS versions of Classic PowerBASIC](#), Windows versions of Classic PowerBASIC employ a completely different philosophy: to generate the smallest and fastest possible code. Consequently, [error handling](#) is placed firmly in the hands of the programmer. Classic PowerBASIC does not stop your program when a run-time error occurs. It is responsibility of the programmer to check for any conceivable errors that may occur after executing a statement. This is especially true with disk access routines. This section describes the types of errors that may be encountered, and follows on with a discussion on error detection and error handling techniques.

Compile-time errors

Compile-time errors are generated when the compiler cannot resolve a problem in your source code while it is compiling. Examples include: typographical errors; assigning incorrect values to variables (such as "x\$ = 5"); and attempting to use a variable name which has not been dimensioned when [OPTION EXPLICIT](#) (or [#DIM ALL](#)) has been turned on.

When a compile-time error is detected, Classic PowerBASIC will display a message box indicating the error code, plus a brief description, along with the line number in the code where the error occurred. The offending line of code will also be displayed. If you are using the Classic PowerBASIC IDE, the caret will move to the offending line once the error dialog has been dismissed.

Run-time errors

Run-time errors are generated when execution of a particular code statement or function results in an error condition being set. Run-time errors caught by Classic PowerBASIC include Disk access violations (i.e., trying to write data to a full disk), out of bounds array and pointer access, and Memory allocation failures. Array bounds and null-pointer checking is only performed when [#DEBUG ERROR ON](#) is used.

Run-time errors can be trapped; that is, you can cause a designated error-handling subroutine to get control should an error occur. Use the [ON ERROR](#) statement to accomplish this. This routine can "judge" what to do next based on the type of error that occurs. File-system errors (for example, disk full) in particular are prime candidates for run-time error-handling routines. They are the only errors that a thoroughly debugged program should have to deal with.

The [ERROR](#) statement (which simulates run-time errors) can be used to debug your error-handling routines. It allows you to deliberately cause an error condition to be flagged. Avoid using error numbers higher than 240, as they are reserved for use in critical error situations which can never be trapped with ON ERROR. Run-time error values are restricted to the range 1 through 255, and the compiler reserves codes 0 through 150, and 241 through 255 for predefined errors. Attempting to set an error value (with the ERROR statement) outside of the valid range 1 to 255 will result in a run-time [Error 5](#) ("Illegal function call")

instead. In addition to the predefined run-time errors, you may also set your own customized run-time error codes in the range 151 through 240. These error codes may be useful to signal specific types of errors in your own applications, ready to be handled by your error trapping code.

In the situation where an undocumented run-time error occurs, the chief suspect is memory corruption. This can typically be caused by writing beyond an array boundary, improper use of pointers, bad Inline Assembly code, etc.

Disk Errors

Disk and I/O errors are *always* trapped at run-time by Classic PowerBASIC. If a run-time Disk or I/O error is detected, the error code is placed in the [ERR](#) system variable. If ON ERROR is enabled, code execution will branch to the designated local error handler.

All error handling in Classic PowerBASIC is local to each [Sub](#), [Function](#), [Method](#), and [Property](#). You cannot create a global error handler routine as you can in some DOS BASICs.

When an error occurs in Classic PowerBASIC, an error code is placed into the ERR system variable. If [Error Trapping](#) has been enabled, execution branches to the error trap. Otherwise, execution continues. If an error occurs and your code does not take care of it, either by using an error trap or by explicitly testing the ERR or [ERRCLEAR](#) variables, your program may produce unpredictable results. For example, in the following code, several problems can occur which would cause the code to fail, and possibly even trigger a General Protection Fault (GPF) in Windows:

```
SUB ReadFile(Filnam$, buffer$(), Lines%)
  RESET Lines%
  OPEN Filnam$ FOR INPUT AS #1
  WHILE ISFALSE EOF(1)
    INCR Lines%
    LINE INPUT #1, buffer$(Lines%)
  WEND
  CLOSE #1
END SUB
```

Here, the ERR variable is not checked after the [OPEN](#) statement to see if it was successful. If the file does not exist or has been locked by another process, a run-time error can occur. In this case, [EOF\(1\)](#) will never be able to return TRUE (non-zero) since the file was not able to be opened, and therefore the end of the file cannot be determined. Further, checking the EOF of a file that has not been opened will trigger yet another run-time error.

The result is that without adequate error testing, this small loop will begin to run continuously.

While certainly a flaw in the code, no harm will come to the program for period. However, a fatal error in the [LINE INPUT#](#) statement is imminent if the *Lines%* variable value exceeds the [UBOUND](#) of the *buffer\$()* array. A fatal error could also occur if *buffer\$()* was not previously dimensioned, or it wasn't dimensioned with enough elements to store the entire file (that is, assuming the file was opened successfully).

In these cases, a General Protection Fault (GPF) is quite likely to occur, as soon as invalid memory addresses begin to be accessed in an attempt to store the string data. You can prevent the array boundary GPF by turning on error checking using the #DEBUG ERROR ON metastatement. However, if the [array](#) was not previously dimensioned or does not have enough space, the code will still fail in its overall objective.

A more robust version of this example code follows:

```
#DEBUG ERROR ON
SUB ReadFile(Filnam$, buffer$(), Lines%)
    LOCAL Temp$
    ON ERROR RESUME NEXT

    RESET Lines%
    OPEN Filnam$ FOR INPUT AS #1
    IF ERR THEN                                'error opening file
        EXIT SUB
    END IF

    WHILE ISFALSE EOF(1)
        INCR Lines%
        LINE INPUT #1, Temp$
        IF ERR THEN EXIT SUB                    'abort if disk error
        buffer$(Lines%) = Temp$
        IF ERR = 9 THEN                        'subscript out of range
            REDIM PRESERVE buffer$(Lines%)    'increase array size
            buffer$(Lines%) = Temp$
        END IF
    WEND
    CLOSE #1
END SUB
```

Numeric Errors

In order to generate tight, fast code, we have eliminated quite a bit of error checking that was done in earlier compilers (such as Division-by-Zero, Numeric Overflow, and most other numeric checking errors). While this results in code that is considerably smaller and faster than any other Windows compiler product, it does put more of an onus on the programmer to write code that is bug-free, or code that does its own [error checking](#) and validation of its data.

For example, an application that performs exponentiation of a negative value to a fractional power ($-5^{1.9}$) will not trigger a run-time error, but the result of the expression will be undefined. Therefore, it makes sense for the application to make some attempt to validate or restrict the numeric range of the arguments of this kind of expression.

See Also

[Error Trapping](#)

Error traps let you intercept and deal with run-time errors, rather than having programs unceremoniously abort or ignore a fatal error, possibly causing loss of data.

There are three steps you must take to trap errors, as described in the following sections:

1. **Set the error trap.** Use the [ON ERROR GOTO](#) statement.
2. **Write the error-handling routine.** The error-handling routine receives control when an error occurs.
3. **Exit the error-handling routine.** You exit the error handler using the [RESUME](#) statement so execution can continue at an appropriate location in the code.

For example, here is a piece of code that fills an array with the filenames from a directory. This section will add complete Error Trapping to prevent [run-time errors](#) when the user chooses a directory that does not have any files or a drive that is not ready.

```
SUB GetFileNames(File() AS STRING)
  DIM CurrentDir AS STRING
  DIM fName AS STRING, Mask AS STRING
  DIM X AS INTEGER

  Mask = "*.*"
  CurrentDir = CURDIR$
  Path = AskUserForPath$()
  fName = DIR$(RTRIM$(Path) + Mask)
  IF LEN(fName) = 0 THEN EXIT SUB
  X = 1

  WHILE LEN(fName)
    Files(X) = fName
    fName = DIR$
    INCR X
  WEND
END SUB
```

See Also

[Error Overview](#)

[How error traps work](#)

[Setting an error trap](#)

[Writing an error handler](#)

[Exiting an error handler](#)

[Error Trapping Summary](#)

How error traps work

[Top](#) [Previous](#) [Next](#)

In Classic PowerBASIC, error codes - returned by the [ERR](#) or [ERRCLEAR](#) functions - and [error traps](#) are local to each [Sub](#), [Function](#), [Method](#), or [Property](#). An error trap will only trap errors that occur within the procedure where it is defined.

Classic PowerBASIC uses the following steps to determine what to do when a [run-time error](#) occurs:

- Does an error trap exist? If so, Classic PowerBASIC uses it.
- If no error trap exists, Classic PowerBASIC places an error code in the ERR and ERRCLEAR system variables and continues execution.

Consider the following:

```
SUB Proc1
  ON ERROR GOTO ErrorTrap
  ' some code goes in here
  CALL Proc2
  ' some more code goes in here

Proc1Resume:
  EXIT SUB

ErrorTrap:
  ' Error-handling code goes in here
  RESUME Proc1Resume
END SUB
```

See Also

[Error Overview](#)

[Error Trapping](#)

[Setting an error trap](#)

[Writing an error handler](#)

[Exiting an error handler](#)

[Error Trapping Summary](#)

Setting an error trap

[Top](#) [Previous](#) [Next](#)

To enable an [error trap](#), use the [ON ERROR GOTO](#) statement where you want trapping enabled within the procedure. The error-handling code must be within that procedure. An error trap is enabled only while the procedure is executing. Use the `ON ERROR GOTO 0` statement where you want trapping disabled within the procedure.

See Also

[Error Overview](#)

[Error Trapping](#)

[How error traps work](#)

[Writing an error handler](#)

[Exiting an error handler](#)

[Error Trapping Summary](#)

When an error occurs and an [error trap](#) invokes your error-handling routine, the first thing the code should do is to determine which error occurred. Classic PowerBASIC's [ERR](#) and [ERRCLEAR](#) functions return the code of the most recent error. You can use one of Classic PowerBASIC's control structures (like [SELECT CASE](#)) to take appropriate action based on the error code. The [ERROR\\$](#) function can be used to help formulate a suitable error message to log or report to the user.

See Also

[Error Overview](#)

[How error traps work](#)

[Setting an error trap](#)

[Exiting an error handler](#)

[Error Trapping Summary](#)

Exiting an error handler

[Top](#) [Previous](#) [Next](#)

You must exit an error handler with the [RESUME LABEL](#) statement. Execution branches immediately to the specified local [label](#), and the original [error trap](#) operation is restored ready to catch the next [run-time error](#).

In the sample program, you want to use RESUME to a specific line. Put the line label **before** the line that requests user input to give the user another chance to enter a correct path.

Here is the sample program, complete with Error Trapping:

```
SUB GetFileNames(File() AS STRING)
  DIM CurrentDir AS STRING
  DIM fName AS STRING, Mask AS STRING
  DIM X AS INTEGER
  ON ERROR GOTO ErrorTrap
  Mask = "*. *"
  CurrentDir = CURDIR$

  GetPath:
  Path = AskUserForPath$()
  fName = DIR$(RTRIM$(Path) + Mask)
  IF LEN(fName) = 0 THEN EXIT SUB
  X = 1
  WHILE LEN(fName)
    Files(X) = fName
    fName = DIR$
    INCR X
  WEND
  EXIT SUB

  ErrorTrap:
  SELECT CASE ERRCLEAR
    CASE 53 : ErrorMessage "No files in this directory."
    CASE 71 : ErrorMessage "Drive not ready."
    CASE 76 : ErrorMessage "That path doesn't exist."
    CASE ELSE : ErrorMessage "Unknown error!"
  END SELECT
  RESUME GetPath
END SUB
```

See Also

[Error Overview](#)

[Error Trapping](#)

[How error traps work](#)

[Setting an error trap](#)

[Writing an error handler](#)

[Error Trapping Summary](#)

[Error Trapping](#) is a useful and powerful feature of Classic PowerBASIC. Many programmers avoid trapping errors because of the substantial penalties other BASIC dialects impose when Error Trapping is turned on. Fortunately, Classic PowerBASIC's hit is much lower. Still, having Error Trapping turned on may increase the size of your executable. You may wish to investigate other ways to accomplish the same results, whilst ensuring the stability of your code.

Finally, Classic PowerBASIC now includes the following list of predefined (built-in) equates to assist in the creation of more verbose error handling code. They include:

<code>%ERR_NOERROR</code>	<code>= 0</code>
<code>%ERR_ILLEGALFUNCTIONCALL</code>	<code>= 5</code>
<code>%ERR_OVERFLOW</code>	<code>= 6 (reserved)</code>
<code>%ERR_OUTOFMEMORY</code>	<code>= 7</code>
<code>%ERR_SUBSCRIPTPOINTEROUTOFRANGE</code>	<code>= 9</code>
<code>%ERR_DIVISIONBYZERO</code>	<code>= 11 (reserved)</code>
<code>%ERR_DEVICETIMEOUT</code>	<code>= 24</code>
<code>%ERR_INTERNALERROR</code>	<code>= 51</code>
<code>%ERR_BADFILENAMEORNUMBER</code>	<code>= 52</code>
<code>%ERR_FILENOTFOUND</code>	<code>= 53</code>
<code>%ERR_BADFILEMODE</code>	<code>= 54</code>
<code>%ERR_FILEISOPEN</code>	<code>= 55</code>
<code>%ERR_DEVICEIOERROR</code>	<code>= 57</code>
<code>%ERR_FILEALREADYEXISTS</code>	<code>= 58</code>
<code>%ERR_DISKFULL</code>	<code>= 61</code>
<code>%ERR_INPUTPASTEND</code>	<code>= 62</code>
<code>%ERR_BADRECORDNUMBER</code>	<code>= 63</code>
<code>%ERR_BADFILENAME</code>	<code>= 64</code>
<code>%ERR_TOOMANYFILES</code>	<code>= 67</code>
<code>%ERR_DEVICEUNAVAILABLE</code>	<code>= 68</code>
<code>%ERR_COMMERROR</code>	<code>= 69</code>
<code>%ERR_PERMISSIONDENIED</code>	<code>= 70</code>
<code>%ERR_DISKNOTREADY</code>	<code>= 71</code>
<code>%ERR_DISKMEDIAERROR</code>	<code>= 72</code>
<code>%ERR_RENAMEACROSSDISKS</code>	<code>= 74</code>
<code>%ERR_PATHFILEACCESSERROR</code>	<code>= 75</code>
<code>%ERR_PATHNOTFOUND</code>	<code>= 76</code>
<code>%ERR_OBJECTERROR</code>	<code>= 99</code>
<code>%ERR_GLOBALMEMORYCORRUPT</code>	<code>= 241 (Previously %ERR_FARHEAPCORRUPT)</code>
<code>%ERR_STRINGSPACECORRUPT</code>	<code>= 242</code>

See Also

[Error Overview](#)

[Error Trapping](#)

[How error traps work](#)

[Setting an error trap](#)

[Writing an error handler](#)

[Exiting an error handler](#)

Error 401 - Expression too long/complex

[Top](#) [Previous](#)
[Next](#)

Expression too long/complex - The expression contained too many operators/operands; break it down into two or more simplified expressions.

Error 402 - Statement too long/complex

[Top](#) [Previous](#)
[Next](#)

Statement too long/complex - The statement complexity caused an overflow of the internal compiler buffers; break the statement down into two or more simplified statements. This error can also occur if a [SELECT CASE](#) structure using the AS CONST optimization causes the internal jump table to exceed the maximum size (approximately 3200 entries or 12 Kb).

Error 403 - #IF nesting overflow

[Top](#) [Previous](#) [Next](#)

#IF nesting overflow - Conditional compilation blocks ([#IF/#ELSE/#ENDIF](#)) can only be nested up to 16 levels deep.

Error 404 - #INCLUDE file/Macro nesting overflow

[Top](#) [Previous](#)
[Next](#)

#INCLUDE file/Macro nesting overflow - Include files and macros may be nested up to twelve levels deep. The most common cause of this error stems from excessive nesting and/or circular references. For example, a nested [#INCLUDE](#) file that includes itself or an ancestor file that in turn includes the file again, etc. Likewise, a macro that references itself either directly or indirectly can cause a circular reference. See the [MACRO](#) statement for more information on the limits of macro expansions.

Error 405 - Block nesting overflow

[Top](#) [Previous](#) [Next](#)

Block nesting overflow - Your program has too many statement block structures nested within each other. In Classic PowerBASIC block structures may be nested 64 levels deep.

Error 406 - Compiler out of memory

[Top](#) [Previous](#) [Next](#)

Compiler out of memory - Available compiler memory for symbol space, buffers, and so on, has been exhausted.

If no more memory is available, separate your program into a small main program which uses the [#INCLUDE](#) metastatement to include the rest of your program. You can also try the following steps:

- Remove unnecessary [line numbers and labels](#).
- Shorten your [variable](#) and [procedure names](#).
- If your code includes [WIN32API.INC](#): Try adding the "code exclusion" [equates](#) such as `%NOGDI = 1` to your code to cause the compiler to ignore large sections of the API file. Please review the first few pages of notes in WIN32API.INC for more information.

Alternatively, create a customized version of WIN32API.INC that contains just the definitions and declarations actually used by your code. The latter solution, whilst more work initially, will have the added benefit of much faster compilation times, and make your code more resistant to changes in subsequent releases of WIN32API.INC.

Error 407 - Source line too long

[Top](#) [Previous](#) [Next](#)

Source line too long - The line of code is too long for the compiler to process. This can also occur if the file contains lines of source code that are not CR/LF delimited. Try breaking the line of code up into smaller logical lines with the use of [line continuation](#) characters, and ensure that the file is using the Win32 standard of CR/LF line delimiting. If you are using a 3rd-party editor, try opening the source code file in the [Classic PowerBASIC IDE](#) and examine the lines where the error occurred -- merged lines here will be a good indication of invalid line delimiting.

Error 408 - Wrong compiler for this program

[Top](#) [Previous](#)
[Next](#)

Wrong compiler for this program - The compiler you are using is not compatible with the compiler version specified by the [#COMPILER](#) metastatement. Use the compiler specified by the `#COMPILER` metastatement. Another approach would be to change the `#COMPILER` settings to match your compiler but, this should be done with caution, since the program may no longer work the same way (or at all) with a different compiler.

Error 409 - Sub/Function/Method/Property is too large

[Top](#) [Previous](#)
[Next](#)

Sub/Function/Method/Property is too large - There is a reasonable limit for the physical size of a single [Sub](#), [Function](#), [Method](#), or [Property](#). The limit is imposed for practical reasons (such as the size of internal compiler buffers), but also for logical suitability. A huge block of code is very difficult to maintain. In the current version of Classic PowerBASIC, this absolute limit is set at approximately 12,000 lines of source code per procedure. Classic PowerBASIC recommends that each procedure perform one logical function, with a general goal of no more than perhaps 100 lines of source code. If you encounter this error, just break up your code into two or more procedures.

Error 411 - "," expected

[Top](#) [Previous](#) [Next](#)

"," expected - The statement's syntax requires a comma (,).

Error 412 - ";" expected

[Top](#) [Previous](#) [Next](#)

";" expected - The statement's syntax requires a semicolon (;).

Error 413 - "(" expected

[Top](#) [Previous](#) [Next](#)

"(" expected - The statement's syntax requires a left parenthesis (()).

Error 414 - ")" expected

[Top](#) [Previous](#) [Next](#)

")" expected - The statement's syntax requires a right parenthesis ()). The compiler encountered text or symbols where a right parenthesis was expected, or a parenthesis is missing. This error can also occur when attempting to pass more than 32 parameters to a [Sub](#), [Function](#), [Method](#), or [Property](#).

Error 415 - "=" expected

[Top](#) [Previous](#) [Next](#)

"="expected - The statement's syntax requires an equal sign (=).

Error 416 - "-" expected

[Top](#) [Previous](#) [Next](#)

"-" expected - The statement's syntax requires a hyphen (-).

Error 417 - "*" expected

[Top](#) [Previous](#) [Next](#)

"*" expected - The statement's syntax requires an asterisk (*).

Error 418 - Statement expected

[Top](#) [Previous](#) [Next](#)

Statement expected - A Classic PowerBASIC statement was expected. Some character could not be identified as a statement, [metastatement](#), or [variable](#).

Error 419 - Label/line number expected

[Top](#) [Previous](#)
[Next](#)

Label/line number expected - A valid [label or line-number](#) reference was expected in an IF, [GOTO](#), [GOSUB](#), or ON statement.

Error 420 - Relational operator expected

[Top](#) [Previous](#)
[Next](#)

Relational operator expected - The compiler has found a string operand in a position where a numeric operand should be, or a type mismatch has been detected.

For example, the statement `X& = Y$` triggers an error because a string cannot be assigned or compared to numeric [variable](#), hence the compiler expected to find an additional operator that would return a numeric result. For example, `X& = Y$ > Z$`.

Error 421 - String operand expected

[Top](#) [Previous](#) [Next](#)

String operand expected - The compiler expected a [string expression](#) and found something else; for example, `X$ = A$ + 3`.

Error 422 - Scalar variable expected

[Top](#) [Previous](#) [Next](#)

Scalar variable expected - The compiler expected a scalar [variable](#) as a formal parameter to a user-defined [function](#). Scalar variables are non-[array](#) variables.

Error 423 - Array variable expected

[Top](#) [Previous](#) [Next](#)

Array variable expected - An [array variable](#) was expected in a [DIM](#) statement.

Error 424 - Numeric variable expected [Top](#) [Previous](#) [Next](#)

Numeric variable expected - A numeric [variable](#) was expected, such as in an [INCR](#) or [DECR](#).

Error 425 - String variable expected

[Top](#) [Previous](#) [Next](#)

String variable expected - A string [variable](#) was expected, such as in a [PUT\\$](#) or a [GET\\$](#) statement.

Error 426 - Variable expected

[Top](#) [Previous](#) [Next](#)

Variable expected - A [variable](#) was expected, but not found. A common cause for this error is the use of a reserved keyword as a variable.

Error 427 - Integer constant expected [Top](#) [Previous](#) [Next](#)

Integer constant expected - An integral [constant](#), [numeric literal](#), or [numeric equate](#) was expected, such as in a named constant assignment.

This error can occur when attempting to use a numeric [variable](#) to dictate the size of the target of a fixed-length or [ASCIIZ](#) string pointer. For example:

```
DIM X AS STRING PTR * Y&
```

is not permitted as this statement could only be evaluated at run-time. However:

```
DIM X AS STRING PTR * 1024
```

is acceptable as the target size is known at compile-time.

Another cause of this error is specifying a non-integer CASE argument in a [SELECT CASE AS CONST](#) block.

Error 428 - Positive integer constant expected

[Top](#) [Previous](#)
[Next](#)

Positive integer constant expected - A positive integer [constant](#) was expected, but not found.

Error 429 - String constant expected [Top](#) [Previous](#) [Next](#)

String constant expected - A string constant was expected, but not found. For example, this error can occur when in a [SELECT CASE](#) AS CONST\$ block when a non-string CASE argument is specified.

Error 430 - Integer variable expected [Top](#) [Previous](#) [Next](#)

Integer variable expected - An integer [variable](#) was expected, but not found

Error 431 - Numeric scalar variable expected

[Top](#) [Previous](#)
[Next](#)

Numeric scalar variable expected - The counter [variable](#) in a [FOR/NEXT](#) counter variable is a BYREF parameter passed to the [Sub/Function/Method/Property](#), a pointer target, a [THREADED](#) variable, an [array](#) variable (non-scalar), or the counter variable is not a numeric data type. Scalar variables are non-array variables.

Error 432 - Long-integer variable expected

[Top](#) [Previous](#)
[Next](#)

Long-integer variable expected - A [Long-integer variable](#) is expected.

Error 433 - Matrix array expected (integer/float)

[Top](#) [Previous](#)
[Next](#)

Matrix array expected (integer/float) - Matrix [arrays](#) may only be of integer class or floating point types.

See Also

[MAT Statement](#)

Error 434 - End of line expected

[Top](#) [Previous](#) [Next](#)

End of line expected - No characters are allowed on a line (except for a [comment](#)) following a [metastatement](#), [END SUB](#), or a [label](#).

Error 435 - #IF expected

[Top](#) [Previous](#) [Next](#)

#IF expected - An [#ENDIF](#) conditional compilation metastatement is missing its accompanying #IF. Look for all #ENDIF [metastatements](#) and figure out where to put the associated #IF.

Error 436 - #ENDIF expected

[Top](#) [Previous](#) [Next](#)

#ENDIF expected - An [#IF](#) conditional compilation [metastatement](#) is missing its accompanying #ENDIF. Examine all #IF metastatements to determine where to put the associated #ENDIF.

Error 437 - AS expected

[Top](#) [Previous](#) [Next](#)

AS expected - The AS reserved word is missing, such as in a [variable](#) declaration.

Error 438 - Member name expected

[Top](#) [Previous](#) [Next](#)

Member name expected - The compiler encountered a statement or other text where a structure member name was expected.

Error 439 - GOSUB expected

[Top](#) [Previous](#) [Next](#)

GOSUB expected - An [ON statement](#) is missing its accompanying GOSUB part.

Error 440 - GOTO expected

[Top](#) [Previous](#) [Next](#)

GOTO expected - An [ON statement](#) is missing its accompanying GOTO part.

Error 441 - IN expected

[Top](#) [Previous](#) [Next](#)

IN expected - The IN reserved word is missing in a [REGEXPR](#), [REGREPL](#), or [REPLACE](#) statement. Check the syntax of the relevant statement in the reference directory section.

Error 442 - THEN expected

[Top](#) [Previous](#) [Next](#)

THEN expected - An IF is missing its accompanying THEN part.

Error 443 - TO expected

[Top](#) [Previous](#) [Next](#)

TO expected - Missing TO in a [FOR statement](#). This can also be reported for a missing TO in the [CALL](#) *FuncName* TO syntax.

Error 444 - WITH expected

[Top](#) [Previous](#) [Next](#)

WITH expected - The WITH reserved word is missing in a [REPLACE statement](#).

Error 445 - OF expected

[Top](#) [Previous](#) [Next](#)

OF expected - Indexed pointers with dual indexes require an "OF Limit" clause on both indexes. For example:

```
x = @w[i& OF m&, j& OF n&]
```

Error 446 - FUNCTION expected

[Top](#) [Previous](#) [Next](#)

FUNCTION expected - The compiler found an END FUNCTION or [EXIT FUNCTION](#) statement without a [FUNCTION](#) defined. When defining a FUNCTION, it must begin with a FUNCTION statement.

Error 447 - IF expected

[Top](#) [Previous](#) [Next](#)

IF expected - The compiler found an END IF or an [EXIT IF](#) statement without a beginning [IF statement](#) defined.

Error 448 - DO loop expected

[Top](#) [Previous](#) [Next](#)

DO loop expected - The compiler found a LOOP or [EXIT LOOP](#) statement without a beginning [DO statement](#) defined.

Error 449 - SELECT expected

[Top](#) [Previous](#) [Next](#)

SELECT expected - When defining a [SELECT CASE](#) statement, you either forgot to include the reserved word SELECT or the compiler ran into an END SELECT or [EXIT SELECT](#) without a beginning SELECT CASE statement. This error can also occur if you try to use the reserved word CASE as a [variable](#) name in your program.

Error 450 - CASE expected

[Top](#) [Previous](#) [Next](#)

CASE expected - When defining a [SELECT CASE](#) statement, you forgot to include the reserved word CASE. This error can also occur if you try to use the reserved word SELECT as a [variable](#) name in your program.

Error 451 - FOR loop expected

[Top](#) [Previous](#) [Next](#)

FOR loop expected - A [NEXT](#), [EXIT FOR](#), or [ITERATE FOR](#) was encountered here without the associated [FOR](#) statement to begin the FOR/NEXT loop.

Error 452 - SUB expected

[Top](#) [Previous](#) [Next](#)

SUB expected - An [END SUB](#) was encountered here without the associated [SUB](#) statement to begin the procedure.

Error 453 - Equate (%xyz) expected

[Top](#) [Previous](#) [Next](#)

Equate (%xyz) expected - The [%DEF\(\)](#) function requires a [numeric](#) or [string equate](#) name as the parameter. It returns [true](#) (non-zero) or [false](#) (zero) to advise whether this equate has been defined in the program.

Error 454 - END FUNCTION expected

[Top](#) [Previous](#) [Next](#)

END FUNCTION expected - A [FUNCTION](#) block was not terminated with an associated [END FUNCTION](#) statement. It's likely you tried to start a new procedure block, without first terminating the current FUNCTION.

Error 455 - END IF expected

[Top](#) [Previous](#) [Next](#)

END IF expected - An [IF](#) block was not terminated with a corresponding END IF statement.

Error 456 - LOOP/WEND expected

[Top](#) [Previous](#) [Next](#)

LOOP/WEND expected - A [DO](#) or [WHILE](#) loop was not terminated with a corresponding LOOP or WEND statement.

Error 457 - END SELECT expected

[Top](#) [Previous](#) [Next](#)

END SELECT expected - A [SELECT CASE](#) statement was not properly terminated with an END SELECT statement.

Error 458 - END SUB expected

[Top](#) [Previous](#) [Next](#)

END SUB expected - A [SUB](#) block was not terminated with an associated [END SUB](#) statement. It's likely you tried to start a new procedure block, without first terminating the current SUB.

Error 459 - NEXT expected

[Top](#) [Previous](#) [Next](#)

NEXT expected - A [FOR](#) loop was not properly terminated with a NEXT statement.

Error 460 - Undefined equate

[Top](#) [Previous](#) [Next](#)

Undefined equate - A named constant ([numeric equate](#) or [string equate](#)) was referenced in your program, but it has not yet been defined.

Error 461 - Array not dimensioned

[Top](#) [Previous](#) [Next](#)

Array not dimensioned - An [array](#) element was referenced here, but the array has not been dimensioned anywhere in the program. You must execute an explicit [DIM](#) statement for every array in a Classic PowerBASIC program.

Error 462 - Undefined Procedure reference

[Top](#) [Previous](#)
[Next](#)

Undefined Procedure reference - You attempted to execute or reference a [SUB](#) or [FUNCTION](#), but it has not been declared or defined anywhere in the program. Check for the possibility of spelling errors.

Error 463 - Undefined label/line reference

[Top](#) [Previous](#)
[Next](#)

Undefined label/line reference - You used a [label](#) or [line number](#), but it does not exist. Check for the possibility of spelling errors. Note that labels and line numbers are local to the routine where they are defined.

Error 464 - Undefined class reference [Top](#) [Previous](#) [Next](#)

Undefined class reference - You used a [CLASS](#) name which does not exist. You must define a CLASS before it can be used. Check for the possibility of spelling errors.

Error 465 - Duplicate definition

[Top](#) [Previous](#) [Next](#)

Duplicate definition - A program element which should only appear once was duplicated in your source code. For example, two [#STACK](#) metastatements could cause this error to be generated. A common source of this problem is multiple [#INCLUDE](#) files which define the same term.

See Also

[Error 466 - Duplicate name definition](#)

Error 466 - Duplicate name definition [Top](#) [Previous](#) [Next](#)

Duplicate name definition - A name (identifier) was used for more than one purpose, causing a fatal conflict. For example, you might have used the name ABC as both a [variable](#) and a [label](#). You must rename one or both uses of this particular name. Classic PowerBASIC always generates this error when it encounters the second use of the name. A popular strategy is to move the second usage as close to the top of the source code as possible, and compile it again. Since the relative positions are reversed, the compiler should now highlight the other usage of the name.

See also:

[Error 465 - Duplicate definition](#)

Error 467 - Duplicate line number

[Top](#) [Previous](#) [Next](#)

Duplicate line number - A [line number](#) was used more than once.

Error 468 - Duplicate equate

[Top](#) [Previous](#) [Next](#)

Duplicate equate - A [numeric](#) or [string equate](#) was defined a second time with a different value. Numeric equate definitions may appear more than once, but the assigned values must remain constant.

Error 469 - Quad integer variable expected

[Top](#) [Previous](#)
[Next](#)

Quad integer variable expected - A [Quad integer](#) variable is required as a parameter in this statement.

Error 471 - Invalid line number

[Top](#) [Previous](#) [Next](#)

Invalid line number - [Line numbers](#) must be in the range 1 through 65535.

Error 472 - Invalid label

[Top](#) [Previous](#) [Next](#)

Invalid label - A [label](#) in your code contains invalid characters, such as the period character.

Error 473 - Invalid numeric format

[Top](#) [Previous](#) [Next](#)

Invalid numeric format - Your program declared a number with more than 18 digits or a floating point number with an E component without the exponent value. This error can also occur if the "&" string concatenation operator is used without leading whitespace. For example: `a$ = a$&b$` should be written `a$ = a$ & b$`

Error 474 - Invalid name

[Top](#) [Previous](#) [Next](#)

Invalid name - A [function](#), [sub](#), [method](#), [property](#), [macro](#), or [label](#) has an invalid name. In the case of a Sub, Function, Method, or Property, the name must begin with a letter and can be followed by other letters, digits, and underscores, but may not include a [type-specifier](#) or period. In the case of a macro you may have a duplicate macro name defined.

Error 475 - Metastatements not allowed here

[Top](#) [Previous](#)
[Next](#)

Metastatements not allowed here - A [metastatement](#) must be the first statement on a line.

Error 476 - Block/scanned statements not allowed here

[Top](#)
[Previous](#)
[Next](#)

Block/scanned statements not allowed here - Block statements (like [WHILE/WEND](#), [DO/LOOP](#), and [SELECT CASE](#)) are not allowed in single line [IF](#) statements. In addition, you may not have a [Sub](#), [Function](#), [Method](#), or [Property](#) definition nested within the body of another definition. A missing END SUB, END FUNCTION, END METHOD, or END PROPERTY can also cause this error.

Error 477 - Syntax error

[Top](#) [Previous](#) [Next](#)

Syntax error - Something is incorrect on the line; however, the compiler could not determine a proper error message or decode the line further. A common cause is mixing two statement keywords together, using a reserved keyword for a [variable](#) name, or attempting to use an undefined [interface](#) member (in an [OBJECT](#) statement) when using [ID Binding](#), etc.

Error 478 - Resource file error

[Top](#) [Previous](#) [Next](#)

Resource file error - The [resource file](#) referenced has not been found or is not identifiable as valid resource file. A common cause of this problem is attempting to use [#RESOURCE](#) with a non-PBR file, or if the PBR file was not able to be opened by the compiler (for example, because the file is locked by another process or application).

Error 479 - Array bounds error

[Top](#) [Previous](#) [Next](#)

Array bounds error - You [dimensioned](#) an [array](#) within a [User-Defined Type](#) that contains invalid array boundaries. For example:

```
TYPE MyType
  ArrayWithinUDT(5 TO 1)
END TYPE
```

Error 480 - Parameter mismatches definition

[Top](#) [Previous](#)
[Next](#)

Parameter mismatches definition - You attempted to reference a procedure using a parameter which does not match (or cannot be converted to) the data type found in the original declaration/definition. This might also be caused by passing too few or too many parameters, misspellings, etc.

Error 481 - Mismatch with prior definition

[Top](#) [Previous](#)
[Next](#)

Mismatch with prior definition - This program element ([TYPE](#), [UNION](#), [SUB](#), [FUNCTION](#), etc.) does not match a declaration or definition found previously in the program. It could be a SUB or FUNCTION which mismatches a declaration, a duplicate TYPE or INTERFACE which is not identical, or another similar condition.

Error 482 - Data type mismatch

[Top](#) [Previous](#) [Next](#)

Data type mismatch - Many Classic PowerBASIC statements and functions require parameters which evaluate to a variable or expression of a particular data type. This error is generated if there is a mismatch with the expected data type. Consult the documentation for the specific statement or function to determine the exact parameter requirements.

Error 483 - Requires Object Procedure (Method/Property)

[Top](#)
[Previous](#)
[Next](#)

Requires Object Procedure (Method/Property) - The statement or function found here is only allowed within a [METHOD](#) or [PROPERTY](#). Elsewhere, it has no valid meaning and must be removed.

Error 484 - Requires procedure (Function/Method...)

[Top](#) [Previous](#)
[Next](#)

Requires procedure (Sub/Function/Method/Property) - The statement or function found here is only allowed within a procedure ([SUB](#), [FUNCTION](#), [METHOD](#) or [PROPERTY](#)). Elsewhere, it has no valid meaning and must be removed.

Error 485 - Dynamic/Field strings not allowed

[Top](#) [Previous](#)
[Next](#)

Dynamic/Field strings not allowed - A [TYPE](#) or [UNION](#) may not include a [dynamic string](#) or a [field string](#) as a member, because the total size of the structure must be known at compile-time. [Fixed-length strings](#) and [ASCIIZ strings](#) should be used instead.

Error 486 - BYVAL option not allowed [Top](#) [Previous](#) [Next](#)

BYVAL option not allowed - Use of the BYVAL option in this context is not allowed. This error is most frequently generated by an attempt to pass an array as a BYVAL parameter. Generally speaking, you should change this to BYREF instead.

Error 487 - Multiple NEXT not allowed [Top](#) [Previous](#) [Next](#)

Multiple NEXT not allowed - Prior versions of Classic PowerBASIC allowed multiple NEXT statements implied by, or separated by commas. This is no longer supported.

Error 488 - Numeric processor overflow

[Top](#) [Previous](#)
[Next](#)

Numeric processor overflow - Execution of this line of source code is complex, and requires more floating point registers than are currently available in the FPU. One solution might be to add the metastatement [#REGISTER NONE](#) to the current procedure, if [register](#) variables are being allocated. Another solution would be to break up the source code into multiple simpler statements.

Error 489 - Invalid string length

[Top](#) [Previous](#) [Next](#)

Invalid string length - You attempted to [DIM](#) a [fixed-length string](#) with a length of zero, or you attempted to create a [string equate](#) whose length exceeds 255 characters. Fixed-length strings must be at least 1 byte long, and individual string equates may not exceed 255 bytes in length

Error 490 - Static array too large

[Top](#) [Previous](#) [Next](#)

Static array too large - You attempted to [dimension](#) a [static array](#) larger than 16 MB in a [User-Defined Type](#).

Error 491 - Invalid register variable

[Top](#) [Previous](#) [Next](#)

Invalid register variable - You specified a [register](#) variable which is not allowed in this context. Register variables must be [LOCAL](#), and must be one of: [Integer](#), [Long](#), [Word](#), [DWord](#), or [Extended float](#). It's also possible this variable was used with a function such as [VARPTR\(\)](#), which requires a memory variable for correct execution.

Error 492 - Invalid SORT function

[Top](#) [Previous](#) [Next](#)

Invalid SORT function - [ARRAY SORT](#) of a custom [array](#) requires a custom user [FUNCTION](#) with a specific signature (2 BYREF parameters, STDCALLcalling conventions, etc.). The function you supplied did not meet these requirements.

Error 493 - Compiler file not found/accessible

[Top](#) [Previous](#)
[Next](#)

Compiler file not found/accessible - A source file could not be found in the specified directory path, or the current directory, or in the search path specified in the compiler [/I command-line](#) option. Alternatively, the file may be locked by another process. Check the directory paths or make sure that the specified file exists, and that another process or application has not locked the file.

Error 494 - ASM not allowed here

[Top](#) [Previous](#) [Next](#)

ASM not allowed here - You tried to use multiple statements on a line containing an [ASM](#) statement. An ASM statement must be the only statement on a line (plus an optional comment or [REM](#) statement).

Error 495 - Compiler file read error

[Top](#) [Previous](#) [Next](#)

Compiler file read error - During the compilation process, the compiler tried to open an [#INCLUDE](#) or [#RESOURCE](#) file, but a disk error was encountered. Verify that the file is present, not locked by another process, and that the disk itself is free from errors.

Error 496 - Destination file write error [Top](#) [Previous](#) [Next](#)

Destination file write error - During compilation the compiler received a disk write error. This can occur if the destination EXE is, for example, still running in memory when you attempt to compile, the target file is write locked by another process or compile session, or the target file is write-protected (read-only).

Error 497 - Assembler syntax error

[Top](#) [Previous](#) [Next](#)

Assembler syntax error - An [ASM](#) statement contains an invalid [assembly-language](#) construction.

Error 498 - Assembler variables must be declared

[Top](#) [Previous](#)
[Next](#)

Assembler variables must be declared - An attempt was made to reference an [assembly variable](#) before it was defined.

Error 499 - Statement must be first on line

[Top](#) [Previous](#)
[Next](#)

Statement must be first on line - A statement, such as [CASE](#), must be the first statement on a line. Split the compound statements apart so that each statement is on a separate line.

Error 500 - Variable name must be unique

[Top](#) [Previous](#)
[Next](#)

Variable name must be unique - Global, Threaded, and Instance variable names must be unique to guarantee access to a specific variable.

Error 502 - COM interface name expected

[Top](#) [Previous](#)
[Next](#)

COM interface name expected - This form of the [LET](#) (assignment) statement is used to create a [COM object](#), one which is created externally using the [COM](#) services provided by Windows. The associated [interface](#) name is not valid.

Error 503 - Multiple Main Functions have been defined

[Top](#) [Previous](#)
[Next](#)

Multiple Main Functions have been defined - The source code contains more than one entry point function. To resolve this error, remove the surplus entry point function(s), leaving just one entry point function appropriate to the type of file being compiled.

Both [Classic PowerBASIC for Windows](#) and the [Classic PowerBASIC Console Compiler](#) directly support three types of entry point functions for executable files ([MAIN](#), [WINMAIN](#), [PBMAIN](#)). All executable programs must contain one of these functions so Windows can determine where program execution should begin.

Conversely, an entry point function is optional in code that produces a dynamic link library (.DLL). However, if an entry point function is required, Classic PowerBASIC for Windows provides three types to choose from ([DLLMAIN](#), [LIBMAIN](#), and [PBLIBMAIN](#)).

Error 504 - Executable requires PBMAIN/WINMAIN function

[Top](#)
[Previous](#)
[Next](#)

Executable requires PBMAIN/WINMAIN function - No [WINMAIN](#) or [PBMAIN](#) function was located in an executable program. Without one of these functions, it is not possible for Windows to execute the program.

Error 505 - Debugging requires EXE file, not DLL

[Top](#) [Previous](#)
[Next](#)

Debugging requires EXE file, not DLL - An attempt was made to launch the [debugger](#) on a DLL rather than an EXE file (PB/Win only). Be sure to use an explicit [#COMPILE EXE](#) metastatement to ensure the compiler generates the correct type of compiled code.

Error 506 - Declaration must precede statements

[Top](#) [Previous](#)
[Next](#)

Declaration must precede statements - You attempted to use a declaration, such as a [#DIM ALL](#) metastatement after executable code. Move the declaration to a position before any statements that generate executable code.

Error 507 - OLE variable expected

[Top](#) [Previous](#) [Next](#)

OLE variable expected - The [OBJECT](#) statement requires that all parameters, return values, and assignment values be in the form of [COM](#)-compatible variables. [Literals](#) and expressions are not allowed. COM-compatible variables include [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [STRING](#), and [VARIANT](#).

Error 508 - INSTANCE not allowed here

[Top](#) [Previous](#)
[Next](#)

INSTANCE not allowed here - [INSTANCE](#) statements may only be placed at the beginning of a [CLASS/END CLASS](#) block, preceding all [INTERFACE](#) blocks and [METHODS](#).

Error 509 - Interface mismatches class

[Top](#) [Previous](#)
[Next](#)

Interface mismatches class - This form of the [LET](#) (assignment) statement is used to create an internal [object](#), but the associated [class](#) and [interface](#) are not defined in the program.

Error 510 - Interface name expected

[Top](#) [Previous](#) [Next](#)

Interface name expected - The compiler encountered a statement or other text where an interface name was expected.

Error 511 - Numeric operand expected

[Top](#) [Previous](#) [Next](#)

Numeric operand expected - The compiler encountered a statement or other text where a numeric operand was expected.

Error 512 - Nested brackets not allowed

[Top](#) [Previous](#)
[Next](#)

Nested brackets not allowed - You attempted to use nested brackets in an OPTIONAL parameter declaration.

Error 513 - "]" expected

[Top](#) [Previous](#) [Next](#)

"]" expected - The statement's syntax requires a closing bracket (]).

Error 514 - Enclosing <...> angle brackets expected

[Top](#) [Previous](#)
[Next](#)

Enclosing <...> angle brackets expected - An INTERFACE definition block member item requires a parameter enclosed with angle brackets to identify the member ID.

Error 515 - Fixup overflow

[Top](#) [Previous](#) [Next](#)

Fixup overflow - You have a jump short instruction that exceeds its maximum length.

Error 516 - DEFtype, Type ID or type-specifier required

[Top](#) [Previous](#)
[Next](#)

DEFtype, Type ID or type-specifier (?%&!#\$), or AS ... required - A [variable](#) with no [type declaration](#) was found and no [DEFtype](#) statement (such as [DEFINT](#)) was found. The compiler was unable to identify the type of variable indicated. The misspelling of variable names commonly causes this error. The DEFtype statement may not be supported in future editions of Classic PowerBASIC. Use explicit declarations wherever possible to maintain future compatibility.

Error 517 - OPTIONAL requires CDECL or SDECL

[Top](#) [Previous](#)
[Next](#)

OPTIONAL requires CDECL or SDECL - The OPTIONAL (or OPT) clause in a [DECLARE](#), [SUB](#), [FUNCTION](#), [METHOD](#), or [PROPERTY](#) statement requires the CDECL or SDECL/STDCALL calling convention. You may not use OPTIONAL or OPT parameters with BDECL calling convention.

Error 518 - "[...]" requires CDECL

[Top](#) [Previous](#) [Next](#)

"[...]" requires CDECL - You attempted to use the classic optional parameter syntax in a [SUB](#), [FUNCTION](#), [METHOD](#), or [PROPERTY](#) without declaring it to use CDECL calling convention ('C' style calling convention).

Error 519 - Missing declaration

[Top](#) [Previous](#) [Next](#)

Missing declaration - You specified that all [variables](#) must be declared before use, but this one was not declared. Use [DIM](#), [GLOBAL](#), [INSTANCE](#), [LOCAL](#), [STATIC](#), or [THREADED](#) to declare the data type and dimensions, if an [array](#). To declare [Register Variables](#) use the [REGISTER](#) statement.

Error 520 - TYPE expected

[Top](#) [Previous](#) [Next](#)

TYPE expected - An [END TYPE](#) was encountered here without the associated [TYPE](#) statement to initiate the data block.

Error 521 - UNION expected

[Top](#) [Previous](#) [Next](#)

UNION expected - An [END UNION](#) was encountered here without the associated [UNION](#) statement to initiate the data block.

Error 522 - END TYPE expected

[Top](#) [Previous](#) [Next](#)

END TYPE expected - The compiler found a [TYPE](#) statement without a terminating END TYPE statement.

Error 523 - END UNION expected

[Top](#) [Previous](#) [Next](#)

END UNION expected - The compiler found a [UNION](#) statement without a terminating END UNION statement.

Error 524 - Undefined type

[Top](#) [Previous](#) [Next](#)

Undefined type - You referenced a [TYPE](#) or [UNION](#) which was not defined. Check for the possibility of spelling errors.

Error 525 - Type ID or specifier (?%&!#)\$) not allowed

[Top](#) [Previous](#)
[Next](#)

Type ID or specifier (?%&!#)\$) not allowed - Members in a [User-Defined Type \(UDT\)](#) or [UNION](#) variable must not include type ID or [type-specifier](#) characters. Change the definition to use the AS type syntax instead.

Error 526 - Period not allowed

[Top](#) [Previous](#) [Next](#)

Period not allowed - Periods are not allowed within any identifier names. They may only be used as a separator for member names. A good alternative is to use an underscore (`_`) character to decorate variable names.

Error 527 - End of statement expected

[Top](#) [Previous](#) [Next](#)

End of statement expected - There were one or more extra characters at the end of this statement.

Error 528 - Type too large

[Top](#) [Previous](#) [Next](#)

Type too large - This [TYPE](#) or [UNION](#) exceeded the 16 Megabyte structure size limit.

Error 529 - Pointer variable error

[Top](#) [Previous](#) [Next](#)

Pointer variable error - You used [pointer](#) variable syntax incorrectly, such as placing a leading "@" on a variable which is not declared as a pointer.

Error 530 - Invalid member name/definition

[Top](#) [Previous](#)
[Next](#)

Invalid member name/definition - This usage of a member name or definition is not allowed in a [TYPE](#), [UNION](#), or INTERFACE. The name could be invalid, or the data type could be disallowed. See the specific statement definition for more information.

Error 531 - Object variable expected

[Top](#) [Previous](#) [Next](#)

Object variable expected - The syntax of this statement or function requires an [object variable](#) here. Substitution with another data type is not possible. See the specific statement definition for more information.

Error 532 - Variant variable expected [Top](#) [Previous](#) [Next](#)

Variant variable expected - The syntax of this statement or function requires a [VARIANT](#) variable here. Substitution with another data type is not possible. See the specific statement definition for more information.

Error 533 - IDispatch object variable expected

[Top](#) [Previous](#)
[Next](#)

IDispatch object variable expected - The [OBJECT](#) statement requires an [object variable](#) which has either been declared as [IDISPATCH](#) (for [late binding](#)), or by a specific dispatch interface (for [ID binding](#)).

Error 534 - Bit field error

[Top](#) [Previous](#) [Next](#)

Bit field error - An error was made in defining a bit field of [BIT/SBIT](#) variables. For example, it could be that the first variable in the bit field did not define the total size (using [IN BYTE|WORD|DWORD](#)), or the total number of bits may have exceeded the maximum of 32.

Error 535 - Dynamic string variable expected

[Top](#) [Previous](#)
[Next](#)

Dynamic string variable expected - The syntax of this statement or function requires a [dynamic string](#) variable here. Substitution with another data type is not possible. See the specific statement definition for more information.

Error 536 - Too many imports

[Top](#) [Previous](#) [Next](#)

Too many imports - The program has exceeded the maximum number of allowed imports.

Error 537 - Pointer expected

[Top](#) [Previous](#) [Next](#)

Pointer expected - This operation expects a [pointer](#). For example:

```
. . . @PtrName[n]
```

Error 538 - Invalid FOR/NEXT limits

[Top](#) [Previous](#) [Next](#)

Invalid FOR/NEXT limits - The specified start, stop and/or increment value(s) for a [FOR/NEXT](#) loop are not within the allowable range for the class of counter [variable](#) used. For example, you attempted to specify an increment value of -1 (a signed value) when the loop counter uses an unsigned variable. This error is also generated if the compiler is able to determine, at compile time, that the start and stop values chosen will prevent the FOR/NEXT from ever executing, e.g., FOR x = 10 TO 1.

Error 539 - Invalid thread function

[Top](#) [Previous](#) [Next](#)

Invalid thread function - A valid thread [Function](#) may only take one 32-bit [LONG](#) or [DWORD](#) parameter, which must be received by value (BYVAL). This error can occur if the thread Function does not match the following syntax:

```
FUNCTION ThreadFuncName (BYVAL param AS {LONG | DWORD}) AS {LONG | DWORD}
```

An error 539 can also occur if the target thread Function is declared to use a DWORD parameter but is passed a Long-integer, or vice-versa. You must pass the correct (matching) data type for the thread Function parameter. For example:

```
THREAD CREATE MyThread(y&) TO hThread???  
[statements]  
FUNCTION MyThread(BYVAL x AS LONG) AS LONG
```

Or

```
THREAD CREATE MyThread(y???) TO hThread???  
[statements]  
FUNCTION MyThread(BYVAL x AS DWORD) AS LONG
```

See Also

[THREAD CREATE statement](#)

Error 540 - Float opcode with a register variable

[Top](#) [Previous](#)
[Next](#)

Float opcode with a register variable - An inline assembler statement ([ASM](#)) referenced an integer class [register](#) variable with a floating point opcode using the FPU. This form of operation is not supported by the CPU and FPU.

Error 541 - Register size conflict

[Top](#) [Previous](#) [Next](#)

Register size conflict - An inline assembler statement ([ASM](#)) used [registers](#) or a memory [operand](#) which conflicted in size. For example, an attempt might have been made to move a value such as:

```
ASM MOV    AX, EBX
ASM SUB    EBX, DL
```

Error 542 - May not be altered

[Top](#) [Previous](#) [Next](#)

May not be altered - An attempt was made to change the value of a read-only parameter. For example, [COMM SET](#) cannot be used with RING, RLSD, RXQUE or TXQUE.

Error 543 - Must be outside Sub/Function/Class...

[Top](#) [Previous](#)
[Next](#)

Must be outside Sub/Function/Class... - This statement/function is only allowed outside of any [Sub](#), [Function](#), [Method](#), or [Property](#) block. It should be moved to the correct location.

Error 544 - Field variable expected

[Top](#) [Previous](#) [Next](#)

Field variable expected - The syntax of this statement or function requires a [field](#) variable here. Substitution with another data type is not possible. See the specific statement definition for more information.

Error 545 - AT expected

[Top](#) [Previous](#) [Next](#)

AT expected - The syntax of this statement or function requires the word AT here. See the specific statement definition for more information.

Error 546 - Use only as a Callback

[Top](#) [Previous](#) [Next](#)

Use only as a Callback - You tried to explicitly call a DDT [Callback](#) function. Callback functions may only be invoked by the [DDT](#) engine or Windows. To reference it indirectly, send an appropriate window message using [CONTROL SEND](#) or [DIALOG SEND](#). To send custom messages, be sure to use message values higher than %WM_USER+500 to avoid conflicts with other notification messages.

Error 547 - Callback function required [Top](#) [Previous](#) [Next](#)

Callback function required - A [Callback](#) Function was named but the target function was not defined as a CALLBACK, or the nominated function was not a Callback Function.
(PB/Win only)

Error 548 - No parameters with Callback

[Top](#) [Previous](#)
[Next](#)

No parameters with Callback - A [Callback](#) Function definition cannot specify parameters. (PB/Win only)

Omit the parameters from the function definition. For example:

```
CALLBACK FUNCTION Dlg1Callback()  
[statements]  
END FUNCTION
```

Error 549 - BYVAL required with pointers

[Top](#) [Previous](#)
[Next](#)

BYVAL required with pointers - [Pointers](#) may only be passed BYVAL. Add an explicit BYVAL to the [Sub/Function/Method/Property](#) declaration and prototype. Previous versions of Classic PowerBASIC used an implied BYVAL.

Error 550 - Too many data statements [Top](#) [Previous](#) [Next](#)

Too many data statements - Data is limited to 64 Kb per [Sub](#), [Function](#), [Method](#), or [Property](#), and 16384 individual data items. Either reduce the number of [DATA](#) statements, or split the data into separate procedures.

Error 551 - Not supported in this version

[Top](#) [Previous](#)
[Next](#)

Not supported in this version - An attempt was made to use a feature that is not supported by this version of the compiler. This error may also occur if a reserved word is used as a [variable](#), [label](#), [Sub](#), [Function](#), [Method](#), or [Property](#) name. For example, using INP or OUT.

Error 552 - TRY statement expected

[Top](#) [Previous](#) [Next](#)

TRY statement expected - Classic PowerBASIC expected to find a [TRY](#) statement at or before the indicated position in the code. Check the syntax of the surrounding code for other syntax errors, such as the misplacement of a CATCH or END TRY statement, conditional compilation excluding required portions of the code, etc.

Error 553 - CATCH statement expected

[Top](#) [Previous](#)
[Next](#)

CATCH statement expected - A [TRY/END TRY](#) block did not include a CATCH statement. Recheck the syntax of the block.

Error 554 - END TRY statement expected

[Top](#) [Previous](#)
[Next](#)

END TRY statement expected - A [TRY/END TRY](#) block appears to be missing its END TRY clause. This can typically occur if an [END SUB](#), [END FUNCTION](#), [END METHOD](#), [END PROPERTY](#) statement was encountered within the TRY/END TRY block.

Error 555 - ON ERROR/RESUME not allowed here

[Top](#) [Previous](#)
[Next](#)

ON ERROR/RESUME not allowed here - An attempt was made to include an [ON ERROR](#) or a [RESUME](#) statement inside a [TRY/END TRY](#) block. Remove the ON ERROR or RESUME statement or move it out of the TRY/END TRY block. [Error handling](#) is automatic within a TRY/END TRY block.

Error 556 - Function restricted to threads

[Top](#) [Previous](#)
[Next](#)

Function restricted to threads - [Functions](#) that are called with [THREAD CREATE](#) may not be called in the conventional manner. This restriction is necessary because thread Functions require additional initialization steps that are not included in standard function code.

One situation that can arise is where a Function may need to be invoked both directly and used as a thread Function. The easiest solution is to create a small wrapper function for the function, then use THREAD CREATE with the wrapper function, or call the original function directly. For example:

```
FUNCTION WorkerFunc(BYVAL x AS LONG) AS LONG
    ' code here
END FUNCTION

FUNCTION WorkerThread(BYVAL x AS LONG) AS LONG
    FUNCTION = WorkerFunc(x)
END FUNCTION

' more code here

' Execute the worker function directly, thus:
lResult& = WorkerFunc(var&)

' Execute the worker thread as a thread, using
' the wrapper function:
THREAD CREATE WorkerThread(var&) TO hThread???
```

Error 557 - Macro too long/complex

[Top](#) [Previous](#) [Next](#)

Macro too long/complex - An attempt was made to create a [MACRO](#) that is too long or complex. An individual macro can contain replacement text of up to approximately 4000 characters, and can specify up to 240 parameters occupy up to approximately 2000 bytes expanded space per macro. Macro substitutions are limited to an expanded total of approximately 16000 characters per line of original source code.

Error 558 - MACRO expected

[Top](#) [Previous](#) [Next](#)

MACRO expected - An END MACRO statement was found without a matching [MACRO](#) statement. Please recheck the syntax of the macro block.

Error 559 - END MACRO expected

[Top](#) [Previous](#) [Next](#)

END MACRO expected - A [MACRO](#) block appears to be missing a terminating END MACRO statement. Please recheck the syntax of the macro block.

Error 562 - INTERFACE expected

[Top](#) [Previous](#) [Next](#)

INTERFACE expected - An END INTERFACE statement was found to be without a matching INTERFACE statement. Please recheck the syntax of the interface definition block.

Error 563 - END INTERFACE expected [Top](#) [Previous](#) [Next](#)

END INTERFACE expected - An INTERFACE statement was found without a matching END INTERFACE statement. Please recheck the syntax of the interface definition block.

Error 564 - MACROTEMP not allowed here

[Top](#) [Previous](#)
[Next](#)

MACROTEMP not allowed here - Classic PowerBASIC encountered a [MACROTEMP](#) statement outside the scope of a [MACRO](#) block.

Error 565 - Macro mismatch with code position

[Top](#) [Previous](#)
[Next](#)

Macro mismatch with code position - The compiler encountered a multi-line [MACRO](#) statement in a non-statement position.

Error 566 - CLASS expected

[Top](#) [Previous](#) [Next](#)

CLASS expected - An [END CLASS](#) statement was encountered here without the associated [CLASS](#) statement to initiate the block.

Error 567 - END CLASS expected

[Top](#) [Previous](#) [Next](#)

END CLASS expected - A [CLASS](#) block was not terminated with an associated [END CLASS](#) statement.

Error 568 - METHOD expected

[Top](#) [Previous](#) [Next](#)

METHOD expected - An [END METHOD](#) statement was encountered here without the associated [METHOD](#) statement to initiate the block.

Error 569 - END METHOD expected

[Top](#) [Previous](#) [Next](#)

END METHOD expected - A [METHOD](#) block was not terminated with an associated [END METHOD](#) statement. It's likely you tried to start a new procedure block, without first terminating the current METHOD.

Error 570 - PROPERTY expected

[Top](#) [Previous](#) [Next](#)

Property expected - An [END PROPERTY](#) statement was encountered here without the associated [PROPERTY](#) statement to initiate the block.

Error 571 - END PROPERTY expected [Top](#) [Previous](#) [Next](#)

END METHOD expected - A [PROPERTY](#) block was not terminated with an associated [END PROPERTY](#) statement. It's likely you tried to start a new procedure block, without first terminating the current PROPERTY.

Error 572 - PROPERTY GET expected [Top](#) [Previous](#) [Next](#)

PROPERTY GET expected - A `PROPERTY = nnnn` statement (for assigning the return value) was found, but it was not located within a [PROPERTY GET](#) block. It is not allowed at any other location in your program.

Error 573 - Valid only in a CALLBACK FUNCTION

[Top](#) [Previous](#)
[Next](#)

Error 573 - Valid only in a CALLBACK FUNCTION - FUNCTION = x,y with two parameters is only valid in a [CALLBACK](#) FUNCTION.

Error 574 - Not allowed in an Event Class

[Top](#) [Previous](#)
[Next](#)

Not allowed in an Event Class - The statement or function found here is not allowed within an [EVENT CLASS](#). It has no valid meaning and must be removed. See the specific statement definition for more information.

Error 575 - EVENT SOURCE is not declared

[Top](#) [Previous](#)
[Next](#)

EVENT SOURCE is not declared - You included code which generates events with the [RAISEEVENT](#) statement, but did not declare an event source with the [EVENT SOURCE](#) statement.

Error 576 - Too many Interfaces

[Top](#) [Previous](#) [Next](#)

Too many Interfaces - Classic PowerBASIC allows up to 32 [interfaces](#) per [CLASS](#), but you have exceeded that limit. You should try to combine two or more of those interfaces.

Error 577 - EVENT INTERFACE expected

[Top](#) [Previous](#)
[Next](#)

EVENT INTERFACE expected - The [EVENT INTERFACE](#) you specified could not be found.

Error 578 - INHERIT of Base Class expected

[Top](#) [Previous](#)
[Next](#)

INHERIT of Base Class expected - Every [INTERFACE](#) must INHERIT from a [base class](#), which may be nested any level. Ultimately, every interface inherits from [IUnknown](#). The INHERIT statement must be the first statement in every INTERFACE block.

Error 579 - BYREF variable or BYVAL/BYREF variant expected

[Top](#)
[Previous](#)
[Next](#)

BYREF variable or BYVAL/BYREF variant expected - The [ISMISSING\(\)](#) function can only detect a missing parameter for a BYREF variable, or a BYVAL/BYREF [variant](#).

Error 580 - Duplicate GUID usage

[Top](#) [Previous](#) [Next](#)

Duplicate GUID usage - You have used a single [GUID](#) to identify two or more elements of your program. Change at least one of the GUIDs to a new value.

Error 581 - Type Library creation error [Top](#) [Previous](#) [Next](#)

Type Library creation error - A system error occurred while creating the COM [Type Library](#). The common cause of this error is using a data type not supported by Type Libraries. Type Libraries only support the following data types: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). Either suppress the creation of a Type Library by using the [#COM TLIB OFF](#) metastatement or by changing the [Methods](#) and [Properties](#) to only use supported data types.

Error 582 - Duplicate Dispatch interface

[Top](#) [Previous](#)
[Next](#)

Too many DISPATCH interfaces - Only one [Dispatch](#) (DUAL) interface is allowed per [CLASS](#).

Error 583 - Unpaired PROPERTY definition

[Top](#) [Previous](#)
[Next](#)

Unpaired PROPERTY definition - If you create both a [PROPERTY GET](#) and a [PROPERTY SET](#), they must be paired. The parameters and the property value must be identical in both forms, and the PROPERTY SET must immediately follow the PROPERTY GET.

Error 584 - Mismatched PROPERTY pair

[Top](#) [Previous](#)
[Next](#)

Mismatched PROPERTY pair - If you create both a [PROPERTY GET](#) and a [PROPERTY SET](#), they must be paired. The parameters and the property value must be identical in both forms, and the PROPERTY SET must immediately follow the PROPERTY GET.

Error 585 - PROPERTY requires BYVAL parameters

[Top](#) [Previous](#)
[Next](#)

PROPERTY requires BYVAL parameters - [PROPERTY](#) methods created in Classic PowerBASIC must have BYVAL parameters.

Error 586 - User Defined Type or AS expected

[Top](#) [Previous](#)
[Next](#)

User Defined Type or AS expected - The name of a [User-Defined TYPE](#), or an "AS <type>" clause is required here.

Error 587 - Invalid Constructor/Destructor

[Top](#) [Previous](#)
[Next](#)

Invalid Constructor/Destructor - [Constructor](#) and [Destructor](#) Methods must be [CLASS METHODS](#). They must take no parameters and return no result.

Error 588 - Indirect operand must be bracketed: [12]

[Top](#) [Previous](#)
[Next](#)

Indirect operand must be bracketed: [12] - An inline assembler ([ASM](#)) opcode which includes indirect addressing must enclose that operand in square brackets.

Error 589 - Dual/IDispatch interface is required

[Top](#) [Previous](#)
[Next](#)

Dual/IDispatch interface is required - This statement or construct may only be used in a [DUAL interface](#).

Error 590 - PROPERTY SET requires at least one parameter

[Top](#)
[Previous](#)
[Next](#)

PROPERTY SET requires at least one parameter - [PROPERTY SET](#) is used to assign a value to an [INSTANCE](#) variable. At least one parameter is mandatory to hold that value.

Error 591 - BYVAL with OUT is not allowed

[Top](#) [Previous](#)
[Next](#)

BYVAL with OUT is not allowed - OUT parameter may not be BYVAL, because those are destroyed before the OUT value could be retrieved.

Error 592 - Return value required

[Top](#) [Previous](#) [Next](#)

Return value required - [GET PROPERTY](#) requires a return value to hold the retrieved value.

Error 593 - Dual or Automation interface is required

[Top](#) [Previous](#)
[Next](#)

Dual or Automation interface is required - [OBJRESULT](#) is only valid in a [DUAL](#) or [IAUTOMATION](#) interface.

Error 594 - Macro ends with continuation '_'

[Top](#) [Previous](#)
[Next](#)

Macro ends with continuation '_' - [MACRO](#) body text may not end with an underscore continuation character.

Error 595 - Object return type required

[Top](#) [Previous](#) [Next](#)

Object return type required - [Component methods](#) in a [Compound Object Reference](#) must each return an [object variable](#) to be used by the next method.

Error 596 - Inherited interface expected

[Top](#) [Previous](#)
[Next](#)

Inherited interface expected - [MYBASE](#) may only be used on an [interface](#) which is derived from an [inherited](#) user-created interface.

Error 597 - Invalid name or sequence in the interface

[Top](#) [Previous](#)
[Next](#)

Invalid name or sequence in the interface - To [OVERRIDE](#) an [inherited METHOD](#), the replacement must have the same name and signature, and appear in the same sequence.

Error 598 - CLASS METHOD name expected

[Top](#) [Previous](#)
[Next](#)

METHOD or PROPERTY name expected - A valid [METHOD](#) or [PROPERTY](#) name must appear in this context.

Error 599 - Invalid within an INTERFACE

[Top](#) [Previous](#)
[Next](#)

Invalid within an INTERFACE - A [CLASS METHOD](#) is invalid within an [interface](#).

Error 600 - Macro phase error, referenced before define

[Top](#)
[Previous](#)
[Next](#)

Macro phase error, referenced before define - A macro was referenced before it was defined.

Error 601 - One INHERIT per interface [Top](#) [Previous](#) [Next](#)

One INHERIT per interface - Classic PowerBASIC offers single [inheritance](#), so just one [INHERIT](#) is allowed per [interface](#). However, the inherited interface may itself inherit from another interface, to virtually any level of nesting.

Error 602 - Hidden interface referenced by COM

[Top](#) [Previous](#)
[Next](#)

Hidden interface referenced by COM - The compiler was not able to create a [Type Library](#). The most likely cause is the use of a Hidden [Interface](#) as a parameter or return value in a [METHOD](#) or [PROPERTY](#) published [AS COM](#).

Error 603 - Incompatible with a Dual/IDispatch interface

[Top](#)
[Previous](#)
[Next](#)

Incompatible with a Dual/IDispatch interface - This data type cannot be passed as a [variant](#).

Error 604 - Incompatible with #COM TLIB generation

[Top](#) [Previous](#)
[Next](#)

Incompatible with #COM TLIB generation - This data type cannot be described in a [Type Library](#).

Error 605 - Macro parameter mismatch

[Top](#) [Previous](#)
[Next](#)

Macro parameter mismatch - A Macro parameter does not match the original definition.

Error 606 - Macro empty parentheses "()" are needed

[Top](#) [Previous](#)
[Next](#)

Macro empty parentheses "()" are needed - This reference to a macro requires empty parentheses to declare there are no macro parameters.

Error 607 - Too many macro expansions

[Top](#) [Previous](#)
[Next](#)

Too many macro expansions - Any one program may expand up to 65535 macros.

Error 613 - Cannot compile - the program is now running

[Top](#)
[Previous](#)
[Next](#)

Cannot compile - the program is now running - The program you are trying to compile is currently executing. You may have to use Task Manager to force the program to end.

Error 801 to 815 - Internal error

[Top](#) [Previous](#) [Next](#)

Internal error - If one of these errors occurs, please report it to the Classic PowerBASIC [Technical Support](#) group.

Error 0 - No error

[Top](#) [Previous](#) [Next](#)

No error (%ERR_NOERROR)

Error 5 - Illegal function call

[Top](#) [Previous](#) [Next](#)

Illegal function call - (%ERR_ILLEGALFUNCTIONCALL) - This is a catch-all error related to passing an inappropriate argument to some statement or [function](#).

There are many things that can cause an Error 5, for example:

- Trying to set [TIMES](#) to an invalid value.
- A record number is too large (or negative) in a [GET](#) or [PUT](#).
- Attempting to use the WIDTH# statement on a non-sequential file.
- The run-time execution of a [LET](#), [LET \(with Objects\)](#), [LET \(with Types\)](#), [LET \(with Variants\)](#), or [OBJECT](#) statement failed (see [OBJRESULT](#) and [OBJRESULT\\$](#) to obtain an extended error code).

Error 6 - Overflow

[Top](#) [Previous](#) [Next](#)

Overflow (%ERR_OVERFLOW) - This error is not currently supported.

Error 7 - Out of memory

[Top](#) [Previous](#) [Next](#)

Out of memory - (%ERR_OUTOFMEMORY) - Many different situations can cause this message, including [dimensioning](#) too large an [array](#), or running out of virtual memory due to insufficient free disk space for the Windows swap file.

Error 9 - Subscript / Pointer out of range

[Top](#) [Previous](#)
[Next](#)

Subscript / Pointer out of range - (%ERR_SUBSCRIPTPOINTEROUTOFRANGE) - You attempted to use a [subscript](#) smaller than the minimum or larger than the maximum value established when the [array](#) was [dimensioned](#). Attempting to use a null or invalid [pointer](#) may also cause this error. Error 9 will only be generated if you have specified [#DEBUG ERROR ON](#).

Error 11 - Division by zero

[Top](#) [Previous](#) [Next](#)

Division by zero (%ERR_DIVISIONBYZERO) - This error is not currently supported.

Error 24 - Device time-out

[Top](#) [Previous](#) [Next](#)

Device time-out - (%ERR_DEVICETIMEOUT) - The specified time-out value for a [UDP](#) or [TCP](#) communications operation has expired.

Error 51 - Internal error

[Top](#) [Previous](#) [Next](#)

Internal error - (%ERR_INTERNALERROR) - A malfunction occurred within the Classic PowerBASIC run-time system, or the operating system reported an error that Classic PowerBASIC was not expecting (or was unable to decipher). For example, attempting to [KILL](#) (delete) an open file can cause this kind of problem.

If you are unable to identify the cause of the problem, contact the Classic PowerBASIC [Technical Support](#) group with information about your program.

Error 52 - Bad file name or number

[Top](#) [Previous](#) [Next](#)

Bad file name or number - (%ERR_BADFILENAMEORNUMBER) - The file number you gave in a file statement does not match the file number given in an [OPEN](#) statement, or the file number may be out of the range of valid file numbers.

Error 53 - File not found

[Top](#) [Previous](#) [Next](#)

File not found - (%ERR_FILENOTFOUND) - The file name specified could not be found on the indicated drive.

Error 54 - Bad file mode

[Top](#) [Previous](#) [Next](#)

Bad file mode - (%ERR_BADFILEMODE) - You attempted a [PUT](#) or a [GET](#) (or [PUT\\$](#) or [GET\\$](#)) on a file opened in [sequential](#) mode.

Error 55 - File is already open

[Top](#) [Previous](#) [Next](#)

File is already open - (%ERR_FILEISOPEN) - You attempted to [OPEN](#) a [file](#) that was already open, or you attempted to [delete](#) an open file.

Error 57 - Device I/O error

[Top](#) [Previous](#) [Next](#)

Device I/O error - (%ERR_DEVICEIOERROR) - A hardware problem occurred when trying to carry out some device-orientated command.

For example, a [COMM](#) connection was lost during a session, or a [TCP/UDP](#) statement failed to be connected, etc. Alternatively, a TCP/UDP port may have been closed unexpectedly or the network refused the connection requested.

If an ERROR 57 occurs with a [TCP OPEN](#) statement under Windows 98 when using a dotted [IP](#) address string (i.e., "202.123.456.1"), then check to ensure that "Client for Microsoft Networks" is installed in the Network applet in Control Panel. Alternatively, manually add a DNS entry in the HOSTS file in the \WINDOWS folder.

For example, add the following line into the HOSTS file, and change the TCP OPEN statement to use the (dummy) domain name instead of the dotted IP address:

```
202.123.456.1  dummyname.com
```

Error 58 - File already exists

[Top](#) [Previous](#) [Next](#)

File already exists - (%ERR_FILEALREADYEXISTS) - The new name argument specified in your [NAME](#) statement already exists.

Error 61 - Disk full

[Top](#) [Previous](#) [Next](#)

Disk full - (%ERR_DISKFULL) - There is not enough free space on the indicated or default disk to carry out a file operation. Create more free disk space and retry your program.

Error 62 - Input past end

[Top](#) [Previous](#) [Next](#)

Input past end - (%ERR_INPUTPASTEND) - You tried to read more data from a file than it had to read. Use the [EOF](#) (end of file) function to avoid this problem. Trying to read from a [sequential file](#) opened for output or append can also cause this kind of error.

Error 63 - Bad record number

[Top](#) [Previous](#) [Next](#)

Bad record number - (%ERR_BADRECORDNUMBER) - A number less than the BASE option specified in the [OPEN](#) statement or a number larger than $2^{63}-1$ was specified as the record argument to a random file [PUT](#) or a [GET](#) statement.

Error 64 - Bad file name

[Top](#) [Previous](#) [Next](#)

Bad file name - (%ERR_BADFILENAME) - The file name specified in a [KILL](#) or [NAME](#) statement contains invalid characters.

Error 67 - Too many files

[Top](#) [Previous](#) [Next](#)

Too many files - (%ERR_TOOMANYFILES) - This error can be caused either by trying to create too many files in a drive's root directory, or by an invalid file name that affects the performance of the Create File system call.

Error 68 - Device unavailable

[Top](#) [Previous](#) [Next](#)

Device unavailable - (%ERR_DEVICEUNAVAILABLE) - You tried to OPEN or ATTACH a device or to a device or graphic without that device present or installed.

For example, opening COM1 on a system without a serial adapter or modem, or attempting to use [TCP/IP](#) or [UDP/IP](#) on a machine without [Winsock](#) 2.0 (or better) installed. Also, trying to attach to a [graphic](#) or [printer](#) that is not available will cause this error.

Error 69 - COMM error

[Top](#) [Previous](#) [Next](#)

COMM error - (%ERR_COMMERROR) - A [communications](#) error occurred. For example, a framing error may have occurred.

Error 70 - Permission denied

[Top](#) [Previous](#) [Next](#)

Permission denied - (%ERR_PERMISSIONDENIED) - You tried to write to a write-protected disk. This error can also be generated as a result of network permission errors, such as accessing a locked file, or a locked record. It can also occur when attempting to open a subdirectory as a file.

Error 71 - Disk not ready

[Top](#) [Previous](#) [Next](#)

Disk not ready - (%ERR_DISKNOTREADY) - The door of a floppy disk drive is open, or there is no disk in the indicated drive.

Error 72 - Disk media error

[Top](#) [Previous](#) [Next](#)

Disk media error - (%ERR_DISKMEDIAERROR) - The controller board of a floppy or hard disk indicates a hard media error in one or more sectors.

Error 74 - Rename across disks

[Top](#) [Previous](#) [Next](#)

Rename across disks - (%ERR_RENAMEACROSSDISKS) - You cannot [rename](#) a directory across disk drives or partitions.

Error 75 - Path/file access error

[Top](#) [Previous](#) [Next](#)

Path/file access error - (%ERR_PATHFILEACCESSERROR) - During a command capable of specifying a path name ([OPEN](#), [NAME](#), or [MKDIR](#), for example), a path was used inappropriately. For example, attempting to [delete a directory](#) that is in-use.

Error 76 - Path not found

[Top](#) [Previous](#) [Next](#)

Path not found - (%ERR_PATHNOTFOUND) - The path you specified during a [CHDIR](#), [MKDIR](#), [OPEN](#), etc, cannot be found.

Error 99 - Object error

[Top](#) [Previous](#) [Next](#)

Object error - (%ERR_OBJECTERROR) - A run-time error occurred involving an [object](#).

Error 241 - Global memory corrupt

[Top](#) [Previous](#) [Next](#)

Global memory corrupt - (%ERR_GLOBALMEMORYCORRUPT) - Classic PowerBASIC detected a global memory corruption.

Typical causes include misuse of pointers, accessing an [array](#) beyond its boundary, or bad [Inline Assembly](#) code. The cause of the problem may actually be in a seemingly unrelated portion of the program, and/or in a DLL or module used by the program.

Error 241 was formerly deemed "Far heap corrupt" (%ERR_FARHEAPCORRUPT). While this [equate](#) remains supported for a short period, source code should be updated to maintain compatibility with future versions of Classic PowerBASIC.

Error 242 - String space corrupt

[Top](#) [Previous](#) [Next](#)

String space corrupt - (%ERR_STRINGSPACECORRUPT) - Classic PowerBASIC detected a memory or string space corruption. Typical causes include misuse of pointers, accessing an [array](#) beyond its boundary, or bad [Inline Assembly](#) code. The cause of the problem may actually be located in a seemingly unrelated portion of the program, and/or in a DLL or module used by the program.

Welcome to Classic PowerBASIC's powerful and improved Dynamic Dialog Tools. DDT allows a BASIC programmer to easily create a Graphical User Interface (GUI) for an application using simple BASIC statements. With DDT, there's no need to stress over learning how to effectively use GUI design software that contains icons you don't understand and also hundreds of cryptic "property" settings. With DDT, your Classic PowerBASIC application or [DLL](#) can create user interface dialogs "on the fly".

For programmers who are familiar with DDT, you will find that Classic PowerBASIC has expanded the DDT implementation even further in this version of Classic PowerBASIC, with advanced features such as User Data storage and accelerator tables.

This chapter describes Classic PowerBASIC's Dynamic Dialog Tools and how to easily create full-featured Graphical User Interfaces in your code.

See Also

[Creating a Dialog](#)

[Adding Controls to the Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Control Styles](#)

[Callbacks](#)

[Dialog Styles](#)

[Menus](#)

[Menu Walkthrough](#)

[More on the Menu](#)

[Menu State](#)

[Menu Example](#)

Creating a Dialog

[Top](#) [Previous](#) [Next](#)

In this example, we will create a simple dialog that asks the user to enter his/her name, providing a text box for input, plus both "OK" and "Cancel" buttons. To create the dialog, first we use the [DIALOG NEW](#) statement:

```
LOCAL hParent AS DWORD
LOCAL hDlg AS DWORD
[statements]
DIALOG NEW hParent, Caption$,,, 160, 50, Style&, exStyle& TO hDlg
```

hParent refers to the parent window handle. If this value is 0 (or %HWND_DESKTOP), the dialog has no parent window, and may be referred to as a "top-level" window. However, if the dialog has a parent window and the dialog is a MODAL dialog, Windows will automatically disable the parent window while the DDT dialog is displayed.

Caption\$ is the text displayed in the caption of the dialog. This may be the name of your program, or it can be used to convey other information to the user.

The next two parameters for the location on the screen are omitted (this causes the dialog to be centered on the screen), and the size is set to 160 dialog units wide by 50 dialog units tall. *Style&* specifies how the dialog is drawn on the screen (with a caption, without a caption, etc). *exStyle&* specifies an extended style attributes for drawing the dialog. For information on the range of possible dialog styles, please see the [DIALOG NEW](#) statement.

Once the dialog has been created, the handle for it is placed in the *hDlg* variable. *hDlg* may be a Long-integer or Double-word variable (i.e., *hDlg&* or *hDlg???*), but a **Double-word variable is recommended**. This handle is used by Windows (and your program) code to identify the dialog. Windows gives each dialog a unique handle value at run-time; no two windows, dialogs, or controls can have the same handle value. This means that the actual handle value will be different every time the dialog is created.

Note that the height and width values determine the client size of the dialog, if the dialog style explicitly includes the %WS_CAPTION style. Otherwise, they are interpreted as the outer dimensions of the complete dialog.

Note: The location and size of a dialog are specified in Dialog Units or, optionally, Pixels.

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Adding Controls to the Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Control Styles](#)

[Callbacks](#)

[Dialog Styles](#)

[Menus](#)

Adding Controls to the Dialog

[Top](#) [Previous](#) [Next](#)

Once the dialog has been created, we can add controls to it. For our example, we will add a text box to let the user type in their name, and also add two BUTTON controls ("OK" and "Cancel"):

```
CONTROL ADD TEXTBOX, hDlg, IdText&  
, "", 14, 21, 134, 12, Style&, exStyle&  
CONTROL ADD BUTTON, hDlg, 1, "&OK", 44, 38, 40, 14, %BS_DEFAULT or %WS_TABSTOP CALL  
Ok  
CONTROL ADD BUTTON, hDlg, 2, "&Cancel", 90, 38, 40, 14 CALL Cancel
```

hDlg refers to the handle of the dialog you're adding the control to, as returned by the [DIALOG NEW](#) statement.

The next parameter *IdText&*, 1, and 2 in the example lines above) is the unique numeric identifier (ID) for the control. Whereas dialog handles are determined by Windows at run-time, controls use ID values that are specified by the programmer. By knowing the dialog handle and a control ID, we can identify and interact programmatically with any control on a DDT dialog using any of the control-related DDT statements.

In general, ID values should be kept within in the range 100 to 65535. It should also be noted that some values below 100 are reserved by Windows for special purposes. For example, the special ID value 1 (%IDOK) is usually assigned to a Button control that is to be activated when the ENTER key is pressed (this would typically be the "OK" button on a dialog). Similarly, the special ID value of 2 (%IDCANCEL) is usually assigned to a Button control that is to be activated when the ESCAPE key is pressed (typically this would be the "Cancel" button).

In general, two controls on a given dialog should not use the same ID value, as it prevents them from being identified uniquely. However, it is common to assign the special value -1& to plain [Label](#) (static) controls that will not have their content, style, or color changed at run-time.

It is always a good idea to plan the values of control identifiers carefully. For example, a set of related Option (radio) controls should use ID values that are ordered sequentially, as this makes it very easy to manipulate them as a group with the [CONTROL SET OPTION](#) statement, etc. Another common scheme is keep all the ID numbers for the controls in a specific range. For example, the first dialog in a program might use controls whose ID values are in the range 100 to 199, the second dialog might use the range 200 to 299, etc.

The identifier parameter is followed by the caption text for the control. The ampersand symbols "&" within the caption text fields is surprisingly helpful - the letter that follows the symbol specifies a command accelerator (hot-key). At run-time, the accelerator character is drawn underscored: OK and Cancel. In this case, the underscored character informs the user that pressing the ALT+O keys has the same effect as using the mouse to click the "OK" button. Similarly, the ALT+C combination will trigger the "Cancel" button.

Coordinates used in the CONTROL ADD statement are specified in the same terms (dialog units or pixels) as the parent dialog. The final Style& (primary style) and exStyle&

(extended style) parameters tell Windows how to draw the control, and how the control should behave. These parameters are optional, and if omitted, receive default styles according to the type of control.

Each type of control has its own unique set of style options.

Most of the equates have been predefined in the DDT.INC and [WIN32API.INC](#) files supplied with Classic PowerBASIC.

It should be noted that explicit (custom) style values replace the default values for the control. That is, custom styles are not additional to the default style values - your code must specify all necessary style parameters. This also applies to the extended styles parameter - if your code specifies a custom primary style, the default extended style will no longer be in effect either. In this case, an explicit extended style may also need to be added to the CONTROL ADD statement if an explicit primary style is specified.

The CONTROL ADD statement for the "OK" button includes the keyword [CALL](#). This tells Windows to call the "OK" function each time the "OK" button is pressed. The "OK" function is simply a [Callback](#) Function that contains the code you want to execute when the button is pressed (or when some other control-related event occurs).

In this example, we want to assign the text from the text box control to a [global string](#), and then close the dialog box. However, we first must check that our code is executed only in response to a "click" event - we would not want our dialog to end if some other notification message was sent to the callback! We do this by testing the values of the message parameters held in the [CB.HNDL](#), [CB.MSG](#), and [CB.CTLMSG](#) system variables:

```
CALLBACK FUNCTION Ok() AS LONG
  IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
    CONTROL GET TEXT CB.HNDL, %IDTEXT TO gsUserName
    DIALOG END CB.HNDL, 1 ' Return 1
    FUNCTION = 1
  END IF
END FUNCTION
```

Similarly, we provide a Callback Function for the "Cancel" button, which will close the dialog box, ignoring any text entered into the text box:

```
CALLBACK FUNCTION Cancel() AS LONG
  IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
    DIALOG END CB.HNDL, 0 ' Return 0
    FUNCTION = 1
  END IF
END FUNCTION
```

Once the dialog has been created and the controls added, we are ready to display the dialog on the screen. In this example, we will create it as a Modal dialog. That means that when the [DIALOG SHOW MODAL](#) statement is executed, the execution of this portion of our program will block (halt) until the dialog is closed: (see [Modal vs. Modeless](#) below for more information on modal and modeless dialogs)

```
LOCAL lResult AS LONG
...
```

```
DIALOG SHOW MODAL hDlg TO lResult
```

During the time that the "main" part of our code is blocked by the modal dialog, DDT may call the code in the Callback Functions in response to user interaction, etc. If no events occur, our code is not executed at all, and therefore uses no CPU time. In this example, the dialog only closes when the user eventually clicks the OK or the Cancel button (or presses the ENTER or ESCAPE keys).

Once the dialog is closed, the *lResult* variable will contain the value set using the [DIALOG END](#) statement, and execution of the statements following the DIALOG SHOW statement will resume. In our example, we use a return value of one (1) to indicate that the user clicked the OK button, and a return value of 0 to indicate the user clicked the Cancel button.

The complete example code can be found in the HELLODDT.BAS file in the \PB\SAMPLES\DDT\HELLODDT folder:

```
#COMPILE EXE
#INCLUDE "DDT.INC"

%IDOK = 1
%IDCANCEL = 2
%IDTEXT = 100
%BS_DEFAULT = 1

' Global variable to receive the user name
GLOBAL gsUserName AS STRING

CALLBACK FUNCTION OkButton()
    IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
        CONTROL GET TEXT CB.HNDL, %IDTEXT TO gsUserName
        DIALOG END CB.HNDL, 1
        FUNCTION = 1
    END IF
END FUNCTION

CALLBACK FUNCTION CancelButton()
    IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
        DIALOG END CB.HNDL, 0
        FUNCTION = 1
    END IF
END FUNCTION

FUNCTION PBMAIN() AS LONG

    LOCAL hDlg AS DWORD
    LOCAL lResult AS LONG

    ' ** Create a new dialog template
    DIALOG NEW 0, "What is your name?", , , 160, 50, 0, 0 TO hDlg

    ' ** Add controls to it
    CONTROL ADD TEXTBOX, hDlg, %IDTEXT, "", 14, 12, 134, 12
    CONTROL ADD BUTTON, hDlg, %IDOK, "OK", 34, 32, 40, 14, %BS_DEFAULT OR %WS_TABSTOP
    CALL OkButton
    CONTROL ADD BUTTON, hDlg, %IDCANCEL, "Cancel", 84, 32, 40, 14 CALL CancelButton

    ' ** Display the dialog
    DIALOG SHOW MODAL hDlg TO lResult
```



```
' ** Check the dialog return result
IF lResult THEN
    MSGBOX "Hello " & gsUserName, &H00002000& ' = %MB_TASKMODAL
END IF

END FUNCTION
```

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Creating a Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Control Styles](#)

[Callbacks](#)

[Dialog Styles](#)

[Menus](#)

Modal vs. Modeless

To support the different ways that applications use dialog boxes, Classic PowerBASIC provides two types of dialog box: modal and modeless.

A modal dialog box requires the user to supply information, or cancel the dialog box, before allowing the application to continue. Applications use modal dialog boxes in conjunction with commands that require additional information before they can proceed.

A modeless dialog box allows the user to supply information and return to the previous task without closing the dialog box. Modal dialog boxes are simpler to manage than modeless dialogs because they are displayed, perform their task, and are destroyed by calling a single [DIALOG SHOW MODAL](#) statement.

In the above example, we display the dialog as modal. The [DIALOG SHOW MODAL](#) statement displays the dialog and waits until your code calls [DIALOG END](#) (or if there is a Close box in the caption, the dialog will end when the Close box is clicked). When Windows displays a modal dialog box, it disables the parent window to keep the user focused on the dialog. When the dialog box is closed, the parent window is automatically re-enabled.

By comparison, a modeless dialog box does not cause your code to stop and wait while the dialog is displayed. An example of a modeless dialog box is the "Cancel" dialog displayed by many programs that print long documents on the printer. The application code sits in a loop sending data to the printer. The "Cancel" dialog allows the user to cancel printing at any time. The following is a simplistic example of this process:

```
DIALOG SHOW MODELESS hDlg TO lResult&
DO
  DIALOG DOEVENTS
  Done& = PrintNextLineFunction()
LOOP UNTIL lResult& = %IDCANCEL OR Done& = %TRUE
```

The [DIALOG DOEVENTS](#) statement is necessary so that Windows can process messages for your modeless dialog. Without it, events such as clicking on the "Cancel" button or redrawing the dialog would not be processed. This loop is known as a Message Pump.

A modeless dialog must always have a message pump running while the dialog is running. Without a message pump, a modeless dialog will not be able to receive messages to redraw itself, etc.

Because of this consideration, applications should be written in such a way as to ensure that the message pump is able to run. The following example is of a modeless dialog message pump that relies on the fact that when the dialog is destroyed, [DIALOG GET SIZE](#) will return 0.

```
DIALOG SHOW MODELESS hDlg TO lResult&
DO
  DIALOG DOEVENTS
  DIALOG GET SIZE hDlg TO x&, y&
LOOP UNTIL ISFALSE (x& * y&)
```

This works fine for applications that have a single modeless dialog showing at any given moment, but this is not always practical. For example, consider an application that uses a tabbed dialog. Typically, this is constructed around a single dialog containing a "Tab Control", plus an additional set of modeless dialogs, each of which would form a "page" of the tabbed dialog.

In this case, we need to reconstruct our message pump so that it terminates only when all of the modeless dialogs have been destroyed. If the main dialog is modal, the application design would become quite complex - the modeless dialogs and the message pump would need to be launched from within the main dialog's Callback Function. Such an approach is technically feasible, but unnecessary. By changing the main dialog from modal to modeless, the whole design can be simplified to use a single message pump.

```
DIALOG SHOW MODELESS hMainDlg TO lResult&
DIALOG SHOW MODELESS hPage1
DIALOG SHOW MODELESS hPage2
' more code here
DO
  DIALOG DOEVENTS TO Count&
LOOP UNTIL ISFALSE Count&
```

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Creating a Dialog](#)

[Adding Controls to the Dialog](#)

[Controls](#)

[Control Styles](#)

[Callbacks](#)

[Dialog Styles](#)

[Menus](#)

A control is a special Window that provides a method for interacting with the user. [Buttons](#), [Combo boxes](#), [List boxes](#), and [Text boxes](#) are all examples of controls. Whenever the user interacts with a control (clicks a button or types into a text box), an event occurs causing Windows to send a message to your application. Your application processes these messages in special functions called [Callback](#) Functions.

When you add a control to a [dialog](#), it is important that each control has a unique numeric identifier. This identifier helps your application to know which control is sending an event. For example, if your program has two buttons in it, the control ID allows you distinguish between them.

As each control is created, Windows assigns a unique window handle to identify the control. Because your program does not assign these handle values, your code cannot directly use them to identify individual controls. Further, each time a control is destroyed and recreated, a new unique handle value is assigned, further complicating the task. The control ID overcomes these problems, as the programmer determines the ID for each control.

Controls are added to your dialog with the CONTROL ADD statement. Make sure that each control you create has a unique numeric identifier, so that you (and Windows) can tell it apart from other controls on the dialog.

Given the ID of a control, DDT provides the [CONTROL HANDLE](#) statement to retrieve the window handle value of the control. If a given ID is duplicated in a dialog, CONTROL HANDLE is only able to identify the first control that matches the ID, and the remaining controls will essentially be ignored. Control ID's can often be duplicated for [Label](#) (static) text controls, provided these controls (and their contents, [color](#), or [styles](#)) are not going to be modified at run-time. If such a Label control is to be modified, its control ID must be unique.

Classic PowerBASIC provides a comprehensive set of statements and functions for dealing with controls. The following is a small sample of these statements and functions with a brief description of the purpose of each:

Function Description

#MESSAGES	Specify which messages should be sent to a Control Callback Function
CB.CTL	Return the ID of the control sending a message to your Callback Function. (Only valid inside a Callback Function).
CB.CTLMSG	Return the notification ID of the control sending a message to your Callback Function. (Only valid inside a Callback Function).
CB.HNDL	Returns the dialog handle sending a message to your Callback Function. (Only valid inside a Callback Function).

<u>CB.LPARAM</u>	Returns the lParam& value sent to your Callback Function. (Only valid inside a Callback Function).
<u>CB.MSG</u>	Returns the wParam& value sent to your Callback Function. (Only valid inside a Callback Function).
<u>CB.NMCODE</u>	Returns the specific notification message describing the event which occurred. (Only valid inside a Callback Function).
<u>CB.NMHDR</u>	Returns the address (a <u>pointer</u>) to the NMHDR UDT for a notification message sent to your Callback Function. (Only valid inside a Callback Function).
<u>CB.NMHDR\$</u>	Returns the contents of the NMHDR UDT as a dynamic string. (Only valid inside a Callback Function).
<u>CB.NMHWND</u>	Returns the handle of the control which sent this message to your Callback Function. (Only valid inside a Callback Function).
<u>CB.NMID</u>	Returns the ID number assigned to this control which sent this message to your Callback Function. (Only valid inside a Callback Function).
<u>CB.WPARAM</u>	Returns the wParam& value sent to your Callback Function. (Only valid inside a Callback Function).
<u>CONTROL DISABLE</u>	Create a control in a dialog. Disable a control so that it can no longer send messages to a callback. If the control is a button, it is grayed. If it is a text box, it becomes grayed out and you can no longer edit the text contained within it.
<u>CONTROL ENABLE</u>	Enable a control that was previously disabled. A control must be enabled in order for it to send notifications to a Callback Function.
<u>CONTROL GET SIZE</u>	Get the size of a control.
<u>CONTROL GET LOC</u>	Get the location of a control inside of its parent dialog.
<u>CONTROL GET TEXT</u>	Retrieve the text from a control, such as a Text box or Label, etc.
<u>CONTROL HANDLE</u>	Return a window handle for a given control.
<u>CONTROL KILL</u>	Remove a control from a dialog.
<u>CONTROL SEND</u>	Send a message to a control.
<u>CONTROL SET FOCUS</u>	Set the keyboard focus to a given control. If the control is a button, it receives keyboard focus. If the control is a text box, the caret is placed in the text box to allow the user to edit the text.
<u>CONTROL SET FONT</u>	Select a font to be used for a particular control.
<u>CONTROL SET IMAGE</u>	Change the image on an image button or image control. Also see <u>CONTROL SET IMAGEX</u> .
<u>CONTROL SET SIZE</u>	Change the size of a control.
<u>CONTROL SET LOC</u>	Change the location of a control within its parent dialog.
<u>CONTROL SET TEXT</u>	Place new text into a control. Any existing text in the control is replaced.

[WINDOW GET ID](#)
[WINDOW GET
PARENT](#)

Returns the Control ID for a given control.
Returns the handle of a controls parent.

For a more comprehensive list of DDT statements and functions, See the [Command Summary for DDT](#).

See Also

[Dynamic Dialog Tools \(DDT\)](#)
[Creating a Dialog](#)
[Adding Controls to the Dialog](#)
[Modal vs. Modeless](#)
[Control Styles](#)
[Callbacks](#)
[Dialog Styles](#)
[Menus](#)

When creating child controls for your dialogs, you are free to use almost any control style permitted by Windows. These styles mostly start with the %WS_ prefix (Window Style), and are included in the [WIN32API.INC](#) file included in your WINAPI directory.

If the style parameter in your CONTROL ADD statements is set to 0, DDT will set default styles automatically for you. The default styles will depend on the type of control you are adding to your dialog. For example, a button will be given the %WS_TABSTOP style.

Note that DDT always gives your controls certain styles, such as %WS_CHILD and %WS_VISIBLE, regardless of the styles you specify. When setting your style parameter, you can safely ignore these two styles and concentrate on the more important styles that are required. This has the advantage of reducing the clutter of your code. The exception is custom controls - in this case you must explicitly specify all required styles.

The "tab-order" of controls (also known as the "z-order") is determined by the order that DDT controls are created at run-time. That is, the first control added to a dialog is the first control in the z-order, the second control added is second, and so forth. When a dialog is initially displayed, keyboard focus is automatically given to the first control in the z-order that has the %WS_TABSTOP style. Each time the TAB key is subsequently pressed, the keyboard focus moves to the next control in the tab-order. To ensure all controls in a dialog can be selected using the TAB key, each control in the dialog should include the %WS_TABSTOP style. The z-order also determines the order that controls are drawn on a dialog, to help ensure that control that overlap one another can be drawn in a predictable manner.

Controls that are disabled (because either they have the %WS_DISABLED style or they have been dynamically disabled with [CONTROL DISABLE](#)) are skipped over.

Most DDT controls are created with the %WS_TABSTOP style by default. However, you should explicitly include the %WS_TABSTOP style in the control style parameter, if your DDT code creates controls with custom (non-default) styles. If you do not include this style, these control(s) may not be able to receive keyboard focus.

The following table lists the default DDT styles for many of the standard controls:

Control type	Default DDT Styles	Hex Value
BUTTON	%WS_TABSTOP	50010000
CHECK3STATE	%WS_TABSTOP, {%BS_AUTO3STATE}	50010006
CHECKBOX	%WS_TABSTOP, {%BS_AUTOCHECKBOX}	50010003
COMBOBOX	%WS_TABSTOP, %CBS_SORT, %CBS_DROPDOWN, {%CBS_HASSTRINGS}	50010302
FRAME	%BS_LEFT, {%BS_TOP, %BS_GROUPBOX}	50000507
GRAPHIC	%WS_CHILD, %WS_VISIBLE, %SS_OWNERDRAW	5001000D
IMAGE	either {%SS_ICON} or {%SS_BITMAP}	50000003 5000000E

IMAGEX	either {%SS_ICON} or {%SS_BITMAP}	50000003 5000000E
IMGBUTTON	either %WS_TABSTOP, {%BS_ICON} or %WS_TABSTOP, {%BS_BITMAP}	50010040 50010080
IMGBUTTONX	either %WS_TABSTOP, {%BS_ICON} or %WS_TABSTOP, {%BS_BITMAP}	50010040 50010080
LABEL	%SS_LEFT	50000000
LINE	%SS_ETCHEDFRAME	50000012
LISTBOX	%WS_TABSTOP, %LBS_SORT, %LBS_NOTIFY, %WS_VSCROLL	50210003
LISTVIEW	%WS_TABSTOP, %LVS_REPORT, %LVS_SHOWSELALWAYS	50000009
OPTION	%WS_TABSTOP, {%BS_AUTORADIOBUTTON}	50010009
PROGRESSBAR	%WS_BORDER	50800000
SCROLLBAR	either {%SBS_HORZ} or {%SBS_VERT}	50000000 50000001
STATUSBAR	%CCS_BOTTOM	50000003
TAB	%WS_CHILD, %WS_TABSTOP	54010000
TEXTBOX	%WS_TABSTOP, %WS_BORDER, %ES_AUTOHSCROLL, %ES_LEFT	50810080
TOOLBAR	%WS_CHILD, %WS_VISIBLE, %WS_BORDER, %CCS_TOP, and %TB_FLAT	50808801
TREEVIEW	%WS_TABSTOP, %TVS_HASBUTTONS, %TVS_LINESATROOT, %TVS_HASLINES, and %TVS_SHOWSELALWAYS	50010027
"custom control"	No default style (%WS_CHILD and %WS_VISIBLE not used)** <u> </u>	0

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Creating a Dialog](#)

[Adding Controls to the Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Callbacks](#)

[Dialog Styles](#)

[Menus](#)

A callback is a Function called by Windows when an event occurs. In the previous [modal dialog example](#), when the OK button is clicked by the user, Windows calls the *OkButton()* function. Classic PowerBASIC's [Dynamic Dialog Tools](#) allows you to create a single callback to handle all events for the [dialog](#), or you can create individual Callback Functions for each control in your dialog. You can even use a combination of the two methods.

Control Callback

If you've used Visual Basic, you'll be familiar with the concept of a Control Callback even though it's not called by that name. A Control Callback is a [function](#) that is called when a %WM_COMMAND or %WM_NOTIFY event is generated for a particular control. In the [earlier example](#), we arranged it so the *OkButton()* function was called when the OK button was clicked. Further, when the Cancel button was clicked, the *CancelButton()* function was called. A Control Callback function is enabled when you execute a CONTROL ADD statement using the CALL CtlProc option at the end.

```
CONTROL ADD BUTTON, hDlg, %IDOK, "OK", 34, 32, 40, 14, %BS_DEFAULT OR %WS_TABSTOP  
CALL OkButton  
CONTROL ADD BUTTON, hDlg, %IDCANCEL, "Cancel", 84, 32, 40, 14 CALL CancelButton
```

Some controls, like [text boxes](#), [list boxes](#), and [combo boxes](#), can generate more than one type of event. In VB, each separate event on each control is handled by a new function. For example, if your VB form includes a list box, it may include a Callback Function such as *List1_Change()* that is called whenever the current selected item changes. In Classic PowerBASIC, only a single Callback Function is needed for each control. When an event occurs, the Callback Function just chooses which events to handle, and which events to ignore. If your Classic PowerBASIC callback wanted to process the Change event for a list box, your code would look like this:

```
CALLBACK FUNCTION List1() AS LONG  
  IF CB.MSG = %WM_COMMAND THEN  
    IF CB.CTLMSG = %LBN_SELCHANGE THEN  
      [your code here]  
      FUNCTION = 1  
    END IF  
  END IF  
END IF  
  
END FUNCTION
```

You can use a combination of the [CB.MSG](#) and [CB.CTLMSG](#) functions to decide exactly which event has occurred. Generally speaking, in a Control Callback, CB.MSG will contain either %WM_COMMAND or %WM_NOTIFY. The CB.CTLMSG will return the specific message is either of those two categories. In this example situation, the control notification [%LBN_SELCHANGE](#) is sent to the callback for the list box whenever the item in the list box changes (the user clicks on the new item or uses the keyboard to select a new item).

All of the control and dialog message equates are located in the DDT.INC file. This

file is simply a subset of the much larger [WIN32API.INC](#) file and is provided only for convenience. Therefore, the use of these two files is mutually exclusive.

If your code processes a message, it should return [TRUE](#) (any non-zero value) by setting `FUNCTION = number` within the Control Callback. This advises that there is no need to process that message further. If you return the value [FALSE](#) (zero), the message is passed on to your Dialog Callback, if you have one. If the message is still unhandled by your Dialog Callback, the [DDT](#) dialog engine itself will handle the message on your behalf. If your code processes a `%WM_NOTIFY` message, the return value is generally ignored. Because of the nature of `%WM_NOTIFY` messages, they are always directed to both Control callbacks and Dialog callbacks to use as needed.

Prior to version 9.0 of Classic PowerBASIC for Windows, Control Callback Functions received only `%WM_COMMAND` messages. Beginning with PB 9.0, `%WM_NOTIFY` messages are sent as well. There are many situations where these added messages will prove to be very important. If your existing callback functions are written with complete error checking (ensuring that `CB.MSG = %WM_COMMAND`), this minor addition will cause no problems. It just presents additional information which can be acted upon, or just ignored. However, if callbacks were written without complete error checking, some ambiguity is possible. In this case, you should either update your Control Callback code, or suppress `%WM_NOTIFY` messages with a [#MESSAGES COMMAND](#) metastatement.

When a Control Callback receives a click notification for a control, the callback will receive a `%WM_COMMAND` message in the `CB.MSG` variable. A common mistake made by programmers is to fail to test both `CB.MSG` and `CB.CTLMSG` parameters before responding to the message. If the message is truly generated from a click event, `CB.CTLMSG` will contain `%BN_CLICKED`. This simple test ensures that your code responds correctly to notification messages.

```
CALLBACK FUNCTION OkButton() AS LONG
  IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
    '...Process the click event here
    FUNCTION = 1
  END IF
END FUNCTION
```

It pays to be sure you are responding to the correct message in your callback. Subtle bugs can occur if you aren't very careful to notice and recognize unanticipated messages.

It should also be noted that there are ranges of notification messages that individual controls can send to the Control Callback or Dialog Callback. However, many of these messages are suppressed unless the controls have been initially assigned a "notify" style. For controls that are members of the Button class ([CHECKBOX](#), [OPTION](#), [FRAME](#), etc.), this is the `%BS_NOTIFY` style. Please refer to the statements for additional information on notification styles for other control types.

Dialog callback

If you review the example code in most Windows programming books (particularly the Windows 32-bit SDK), you will see that most of the examples create a single callback for the entire dialog. Each time the user presses a button, a message is sent to this Callback Function. Within this Callback Function, there is often a large [SELECT CASE](#) or [IF/ELSEIF/THEN](#) structure, designed to pick out the incoming event messages and then process the selected messages.

C programmers are usually quite familiar with this concept, and often resort to using "Message Cracker" functions to separate their event handling code into a set of independent functions. On the other hand, Classic PowerBASIC's DDT takes much of this drudgery away. By permitting separate callbacks for each CONTROL ADD statement, you become free to enclose your event handling code in separate functions, just like a C programmer may do, but without the confusing macros C programmers are often forced to use.

DDT gives the programmer the choice of either using a single callback to handle all dialog and control events, or writing a callback for each (or any) specific control. If you intentionally omit a callback for a particular control, the programmer has the choice of handling messages for that control within the dialog Callback Function, or ignoring them altogether.

In addition to handling control messages within the dialog callback, this Callback Function also provides a way to handle events that concern the actual dialog box itself. For example, handling a %WM_PAINT message, or notification that the dialog was minimized, etc.

A Dialog Callback function is enabled when you execute a DIALOG SHOW statement using the CALL DlgProc option.

```
DIALOG SHOW MODELESS hDlg CALL DlgProc TO lResult&
```

or:

```
DIALOG SHOW MODAL hDlg CALL DlgProc TO lResult&
```

These two lines of code specify that dialog related event messages should be directed to the Callback Function *DlgProc()*. If we rewrote the earlier DDT example to use a single Dialog Callback instead of individual Control Callback Functions, the function might look something like this:

```
CALLBACK FUNCTION DlgProc()  
  SELECT CASE CB.MSG  
    CASE %WM_COMMAND  
      IF CB.CTLMSG = %BN_CLICKED THEN  
        IF CB.CTL = %IDOK THEN  
          DIALOG END CB.HNDL, 1  
          FUNCTION = 1  
        ELSEIF CB.CTL = %IDCANCEL THEN  
          DIALOG END CB.HNDL, 0  
          FUNCTION = 1  
        END IF  
      END IF
```

```
END IF
END SELECT
END FUNCTION
```

To complete this stage of modifications, you would also remove the "CALL OkButton" and "CALL CancelButton" parameters from the CONTROL ADD lines. Once changed, this modified code produces the identical behavior of the original example with only a single Callback Function.

This simple example only scrapes the surface of what can be achieved in a Dialog Callback Function. For example, by intercepting a %WM_ERASEBKGND message, you could draw onto the dialog client area, producing colorful dialogs with ease.

Callback Return Values

Callback functions always return a long integer result. The primary purpose of this return value is to tell the Classic PowerBASIC DDT engine and the Windows operating system whether your Callback Function has processed this particular message. If you return the value TRUE (any non-zero value), you are asserting that the message was processed and no further handling is needed. If you return the value FALSE (zero), the Classic PowerBASIC DDT engine will manage the message for you, using the default message procedures in Windows. If you do not specify a return value in the function, Classic PowerBASIC chooses the value FALSE (zero) for you.

The term "process a message" may have many meanings. If it's a simple notification of a change in focus or style, which has no impact on your program, you may decide to consider it processed, yet do nothing. In other cases, your reaction could be quite complex and involved. As the programmer, that's your decision to make. But, regardless of your reaction, you should consider a message "processed" (returning a true value) whenever no further handling of the message (by DDT or Windows) is needed.

In some cases, especially when dealing with Common Controls and custom controls, you may be required to return a second result value through a special Windows data area named DWL_MSGRESULT. When you complete a Callback Function, Classic PowerBASIC automatically copies any non-zero return value to DWL_MSGRESULT, if you haven't done so already. Therefore, it's generally safe to ignore this requirement in your code.

In most cases, when you process a message, you'll return a generic value for TRUE, such as: FUNCTION = 1. However, some messages require that you return a special value for TRUE, such as a graphical brush handle. As long as the value is non-zero, you can return it in the normal manner (with FUNCTION=n), since any non-zero value automatically implies that the message was processed.

That said, there are a few unique messages which may require special handling. Luckily, they're rare, but some just "break all the rules" listed above. For example, you might find one which requires a zero result, even when you have processed the message. You may find another which requires the return value be different from DWL_MSGRESULT. For these very special cases, you can simply specify two return values:

```
FUNCTION = 1, BrushHandle&
```

In this form, the first numeric expression specifies the value to be returned from the Callback Function. The second numeric expression tells the value to be assigned to DWL_MSGRESULT. When you use this double parameter assignment, the results are absolute. Classic PowerBASIC assumes you have processed the message, regardless of the values given. Classic PowerBASIC makes no other assumptions of any kind about these values. A double parameter function assignment is only allowed in a Callback Function.

Previous versions of Classic PowerBASIC did not offer a double parameter form of function return. This caused some difficulty with a few Windows messages which required a special return value of zero. If you return a value of zero (0) with the single parameter form, it implies the message was not processed at all by the Callback. This issue is totally circumvented by the double parameter form.

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Creating a Dialog](#)

[Adding Controls to the Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Control Styles](#)

[Dialog Styles](#)

[Menus](#)

Dialog Styles

Like [control styles](#), DDT provides a default style for a dialog window, if the [DIALOG NEW](#) statement does not specify a specific style parameters.

The default style comprises the combination of %WS_POPUP, %WS_CAPTION, %DS_SETFONT, %DS_NOFAILCREATE, %DS_MODALFRAME, and %DS_3DLOOK. These equates are equivalent to a style of &H080C00D4. The extended style default is zero.

If you explicitly specify %WS_CAPTION in your DIALOG NEW statement, DDT will interpret the width and height values as client dimensions, rather than as overall dialog dimensions. This can be very useful for the times when you need to build a dialog with particular client dimensions.

You can create dialogs using combinations of the following styles:

Style Equate	Description
%WS_BORDER	Dialog has a thin-line border.
%WS_CAPTION	Dialog has a title bar (includes the %WS_BORDER style).
%WS_HSCROLL	Dialog contains a horizontal scroll bar.
%WS_MAXIMIZE	Dialog is initially maximized.
%WS_MAXIMIZEBOX	Dialog has a Maximize button, but must be used in conjunction with the %WS_SYSMENU style. You cannot combine this style with the %WS_EX_CONTEXTHELP extended style.
%WS_MINIMIZE	Dialog is initially minimized.
%WS_MINIMIZEBOX	Dialog has a Minimize button, but must be used in conjunction with the %WS_SYSMENU style. You cannot combine this style with the %WS_EX_CONTEXTHELP extended style.
%WS_SIZEBOX	Dialog has a resizable border. Equivalent to the %WS_THICKFRAME style.
%WS_SYSMENU	Dialog contains a system-menu on its title bar. Must be used in conjunction with the %WS_CAPTION style.
%WS_THICKFRAME	See %WS_SIZEBOX.
%WS_VSCROLL	Dialog contains a vertical scroll bar.
%DS_3DLOOK	Dialog uses a non-bold font and uses three-dimensional borders around child controls. Not required with applications marked for #OPTION VERSION4 or #OPTION VERSION5 , as Windows provides this style automatically.
%DS_CENTER	Centers the dialog box in the region of the screen that is not obscured by the taskbar and tray (i.e., the work area).
%DS_CENTERMOUSE	Centers the mouse cursor in the dialog.
%DS_CONTEXTHELP	Places a "question mark" button in the title bar of the dialog. If this

	button is clicked, the cursor changes to a pointer with a question mark. If the next click is on a control in the dialog, the control's Callback Function will receive a %WM_HELP message. When a dialog containing this style is created, Windows automatically adds the %WS_EX_CONTEXTHELP extended style. %DS_CONTEXTHELP is mutually exclusive with the %WS_MAXIMIZEBOX and %WS_MINIMIZEBOX styles.
%DS_CONTROL	Dialog operates as a child of another dialog. For example, a modeless dialog is able to operate as a child window of a tab control (although the parent must be the tab control's owner, not the tab control itself). This style permits the TAB key to move from control to control in both the parent and the modeless dialog seamlessly, provided the parent includes the extended style %WS_EX_CONTROLPARENT.
%DS_FIXEDSYS	Dialog uses the %SYSTEM_FIXED_FONT instead of the %SYSTEM_FONT.
%DS_MODALFRAME	Used in combination with %WS_CAPTION and %WS_SYSMENU to produce a dialog with a title bar and system-menu.
%DS_NOFAILCREATE	Dialog is created even if an error occurs during creation. Such an error may occur if a child control cannot be created successfully.
%DS_SETFONT	During dialog creation, the child controls in the dialog will be sent a %WM_SETFONT message in order to receive the handle of the font specified by the dialog.

See Also

[Dynamic Dialog Tools \(DDT\)](#)

[Creating a Dialog](#)

[Adding Controls to the Dialog](#)

[Modal vs. Modeless](#)

[Controls](#)

[Control Styles](#)

[Menus](#)

Just like regular GUI windows and [dialog boxes](#), [DDT](#) dialogs can use menus too. With just a handful of statements, you can create a menu and add or remove items, depending on the context of your application.

A [menu bar](#) is positioned just below the caption bar of a dialog box. From this menu bar, [popup menus](#) (or sub-menus as they are also known) can be displayed, each containing commands. Popup menus may contain even deeper levels of popup menus.

Menus are constructed in a hierarchical manner: the top-most level is positioned on the menu bar, and the lower levels of the menu are the popup portions. The items on the menu bar are always visible, but the popup menus are never visible until a menu bar item is either clicked by the mouse, or activated by a command accelerator (hot-key) which is indicated by an underscored character in the menu item text.

Please note that command accelerators differ slightly from keyboard accelerators. The latter are configured and described in the [ACCEL ATTACH](#) statement topic.

Typically, a popup menu contains a range of associated commands. For example, a FILE popup menu usually contains a range of commands to permit the opening, saving and closing of files, etc.

When the user activates a popup menu item, and a command is selected, a %WM_COMMAND message is sent to the dialog Callback Function to notify the program that a menu item has been selected.

See Also

[Menu Walkthrough](#)

[More on the Menu](#)

[Menu State](#)

[Menu Example](#)

In order to create an example menu for our [DDT](#) dialog, we will need one [Double-word](#) variable to hold the handle of the menu, and one for each of the popup menu levels that our menu will contain. In the following code, we will work towards creating a menu with two items on the menu bar (therefore two popup menus). In all, we will need three 32-bit variables:

```
DIM hMenu AS DWORD
DIM hPopup1 AS DWORD
```

To begin creating our menu, we use the [MENU NEW BAR](#) statement:

```
MENU NEW BAR TO hMenu
```

The value returned in *hMenu* is termed the menu handle. We use this handle to attach each of our popup menus. In order to create these popup menus, we will also need to create a handle:

```
MENU NEW POPUP TO hPopup1
```

Now we will "glue" our new popup menu to the menu bar. We do this using the [MENU ADD POPUP](#) statement, which results in an entry on the menu bar labeled "*File*", complete with a command accelerator:

```
MENU ADD POPUP, hMenu, "&File", hPopup1, %MF_ENABLED
```

The ampersand character in "*&File*" means that pressing ALT+F on the keyboard, in addition to the conventional mouse click, can open the menu. The *hPopup1* parameter instructs the DDT engine to attach the menu to the menu bar (*hMenu*), and it is initially enabled.

Using the handle returned in *hPopup1*, we can begin adding items to the newly created popup menu. For each menu item that is a command (Open, Save, etc), we assign an ID value and specify the state of the item. When the user clicks on a menu item, the dialog [Callback](#) Function receives a %WM_COMMAND message.

In turn, we can then use the [CB.CTL](#) function to obtain this ID value, to determine which menu item the user has selected. The state parameter allows us to specify whether the menu item is initially enabled, grayed (disabled), checked, or unchecked, etc.

Now let's begin to add items to our new popup menu:

```
MENU ADD STRING, hPopup1, "&Open", 201, %MF_ENABLED
MENU ADD STRING, hPopup1, "&Exit", 202, %MF_ENABLED
MENU ADD STRING, hPopup1, "-", 0, 0
```

Here we created two items that form part of our first popup menu. These menu items have the ID values 201 and 202 respectively, and each is initially enabled. The third item is a special type of menu item, called a separator. A separator is a horizontal line within the menu, and can be used to visually separate groups of menu items from each other within the same popup menu.

We recommend using equates for the ID parameters, as they make your code more readable and maintainable. For this example we use hard-coded values simply for

Let's add an additional popup menu to this original popup menu, just to demonstrate how simple it can be to create menus with multiple "layers". First, we will need to create a new popup menu handle:

```
MENU NEW POPUP TO hPopup2
```

Using this new popup menu handle, we attach menu items in exactly the same order as we did as before:

```
MENU ADD STRING, hPopup2, "Option &1", 403, %MF_ENABLED  
MENU ADD STRING, hPopup2, "Option &2", 404, %MF_ENABLED
```

Now comes the tricky part... we must attach this new menu to the previous menu, rather than the menu bar:

```
MENU ADD POPUP, hPopup1, "&More Options", hPopup2, %MF_ENABLED
```

This statement "glues" the second popup menu to the end of the first popup menu. If we changed the *hPopup1* parameter to *hMenu*, the popup menu would appear on the menu bar. Making multiple level menus is that simple!

With our menu created, we then attach the menu to our DDT dialog:

```
MENU ATTACH hMenu, hDlg
```

This code is almost self-explanatory - DDT is instructed to attach our menu structure to the dialog handle contained in *hDlg*. The only thing left now is to show the dialog, complete with a menu.

```
DIALOG SHOW MODAL hDlg, CALL DlgProc TO lResult
```

See Also

[Menus](#)

[Menu Walkthrough](#)

[More on the Menu](#)

[Menu State](#)

[Menu Example](#)

When adding new menu items to a menu, additional parameters may be included in the following statements:

```
MENU ADD POPUP, hMenu, txt$, hPopup, state&[, AT [BYCMD] position&]
MENU ADD STRING, hMenu, txt$, hPopup, state&[, AT [BYCMD] position&] [, CALL
callback]
```

AT *position*& An optional position parameter that allows the programmer to specify an absolute position of the menu item within the popup menu, inserted immediately before the value of *position*&. Omitting this parameter causes the menu item to be appended to the menu at the "current position". Position values are indexed to 1. For example:

```
' Insert a new menu item at position 3 in the popup menu hPopup
position& = 3
MENU ADD STRING, hPopup, "&Print", %id_Print, _
ItemState&, AT position&
```

BYCMD The BYCMD keyword (also applicable to other forms of the MENU statement) changes the interpretation of *position*& to an identifier value, rather than an absolute position value. For example:

```
' Insert the "Print Setup" menu item before the "Print" menu item
Position& = %id_Print
MENU ADD STRING, hPopup, "Print Se&tup", _
%id_PrintSetup, ItemState&, AT BYCMD Position&
```

callback ([MENU ADD STRING](#) only) The Callback parameter provides a mechanism to specify a [Callback](#) Function that is executed, to process %WM_COMMAND messages for the menu item.

See Also

[Menus](#)

[Menu Walkthrough](#)

[Menu State](#)

[Menu Example](#)

The MENU ADD statements provide for an ItemState& parameter. For popup menus, this may be either %MF_ENABLED or %MF_DISABLED. For menu items, the state may be one of %MF_ENABLED, %MF_DISABLED, %MF_CHECKED, %MF_UNCHECKED, %MF_GRAYED, or %MF_SEPARATOR.

These equates are included in the DDT.INC file, which is a subset of the [WIN32API.INC](#) file. A DDT application that requires the use of these equates would normally include a line such as #INCLUDE "DDT.INC". If your application requires the inclusion of the WIN32API.INC declaration file, you should omit the inclusion of DDT.INC.

A [DDT](#) menu requires the parent DDT dialog to contain at least one child control for the menu to operate correctly. This control may be a [BUTTON](#) or [LABEL](#), etc, and the control may be located out of the visible client area of the dialog if necessary.

The Dessert Menu

In addition to creating menus dynamically, DDT provides a rich set of additional menu functions to allow you to manipulate your menus at run-time. The following is a brief summary of these functions:

MENU DELETE	Delete (remove) a menu item from a menu, or a popup menu from a menu bar.
MENU DRAW BAR	Redraw the menu bar for a given menu. This must be used if a menu is changed at run-time, regardless of whether the menu is visible or not.
MENU GET STATE	Obtain the current state of a menu item (%MF_ENABLED, etc). If the menu item is a separator, the returned value will be %MF_SEPARATOR.
MENU GET TEXT	Retrieve the text for a given menu item.
MENU SET STATE	Set the current state of a menu item.
MENU SET TEXT	Change the text of a specific menu item, and can be used to change the command accelerator of the item.

For a more comprehensive list of menu statements and functions, See the [Command Summary for DDT](#).

See Also

[Menus](#)

[Menu Walkthrough](#)

[More on the Menu](#)

[Menu Example](#)

Menu Example

In the following code example, we create a dialog with a menu, outlining the concepts discussed in this chapter. Feel free to use this code as a base for your own DDT projects. This example is also available in your Classic PowerBASIC installation, in the \PBWIN\SAMPLES\DDT\MENU folder.

```
'=====
'
' Simple example of an application that has a menu and
' requires absolutely no API calls!
'
'=====

#COMPILE EXE

%IDOK = 1
%IDCANCEL = 2
%IDTEXT = 100
%BN_CLICKED = 0
%BS_DEFAULT = 1
%MF_ENABLED = 0
%WM_COMMAND = &H111

%ID_OPEN = 401
%ID_EXIT = 402
%ID_OPTION1 = 403
%ID_OPTION2 = 404
%ID_HELP = 405
%ID_ABOUT = 406

'-----

' ** Global variable to receive the user name
GLOBAL gsUserName AS STRING

'-----

CALLBACK FUNCTION OkButton()
  IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
    CONTROL GET TEXT CB.HNDL, %IDTEXT TO gsUserName
    DIALOG END CB.HNDL, 1
    FUNCTION = 1
  END IF
END FUNCTION

'-----

CALLBACK FUNCTION CancelButton()
  IF CB.MSG = %WM_COMMAND AND CB.CTLMSG = %BN_CLICKED THEN
    DIALOG END CB.HNDL, 0
    FUNCTION = 1
  END IF
END FUNCTION

'-----
```

```

CALLBACK FUNCTION DlgProc()
IF CB.MSG = %WM_COMMAND THEN
    IF CB.CTL => %ID_OPEN AND CB.CTL <= %ID_ABOUT THEN
        MSGBOX "WM_COMMAND received from a menu item!", &H00002000& ' = %MB_TASKMODAL
        FUNCTION = 1
    END IF
END IF
END FUNCTION

'-----

FUNCTION PBMAIN () AS LONG
LOCAL hDlg AS DWORD
LOCAL lResult AS LONG
LOCAL hMenu AS DWORD
LOCAL hPopup1 AS DWORD
LOCAL hPopup2 AS DWORD

' ** First create a top-level menu:
MENU NEW BAR TO hMenu

' ** Add a top-level menu item with a popup menu:
MENU NEW POPUP TO hPopup1
MENU ADD POPUP, hMenu, "&File", hPopup1, %MF_ENABLED
MENU ADD STRING, hPopup1, "&Open", %ID_OPEN, %MF_ENABLED
MENU ADD STRING, hPopup1, "&Exit", %ID_EXIT, %MF_ENABLED
MENU ADD STRING, hPopup1, "-", 0, 0

' ** Now we can add another item to the menu that will bring up a sub-menu.
' First we obtain a new popup menu handle to distinguish it from the first
' popup menu:
MENU NEW POPUP TO hPopup2

' ** Now add a new menu item to the first menu.
' This item will bring up the sub-menu when selected:
MENU ADD POPUP, hPopup1, "&More Options", hPopup2, %MF_ENABLED

' ** Now we will define the sub menu:
MENU ADD STRING, hPopup2, "Option &1", %ID_OPTION1, %MF_ENABLED
MENU ADD STRING, hPopup2, "Option &2", %ID_OPTION2, %MF_ENABLED

' ** Finally, we'll add a second top-level menu and popup.
' For this popup, we can reuse the first popup variable:
MENU NEW POPUP TO hPopup1
MENU ADD POPUP, hMenu, "&Help", hPopup1, %MF_ENABLED
MENU ADD STRING, hPopup1, "&Help", %ID_HELP, %MF_ENABLED
MENU ADD STRING, hPopup1, "-", 0, 0
MENU ADD STRING, hPopup1, "&About", %ID_ABOUT, %MF_ENABLED

' ** Create a new dialog template
DIALOG NEW 0, "What is your name?", , 160, 60, 0, 0 TO hDlg

' ** Add controls to it
CONTROL ADD TEXTBOX, hDlg, %IDTEXT, "", 14, 12, 134, 12, 0
CONTROL ADD BUTTON, hDlg, %IDOK, "OK", 34, 32, 40, 14, %BS_DEFAULT CALL OkButton
CONTROL ADD BUTTON, hDlg, %IDCANCEL, "Cancel", 84, 32, 40, 14, 0 CALL CancelButton

MENU ATTACH hMenu, hDlg

' ** Display the dialog
DIALOG SHOW MODAL hDlg, CALL DlgProc TO lResult

```

```
' ** Check the dialog return result
IF lResult THEN
    MSGBOX "Hello " & gsUserName
END IF
END FUNCTION
```

'-----

See Also

[Menus](#)

[Menu Walkthrough](#)

[More on the Menu](#)

[Menu State](#)

Classic PowerBASIC offers three distinct ways to store and retrieve information from disk: sequential, random, and binary file input and output. Each has its advantages and disadvantages; the one that works best for you will depend on your application.

See Also

[Sequential Files](#)

[Random Access Files](#)

[Binary Files](#)

Sequential [file](#) techniques provide a straightforward way to read and write files. Classic PowerBASIC's sequential file commands manipulate *text* files: files of ASCII characters with carriage-return/linefeed pairs separating records.

Quite possibly, the best reason for using sequential files is their degree of portability to other programs, programming languages, and computers. Because of this, you can often look at sequential files as the common denominator of data processing, since they can be read by word-processing programs and editors (such as Classic PowerBASIC's), absorbed by other applications (such as database managers), and sent over the Internet to other computers.

The idea behind sequential files is simplicity itself: write to them as though they were the screen and read from them as though they were the keyboard.

Create a sequential file using the following steps:

1. Open the file in sequential output mode. To create a file in Classic PowerBASIC, you must use the [OPEN](#) statement. Sequential files have two options to prepare a file for output:

OUTPUT: If a file does not exist, a new file is created. If a file already exists, its contents are erased, and the file is then treated as a new file.

APPEND: If a file does not exist, a new file is created. If a file already exists, Classic PowerBASIC appends (adds) data at the end of that file.

2. Output data to a file. Use [WRITE#](#) or [PRINT#](#) to write data to a sequential file.
3. Close the file. The [CLOSE](#) statement closes a file after the program has completed all I/O operations.

To read a sequential file:

1. First, OPEN the file in sequential INPUT mode. This prepares the file for reading.
2. Read data in from the file. Use Classic PowerBASIC's [INPUT#](#) or [LINE INPUT#](#) statements.
3. Close the file. The CLOSE statement closes a file after the program has completed all I/O operations.

The drawback to sequential files is, not surprisingly, that you only have sequential access to your data. You access one line at a time, starting with the first line. This means if you want to get to the last line in a sequential file of 23,000 lines, you will have to read the preceding 22,999 lines.

Sequential files, therefore, are best suited to applications that perform sequential processing (for example, counting words, checking spelling, printing mailing labels in file order) or in which all the data can be held in memory simultaneously. This allows you to read the entire file in one fell swoop at the start of a program and to write it all back at the end. In between, the information can be stored in an array (in memory) which *can* be

accessed randomly.

Although the [SEEK](#) statement can be used to change the point in the file where the next read or write will occur, the calculations required to determine the position of the start of each record in a sequential file would add considerable overhead. Sequential files typically consist of records of varying sizes. Either you would have to maintain a separate index file indicating the starting byte position of each record, or you would have to seek randomly until you found the correct position. However, SEEK does have its uses with sequential files. For instance, after reading an entire file, you could use SEEK to reposition the file pointer to the start of the file, in order to process the data a second time. This is certainly quicker than closing and re-opening the file.

Sequential files lend themselves to database situations in which the length of individual records is variable. For example, suppose an alumni list had a comments field. Some people may have 100 bytes or more of comments. Others, perhaps most, will have none. Sequential files handle this problem without wasting disk space.

Finally, the OPEN statement provides an optional LEN parameter for use with sequential files. This instructs Classic PowerBASIC to use internal buffering to speed up reading of sequential files, using a buffer of the size specified by the LEN parameter. A buffer of 8192 bytes is suggested for best general performance, especially when networks are involved. However, this value can be increased in size to gain additional performance - the best value will always be specific to a particular combination of hardware and software, and may vary considerably from PC to PC, network to network, etc.

See Also

[Files](#)

[Random Access Files](#)

[Binary Files](#)

Random access [files](#) consist of records that can be accessed in any sequence. This means the data is stored exactly as it appears in memory, thus saving processing time (because no translation is necessary) both in when the file is written and in when it is read.

Random files are a better solution to database problems than [sequential files](#), although there are a few disadvantages. For one thing, random files are not especially transportable. Unlike sequential files, you cannot peek inside them with an editor, or type them in a meaningful way to the screen. In fact, moving a Classic PowerBASIC random file to another computer or language will probably require that you write a translator program to read the random file and output a text (sequential) file.

One example of the transportability problem strikes close to home. Interpretive BASIC uses Microsoft's non-standard format for floating point values, and Classic PowerBASIC uses IEEE standard floating-point conventions, this means you cannot read the floating-point fields of random files created by Interpretive BASIC with a Classic PowerBASIC program, or vice versa, without a bit of extra work.

The major benefit of random files is implied in their name: every record in the file is available at any time. For example, in a database of 23,000 alumni, a program can go straight to record number 11,663 or 22,709 without reading any of the other records. This capability makes it the only reasonable choice for large files, and probably the better choice for small ones, especially those with relatively consistent record lengths.

However, random access files can be wasteful of disk space because space is allocated for the longest possible field in every record. For example, a 100-byte comment field forces every record to use an extra 100 bytes of disk space, even if only one in a thousand actually uses it.

At the other extreme, if records are consistent in length, especially if they contain mostly numbers, random files can save space over the equivalent sequential form. In a random file, every number of the *same* type ([Integer](#), [Long-integer](#), [Quad-integer](#), [Byte](#), [Word](#), [Double-word](#), [Single-precision](#), [Double-precision](#), [Extended-precision](#) or [Currency](#)) occupies the same amount of disk space, regardless of the value itself. For example, the following five Single-precision values each require four bytes (the same space they occupy in memory):

```
0
1.660565E-27
15000.1
641
623000000
```

By contrast, numbers in a sequential file require as many bytes as they have ASCII characters when printed (plus one for the delimiting comma if [WRITE#](#) was used instead of [PRINT#](#)). For example:

```
WRITE #1, 0;0      ' takes 3 bytes
PRINT #1, 0;0      ' takes 5 bytes
PRINT #1, 1.660565E-27 ' takes 13 bytes
```

You can create, write, and read random access files using the following steps:

1. First, [OPEN](#) the file and specify the length of each record:

```
OPEN filespec FOR RANDOM AS [#]filenum [LEN = recordsize]
```

The LEN parameter indicates to Classic PowerBASIC the total size of each record in bytes. If you do not specify a LEN parameter, Classic PowerBASIC assumes 128.

Unlike sequential files, you do not have to declare whether you are opening for input or output because you can simultaneously read and write a random file.

2. Define a structure for records in the file using the [TYPE](#) statement.

```
TYPE StudentRecord
  LastName      AS STRING * 20 ' A 20-character string
  FirstName     AS STRING * 15 ' A 15-character string
  IDnum         AS LONG        ' Student ID, a Long-integer
  Contact       AS STRING * 30 ' Emergency contact person
  ContactPhone  AS STRING * 14 ' Their phone number
  ContactRel    AS STRING * 8  ' Relationship to student.
  AverageGrade AS SINGLE      ' Single-precision % grade
END TYPE
```

```
DIM Student AS StudentRecord
```

3. Fill the [UDTs](#) members with the values you want, and write records to the file using the [PUT](#) statement.

```
Student.LastName = "Anderson"
Student.FirstName = "Bob"
Student.IDnum = 494425610
Student.Contact = "Ma Anderson"
Student.ContactPhone = "(800) BOBSMOM"
Student.ContactRel = "Mother"
Student.AverageGrade = 98.9
```

```
PUT #fileNumber, recordNumber, Student
```

4. Read records from the file using the [GET](#) statement.

```
GET #fileNumber, recordNumber, Student
```

5. When finished, [CLOSE](#) the file.

See Also

[Files](#)

[Sequential Files](#)

[Binary Files](#)

Classic PowerBASIC's binary file technique, an extension to Interpretive BASIC, allows you to treat any file as a numbered sequence of bytes without regard to anything, including the following: ASCII characters, number versus string considerations, record length, carriage returns. With the binary approach to a file problem, you read and write a file by specifying exactly which bytes to read or write. This is similar to the services provided by Windows API functions used for reading and writing files.

Flexibility always comes at a price. Binary files require that you do all the work to decide what goes where. Binary may be the best option when dealing with alien files that aren't in ASCII format; for example, a file created by a spreadsheet or database product. Of course, you will have to know the precise structure of the file before you can even attempt to break it down into numbers and strings agreeable to Classic PowerBASIC.

Every file opened in binary mode has an associated position indicator that points to the place in the file that will be read or written to next. Use the [SEEK statement](#) to set the position indicator, and the [SEEK function](#) to read it.

Binary files are accessed in the following way:

1. First, [OPEN](#) the file in BINARY mode. You need not specify whether you are reading or writing; you can do either, or both.
2. To read the file, use SEEK to position the file pointer at the byte you want to read. Then use [GET\\$](#) to read a specified number of characters into a string variable.
3. To write to the file, load a string variable with the information to be written. Then use SEEK to position the point in the file to which it should be written, and use [PUT\\$](#) to write the data.
4. When finished, [CLOSE](#) the file.

See Also

[Files](#)

[Sequential Files](#)

[Random Access Files](#)

This version of Classic PowerBASIC offers an excellent graphics package for most any programming need. It's fast. It's complete. And it handles all those messy Windows details for you... automatically!

First, it's good to know that graphics in Classic PowerBASIC are persistent. Create it once... and forget it. You'll never worry about redrawing when your window is minimized or temporarily covered. Classic PowerBASIC handles everything. Automatically!

So, how about a quick overview? Just what can you do? First, how about some fancy text? Any font. Any size. Any color. Bold. Italic. Underline and Strikeout. Mix any combination of fonts on a single Window. Print just about anything, just about anywhere. Then add bitmaps. Stretch them or shrink them. Copy them or change them. Circles, ovals, lines and boxes. Fat lines, skinny lines, ellipses, rounded rectangles. Filled forms or empty. Colors or not. You'll create a custom scaling system -- even with fractional floating point coordinates!

So, let's get started. You should know that almost every graphical function name starts with the word GRAPHIC. You'll find all of them together in the help file or the book.

Step one -- you'll need a canvas. A place to create these works of art. So, create a [GRAPHIC WINDOW](#). Or two. Or ten. They'll be visible right away and give you quick feedback.

```
GRAPHIC WINDOW "PowerGraphics", 600, 200, 400, 300 TO hWin???
```

You'll get a new window, with the title "PowerGraphics". It's positioned on the upper right side of the screen at x=600, y=200. It's 400 pixels wide, and 300 pixels high.

A second option is a memory bitmap. These aren't visible at all. You create your image "behind-the-scenes", then copy or stretch it to a visible window whenever you're ready. Use [GRAPHIC BITMAP NEW](#) for a blank bitmap, or [GRAPHIC BITMAP LOAD](#) to get one from a [resource](#) or a disk file. You can have one window or five. One bitmap or twenty. As each is created, it returns a handle that you need to save. That's how you'll identify each of your canvases.

Step two Use [GRAPHIC ATTACH](#) to choose a "graphic target". This tells Classic PowerBASIC which window or memory bitmap to use, for the actions which follow. Until you execute another GRAPHIC ATTACH to change it again. Move back and forth, as often as necessary. There is no limitation here.

Step three Draw-Draw-Draw. Arcs. Circles. Lines. Boxes. Text. Display them. Copy them. Save them to disk.

Step four Clean up when you're done. You must close every graphic window with [GRAPHIC WINDOW END](#), and every memory bitmap with [GRAPHIC BITMAP END](#).

It's just that simple!

Some GRAPHIC functions use the concept of an implied "graphic position" to determine the

default point on the graphic target where the next operation will take place. In Classic PowerBASIC, we use the keyword POS to refer to this position (See [GRAPHIC GET POS](#) and [GRAPHIC SET POS](#) to alter or retrieve this position). POS is also commonly known as the LPR (Last Point Referenced) or even NPR (Next Point Referenced). For most purposes, you can consider these three terms to be synonymous.

When a Graphic Window or Graphic Bitmap is created, the default POS is set to (0,0), which is the upper left corner. Unless you specify otherwise, the first graphical operation starts at that point, and the completion point is then saved as the new POS. So, if you draw a line from (0,0) to (100,100), that last point (100,100) is saved as the new POS. The next line you draw would then, by default, start at (100,100), and then automatically save its completion point as the updated POS for next time.

The "Graphic Position" (POS) is used by [GRAPHIC LINE](#), [GRAPHIC PAINT](#), [GRAPHIC PRINT](#), and [GRAPHIC SET PIXEL](#). Other graphic functions neither use nor update POS.

Other GRAPHIC functions, namely those involved with the drawing of curves ([GRAPHIC ARC](#), [GRAPHIC ELLIPSE](#), and [GRAPHIC PIE](#)), utilize the concept of a "bounding rectangle" to determine their size and position on the graphic target. A bounding rectangle is defined as the smallest rectangle which can be drawn around the circle or ellipse. For example, let's say you wish to draw a circle centered at position (200,200), which has a radius of 50 pixels. The upper left corner (x1,y1) of the bounding rectangle would be at (150,150), while the lower right corner of the bounding rectangle would be at (250,250).



See Also

[Graphic Commands](#)

Classic PowerBASIC supports two general classes of printers. We categorize them as Line Printers or Host Printers. Generally speaking, we recommend using Host Printers whenever possible, as they have far greater capabilities, including an extensive graphics package.

A line printer is one which will accept standard ASCII text and associated control codes, such as CR, LF, and FF. A line printer is identified by the port to which it is attached (LPT1, etc.) because data is sent directly to the port, not through a device driver. Print to Line Printers by using the LPRINT family of functions.

A host printer is one which works through the Windows printing system and a Windows printer driver. These printers are sometimes known as "Windows-only printers" or "GDI printers". They achieve device independence because the printer driver handles the task of converting ASCII text into the manufacturers proprietary binary format used by the printer. Print to Host Printers by using the XPRINT family of functions.

An interesting feature of this version is the new [PRINTER\\$](#) function. This will let you retrieve both the printer name and the port name for every printer connected to the computer. Also, the new [XPRINT ATTACH](#) statement will optionally display a Printer Common Dialog to assist the user in selecting a printer, and the associated options.

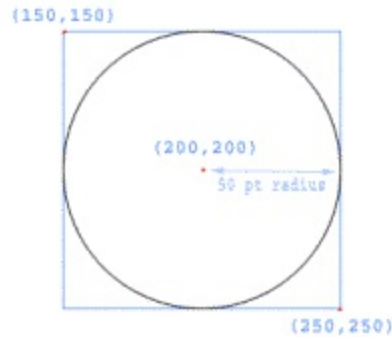
In contrast to single-tasking systems like DOS, you'll need to select a printer when you're ready to print. Use either [LPRINT ATTACH](#) or XPRINT ATTACH to do that. That assures two applications won't try to print to the same printer at the same time. Then print your report. Print your graphics. Print your charts. When you're done, don't forget to detach the printer with [LPRINT CLOSE](#) or [XPRINT CLOSE](#). This frees up the printer for another application to use. Perhaps even more important, Host Printers normally won't even begin to print to the physical paper until the print job is closed!

Some XPRINT functions use the concept of an implied "XPrint Position", to determine the default point on the host printer page where the next operation will take place. In Classic PowerBASIC, we use the keyword POS to refer to this position (See [XPRINT GET POS](#) and [XPRINT SET POS](#) to alter or retrieve this position). POS is also commonly known as the LPR (Last Point Referenced) or even NPR (Next Point Referenced). For most purposes, you can consider these three terms to be synonymous.

When a new host printer page is created (with XPRINT ATTACH of a host printer, or [XPRINT FORMFEED](#) which ends a printer page), the default POS is set to (0,0), which is the upper left corner. Unless you specify otherwise, the first XPRINT operation starts at that point, and the completion point is then saved as the new POS. So, if you draw a line from (0,0) to (100,100), that last point (100,100) is saved as the new POS. The next line you draw would then, by default, start at (100,100), and then automatically save its completion point as the updated POS for next time.

The "XPrint Position" (POS) is used by [XPRINT](#), [XPRINT LINE](#), and [XPRINT SET PIXEL](#). Other XPRINT functions neither use nor update POS.

Other XPRINT functions, namely those involved with the drawing of curves ([XPRINT ARC](#), [XPRINT ELLIPSE](#), and [XPRINT PIE](#)), utilize the concept of a "bounding rectangle" to determine their size and position on the host printer page. A bounding rectangle is defined as the smallest rectangle which can be drawn around the circle or ellipse. For example, let's say you wish to draw a circle centered at position (200,200), which has a radius of 50 pixels. The upper left corner (x1,y1) of the bounding rectangle would be at (150,150), while the lower right corner of the bounding rectangle would be at (250,250).



See Also

[Printing Commands](#)

This section introduces telecommunications. For many programmers, writing a communications program is difficult. It is not that the programs themselves are especially long; it is that the procedures and terminology are unfamiliar. That means programs take longer to write and [debug](#), making writing such programs frustrating. To compound the problem, performing serial communications using the Windows API can be a daunting task.

In this section, we define common communications terms and discuss some of the more regular ways of transmitting and receiving data, using the native COMM statements and related features of Classic PowerBASIC. There are plenty of examples and some working Classic PowerBASIC program code (included on your distribution disks) for you to try out. Feel free to use this code as a starting point for your own communications programs.

Before presenting any sample code or delving further into the mysteries of communications, let's define a few terms:

ACK	An acknowledgment signal sent by the receiver of a message.
Asynch	Asynchronous; not synchronous. The receiver and transmitter are free to send signals without matching clocks.
Baud Rate	Baud (from J. M. E. Baudot, a communications pioneer) refers to the total number of signal changes that could possibly be sent between transmitter and receiver per second. Signal changes do not necessarily mean bits, and not all bits are necessarily data, so baud rate isn't equivalent to a fixed number of characters (or even a fixed number of bits) per second.
Buffer	An area of memory used to hold transmitted or received signals before processing them.
CD	Carrier Detect. A signal used to tell that a carrier has been detected; the DCE (modem) has connected with another computer and is ready for use.
CRC	Cyclic Redundancy Check. A way of summing the data bits sent between transmitter and receiver so as to detect transmission errors.
CTS	Clear To Send. A handshaking signal that indicates the receiver is ready to receive data. Typically, a modem uses this signal for to control data flow from the computer. CTS (and sometimes DSR) are often used in response to an RTS signal.
DCE	Data Circuit-terminating Equipment. Typically, a DCE is a modem. A DCE is often inaccurately referred to as the "Data Communications Equipment".
DSR	Data Set Ready. A handshaking signal that serves to indicate that an RS-232C non-terminal device (usually a modem) is ready to receive data.

Often used with CTS.

DTE	Data Terminal Equipment. Typically the computer.
DTR	Data Terminal Ready. A handshaking signal that indicates that an RS-232C serial terminal device (the computer) is ready to receive data.
Handshaking	A process whereby the receiver and transmitter match signals and correctly determine each other's status. See CTS, DSR, and RTS.
Modem	Modulator/Demodulator. A device used to convert digital signals to sounds that can be carried over standard telephone lines, and to convert such sounds back to digital signals.
NAK	Negative Acknowledgment. A signal sent from receiver to transmitter, indicating that an error was detected in the last message.
Null Modem	A way of connecting receive and transmit lines, so that one computer can send or receive signals directly from another without having to go through a modem. This typically requires a "null-modem" or "cross-over" cable to ensure the signals are correctly interconnected between devices.
Parity	One bit (the high-order bit) per byte sent or received, used to detect some of the possible transmission errors. Parity may be even, odd, none, mark (always on), or space (always off).
Port	A term used to refer to any one of the possibly several communications devices available to the operating system. Usually COM1, COM2, etc.
Protocol	A way of controlling transmissions or receptions. A protocol consists of a set of rules describing the form of a valid transmission, the proper response when a transmission is received, and ways of detecting and correcting errors.
RS-232	A standard for wiring on serial communications ports, that describes which wires should carry which signals at what voltage. There are two basic standards: one for transmitting equipment (DTE) and one for receiving equipment (DCE).
RTS	Request To Send. A signal raised by the transmitter, to which the receiver should reply with CTS and/or DSR.
Serial	Refers to signals sent one bit after the other, as opposed to parallel. Parallel signals are sent more than one bit at a time.
Stop Bits	The number of bits added to each byte of data transmitted, to allow the receiver to get in step with the transmitter.
Synch	Synchronous. The receiver and transmitter match their clocks so that each will send and receive only at specific times.

See Also

[Communications Basics](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Start and Stop bits](#)

[Opening a communications port](#)

[Reading and writing data](#)

[A simple communications program](#)

To communicate from one computer to another, you need a communications program on each machine, and some way to connect them (telephone lines, for example). Sometimes, those programs are built into the operating system. Even when that's the case, there will be times when you want to do something faster, more reliably, or in a different manner than what has been provided.

To communicate, you will need a way for each program to inform the other that:

1. It is ready/not ready to transmit/receive data.
2. Data has been received and is correct.
3. Data has been received and is not correct.
4. Transmission is over.

If it is not important to check for, or correct errors, items 2 and 3 in the previous list can be ignored. Those capabilities are often skipped when the programs are to be used for simple communications - short text or brief typed messages. Even the other two parts can be dropped if the programs are closely monitored, or if errors will not matter very much.

However, it is important to realize that [receiving and transmitting data](#) is not always quite as simple as it might appear, especially when you are coding under Windows. For example, suppose your program is receiving data and saving material to a disk file. What should happen if more data is received while the program is writing data to disk?

Alternatively, suppose the user presses a key that means "clear the screen and display a menu" while data is being received?

Situations like this are common. The standard way of handling them is to use a [buffer](#).

See Also

[Serial Communications](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Start and Stop bits](#)

[Opening a communications port](#)

[Reading and writing data](#)

[A simple communications program](#)

Buffers can be useful in solving various types of communication or data transfer problems. For example, a printer typically includes a buffer of at least a few thousand characters. Since the computer can send material to the printer faster than it is capable of putting the material on paper, the buffer serves three purposes:

1. To even the workload for the printer
2. To allow the computer to finish sending material sooner
3. To handle occasions when the printer cannot accept more material

If the printer did not have a buffer, the computer would be forced to send data one character at a time. Until each character is received and printed, the computer should not send more. To prevent this, the printer sends a busy signal back to the computer. When it gets this signal, the computer stops until the printer sends a "ready" signal. This "ready/busy" signaling is called *handshaking*.

Visualize what is taking place. The printer sends a ready signal; the computer sends a character; the printer sends a busy signal, forcing the computer to wait while it prints the character; and then the whole process repeats. That is a lot of signaling for just one transmitted character!

Further, there is a possibility of error. If the computer is fast or the printer is slow (or both), it's possible for the computer to send the next character before the printer is able to signal that it's busy - something called *buffer overflow*. This can also happen if there is something wrong with the handshaking signals. When this happens, the printer fails to print one or more characters. Those characters have been sent by the computer, but cannot be received by the printer because there is no place to put them.

With a buffer, the printer sends a busy signal only when the buffer is full (or nearly so). That way, even if additional characters have already been sent, there will be room to store them before they are printed. Most of the time, the computer sends and the printer receives. As a result, far less signaling is necessary, and more actual data is transmitted. Therefore, a buffer makes communications between computers and printers more efficient. Since there is room to store characters transmitted, there is less chance that a character will be missed; so a buffer makes transmissions more error free, too.

In general, all communications are affected by buffering in the same way. For that reason, Classic PowerBASIC allows you to set aside one communications buffer for received data and a separate buffer for transmitted data. In your programs, you have two responsibilities: to make sure that the buffer you use is large enough, and to empty the buffer as often as needed to prevent a buffer overflow.

How large a buffer will you need? It depends on the sort of program you are writing, and is often a matter of trial and error. At low baud rates (up to 300 baud), 256 bytes is probably adequate. Under some circumstances, 256 bytes may well be adequate at 1200 baud or higher; it all depends on how often your program checks the buffer and empties it. It's probably a good idea to use a buffer of 1024 bytes or more for 1200 baud, and it's not

at all uncommon to use buffers of 4 Kilobytes or more. With the large amount of data memory available to your applications with Classic PowerBASIC, you could specify a receive buffer of 1 MB (or even more) and have little impact on system memory.

See Also

[Serial Communications](#)

[Communications Basics](#)

[Parity and general error checking](#)

[Start and Stop bits](#)

[Opening a communications port](#)

[Reading and writing data](#)

[A simple communications program](#)

The ASCII character set contains 128 defined characters. It takes 7 bits to represent all 128 characters. Since there are 8 bits per byte, the eighth bit can be used to detect errors. One sort of checking adds up all the on (1) bits in each character transmitted. If the number of bits is even, the eighth bit is turned on when the character is transmitted; that forces the total number of on bits to be odd and is called odd parity. When the receiver gets the character, it performs the same procedure in reverse. If it gets the same answer as was encoded in the eighth bit, the character is accepted. If it does not, the character is in error. A related method (even parity), sums the bits and turns the parity bit on if the number of bits is odd, forcing the total number of on bits to be even.

Either method of checking the correctness of received characters is called a parity check. Unfortunately, the method can easily be thwarted if the errors are bad enough. If the transmission is relatively clean and there are few errors, a simple parity check of this sort can be reasonably effective. If any even number of data bits are reversed (on to off or vice versa), or if any odd number of bits are wrong and the parity bit is also incorrect, the parity check will fail to detect an error.

Most communications programs do not rely on parity checks, however. That is especially true if you must send a full 8 bits of data, as is the case when sending executable programs, spread sheets, some kinds of word processing files, and any kind of binary data. You should set parity to none or off whenever you need to send a full 8 bits of data.

See Also

[Serial Communications](#)

[Communications Basics](#)

[Communication Buffers](#)

[Start and Stop bits](#)

[Opening a communications port](#)

[Reading and writing data](#)

[A simple communications program](#)

Stop bits are a way for a computer to "catch its breath" while sending or receiving data, while still letting the other end know that the connection is still there and is still valid; they're also used in error detection.

Stop bits are rather like [parity bits](#). They are sent with the data, but they are not part of the data. Unlike parity bits, they are not turned on and off by the number of bits in the data; instead, they are always on. If one or more of the stop bits are missing, it constitutes a framing error.

Some computers also use start bits for a similar purpose; however, that is not as common a practice as it used to be.

The number of stop bits generally increases with higher baud rates. At 300 baud, usually 0 or 1 stop bits are used. At 1200 baud, 1 or 2 stop bits are most common.

If you connect with another computer and everything seems to be correct, but you cannot read the material you're receiving, one of three possible problems is likely. Either the baud rates are set wrong, the parity is wrong, or the number of stop (or start) bits is incorrect. If the baud rate is correct and the errors are framing errors, it is probably the number of stop bits.

See Also

[Serial Communications](#)

[Communications Basics](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Opening a communications port](#)

[Reading and writing data](#)

[A simple communications program](#)

Before we can actually open a [communications port](#), we must first obtain a Classic PowerBASIC *file number* so we may manage input and output to the communications port. The best method of obtaining a file number is to use the [FREEFILE](#) function:

```
DIM hComm AS LONG
hComm = FREEFILE
```

The general way of opening a communications port in a Classic PowerBASIC program is with a [COMM OPEN](#) statement. The syntax is similar to a simple [random-access file OPEN](#), where *n* is the communications port number

```
COMM OPEN "COMn" AS #hComm
```

Note the trailing colon typical in DOS communications is not permitted with COMM OPEN. If you are familiar with serial communications with DOS compilers (where all of the communications parameters are configured within a single OPEN statement), you will realize that we must instead configure these parameters individually. For this purpose, Classic PowerBASIC offers the [COMM SET](#) statement:

```
COMM SET #hComm, Comfunc = value
```

Although configuring a serial port for communications can mean using quite a few COMM SET statements, Classic PowerBASIC offers a greater control of the serial port than was possible before, plus a completely new method of querying existing settings and status. Retrieving a setting is performed with the [COMM](#) function, which returns a [Long-integer](#) value:

```
x% = COMM(#hComm, Comfunc)
```

Comfunc **must** be one of the following keywords:

Comfunc	Return values (TRUE <> 0, FALSE = 0)
BAUD	Port Baud Rate (9600, 14400, 19200, etc). See notes below.
BREAK	TRUE/FALSE Break is asserted. Break is generally used to "get the attention" of the connected modem, terminal or system.
BYTE	Number of bits per byte (4, 5, 6, 7, or 8).
CD	TRUE/FALSE Carrier Detect state. Synonym for RLSD (<i>READ-ONLY</i>). When CD is TRUE, the DCE (modem) has a suitable connection on the communications channel present. When CD is FALSE, there is no suitable connection.
CTS	TRUE/FALSE Clear-To-Send state is returned (<i>READ-ONLY</i>).
CTSFLOW	TRUE/FALSE Enable CTS output flow control (Input signal). When CTSFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the CTS signal is set to logic low by the DCE (modem). Transmission continues when the DCE (modem) sets the CTS signal back to logic high. The CTS signal is usually used in response to an RTS signal.

DSR	TRUE/FALSE Data-Set-Ready state is returned (READ-ONLY).
DSRFLOW	TRUE/FALSE Enable DSR output flow control (Input signal). When DSRFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the DSR signal is set to logic low by the DCE (modem). Transmission is enabled when the DSR signal returns to logic high. The DSR signal is often used in conjunction with CTS in response to a RTS signal.
DSRSENS	TRUE/FALSE Enable DSR sensitivity. When DSRSENS is enabled, data received by the DTE (computer) is placed into the receive buffer only if DSR is set to logic high. If DSR is set low, received data is discarded. Enabling DSRSENS allows DSR to enable or disable the DTE (the computer) to receive data from the DTE (the modem). DSRSENS is rarely used in practical communications situations.
DTRFLOW	TRUE/FALSE Enable DTR handshaking flow control (Output signal). When DTRFLOW is enabled, it signals that the DCE (modem) should prepare to connect to the communications channel. DTR is usually used for modem on-hook/off-hook control, but can also be used in conjunction with DSR for handshaking.
DTRLINE	TRUE/FALSE Enable DTR line. When enabled, DTRLINE leaves the DTR line active when the port is closed by the DTE (computer). This ensures that the DCE (modem) does not close the communications channel when the port is closed.
NULL	TRUE/FALSE Null (\$NUL) bytes are discarded when read.
PARITY	TRUE/FALSE Enable parity checking. This mode must be enabled for the other Parity options to be selected.
PARITYCHAR	Character to use for parity error replacement. PARITY must be enabled.
PARITYREPL	TRUE/FALSE Enable character replacement on parity error. PARITY must be enabled.
PARITYTYPE	0 = None, 1 = Odd, 2 = Even, 3 = Mark, 4 = Space. PARITY must be enabled. Default = 0.
RING	TRUE/FALSE Ring indicator is on (READ-ONLY). When RING returns TRUE, a ringing signal is being received on the communications channel (by the modem). RING approximates the state of the ringing signal; however, it may not report accurately in all Windows platforms.
RLSD	Receive-line-signal-detect (READ-ONLY). See CD/Carrier Detect above.
RTSFLOW	Ready To Send (Output signal). 0 = Disable, 1 = Enable, 2 =

	Handshake, 3 = Toggle. Toggle is used for half-duplex (2-wire) operations to "reverse" the line. While the DTE (computer) is busy sending data, it raises the RTS signal, and the DCE (modem) blocks its data receive channel. When RTS signal reverts to logic low, the DCE (modem) reverts to transmit mode and the DTE (computer) switches to receive mode.
	Handshake mode causes the DTE (computer) to check the receive buffer (RXQUE) after each character is placed into the buffer. When the buffer is 5/6th full, the RTS signal is dropped. When the receive buffer drops to below 1/6th full, RTS is raised again.
RXBUFFER	Size of the receive buffer in bytes.
RXQUE	Bytes currently in the receive buffer (<i>READ-ONLY</i>).
STOP	0 = 1 stop bits , 1 = 1.5 stop bits, 2 = 2 stop bits.
TXBUFFER	Size of the transmit buffer in bytes. In some cases, Windows may not be able to report the transmit size.
TXQUE	Bytes currently in the transmit buffer (<i>READ-ONLY</i>).
XINPFLOW	TRUE/FALSE Enable XON/XOFF input flow control. When the DTE (computer) receive buffer is full, an XOFF character is sent to the DCE (modem) to instruct it to halt transmission. When the DCE is ready to resume transmission, an XON character is sent to the DCE. Typically, XOFF is sent when the receive buffer has less than 1/16th remaining, and XON is sent when the receive buffer drops to less than 1/16th of its maximum size. Default = FALSE.
XOUTFLOW	TRUE/FALSE Enable XON/XOFF out flow control. When enabled, the DCE (modem) sends an XOFF to the DTE (computer) to halt data transmission to the DCE. When the DCE is ready to receive more data, an XON character is sent. XOUTFLOW typically uses the same 1/16th rules as XINPFLOW. Default = FALSE.

Common baud rates range from 110 to 256000. There are equates defined in the [WIN32API.INC](#) file, prefixed with %CBR_ to assist you with specifying a common baud rate, but you are not restricted to a limited set of rates.

To open a communication port and initialize it for use, you will need to set the following parameters. The values are for demonstration purposes, you may choose your own settings as necessary.

```
' Minimum recommended settings
COMM SET #hComm, BAUD      = 9600      ' 9600 baud
COMM SET #hComm, BYTE      = 8          ' 8 bits
COMM SET #hComm, PARITY    = %FALSE    ' No parity
COMM SET #hComm, STOP      = 0          ' 1 stop bit
COMM SET #hComm, TXBUFFER  = 2048      ' 2 Kb transmit buffer
COMM SET #hComm, RXBUFFER  = 4096      ' 4 Kb receive buffer
```

```
' Optional settings for flow control
COMM SET #hComm, CTSFLOW = 1      ' Enable CTS flow control
COMM SET #hComm, RTSFLOW = 1      ' Enable RTS flow control
COMM SET #hComm, XINPFLOW = 0      ' Disable XON/OFF Input
                                   ' flow control
COMM SET #hComm, XOUTFLOW = 0      ' Disable XON/XOFF output
                                   ' flow
```

When we have finished using our communication channel, we can terminate it using the [COMM CLOSE](#) function:

```
COMM CLOSE #hComm
```

If any errors occur when attempting to open the communications port, or as a result of an invalid *Comfunc* value, Classic PowerBASIC will set the [ERR](#) system variable.

See Also

[Serial Communications](#)

[Communications Basics](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Start and Stop bits](#)

[Reading and writing data](#)

[A simple communications program](#)

To complement the new [COMM OPEN](#) statement, Classic PowerBASIC introduces four new COMM statements to help you write serial communications programs:

```
COMM PRINT #hComm, expr [;]
COMM SEND #hComm, expr
COMM RECV #hComm, count, expr
COMM LINE [INPUT] #hComm, expr
```

[COMM PRINT](#) and [COMM SEND](#) are used to send data out of the communications port (via the transmit buffer). COMM PRINT sends the data specified by *expr* followed by a CR/LF byte pair {[\\$CRLF](#) or [CHR\\$\(13,10\)](#)}. By adding a trailing semicolon to the COMM PRINT statement, Classic PowerBASIC suppresses these CR/LF bytes. COMM SEND is identical to COMM PRINT with a trailing semicolon.

[COMM RECV](#) and [COMM LINE](#) [INPUT] are used to receive data from a communications port (via the receive buffer). The [COMM](#)(#hComm, RXQUE) function can be used to identify the number of bytes that can be retrieved with COMM RECV. COMM LINE is used to return a CR/LF delimited "line" of data from the receive buffer.

If your communications application is primarily dealing with binary data transmission and reception, COMM SEND and COMM RECV will suit this purpose exactly. COMM PRINT and COMM INPUT are very useful for sending "AT" commands to a modem and receiving the modem response. For example:

```
COMM PRINT #hComm, "AT"
SLEEP 1000 ' Give modem time to respond
WHILE COMM(#hComm, RXQUE)
  COMM LINE #hComm, a$
  ' Display "AT" (the modem echo),
  ' followed by "OK" (the modem response)
  #IF %DEF(%PB_CC32)
    PRINT a$
  #ELSE
    MSGBOX a$
  #ENDIF
WEND
```

The COMM RESET statement allows you to switch off all flow control during a serial communications session.

```
COMM RESET #hComm, FLOW
```

See Also

[Serial Communications](#)

[Communications Basics](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Start and stop bits](#)

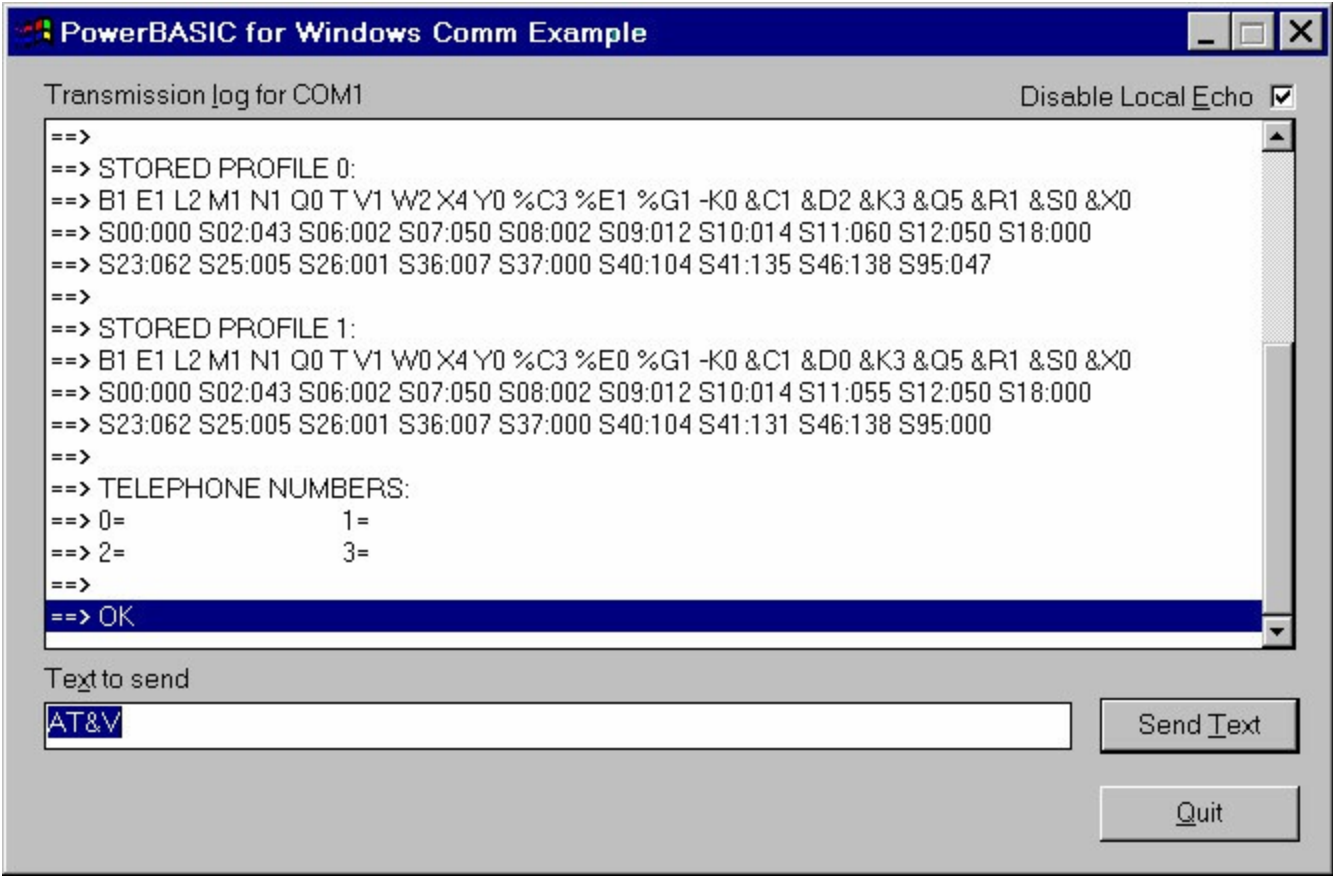
[Opening a communications port](#)

[A simple communications program](#)

A simple communications program

Let's assume you want a simple communications program to use for accessing a local computer bulletin board. You know the parameters for the board: it is 14400 baud, 8 data bits, one stop bit, and no parity.

You want to display data on your screen, be able to type data, and have it sent to the bulletin board. You intend to use a modem connected to COM1. The following short program serves as a starting point, and uses Classic PowerBASIC's new [DDT](#) features to create the user interface:



```
'-----
'
'  Serial Communications Example for Classic PowerBASIC for Windows
'      Copyright (C) 2004-2009 Classic PowerBASIC, Inc.
'
'  Be sure to set the $ComPort constant to the appropriate
'  COM port before compiling this example!
'-----
#COMPILE EXE
#DIM ALL
#INCLUDE "WIN32API.INC"

$ComPort      = "COM1"
$AppTitle     = "Classic PowerBASIC for Windows Comm Example"
%IDD_MAIN     = 100
%IDC_LISTBOX1 = 101
%IDC_EDIT1    = 102
```

```
%IDC_SEND          = 103
%IDC_QUIT           = 106
%IDC_ECHO           = 107
```

```
GLOBAL hComm        AS LONG
GLOBAL Updating      AS LONG
GLOBAL hThread       AS DWORD
GLOBAL ThreadClose   AS LONG
```

```
DECLARE FUNCTION StartComms          AS LONG
DECLARE FUNCTION SendLine(ASCIIZ)    AS LONG
DECLARE FUNCTION ReceiveData(BYVAL LONG) AS LONG
DECLARE FUNCTION EndComms             AS LONG
DECLARE FUNCTION AddLine(BYVAL LONG, BYVAL LONG, ASCIIZ) AS LONG
```

```
CALLBACK FUNCTION Dialog_Callback() AS LONG
    SELECT CASE CB.MSG
        CASE %WM_INITDIALOG
            ' Set focus to the edit control
            CONTROL SET FOCUS CB.HNDL, %IDC_EDIT1

            ' Set SELECTION range to highlight the initial entry
            CONTROL SEND CB.HNDL, %IDC_EDIT1, %EM_SETSEL, 0, -1

            ' Return 0 to stop dialog box engine setting focus
            FUNCTION = %FALSE
    END SELECT
END FUNCTION
```

```
CALLBACK FUNCTION Send_Callback() AS LONG
    DIM SendText AS ASCIIZ * 1024, ListCount AS LONG
    DIM lResult AS LONG, hListBox AS DWORD

    ' Obtain the text to send from the edit control
    CONTROL GET TEXT CB.HNDL, %IDC_EDIT1 TO SendText

    ' Set the update flag
    Updating = %TRUE

    ' Send the line to the comm port
    IF SendLine(SendText) THEN
        SendText = "Transmission Error!"
    ELSE
        ' Check the Echo mode state
        CONTROL GET CHECK CB.HNDL, %IDC_ECHO TO lResult
        IF ISTRUE lResult THEN SkipEcho
    END IF

    ' Add the echo to the listbox
    CALL AddLine(CB.HNDL, %IDC_LISTBOX1, "<== " + SendText)
```

```
SkipEcho:
    ' Set the SELECTION range for the edit control so the
    ' next keypress "clears" the existing text
    CONTROL SEND CB.HNDL, %IDC_EDIT1, %EM_SETSEL, 0, -1

    ' restore the keyboard focus to the edit control
    CONTROL SET FOCUS CB.HNDL, %IDC_EDIT1

    ' Release the update flag
    Updating = %FALSE
```

```

FUNCTION = %TRUE
END FUNCTION

```

```

CALLBACK FUNCTION Quit_Callback() AS LONG
' Kill the dialog and let PBMAIN() continue
DIALOG END CB.HNDL, 0

```

```

FUNCTION = 1
END FUNCTION

```

```

FUNCTION AddLine(BYVAL hWnd AS DWORD, BYVAL nID AS LONG, SendText AS ASCIIZ) AS LONG
DIM ListCount AS LONG

```

```

' Find the current listbox count
LISTBOX GET COUNT hWnd, nID TO ListCount

```

```

' Update the listbox
LISTBOX ADD hWnd, nID, SendText

```

```

' Scroll the new item into view
LISTBOX SELECT hWnd, nID, ListCount + 1
END FUNCTION

```

```

FUNCTION PBMAIN
' Build our GUI interface.
DIM hDlg AS DWORD, Txt(1 TO 1) AS STRING, lResult AS LONG

```

```

' Initialize the port ready for the session
IF ISFALSE StartComms THEN
    MSGBOX "Failure to start communications!",, $AppTitle
    EXIT FUNCTION
END IF

```

```

Txt(1) = "Listbox holds the transmission I/O stream..."

```

```

' Create a modal dialog box
DIALOG NEW 0, $AppTitle,,, 330, 203, %WS_POPUP OR %WS_VISIBLE OR %WS_CLIPCHILDREN OR
    %WS_CAPTION OR %WS_SYSMENU OR %WS_MINIMIZEBOX, 0 TO hDlg

```

```

' Add our application controls
CONTROL ADD LABEL, hDlg, -1, "Transmission &log for " & $ComPort, 9, 5, 100, 10, 0

```

```

CONTROL ADD LISTBOX, hDlg, %IDC_LISTBOX1, Txt(), 9, 15, 313, 133, %WS_BORDER OR _
    %LBS_WANTKEYBOARDINPUT OR %LBS_DISABLENOSCROLL OR %WS_VSCROLL OR %WS_GROUP OR _
    %WS_TABSTOP OR %LBS_NOINTEGRALHEIGHT

```

```

CONTROL ADD LABEL, hDlg, -1, "Text to send", 9, 151, 100, 10, 0

```

```

CONTROL ADD TEXTBOX, hDlg, %IDC_EDIT1, "ATZ", 9, 161, 257, 12, %ES_AUTOHSCROLL OR _
    %ES_NOHIDESEL OR %WS_BORDER OR %WS_GROUP OR %WS_TABSTOP

```

```

CONTROL ADD BUTTON, hDlg, %IDC_SEND, "Send &Text", 273, 160, 50, 14, %WS_GROUP OR _
    %WS_TABSTOP OR %BS_DEFPUSHBUTTON CALL Send_Callback

```

```

CONTROL ADD BUTTON, hDlg, %IDC_QUIT, "&Quit", 273, 182, 50, 14, %WS_GROUP OR
%WS_TABSTOP
    CALL Quit_Callback

```

```

CONTROL ADD CHECKBOX, hDlg, %IDC_ECHO, "Disable Local "+ "&Echo", 252, 5, 70, 10, _
    %WS_GROUP OR %WS_TABSTOP OR %BS_AUTOCHECKBOX OR %BS_LEFTTEXT

```

```

' Erase our array to free memory no longer required
REDIM Txt()

```

```

' Create a "listen" Thread to monitor input from the modem

```

```

THREAD CREATE ReceiveData(hDlg) TO hThread

' Start the dialog box & run until DIALOG END executed.
DIALOG SHOW MODAL hDlg, CALL Dialog_Callback TO lResult

' Close down our "listen" Thread
ThreadClose = %TRUE

DO
    THREAD CLOSE hThread TO lResult

    ' Release time-slice for improved multitasking
    SLEEP 0
LOOP UNTIL ISTRUE lResult

' Flush & close the comm port
CALL EndComms

FUNCTION = %TRUE
END FUNCTION

FUNCTION StartComms AS LONG
    hComm = FREEFILE
    COMM OPEN $COMPORT AS #hComm
    IF ERRCLEAR THEN EXIT FUNCTION      ' Port problem?

    COMM SET #hComm, BAUD      = 14400 ' 14400 baud
    COMM SET #hComm, BYTE      = 8    ' 8 bits
    COMM SET #hComm, PARITY    = %FALSE ' No parity
    COMM SET #hComm, STOP      = 0     ' 1 stop bit
    COMM SET #hComm, TXBUFFER = 4096   ' 4 Kb transmit buffer
    COMM SET #hComm, RXBUFFER = 4096   ' 4 Kb receive buffer

    FUNCTION = %TRUE
END FUNCTION

FUNCTION SendLine(SendText AS ASCIIZ) AS LONG
    COMM PRINT #hComm, SendText
END FUNCTION

FUNCTION ReceiveData(BYVAL hWnd AS DWORD) AS LONG
    DIM InboundData AS STRING
    DIM Stuf AS STRING, ListCount AS LONG
    DIM Qty AS LONG, x AS LONG, a AS STRING

    WHILE ISFALSE ThreadClose
        ' Test the RX buffer
        Qty = COMM(#hComm, RXQUE)

        ' Abort this iteration if sending
        IF ISFALSE Qty OR Updating THEN
            SLEEP 100
            ITERATE LOOP
        END IF

        ' Read incoming characters
        COMM RECV #hComm, Qty, Stuf

        InBoundData = InBoundData & Stuf

        ' strip out LF characters

```

```

REPLACE CHR$(10) WITH "" IN InBoundData

' process only complete lines of data terminated by CR
WHILE INSTR(InboundData, CHR$(13))
    ' Display the data
    CALL AddLine(hWnd, %IDC_LISTBOX1, "==> " + EXTRACT$(InBoundData, CHR$(13)))

    ' reduce the buffer to remove the "displayed" line
    InBoundData = STRDELETE$(InBoundData, 1, LEN(EXTRACT$(InBoundData, CHR$(13))) +
1)
WEND
WEND

FUNCTION = %TRUE
END FUNCTION

FUNCTION EndComms() AS LONG
    DIM dummy AS STRING

    ' Flush the RX buffer & close the port
    SLEEP 1000

    IF COMM(#hComm, RXQUE) THEN
        COMM RECV #hComm, COMM(#hComm, RXQUE), dummy
    END IF

    COMM CLOSE #hComm
END FUNCTION

```

This short program allows you to connect with the bulletin board, but it will not dial the number of the bulletin board through the program itself. You can do that easily though, in one of two ways:

You can dial the bulletin board manually. When you're done dialing, connect the telephone line to the modem (or press a button on your modem, switching the line from the telephone back to the modem). The program should now be ready to receive whatever the bulletin board sends.

You can send the appropriate signals directly to the modem itself. Most modems recognize a common command set originated by the Hayes Company. To initialize the modem and dial, you would enter the following commands:

```

ATZ
ATDT18005551212

```

Note: some modems require capital letters for AT commands. Lowercase letters will not work.

After you have entered the ATZ command, the modem responds. You will see the message "OK" on your screen. After you have entered ATDT and the telephone number, the modem's lights flicker for a moment. If your modem is capable of making a sound, you should hear the sounds of the number being dialed, and the telephone ringing at the other end.

If the number is busy, you may hear a busy signal through your modem speaker, or you may not hear anything more. If the connection is made, you may see some garbage characters on your screen.

At this point, many users become concerned and think that something must be wrong. Why are there illegible characters on screen? Relax: this happens often. The computer you called does not yet know what baud rate and communications parameters you are using. In most cases, you should press ENTER a few times; the computer at the other end will use that character to determine what your parameters are and will adjust itself accordingly. Soon afterward, you should see a welcoming message. You may now type whatever you like.

If you see double lines of characters, click on the Disable Local Echo button. This simply prevents the code from adding your characters to the transmission log window.

If you wish to send a stream of AT commands to a modem in quick succession, you may be required to add a small delay between each AT command, in order to give the modem time to decode each command and respond appropriately. A delay of 100 to 200 milliseconds (mSec) is usually sufficient.

Using disk files

The sample program does not let you save material to a disk file, or send data from a [disk file](#) to the bulletin board. Nevertheless, those two options are very useful. How do you do it?

Let us suppose you wanted to send a disk file to the bulletin board. To do that, the routine that sends your keystrokes to the bulletin board must be altered. The usual way to do this is to assign a special keystroke a different meaning: instead of being sent, it is interpreted as a command to get the name of a disk file, read that disk file, and send it to the bulletin board.

Let's add a new button to our dialog window to provide access to this feature - we will label this button Send File. In addition, we must also add a [Callback Function](#) to handle the event from this button. Lets start by adding the following equate definition to the block near the beginning of the file:

```
%IDC_SENDFILE = 104
```

Now we will insert the new Callback Function to the code. We'll add this immediately after the Send Callback() function ends:

```
CALLBACK FUNCTION SendFile_Callback() AS LONG
    STATIC SendFileName AS STRING
    LOCAL hReadFile AS LONG, FileLen AS LONG, Chunk AS LONG
    LOCAL i AS LONG, Buff1 AS STRING

    Buff1 = INPUTBOX$("Name of disk file to transmit?", $AppTitle, SendFileName)
    IF ISFALSE LEN(Buff1) OR ISFALSE LEN(DIR$(Buff1)) THEN EXIT FUNCTION

    SendFileName = Buff1
    CALL AddLine(CB.HNDL, %IDC_LISTBOX1, "Wait... Sending " & SendFileName)
    DIALOG DOEVENTS

    ' send the file
    hReadFile = FREEFILE
    OPEN SendFileName FOR BINARY AS #hReadFile ' Binary mode
```

```

FileLen = LOF(hReadFile)          ' File length
Chunk   = MAX$(32, COMM(#hComm, TXBUFFER) \ 2) ' 1/2*Buf

FOR ix = 1 TO FileLen \ Chunk
    GET$ #hReadFile, Chunk, Buff1    ' Read a chunk
    COMM SEND #hComm, Buff1          ' and send it
    SLEEP 0
NEXT i

IF FileLen MOD Chunk <> 0 THEN      ' More to send?
    GET$ #hReadFile, FileLen MOD Chunk, Buff1
    COMM SEND #hComm, Buff1
END IF

CLOSE #hReadFile
CALL AddLine(CB.HNDL, %IDC_LISTBOX1, "Transmission complete!")
END FUNCTION

```

Finally, we insert the code that adds a new control button on the dialog box. Add the following line to the group of CONTROL ADD statements in the [PBMAIN](#) function.

```

CONTROL ADD BUTTON, hDlg, %IDC_SENDFILE, "&Send File", 9, 182, 50, 14, %WS_GROUP OR _
%WS_TABSTOP CALL SendFile_Callback

```

The routine works, but there's no error checking in it. If the disk file does not exist, nothing is sent, but a zero-length file is created. If you enter an illegal file name, the program will set the [ERR system variable](#) to indicate that [a potentially fatal] error has occurred. You'll probably want to add some kind of [error checking](#) to the program for those reasons.

To receive a disk file, we will add yet another button to the dialog window titled Receive File. However, things are not quite as simple as the code we added to send a file: you must be able to use the program at the same time as the data is stored, as it comes in from the serial port. We also need a way to stop receiving a disk file.

First, we will add another equate to the beginning of the file, exactly as before:

```
%IDC_RECEIVEFILE = 105
```

Add the following line at the end of the [GLOBAL](#) variable declarations, just below the equates:

```
GLOBAL hWriteFile AS LONG
```

Next, add the Callback Function code, immediately after the SendFile_Callback() function that we just added.

```

CALLBACK FUNCTION ReceiveFile_Callback() AS LONG
    STATIC ReceiveFileName AS STRING
    LOCAL Buff2 AS STRING

    ' First check if file is already open
    IF hWriteFile THEN
        ' Close the file
        CLOSE #hWriteFile

        CALL AddLine(CB.HNDL, %IDC_LISTBOX1, "Finished writing file!")

        ' Update the button label
        CONTROL SET TEXT CB.HNDL, %IDC_RECEIVEFILE, "&Receive File"

    RESET hWriteFile

```

```

EXIT FUNCTION
END IF

' Create a new file
Buff2 = INPUTBOX$("Output file name?", $AppTitle, ReceiveFileName)
IF ISFALSE LEN(Buff2) THEN EXIT FUNCTION

ReceiveFileName = Buff2
hWriteFile = FREEFILE

OPEN ReceiveFileName FOR APPEND AS #hWriteFile
IF ERRCLEAR THEN
    ' Error opening the file
    RESET hWriteFile
ELSE
    ' Update the dialog
    CALL AddLine(CB.HNDL, %IDC_LISTBOX1, "Receiving data stream to " &
ReceiveFileName)
    CONTROL SET TEXT CB.HNDL, %IDC_RECEIVEFILE, "Stop &Receive"
END IF
END FUNCTION

```

Now add the CONTROL ADD statement into PBMAIN in the same manner as before.

```

CONTROL ADD BUTTON, hDlg, %IDC_RECEIVEFILE, "&Receive File", 62, 182, 50, 14,
%WS_GROUP OR _
%WS_TABSTOP CALL ReceiveFile_Callback

```

Finally, to ensure that the disk file is closed correctly, if the program is closed before the file is closed, insert the following lines just before the END FUNCTION within PBMAIN.

```

IF hWriteFile THEN CLOSE #hWriteFile

```

When we click on the new Receive File button, we enter the file name that will be used to save the data. At this point, the output file is opened. The received data will be appended to the end of any existing file of that name. However, we have not provided any way to actually save any of that information. To do that, add one more small line of code to the ReceiveData() function, immediately after the line:

```

InBoundData = InBoundData & Stuf

```

The added line reads:

```

' If Receive mode is on, write raw data to the file
IF hWriteFile THEN PRINT #hWriteFile, Stuf;

```

Finishing touches

If we examine this example file, we find that we have overlooked one problem: if the program is terminated while the output file is in use, the file is not closed.

While this is not a fatal condition, it is a poor approach to program design: we should always close the files we have opened. Remembering to perform this chore will stand you in good stead when it comes to using the Windows API functions. In many cases, failing to close a registry key or delete a GDI object can cause both deceptive and difficult bugs to locate; or memory leaks that reduce system memory even after your program has ended. The golden rule should always be before you leave, clean up after yourself.

So, faced with this problem, how do we know if the output file is open before we end the

program? Simple... we set the global variable that holds the file number when the file was open. If this number is non-zero (logical TRUE), we can simply assume we need to close the file before finally exiting the program.

After the line that reads:

```
CALL EndComms
```

We add the following line to the file:

```
IF hWriteFile THEN CLOSE #hWriteFile
```

In this instance, we control three possible scenarios with only one line of code:

1. the output file feature was not used (hFile2 = 0)
2. the output file remained open when the program was about to end (hFile2 <> 0)
3. the output file had been used, but had been closed before program termination (hFile2 <> 0)

It is true that we could have just closed the file associated with hWriteFile regardless of the state of the file or the value of the file number. However, in most programming circles, that is considered to be a poor approach. It is always better to write code that is fail-safe in as many conditions as possible.

The final program can be found in the PB\SAMPLES\COMM folder of your Classic PowerBASIC installation. It is not very large, but it handles a surprising number of ordinary communications tasks. It lacks some error checking, as has been noted. If you choose to modify this program, you might want to put some error checking in. You might also want to test for such problems as the [List box](#) control filling up to the limits of the operating system (i.e., 32767 entries in Windows 95/98/ME), and even add a few more buttons to send certain preformatted strings to modem, for example "ATZ" or "ATDT555-1234".

Compared to DOS applications, this communications application may seem overly complex. This is because we simply cannot afford to use 100% of the CPU just to monitor a serial port. If we did, your multitasking operating system would suddenly take a huge drop in performance. If you examine the code a little more deeply, you will see it takes advantage of a very handy feature of 32-bit Windows: [multi-threading](#).

This communications program consists of two threads in total: (1) the main thread handles the user commands and sending data to the modem; (2) the second thread simply monitors the serial port for receive data. If we used only one single thread in this application, the code would need to share its time between both data reception and transmission, but by using two, we ensure that the CPU is not heavily loaded unnecessarily. Using a second thread in this way effectively splits the application into two (almost) independent sections. The only time these threads need to be aware of each other is when one is writing to the list box control. To handle this, we used a GLOBAL variable to signal when data was being displayed; temporarily "locking" the other thread until the task was complete.

For further experimentation, you could split the main thread down even further and create a separate thread just for writing data to the serial port. You could even try replacing the [TEXTBOX](#) control with a [COMBOBOX](#) so users can scroll back through the most recent

"send" strings, providing a simple "history" feature.

See Also

[Communications Basics](#)

[Communication Buffers](#)

[Parity and general error checking](#)

[Start and Stop bits](#)

[Opening a communications port](#)

[Reading and writing data](#)

Network Communications is one of the hottest programming topics today. Whether you need to send an email message to an SMTP server on your Intranet, or transfer a file from a remote Internet server halfway around the world, Classic PowerBASIC can handle your network communications requirements.

Networks typically consist of many computers, all with a number of different hardware architectures and operating systems. Your local area network might have machines running Windows, Linux, DOS, OS/2, or Mac OS. Your network may use IPX, ATM, or some other transmission protocol for sending data packets from one computer to the next. The architects of the Internet needed a transmission protocol that could be used on any platform.

See Also

[The Internet Protocol \(IP\)](#)

[User Datagram Protocol \(UDP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Winsock](#)

[Request for Comments \(RFC\)](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

The Internet Protocol (IP)

[Top](#) [Previous](#) [Next](#)

The Internet Protocol was designed for transmitting blocks of data called datagrams from a source location, to one or more destinations. It is specifically limited to provide the functions necessary to deliver a datagram from a source to a destination over an interconnected system of networks. There is no functionality for data reliability, flow-control, or sequencing. It works by using one local network to connect with another local network, until the datagram is delivered to its destination - although, a datagram does not have to leave the local network at all if the destination does not reside outside of the network.

The source and the destination are specified as numeric addresses, also known as IP addresses. An IP address consists of four bytes. The combined sequence of the four bytes is unique for each connection to the network (a single computer can have more than one connection to the network, and therefore can have more than one IP address).

Let's say that you want to send the message "Hello" from your computer to another computer on the network. Your computer cannot simply transmit the 5 bytes of your message over the network cable. It first has to create a datagram. In simple terms, the datagram would include the identity of your computer (the sender), the identity of the computer you are sending the message to, and some kind of checksum that allowed the receiving computer to verify that the datagram arrived intact. Your computer would deliver that datagram to a host or gateway on your network.

The gateway will then determine where the datagram needs to go next. If the destination computer is on the same local network, it may simply deliver it to the destination. If the destination is outside of the local network, the datagram is delivered to another host or gateway "downstream" from your network. After that, either that host or gateway would then send the datagram even further downstream toward the destination; or if the destination resides on the local network of the current host or gateway, it will deliver the datagram to the final destination itself.

During this journey, it is possible for the datagram to become corrupted, be misrouted (and lost), or simply expire because the journey was too long. The Internet Protocol does not provide any notification capabilities to inform the sender of a delivery problem. It is also possible for large datagrams to be chopped into multiple smaller datagrams if any host or gateway along the path cannot handle the size of the datagram. Each datagram is then broken into as many smaller datagrams as it needs to hold all of the data. Those datagrams then have to be reassembled at the destination.

See Also

[User Datagram Protocol \(UDP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Winsock](#)

[Request for Comments \(RFC\)](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

User Datagram Protocol (UDP)

[Top](#) [Previous](#) [Next](#)

Obviously, writing code to deal with reassembling fragmented datagrams would make you think twice about how badly your application needs to communicate over a network. Fortunately, the Internet architects provided a protocol layer that sits on top of the [Internet Protocol](#).

UDP uses the Internet Protocol to send datagrams from a source to a destination. When the datagram arrives at the destination, it hands the complete datagram packet to the client. If the datagram was fragmented along the way, it reassembles the fragments into a complete datagram beforehand.

Like the Internet Protocol it uses, UDP does not guarantee delivery of a datagram. Its purpose is simply to format a datagram with your data, send it via the Internet Protocol to a destination, and at the destination, deliver the complete datagram to a client.

One interesting aspect of the Internet Protocol is that datagrams can be delivered to a destination in a different sequence than the one in which they were sent. For example: your application sends two datagrams to another computer. The first datagram is routed along a longer path than the second datagram, and therefore arrives at the destination after the second datagram has arrived.

See Also

[The Internet Protocol \(IP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Winsock](#)

[Request for Comments \(RFC\)](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

Transmission Control Protocol (TCP) [Top](#) [Previous](#) [Next](#)

TCP is a connection-based protocol layer that guarantees delivery of data to the destination. The reliability and flow control of TCP requires that status information be sent with each datagram indicating sequence numbers and checksums. So that TCP transmissions can recover from data that is damaged, lost, duplicated, or delivered out of order, each datagram is checked for its sequence number, and the data is verified against the checksum. An acknowledgment (ACK) is then required from the recipient for each successful datagram received. If an ACK is not received within a timeout period, the datagram is resent.

Unlike [UDP](#), TCP does not reassemble fragmented datagrams into the original data packet. It simply extracts the data portion of the datagram and adds it to the incoming data stream. This can be problematic if a source has sent 20 bytes of data that is fragmented into two datagrams with 10 bytes each. TCP will give the first 10 bytes to the client without waiting for the next 10 bytes to arrive. UDP will give all 20 bytes of the data, or nothing.

See Also

[The Internet Protocol \(IP\)](#)

[User Datagram Protocol \(UDP\)](#)

[Winsock](#)

[Request for Comments \(RFC\)](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

In Windows, Microsoft has encapsulated the [Internet Protocol](#) **and** the [TCP](#) and [UDP](#) protocol layers into the Windows Sockets Layer, or "Winsock". Winsock allows an application to send datagrams using either TCP or UDP without having to do low-level programming to create IP datagram packets, deal with receipts and acknowledgments, or reassemble fragmented datagrams.

Classic PowerBASIC further encapsulates the process by handling DNS resolution of IP addresses, and presents statements familiar to the programming model used by BASIC programmers. You are free to concentrate on the data being sent and ignore the details of sending it.

Classic PowerBASIC requires version 2.0 or later of Winsock.

See Also

[The Internet Protocol \(IP\)](#)

[User Datagram Protocol \(UDP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Request for Comments \(RFC\)](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

Request for Comments (RFC)

[Top](#) [Previous](#) [Next](#)

All of the technical specifications for the Internet are contained in white papers called "Request for Comments". For example, the RFC document describing the [UDP](#) protocol is RFC768.TXT and can be downloaded from <http://www.rfc-editor.org>.

See Also

[The Internet Protocol \(IP\)](#)

[User Datagram Protocol \(UDP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Winsock](#)

[TCP clients and servers](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

The [Internet Protocol](#) driver in [Winsock](#) actually sends and receives the datagrams itself, and the [UDP](#) and [TCP](#) layers take care of data integrity. However, to actually communicate with another computer over the Internet your code will have to handle the data itself. That is typically done using a high-level protocol such as [SMTP](#), POP3, FTP, and others.

Think of [IP](#) as the telephone wire that carries a voice from a transmitter of one telephone to a receiver of another telephone. UDP and TCP are simply different types of telephones that make sure each sound is received exactly as it was sent. Therefore, SMTP, POP3, FTP, etc, should be considered the language that you use to speak. Both the caller and the person being called need to speak the same language if they wish to understand the conversation. Obviously, you cannot speak Latin to a person who only understands English or French.

See Also

[The Internet Protocol \(IP\)](#)

[User Datagram Protocol \(UDP\)](#)

[Transmission Control Protocol \(TCP\)](#)

[Winsock](#)

[Request for Comments \(RFC\)](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

[An ECHO client and server using TCP](#)

Simple Mail Transfer Protocol (SMTP) [Top](#) [Previous](#) [Next](#)

One of the easiest high-level [TCP](#) protocols to use is [SMTP](#) for sending an email message. This application simply connects to an SMTP server via TCP, identifies itself, and identifies who the message is for, sends the text of the message, and finally says goodbye. As each line of text is sent to the server, it returns a status code and message to indicate progress. The following code demonstrates this:

```
' Be sure to change the following two string equates
' to the name of your SMTP mail server and your email
' address.
#COMPILE EXE
```

```
$mailhost = "mailserver.mydomain.com"
$mailfrom = "email@address.com"
```

```
FUNCTION PBMAIN() AS LONG
' Get the local computer's IP address and name
HOST ADDR TO ip&
HOST NAME ip& TO hostname$

' ** Connect to the mailhost
hTCP& = FREEFILE
TCP OPEN "smtp" AT $mailhost AS hTCP&
IF ERR THEN
    MSGBOX "Error connecting to mailhost"
    EXIT FUNCTION
ELSE
    TCP LINE hTCP&, buffer$
    IF LEFT$(buffer$, 3) <> "220" THEN
        MSGBOX "Mailhost Error: " & buffer$
        EXIT FUNCTION
    END IF
END IF

' Get the local computer's IP address and name
HOST NAME TO hostname$

' ** Greet the mailhost
TCP PRINT hTCP&, "HELO " + hostname$
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "250" THEN
    MSGBOX "HELO Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF

' ** Tell the mailhost who we are
TCP PRINT hTCP&, "MAIL FROM:<" & $mailfrom & ">"
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "250" THEN
    MSGBOX "MAIL FROM Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF
```

```

' ** Tell the mailhost who we want to send the message to
TCP PRINT hTCP&, "RCPT TO:<info@Classic PowerBASIC.com>"
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "250" THEN
    MSGBOX "RCPT TO Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF

' ** Now we can send the message
TCP PRINT hTCP&, "DATA"
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "354" THEN
    MSGBOX "DATA Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF

TCP PRINT hTCP&, "From: " & $mailfrom
TCP PRINT hTCP&, "To: info@Classic PowerBASIC.com"
TCP PRINT hTCP&, "Subject: Greetings!"
TCP PRINT hTCP&, ""
TCP PRINT hTCP&, "Just wanted to say hello."
TCP PRINT hTCP&, "This TCP stuff is great!"

' ** End of the message
TCP PRINT hTCP&, "."
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "250" THEN
    MSGBOX "DATA Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF

' ** Say goodbye
TCP PRINT hTCP&, "QUIT"
TCP LINE hTCP&, buffer$
IF LEFT$(buffer$, 3) <> "221" THEN
    MSGBOX "QUIT Error: " & buffer$
    TCP CLOSE hTCP&
    EXIT FUNCTION
END IF

TCP CLOSE hTCP&
END FUNCTION

```

The SMTP protocol is fully described in RFC 821 <http://www.rfc-editor.org>

See Also

[TCP and UDP Communications](#)

[TCP clients and servers](#)

[An ECHO client and server using TCP](#)

An ECHO client and server using TCP

[Top](#) [Previous](#) [Next](#)

The simplest [TCP server](#) application is an Echo Server (RFC 862). It simply listens to port 7, and when it receives a data packet, it returns the data packet back to the client.

Writing a TCP server in Classic PowerBASIC is quite straightforward, but your application must contain a (GUI) window or [dialog](#) to receive notification requests from [Winsock](#). It is therefore necessary to either: (1) create a dialog with [DDT](#), or (2) use the Windows API to create a GUI window for the application to receive these notifications. The following function will register a window class, and create a hidden window that can be used by your server.

```
FUNCTION MakeWindow() AS DWORD
    LOCAL wce          AS WndClassEx
    LOCAL szClassName AS ASCIIZ * 80
    LOCAL hWnd         AS DWORD
    STATIC registered AS LONG

    IF ISFALSE registered THEN
        szClassName      = "PBTCPCOMM"
        wce.cbSize       = SIZEOF(wce)
        wce.style         = %NULL
        wce.lpfnWndProc   = CODEPTR(TcpProc)
        wce.cbClsExtra   = 0
        wce.cbWndExtra    = 0
        wce.hInstance     = GetModuleHandle(BYVAL %NULL)
        wce.hIcon         = %NULL
        wce.hCursor       = %NULL
        wce.hbrBackground = %NULL
        wce.lpszMenuName  = %NULL
        wce.lpszClassName = VARPTR(szClassName)
        wce.hIconSm       = %NULL
        RegisterClassEx wce
        registered = %TRUE
    END IF

    hWnd = CreateWindow("PBTCPCOMM", "", 0,0,0,0,0, %NULL, %NULL, _
        GetModuleHandle(BYVAL %NULL), BYVAL %NULL)
    ShowWindow hWnd, %SW_HIDE

    FUNCTION = hWnd
END FUNCTION
```

To create a TCP server, your program must first open a socket using the [TCP OPEN SERVER](#) statement. Then, when a client contacts your server, this socket will receive the notification. To specify which notifications your code will process, use the [TCP NOTIFY](#) statement:

```
%TCP_ACCEPT = %WM_USER + 4093 ' user-defined message value
...
hServer = FREEFILE
TCP OPEN SERVER PORT 7 AS hServer
TCP NOTIFY hServer, ACCEPT TO hWnd AS %TCP_ACCEPT
```

TCP NOTIFY tells Winsock that it should send the %TCP_ACCEPT message to the

window specified by hWnd. Your [callback](#) will then include a message handler for the %TCP_ACCEPT message. The lParam& parameter to your callback will tell you what type of notification was sent:

```
%TCP_ECHO = %WM_USER + 4094 ' user-defined message value
...
CASE %TCP_ACCEPT
    SELECT CASE LO(WORD, lParam&)

        '* An ACCEPT notification was sent
        CASE %FD_ACCEPT
            hEcho = FREEFILE
            TCP ACCEPT hServer AS hEcho
            TCP NOTIFY hEcho, RECV CLOSE TO hWnd AS %TCP_ECHO

        .
        . 'other notification code goes here
        .
    END SELECT
```

Once your code receives the ACCEPT notification, it uses the [TCP ACCEPT](#) statement to "close" the socket. A new socket is created for the actual communication with the client. The original socket (hServer) is used strictly to process ACCEPT notifications only. TCP NOTIFY is then used with the new socket handle to process RECV and CLOSE notifications.

When the Echo Client sends its message to your server, a RECV notification will be sent to your window. Your code can then log the incoming message, and send it right back to the client. When the CLOSE notification is received, you can close the socket:

```
CASE %TCP_ECHO
    SELECT CASE LO(WORD, lParam&)

        CASE %FD_READ
            IF hEcho <> %INVALID_SOCKET THEN
                TCP RECV hEcho, 1024, buffer
                TCP SEND hEcho, buffer
                LogEvent $DQ + Buffer + $DQ
            END IF

        CASE %FD_CLOSE
            TCP CLOSE hEcho
            hEcho = %INVALID_SOCKET

    END SELECT
```

To connect with the Echo Server, our Client simply needs to open a socket at port 7, send a string, and display the string echoed back from the server.

```
FUNCTION PBMAIN() AS LONG
    LOCAL hSocket AS LONG

    hSocket = FREEFILE
    TCP OPEN PORT 7 AT "" AS hSocket
    IF ERR THEN
        MSGBOX "OPEN Error" + STR$(ERR)
        EXIT FUNCTION
    END IF

    IF LEN(COMMAND$) = 0 THEN
```

```

    TCP SEND hSocket, "This is a test"
ELSE
    TCP SEND hSocket, COMMAND$
END IF

TCP RECV hSocket, 1024, buffer$
IF ERR THEN
    MSGBOX "RECV Error" + STR$(ERR)
    EXIT FUNCTION
END IF

MSGBOX buffer$

TCP CLOSE hSocket
END FUNCTION

```

The complete Echo Server and Echo Client sample can be found in your PB\SAMPLES\INTERNET\TCP folder.

Finally, it should be noted that there is no direct correlation between the number of TCP SEND statements executed, compared to the number of %FD_READ messages received. This is because Winsock may concatenate multiple data packets and issue a lesser number of %FD_READ messages in response. Therefore, it is usually necessary to construct your code so that it continues to read data from the incoming data stream until either the returned string is empty, or an error is detected. For example:

```

DIM InBuffer AS STRING
...
CASE %FD_READ
    InBuffer = ""
    IF hEcho = %INVALID_SOCKET THEN EXIT SELECT

    DO
        TCP RECV hEcho, 1024, buffer
        IF LEN(buffer) = 0 OR ISTRUE ERR THEN EXIT LOOP
        InBuffer = InBuffer + buffer
        TCP SEND hEcho, buffer
        LogEvent $DQ + Buffer + $DQ
    LOOP
...

```

See Also

[TCP and UDP Communications](#)

[Simple Mail Transfer Protocol \(SMTP\)](#)

What is an object, anyway?

[Top](#) [Previous](#) [Next](#)

An object is a pre-defined set of data ([variables](#)), neatly packaged with a group of subroutines (code) which manipulate the data and provide any other functionality you need.

For example, a [string array](#) containing names and addresses (data) might be packaged with a subroutine (code) that displays a popup [dialog](#) to edit the data, another subroutine (code) to [print](#) mailing labels, and so forth. That's a great candidate for an object.

In short, an object is a complete little programming package, code and data, all in one tightly contained place. It's safer and protected, easier to debug, maintain, and reuse. An object can be written to perform most any task you might imagine.

In object terminology, a [CLASS](#) is used to define an object. A CLASS is much like an enhanced [user-defined type](#); it's a description of both the variables and the subroutines which make up the object. When you instruct the compiler to create an object, it uses the definitions found in the CLASS to do so. It allocates memory for the variables, establishes [pointers](#) to the subroutines, and makes this new object available to your program.

Each time you create a new OBJECT, it is called an INSTANCE of its definition (an instance of the CLASS). That's why these variables are called [INSTANCE](#) variables. When you create multiple objects (from the same CLASS definition), each instance gets its own individual copy of these INSTANCE variables, and each instance gets individual access to the subroutines.

In Classic PowerBASIC, objects are optional. Objects are a great programming tool, but your existing code remains fully functional. Standard [Subs](#) and [Functions](#) will always be supported, so you can blend the techniques at a comfortable pace.

Classic PowerBASIC objects are practical. They're lightning fast with very little overhead. We've tried very hard to give you the best ratio of straightforward design to performance and features. We think you'll find Classic PowerBASIC objects very hard to resist.

Thousands of books have been written to describe objects and object oriented programming. In most cases, the buzz words and abstract definitions make it seem as though they're designed to confuse, not enlighten. We'll try to limit the use of strange descriptors, but some of it just can't be avoided. In these cases, we'll try to give you clear definitions as they're needed.

A key trait of Classic PowerBASIC objects (and objects in general) is the concept of encapsulation. Data is "hidden" within the object, so INSTANCE variables cannot be accessed from the outside.

INSTANCE variable data may only be set, altered, or retrieved by the subroutines in the object. These variables are hidden from the rest of the program.

Over the years, objects have gained a reputation for slow, bloated programming. In many cases, this reputation was well deserved. But don't let that fool you. With Classic PowerBASIC, you'll find you have a whole new "Object World"! All the power, yet all the performance, too. Classic PowerBASIC objects give you every ounce of performance

possible... the same breathtaking speed as procedural programming!

See Also

[Where are objects located?](#)

[Why should I use objects?](#)

[What are the parts of an object?](#)

[Are there other important "Buzz-Words"?](#)

Where are objects located?

[Top](#) [Previous](#) [Next](#)

Since an [object](#) is a complete programming package (sort of like the idea of a sub-program), it can be located in many different places. However, regardless of where the object is found, Classic PowerBASIC will still handle all the messy details for you... automatically.

In many cases, objects will be located right within your main program. You can create a single, self-contained program, with one object or a thousand objects. Get all the power of objects, but keep the details private -- for your eyes only.

Objects can be located in a [Dynamic Link Library](#) (DLL). This is usually called a [COM](#) object, but is also known as an OCX or an ActiveX object. The actual file extension is largely irrelevant. The subroutines offered by these objects are generally available to any program which knows the subroutine definitions, and wishes to access them. This type of object is known as an "in-process" object because it is loaded into the address space of the calling application, just like a standard DLL.

Objects can also be located in an executable program (EXE). In this case, the calling application is frequently called a "controller", as it can control how the executable operates by manipulating its objects. A good example of this functionality is Microsoft Word -- by simply calling object subroutines, you can load a DOC file, display it to the user, make changes, then save the new document. All under the control of your calling application. Once again, the object subroutines are generally available to any program which knows the subroutine definitions. This type of object is known as an "out-of-process" object because it does not share address space with the calling application.

Whenever an object is accessed outside of your program, Classic PowerBASIC uses the COM (Component Object Model) services of Windows to make the "connection" for you. COM is an important tool which will open many opportunities for you. But more about COM later...

See Also

[What is an object, anyway?](#)

[Why should I use objects?](#)

[What are the parts of an object?](#)

[Are there other important "Buzz-Words"?](#)

Why should I use objects?

- [Objects](#) help you maintain your code. Objects break up your project into small, easily viewed parts. Usually, the input and output is clearly defined. You have all of the code and all of the data right at your fingertips.
- Objects help you write bug-free code. When you keep an object small and well-defined, you greatly enhance the stability of your programs. Consider the comparison to procedural programming: With standard [Subs](#) and [Functions](#), it's typical to create the data ([variables](#)) in the calling code, but manipulate the data in the target procedures when they are executed. This separation of code and data has caused some of the most insidious bugs known to programmers. When you need to extend the range of data to a larger data type, it's easy to change the code. A piece of cake, so to speak. But what about the data? Now you must search out every reference to every involved Sub and Function. Find every data creation, every data change, and every other reference to these variables. What are the chances of missing a critical one? Far too great to ignore.
- Objects help you re-use your code. Since the object contains all the subroutines, and all the data, how could it be easier? Put one object source in one [include](#) file... Or put it in one [DLL](#)... Just use it when you need it!
- Objects help with team programming. Objects are self-contained. All of the subroutines and all of the data, all in one concise place. It's easy to create a precise definition for each object, and there's little dependency between the implementation of various objects. Each team member builds an object, one at a time, so it all comes together neatly in the end.
- Objects are an increasingly popular standard. Do you need to access the Windows API? Many of the newer API functions (DirectX graphics, for example) use only an object interface, and nothing else. If you don't use objects, you simply can't access them. Do you want to control an important application, like an Internet browser, word processor, or spreadsheet? COM objects are the only way to do it. As time goes by, objects will only become more embedded in day-to-day programming. Don't be left behind!

See Also

[What is an object, anyway?](#)

[Where are objects located?](#)

[What are the parts of an object?](#)

[Are there other important "Buzz-Words"?](#)

What are the parts of an object?

- **METHOD**: A subroutine, very similar to a user-defined [Sub/Function](#). A method has the special attribute that it can access the variables stored in the [object](#). A method can return a value like a Function, or return nothing, like a Sub.
- **PROPERTY**: This is a METHOD, but in a specific form, for a specific purpose. A PROPERTY has all the attributes of a standard METHOD. It has a special syntax, and is specifically used to read or write private data to/from the internal variables in an object. This helps to maintain the principle of "encapsulation". Properties are usually created in pairs, a GET PROPERTY to read a variable, and a SET PROPERTY to write to a variable. Paired properties use the same name for both, since Classic PowerBASIC will choose the correct one based upon the usage in your source code. You should note this important fact: Since a PROPERTY is a form of METHOD, all of the documentation about METHODS also applies to PROPERTIES, unless we specifically state otherwise.
- **INTERFACE**: A definition of a set of methods and properties which are implemented on an object. You might think of it as a list of [DECLARE](#) statements where the sequence of the Declares must be preserved. Remember, the interface is just the definition, not the actual code. Every interface is associated with a [GUID](#) (a 128-bit number or string) which uniquely identifies this particular interface from all other interfaces, anywhere in the world. This identifier is called the Interface ID, or IID for short. An interesting note is that one particular interface definition may become a part of several different [classes](#) and objects. In fact, the internal code for an interface in CLASS A may be entirely different from the internal code for the same interface in CLASS B. Method names, parameters, and return values must be identical, but the internal code could vary significantly. An important point: interfaces are immutable. Once an interface has been defined and published, the Method and Property definitions (sequence, names, parameters, return values, etc.) may never be altered. If you find you must change or extend an interface, you would usually define a new interface instead.
- **CLASS**: A definition of a complete object, which may include one or more interfaces. This is the place where you declare [INSTANCE](#) variables, and write your code for the enclosed METHOD and PROPERTY procedures. While some object implementations allow only a single interface per class, Classic PowerBASIC objects (and COM objects in general) support the idea of optional multiple interfaces. Still, remember that a CLASS is the complete definition of an object. It defines all of the code and all of the data which will be found in a particular object. For this reason, there is only one copy of a CLASS. Every class is associated with a GUID (a 128-bit number or string) which uniquely identifies this particular class from all others, anywhere in the world. This identifier is called the Class ID, or [CLSID](#). A friendlier version of the CLSID is a shorter text name, which also identifies the Class. This text name is known as the Program ID ([PROGID](#)), though it's possible this PROGID may not be totally unique.

As it's a simpler construct, it might be duplicated in another program.

- [CLASS METHOD](#): This is a private method, which may only be called from within the same CLASS. It is not a part of any interface, so it is never listed there. It is called a CLASS METHOD because it is a member of the class, not an interface. It is not visible to any code outside the class where it is defined. Code in a CLASS METHOD may call other CLASS METHODS in the same CLASS. Class Properties do not exist because there is no need for them. Within the object, variables can be accessed directly, so there is no need to use a PROPERTY procedure as an intermediary.
- [CONSTRUCTOR](#): This is a special form of CLASS METHOD, which is executed automatically whenever an object is created. It is optional, but if present, it must be named CREATE.
- [DESTRUCTOR](#): This is a special form of CLASS METHOD, which is executed automatically whenever an object is destroyed. It is optional, but if present, it must be named DESTROY.
- [OBJECT](#): An instance of a class. When you create an object in your running program, using the [LET \(with objects\)](#) statement, or its implied form, Classic PowerBASIC allocates a block of memory for the set of instance variables you defined, and establishes a virtual function table (a set of function code pointers) for each of the interfaces. You can create any number of OBJECTS based upon one CLASS definition.

It might be useful to think of an OBJECT in terms of an electrical appliance, like a television set. The TV is the equivalent of an OBJECT, because it's an instance of the plans which define all the things which make it a television. Of course, those plans are the equivalent of a CLASS. You can make many instances of a television from one set of plans, just as you can make many OBJECTS from one CLASS. The individual buttons and controls on the television are the equivalent of METHODS, while all of the controls, taken as a whole, are equivalent to the INTERFACE.

We don't need to know how a television works internally to use it and benefit from it. Likewise, we don't need to know how an object works internally to use it and benefit from it. We only need to know the intended job of the object, and how to communicate with it from the outside. The internal workings are well "hidden", which is called encapsulation. Since we can't "look inside" an Object, it's not possible to directly manipulate internal variables or memory. This provides an increased level of security for the internal code and data.

- [INSTANCE DATA](#): Each CLASS defines some INSTANCE variables which are present in every object. When you create multiple objects (of the same class), each object gets its own unique copy of them. These variables are called INSTANCE variables because a new set of them is created for each instance of the object. For example, if you created a CUSTOMER object for each customer of your business, you might have INSTANCE variables for the Name, Address, Balance owed, etc. Each object would have its own set of INSTANCE variables to describe the attributes of that

particular customer. INSTANCE variables are always private to the object. They can be accessed directly from any METHOD on the object, but they are invisible to any code outside of the object.

- VIRTUAL FUNCTION TABLE: Commonly called a VFT or VTBL, this is a set of function code pointers, one for each METHOD or PROPERTY in an interface. This is a tool used internally to direct program execution to the correct method or property you wish to execute. While it is a vital and integral part of every object, you need give it no concern other than to be aware of its existence. Classic PowerBASIC manages these items for you, with no programmer intervention required.

See Also

[What is an object, anyway?](#)

[Where are objects located?](#)

[Why should I use objects?](#)

[Are there other important "Buzz-Words"?](#)

Are there other important "Buzz-Words"?

[Top](#) [Previous](#)
[Next](#)

- [GUID](#): This is a "Globally Unique Identifier", a very large number which is used to uniquely identify every [interface](#), every [class](#), and every [COM](#) application or library which exists anywhere in the world. GUID's identify the specific components, wherever and whenever they may be used. A GUID is a 16-byte (128-bit) value, which could be represented as an integer or a string. This 128-bit item is large enough to represent all the possible values, anywhere in the world. The Classic PowerBASIC [GUID\\$\(\)](#) function (or a [hot-key](#) in the Classic PowerBASIC IDE) can generate a random GUID which is statistically guaranteed to be unique from any other generated GUID. Each of these identifying GUID's may be assigned by the programmer, or they will be randomly assigned by the Classic PowerBASIC compiler. When a GUID is written in text, it takes the form: {00CC0098-0000-0000-0000-0000000000FF}.
- [DIRECT INTERFACE](#): This is the most efficient form of interface. When you call a particular [METHOD](#) or [PROPERTY](#), the compiler simply performs an indirect jump to the correct entry point listed in the virtual function table (VFT or VTBL). This is just as fast as calling a standard [Sub](#) or [Function](#), and is the default access method used by Classic PowerBASIC.
- [DISPATCH INTERFACE](#): This is a slow form of interface, originally introduced as a part of Microsoft Visual Basic. When you use DISPATCH, the compiler actually passes the name of the METHOD you wish to execute as a text string. The parameters can also be passed in the same way. The [object](#) must then look up the names, and decide which METHOD to execute, and which parameters to use, based upon the text strings provided. This is a very slow process. Many scripting languages still use DISPATCH as their sole method of operation, so continued support is necessary.
- [DUAL INTERFACE](#): This is a combination of a Direct Interface and a Dispatch Interface. This most flexible form allows either option to be used, depending upon how the calling application is implemented.
- [AUTOMATION](#): This is a special calling convention, defined by MS later in the evolution of COM and objects. An Automation object is simply one which adheres to the rules for Automation COM Objects. It may offer just a direct interface, just a Dispatch interface, or both of them (DUAL). It should be noted that some programmers use the word AUTOMATION to mean DISPATCH. Even though that's not correct, you should keep the possibility in mind whenever you encounter the term. Automation Methods must use parameters, return values, and assignment variables which are AUTOMATION compatible: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). A [User Defined Type](#) used as a return value or parameter will be converted to a BYVAL

DWORD. All Automation Methods return a hidden result code which is called the HRESULT. This is not really a handle, as the name suggests, but a result code to report the success or failure of a call to a METHOD or PROPERTY.

- [IUNKNOWN](#): This is the name of a special interface which is the basis for every object. It has three methods, which are always defined as the first three methods in every interface. These 3 methods are used by compilers (Classic PowerBASIC or others) to look up other interfaces on the object, and to keep track of usage of this object. While IUNKNOWN is mandatory for every object, you won't ever need to reference it directly. Classic PowerBASIC handles all those messy details automatically.
- [OBJECT REFERENCE](#): This is a reference (a pointer) to an object, which is the only way objects are used. In Classic PowerBASIC, an object variable initially contains [NOTHING](#). When you create an object, or duplicate one, a reference to that object is placed in an object variable by the compiler. That is, a pointer to the object is automatically inserted in the object variable. It is now considered to contain an OBJECT REFERENCE until such time as the reference is deleted or set to NOTHING.
- [COMPONENT](#): An object that encapsulates code and data, providing a set of publicly available services.
- MONIKER: An object that implements the IMoniker interface. A moniker acts as a name that uniquely identifies a COM object. In the same way that a path identifies a file in the file system, a moniker identifies a COM object in the directory namespace.

See Also

[What is an object, anyway?](#)

[What are the parts of an object?](#)

[What does a Class look like?](#)

[What is a Base Class?](#)

What does a Class look like?

[Top](#) [Previous](#) [Next](#)

Here is the Classic PowerBASIC source code for a very simple [class](#). It has just one interface and one instance variable.

```
CLASS MyClass
  INSTANCE Counter AS LONG
  INTERFACE MyInterface
    INHERIT IUNKNOWN          ' inherit the base class
    METHOD BumpIt(Inc AS LONG) AS LONG
      Temp& = Counter + Inc
      INCR Counter
    METHOD = Temp&
  END METHOD
END INTERFACE
' more interfaces could be implemented here
END CLASS
```

Just like other blocks of code in Classic PowerBASIC, a class is enclosed in the [CLASS](#) statement and the [END CLASS](#) statement. Every class is given a text name (in this case "MyClass") so it can be referenced easily in the program.

The [INSTANCE](#) statement describes INSTANCE [variables](#) for this class. Each [object](#) you create from this class definition contains its own private set of any INSTANCE variables. So, if you had a SHIRT class, you might have an instance variable named COLOR, among others. Then, if you create two objects from the class, the COLOR instance variable in the first object might contain WHITE, while the second might be BLUE.

Next comes the [INTERFACE](#) and [END INTERFACE](#) statements. They define the one interface in this class, and they enclose the [methods](#) and [properties](#) in this interface. Every interface is given a text name (in this case "MyInterface") so it can be referenced easily in the program. You could add any number of additional interfaces to this class if it suited the needs of your program.

The first statement in every Interface Block is the [INHERIT](#) statement. As you learned earlier, every interface must contain the three methods of [IUNKNOWN](#) as its first three methods. In this case, INHERIT is a shortcut, so you don't have to type the complete definitions of those methods in every interface. There are more complex (and powerful) ways to use INHERIT as well, but more about that later.

Finally, we have the [METHOD](#) and [END METHOD](#) statements. They are just about identical to a [FUNCTION](#) block, but they may only appear in an interface. In this case, the METHOD is named "BumpIt". It takes one ByRef parameter, and returns a [long integer](#) result.

How do you reference this object?

```
FUNCTION PBMAIN()
  DIM Stuff AS MyInterface
  LET Stuff = CLASS "MyClass"
  x& = Stuff.BumpIt(77)
END FUNCTION
```

The first line of [PBMain](#) ([DIM](#)...) defines an object variable for an interface of the type "MyInterface". The [LET](#) statement creates an object of the CLASS "MyClass", and assigns

an object reference to the object variable named "Stuff". The next line tells Classic PowerBASIC that you want to execute the method "Bumplt", and assign the result to the variable "x&". It's just that simple!

See Also

[What is an object, anyway?](#)

[What is a Base Class?](#)

[What does an Interface look like?](#)

[Just what is COM?](#)

What is a Base Class?

[Top](#) [Previous](#) [Next](#)

The term "Base Class" is truly a misnomer, since it's actually an [interface](#). The truth is, this term probably originated from those who use a programming language which supports only one interface per [class](#). (Note: Classic PowerBASIC allows an unlimited number of interfaces.) On those limited platforms, the distinction between a class and an interface tends to blur. However, since the term "Base Class" enjoys fairly wide usage already, it's probably best if we just learn to live with it and love it.

Every Classic PowerBASIC interface must ultimately derive from the [IUnknown](#) interface, since it provides information about an [object](#) that the compiler must have to manage these affairs accurately. [Previously](#), we discussed the concept of adding INHERIT IUNKNOWN as the first line of every Interface Block. In that way, Classic PowerBASIC just inserts the necessary source code for you, so that the new interface you are creating will derive all the functionality of IUNKNOWN, but still save you from all of that typing. What we didn't tell you at first was that there are really 3 System Base Classes in Classic PowerBASIC. The other two can be used, because they, too, are derived from IUNKNOWN.

So, the real definition of a Base Class is "The interface from which a newly created interface is derived". To implement any of the system interfaces, you would just use [INHERIT](#) followed by the Base Class name as the first line of the interface block. They are:

INHERIT IUNKNOWN

If this option is chosen, your [methods](#) may only be accessed using a [Direct Interface](#), the most efficient form of access. It uses the STDCALL calling conventions, and uses return value conventions normally associated with C++. This style of Base Class is also known as a CUSTOM INTERFACE, so you can use "INHERIT CUSTOM" in place of "INHERIT IUNKNOWN" if that's more comfortable for you.

INHERIT IAUTOMATION

If this option is chosen, your methods may only be accessed using a Direct Interface, the most efficient form of access. It uses the STDCALL calling conventions, and uses return value conventions involving a hidden parameter on the [stack](#). Automation Methods must use parameters, return values, and assignment variables which are AUTOMATION compatible: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). A [User Defined Type](#) used as a return value or parameter will be converted to a BYVAL DWORD. All Automation Methods return a hidden result code which is called the HRESULT. This is not really a handle, as the name suggests, but a result code to report the success or failure of a call to a [METHOD](#) or [PROPERTY](#). "AUTOMATION" is a synonym for "IAUTOMATION", so you can substitute "INHERIT AUTOMATION" in your code if that's more comfortable for you. Automation Interfaces have become more popular than Custom Interfaces in recent times, likely due to availability of the HRESULT hidden result code.

INHERIT IDISPATCH

If this option is chosen, Classic PowerBASIC will automatically create a DUAL Interface for you. That means your methods can be accessed using a Direct Interface (using Automation conventions described above), or the slower [DISPATCH](#) Interface, if that's what is needed. This is certainly the most flexible Base Class, and the only one which should be used if your methods will be accessed by code from a programming language other than Classic PowerBASIC. In a DUAL interface, both forms return the HRESULT hidden result to report the success or failure of the operation. You may use the term "INHERIT DUAL" in place of "INHERIT IDISPATCH", if that's more comfortable for you. While a class may have any number of direct interfaces, only one DUAL or IDISPATCH interface is allowed.

See Also

[What is an object, anyway?](#)

[What does a Class look like?](#)

[What does an Interface look like?](#)

[Just what is COM?](#)

What does an Interface look like?

[Top](#) [Previous](#) [Next](#)

An [INTERFACE](#) is a definition of a set of [methods](#) and [properties](#) which may be implemented on an [object](#). Think of it as much like a [TYPE](#) declaration, except that it contains Method and Property declarations instead of member variables. One interface definition may be used in many different [classes](#) and objects.

An Interface may appear in two general forms: the declaration form and the implementation form.

In the declaration form, the Interface just provides the "signature" of the member methods, without any other source code:

```
INTERFACE MyInterface
  INHERIT IAutomation
  METHOD Method1(parm AS LONG)
  PROPERTY GET Prop1() AS STRING
  PROPERTY SET Prop1(BYVAL TEXT AS STRING)
END INTERFACE
```

This type of declaration interface can be used to provide a description of external interfaces, which you plan to access through [COM](#) services, or just as additional self-documentation of internal code.

In the implementation form, it is part of a CLASS definition, so it contains the complete source code to implement each of the member Methods and Properties.

```
CLASS AnyClass
  INTERFACE AnyInterface
    INHERIT IAutomation
    METHOD Method1(parm AS LONG)
      CALL abc(parm)
    END METHOD

    METHOD Method2(parm AS LONG)
      CALL abc(Parm*1)
    END METHOD
  END INTERFACE
END CLASS
```

In this case, you have the complete definition of an object, with code implemented so the methods can be called and executed.

The first entry in every INTERFACE block must be the [base class](#) upon which it is built. In Classic PowerBASIC, you choose one of the System Base Classes ([IUnknown](#), [IAutomation](#), or [IDispatch](#)), or you might decide to inherit a User Base Class instead.

```
INTERFACE CustomIface
  INHERIT IUNKNOWN
  METHOD MethodDef()...
END INTERFACE
```

The above code defines a custom interface whose methods are available for direct access only. It uses custom calling conventions and does not support an HRESULT ([OBJRESULT](#)) return value.

```
INTERFACE AutoIface
  INHERIT IAutomation
  METHOD MethodDef()...
```

```
END INTERFACE
```

The above code defines an automation interface whose methods are available for direct access only. It uses automation calling conventions and always supports an HRESULT (OBJRESULT) return value. The above two forms will typically be used for internal objects, since they offer the best performance. Every Classic PowerBASIC interface and every COM interface must ultimately inherit from IUnknown. As required base classes, the IUnknown and IAutomation declarations are built into the Classic PowerBASIC Compiler.

```
INTERFACE DispatchIface
    INHERIT IDISPATCH
    METHOD MethodDef()...
END INTERFACE
```

The above code defines a dual interface whose methods are available for both direct access and Dispatch access. This is the form you will typically use for COM objects, since it offers the best compatibility with varied client modules.

Every method and property in a dual interface needs a positive, [long integer](#) value to identify it. That integer value is known as a DispID (Dispatch ID), and it's used internally by COM services to call the correct function on a Dispatch interface. You can specify a particular DispID by enclosing it in angle brackets immediately following the Method/Property name in an Interface definition block.

```
INTERFACE DualIface
    INHERIT IDISPATCH
    METHOD MethodOne <76> ()
    METHOD MethodTwo <77> ()
END INTERFACE
```

If you don't specify a DispID, Classic PowerBASIC will assign a random value for you. This is fine for internal objects, but may cause a failure for published COM objects, as the DispID could change each time you compile your program.

See Also

[What is an object, anyway?](#)

[What does a Class look like?](#)

[What is a Base Class?](#)

[Just what is COM?](#)

Just what is COM?

[Top](#) [Previous](#) [Next](#)

COM is an acronym. It represents the words "Component Object Model".

The short answer is that COM defines a way to communicate between modules of code. The slightly longer answer follows.

You should know that every [object](#) created or defined in Classic PowerBASIC is fully compatible with the COM specification. Many popular compilers are not able to make that claim accurately. The COM specification defines a standardized method of communication between modules of code (frequently called [components](#)), regardless of the platform or the tool used to create them. COM components are reusable chunks of code and associated data, which may be accessed by other "COM-Aware" components and applications.

One of the most frustrating things about this technology has been the ever-changing list of buzz-words used to describe it. We've evolved through OLE, VBX, and OCX, to COM, ActiveX, and more. Though nuances of difference abound, the important thing to remember is that COM and ActiveX describe a means of accessing code and data located outside of the current module. COM+ refers to some extensions which are specific to Win2000, WinXP, and WinVista. Throughout this discussion, we'll use the terms COM Object and ActiveX Object to describe components: reusable chunks of code and associated data.

Prior versions of Classic PowerBASIC introduced client COM services, which were accessible through the COM [DISPATCH](#) interface. While the DISPATCH interface is very flexible and easy-to-use, that very flexibility adds a level of overhead which is unacceptable for many applications. This version of Classic PowerBASIC adds the capability to create and access COM objects through a [DIRECT INTERFACE](#) or a [DISPATCH INTERFACE](#).

All objects in Classic PowerBASIC, COM or not, follow all the guidelines and implementation rules established for COM Objects. This simplifies usage by the programmer, yet adds no measurable overhead at run-time. Classic PowerBASIC encapsulates all the low-level details of the actual COM communication process. This provides a straightforward way to load and communicate with a COM component using BASIC syntax. You'll find that the Classic PowerBASIC object implementation is very efficient, with virtually no degradation of execution speed as compared to standard [Subs](#) and [Functions](#).

See Also

[What is an object, anyway?](#)

[What is a COM component?](#)

[How do you publish an object?](#)

[How are GUID's used with objects?](#)

What is a COM component?

[Top](#) [Previous](#) [Next](#)

A [COM](#) component is commonly referred to as a COM [Object](#). We can visualize a COM component or Object as simply a "black box" that comprises a set of [methods](#) and associated data. Internally, these Objects contain reusable code (Methods), and provide ways for an application to call the object's Methods and read/write its associated data through its [Interfaces](#). Notice that this is the same definition as an object internal to your program. The difference is that COM offers a way to perform this functionality on an object external to your program.

A COM Component is generally known as a COM SERVER, because it serves up information or actions requested by a COM CLIENT. A COM SERVER makes its Methods and [Properties](#) public, so that a COM CLIENT can call them as needed.

COM Components usually take the form of an EXE, or [DLL](#)/OCX file, but the actual file extension is largely irrelevant. However, DLL/OCX versions of a component are generally referred to as "in-process", since they are loaded into the address space of the calling application. EXE-versions are typically "out-of-process" because they will not share the address space of the calling application.

To summarize, a COM Object (COM Server) is a special form of code library (similar to a standard DLL) that conforms to the COM specification. It provides at least one public interface, and is identified by a globally unique [PROGID](#) and [CLSID](#).

Every class is associated with a [GUID](#) (a 128-bit number or string) which uniquely identifies this particular [class](#) from all others, anywhere in the world. This identifier is called the Class ID, or CLSID. A friendlier version of the CLSID is a shorter text name, which also identifies the Class. This text name is known as the PROGID, though it's possible this PROGID may not be totally unique. As it's a simpler construct, it might be duplicated in another program. These identifiers are stored in the Windows Registry when the COM component is installed and registered. Classic PowerBASIC programmers reference COM components by their PROGID string, and rarely by their CLSID. However, since these two items are stored in pairs, it is straightforward to retrieve the matching PROGID for a known CLSID, and vice versa.

As mentioned earlier, you don't need to know how a television works internally to use it and benefit from it. Likewise, you don't need to know how a COM Object works internally to use it and benefit from it. You only need to know the intended job of the object, and how to communicate with it from the outside. The internal workings are well "hidden", which is called encapsulation. Since we aren't able to "look inside" a COM Object, it's not possible to directly manipulate internal variables or memory. This provides a increased level of security for the internal code and data.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

How do you publish an object?

How are GUID's used with objects?

How do you publish an object?

[Top](#) [Previous](#) [Next](#)

Publishing an [object](#) means making it available for access and use by other applications through the facilities of the [COM](#) Services of Windows. With some compilers, this requires pages upon pages of code. With Classic PowerBASIC, you'll find it's fairly straightforward. Just add a Class Id (CLSID) [GUID](#) and the words "AS COM" to the end of the [CLASS](#) statement. Then, add an Interface ID (IID) to the end of the [INTERFACE](#) statement. Believe it or not, that's just about it!

```
$MyClassGuid = GUID$("{00000099-0000-0000-0000-000000000008}")
$MyIfaceGuid = GUID$("{00000099-0000-0000-0000-000000000009}")
```

```
CLASS MyClass $MyClassGuid AS COM
  INTERFACE MyInterface $MyIfaceGuid
    INHERIT IAutomation
    METHOD Method1(parm AS LONG)
      CALL abc(parm)
    END METHOD
  END INTERFACE
END CLASS
```

Classic PowerBASIC handles all the messy details of COM for you. The name of the CLASS (in this case MyClass) will be used as the [ProgID](#) for COM registration of the [DLL](#). The GUID's you selected will be used for the [CLSID](#) and IID, so you're ready to go...

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

[How are GUID's used with objects?](#)

What is inheritance?

Inheritance is all about code reuse. You can reuse the definitions of an [interface](#), or you can reuse complete sections of code.

INTERFACE INHERITANCE is defined by [COM](#) standards, and available for use by any COM [object](#). This form of inheritance applies only to the definition of each item in an interface, rather than the underlying code. Interface inheritance gives you the option to use one interface in multiple [classes](#) (objects). Because the interface definition remains identical in each instance, you can often use the identical (or similar) code to manipulate different objects. With this form of inheritance, the programmer must provide appropriate code for each of the [Methods](#) and [Properties](#) in every implementation of the interface.

IMPLEMENTATION INHERITANCE is the process whereby a CLASS derives all of the functionality of an interface implemented elsewhere. That is, the derived class now has all the methods and properties of this new, extended version of a [Base Class](#)! This form of inheritance is offered by Classic PowerBASIC, even though it is not required by the COM Specification.

You can extend the functionality of an interface you created earlier by adding new methods and properties to the derived interface/class. The syntax for adding extra methods (not in the Base Class) is the same as adding methods to a standard class -- just add methods and properties, as always.

You can add to, or replace, the functionality of a particular method or property by coding a replacement which is preceded by the word [OVERRIDE](#). The overriding method must have the same name and signature (parameters, return value, etc.) as the one it replaces. When you implement a new method in a derived class, you may call a method in the Base Class by using the pseudo-object [MYBASE](#). This allows you to extend the original functionality, or replace it entirely.

Inheritance is implemented by use of the INHERIT statement within an [INTERFACE / END INTERFACE](#) block. The word INHERIT is followed by the class name and interface name of the code to be inherited. Both are necessary, because COM allows you to have multiple implementations of any particular interface.

```
CLASS MyClass
  INTERFACE MyFace
    INHERIT IDISPATCH
    METHOD aaa()
      ' code...
    END METHOD
    METHOD bbb()
      ' code...
    END METHOD
    METHOD ccc()
      ' code...
    END METHOD
    METHOD ddd()
      ' code...
    END METHOD
  END INTERFACE
```

```
END CLASS
```

```
CLASS TheClass
  INTERFACE TheFace
    INHERIT MyClass, MyFace
    OVERRIDE METHOD bbb()
      ' new code
    END METHOD
    OVERRIDE METHOD ddd()
      ' new code
    END METHOD
    METHOD xxx()
    END METHOD
  END INTERFACE
END CLASS
```

Note that the derived interface "TheFace" first inherits IDISPATCH, and then, all four methods from "MyFace" (aaa,bbb,ccc,ddd). However, because of the OVERRIDE statements, both bbb() and ddd() are replaced by newer versions of these methods. Note that a derived class may be inherited by yet another class, repetitively. The depth of this inheritance is limited only by available memory.

The pseudo-object MYBASE may be used within a derived class to access a method in the original base class. For example, if you placed:

```
MyBase.bbb()
```

in the above derived code, it would execute the method bbb() in the parent interface/class. You could then use the results to extend or modify actions in your newer code.

When you inherit an interface, the inherited [constructor and destructor](#) methods (CREATE and DESTROY) are disabled, in case you wish to change their functionality in the derived interface. If you wish to execute them as-is, you can simply add [MYBASE.CREATE](#) and/or [MYBASE.DESTROY](#) in the derived CREATE/DESTROY methods.

See Also

[What is an object, anyway?](#)

[What does an Interface look like?](#)

[Just what is COM?](#)

[How do you create an object?](#)

How do you create an object?

[Top](#) [Previous](#) [Next](#)

This operation is frequently known as "Creating an INSTANCE of an [OBJECT](#)." Yes, this is just one more buzz-word -- but you'll hear it frequently.

In order to create an object, you first need an OBJECT VARIABLE. This object variable can be located most anywhere in your program, and have any scope: [LOCAL](#), [GLOBAL](#), [THREADED](#), etc. This object variable is declared by using the name of the [interface](#) you wish to access on the object. This is done so that Classic PowerBASIC knows which [Methods](#) can be called via this variable. This variable is expected to be a "container" for an [OBJECT REFERENCE](#) (that is, a pointer to the actual object). Initially, this variable is automatically set to "[NOTHING](#)". If you wish to use the generic DISPATCH interface to access the object, you would use the name [IDISPATCH](#) instead.

```
LOCAL object1 AS MyInterface
LOCAL object2 AS IDISPATCH
```

There is actually one more special case, that of an [IDBIND DISPATCH](#) interface. Since object creation works the same on those interfaces, as well, we'll have more on that special topic in a later section. So, now that you have two empty object variables, what do you do with them? Use the assignment statement ([LET](#)) to create an object!

To create an object, you need to specify a [CLASS](#) and an INTERFACE. The interface is implied by the object variable you use, so it only remains that you specify the CLASS name. If the requested CLASS is internal to your program, use the word CLASS:

```
LET object1 = CLASS "MyClass"
```

The class name ("MyClass") must be specified as a quoted [string literal](#), which is the name of a class implemented within the program. Since the class is internal (the name is known at compile-time), you may not use a string variable or expression. Upon execution, a new object is created, and a reference to that object is assigned to the object variable *object1*. The interface requested is determined by the original declaration of *object1*. If the interface name is DISPATCH, you can call the methods with the [OBJECT](#) statement -- otherwise, regular Method and Property references are used for direct interfaces.

```
LET objvar = NEWCOM PROGID$
LET objvar = GETCOM PROGID$
LET objvar = ANYCOM PROGID$
```

This form of the LET statement is used to obtain an object reference external to the program using the [COM](#) facilities of Windows. If the requested object is in a [DLL](#) (an in-process server), you will always use the [NEWCOM](#) option, as you're asking Windows to supply a new object. If the request is successful, an OBJECT REFERENCE (a pointer to the object) is assigned to the *objvar*.

If the requested object is in an EXE (out-of-process server), you may use any of the three options. If the director word NEWCOM is specified, a new instance of a COM application is created. With [GETCOM](#), an interface will be opened on an existing, running application, which has been registered as the active automation object for its class. With [ANYCOM](#), the compiler will first try to use an existing, running application if available, or a new

instance if not.

Of course, as with any other [LET](#) (assignment) statement, you are free to simply omit the word LET entirely.

If an object creation or assignment fails for any reason, the object variable is set to NOTHING. If this statement fails, no errors are generated, nor is an [OBJRESULT](#) set. You should test for success of the operation with [ISOBJECT](#)(*objvar*) before trying to use the object or execute its methods.

But what about the rare case when there's no [ProgID\\$](#) available? There's an answer for that, too.

```
LET objvar = NEWCOM CLSID ClassID$  
LET objvar = GETCOM CLSID ClassID$  
LET objvar = ANYCOM CLSID ClassID$
```

This new form also obtains a COM object reference, just as in the previous example. However, it is only used in the unusual case of a COM Object which has no ProgID. It works exactly as the original form above, except that it describes the requested object by its 16-byte [GUID](#) which is the ClassID of the object.

```
LET objvar = NEWCOM CLSID ClassID$ LIB DLLPath$
```

Classic PowerBASIC offers the unique ability to create and reference COM objects without any reference to the registry at all. As long as you know the [CLSID](#) (Class ID) and the file path/name of the DLL to be accessed, you can do so with no registry access at all. You don't need a special type of COM server. This technique can be used with any server, whether created by Classic PowerBASIC or another compiler. By using this method of object creation, there is simply no need for the server to be registered at all. That allows you to keep local copies of the COM servers you use, with no chance they will be altered or replaced by another application. You use the above form, where the clause "CLSID ClassID\$" identifies the 16-byte Class ID, and the clause "LIB DLLPath\$" identifies the file path and file name of the COM Server. Once you've obtained the COM object reference in objvar, it is used exactly as you would with a traditional object.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you duplicate an object variable?](#)

[How do you call a Direct Method?](#)

How do you duplicate an object variable?

[Top](#) [Previous](#)
[Next](#)

In the [previous section](#), you learned to create an [object](#), which assigns an OBJECT REFERENCE to the object variable:

```
LOCAL object1 AS MyInterface
LET object1 = CLASS "MyClass"
```

What if you need to duplicate it? Well, you first must decide whether you want to create a completely new object, or if you just want a second object [variable](#) which points the same object. This is a very important distinction. With two objects, they each have their own set of [INSTANCE](#) variables. The variables in each set remain independent of the other set until they are destroyed. You would create two objects by writing:

```
LOCAL object1, object2 AS MyInterface
LET object1 = CLASS "MyClass"
LET object2 = CLASS "MyClass"
```

If you have two object variables pointing to the same object, they would share the same set of INSTANCE variables. You would create two OBJECT REFERENCES to one OBJECT by writing:

```
LOCAL object1, object2 AS MyInterface
LET object1 = CLASS "MyClass"
LET object2 = object1
```

Of course, now we can take this one step further. You already know that an OBJECT may have two (or even more) [interfaces](#) defined in a [CLASS](#). How would you actually use two interfaces on the same object? Just declare an object variable for each interface, much like:

```
LOCAL object1 AS MyInterface
LOCAL object2 AS HisInterface
LET object1 = CLASS "MYClass"
LET object2 = object1
```

The code is very much like the preceding example, except that the two object variables are declared as two different interfaces. When the last line is executed, Classic PowerBASIC looks at the object variables to determine if they represent the same interface or not. If they do, it simply creates an extra variable, pointing to the same object. If they differ, Classic PowerBASIC checks object to ensure the new interface is supported. If so, it creates a new OBJECT REFERENCE via the new interface, and assigns it to *object2*. It's just that simple!

The final issue in this topic is how to destroy an object variable. Generally speaking, you do nothing at all. When an object variable goes out of [scope](#), Classic PowerBASIC will handle all the messy details for you. For the most part, just forget about it. However, in the rare case that you need to destroy an object variable at a specific time and place, you can do so with the following statement:

```
object1 = NOTHING
```

Setting an object variable to [NOTHING](#) handles it all for you.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you create an object?](#)

[How do you call a Direct Method?](#)

How do you call a Direct Method?

[Top](#) [Previous](#) [Next](#)

First, you should remember that [INSTANCE](#) variables may only be accessed from within the [object](#). The only way to access them from the "outside", is by a parameter or return value of a [METHOD](#) or [PROPERTY](#) function. Of course, Methods and Properties may also utilize the other data [scopes](#): [Global](#), [Local](#), [Static](#), and [Threaded](#).

In Classic PowerBASIC, the basic unit of code in an [object](#) is the METHOD. A METHOD is a block of code, very similar to a user-defined function. Optionally, it can return a value, like a [FUNCTION](#), or merely act as a subroutine, like a [SUB](#). Methods are implemented when you write:

```
METHOD NAME [ALIAS "altname"] (var AS type...) [AS TYPE]
[statements]
METHOD = expression
END METHOD
```

Methods can only be called through an object variable, which is an integral part of the calling syntax. The object variable must be valid, that is, it must contain a valid [object reference](#) which was assigned to it with the [LET](#) statement. If you attempt to call a method on a null object, you'll likely experience a GPF and a total failure of your program. Methods may be called by writing:

```
DIM ObjVar AS MyInterface
LET ObjVar = CLASS "MyClass"
```

```
[CALL] objvar.Method1(param)
```

Note the word [CALL](#) is optional. This example shows how to call "Method1" when "Method1" does not return a value. If it did have a return value, use this form instead:

```
var = ObjVar.Method1(param)
```

A PROPERTY is a special type of METHOD, which is only designed to GET or SET INSTANCE data in an object. While the work of a PROPERTY could readily be accomplished with a standard METHOD, this distinction is convenient to emphasize the concept of encapsulation of INSTANCE data within an object. There are two forms of PROPERTY procedures, PROPERTY GET and PROPERTY SET. As implied by the names, the first form is used to retrieve a data value from the object, while the second form is used to assign a value. Properties are implemented:

```
PROPERTY GET NAME [ALIAS "altname"] (BYVAL var AS type...) [AS TYPE]
[statements]
PROPERTY = expression
END PROPERTY
```

```
PROPERTY SET NAME [ALIAS "altname"] (BYVAL var AS type...)
[statements]
variable = value
END PROPERTY
```

When you use PROPERTY SET, the last (or only) parameter is used to pass the value to be assigned. A PROPERTY may be considered "Read-Only" or "Write-Only" by simply omitting one of the definitions. However, if both GET and SET forms are defined for a

particular Property, parameters and the property must be identical in both forms, and they must be paired. That is, the PROPERTY SET must immediately follow the PROPERTY GET. It's important to note that all PROPERTY parameters must be declared as BYVAL. Properties can only be called through an object variable, which is an integral part of the calling syntax. The object variable must be valid, that is, it must contain a valid object reference which was assigned to it with the LET statement.

You can access a PROPERTY GET with:

```
DIM ObjVar AS MyInterface
LET ObjVar = CLASS "MyClass"
```

```
var = ObjVar.Prop1(param)
```

You can access a PROPERTY SET with:

```
DIM ObjVar AS MyInterface
LET ObjVar = CLASS "MyClass"
```

```
[CALL] ObjVar.Prop1(param) = expr
```

Note that the choice of Property procedure is syntax directed. In other words, depending upon the way you use the name, Classic PowerBASIC will automatically decide whether the GET or SET PROPERTY should be called.

In every Method and Property, Classic PowerBASIC automatically defines a pseudo-variable named [ME](#), which is treated as a reference to the current object. Using ME, it's possible to call any other Method or Property which is a member of the class:

```
var = ME.Method1(param)
```

Methods and Properties may be declared (using AS type...) to return a string, any of the numeric types, a specific class of object variable (AS MyInterface), a [Variant](#), or a user defined Type.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you create an object?](#)

[What is a Compound Object Reference?](#)

What is a Compound Object Reference?

[Top](#) [Previous](#)
[Next](#)

There is an interesting "shortcut" available to you by using "Compound [Object](#) References". In some cases, you'll find that you can combine two, three, or more method calls into a single line of Classic PowerBASIC source code.

The notion here is that you may need to execute a [METHOD](#) which returns an object variable, just so you can use that temporary object variable to call another method. In fact, you may even find you need to nest this type of operation several levels deep! While this is certainly workable, you may find yourself with a maze of temporary objects and object variables, all of which need to be destroyed at some point.

For example, assuming you have an object variable named MyDBase, which is an instance of the [interface](#) named DataBase. The interface DataBase offers a method named ErrorObject which returns an Errors object. Errors is a second interface, which has a method named Count. Count returns a [long integer](#), to tell the number of errors which have occurred. In order to retrieve Count, you would normally have to write:

```
LOCAL MyErrors AS Errors
LET MyErrors = MyDBase.ErrorObject
ErrorCount& = MyErrors.Count
MyErrors = NOTHING
```

However, with Compound Object References, this can be combined into a single line of code:

```
ErrorCount& = MyDBase.ErrorObject.Count
```

In particular, note that the temporary object called MyErrors is gone completely, since Classic PowerBASIC automatically handles the lifetime of temporary objects. You can even declare the methods and [properties](#) with parameters, if it's appropriate to allow:

```
ErrorCount& = MyDBase.ErrorObject(item&).Count
```

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you create an object?](#)

[What is an HRESULT?](#)

What is an HRESULT?

[Top](#) [Previous](#) [Next](#)

[Methods](#) may optionally have an explicit return value which you specifically declare. However, in addition to this, all [Automation](#) or [Dispatch](#) Methods and Properties have another "Hidden Return Value", which is cryptically named HRESULT. While the name would imply a handle for a result, it's really not a handle at all, but just a [long integer](#) value, used to indicate the success or failure of the Method. After calling a Method or [Property](#), you can retrieve the HRESULT value with the Classic PowerBASICfunction [OBJRESULT](#). The most significant bit of the value is known as the severity bit. That bit is 0 (value is positive) for success, or 1 (value is negative) for failure. The remaining bits are used to convey error codes and additional status information. If you call any [object](#) Method/Property (either Dispatch or [Direct](#)), and the severity bit in the returned HRESULT is set, Classic PowerBASIC generates Run-Time [error 99](#): Object error. When you create a Method or Property, Classic PowerBASIC automatically returns an HRESULT of zero, which implies success. You can return a non-zero HRESULT value by executing a METHOD OBJRESULT = *expr* within a Method, or PROPERTY OBJRESULT = *expr* within a Property.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[How do you create an object?](#)

[How do you register a COM Component?](#)

How do you register a COM Component?

[Top](#) [Previous](#)
[Next](#)

All [COM Components](#) (COM Servers) must be listed in the system registry. A variety of information is kept there, but the most important is the definition of the [PROGID](#) and the [CLSID](#). These are the terms used to uniquely identify the component, so that the operating system can locate them for a client program that wants to use their services. Classic PowerBASIC COM DLL's provide self-registration and unregistration services by automatically exporting two Subs:

```
Declare Function DllRegisterServer    alias "DllRegisterServer"    as long
Declare Function DllUnregisterServer  alias "DllUnregisterServer"  as long
```

You could write a small executable program to call these registration functions, or use the Microsoft registration utility (REGSVR32.EXE) for that purpose. REGSVR32.EXE is included with Windows.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a COM component?](#)

[How do you publish an object?](#)

[What is a Class Method?](#)

What is a Class Method?

[Top](#) [Previous](#) [Next](#)

A CLASS METHOD is one which is private to the [class](#) in which it is located. That is, it may only be called from a [METHOD](#) or [PROPERTY](#) in the same class. It is invisible elsewhere. The CLASS METHOD must be located within a [CLASS](#) block, but outside of any [INTERFACE](#) blocks. This shows it is a direct member of the class, rather than a member of an interface.

```
CLASS MyClass
  INSTANCE MyVar AS LONG

  CLASS METHOD MyClassMethod(BYVAL param AS LONG) AS STRING
    METHOD = "My" + STR$(param + MyVar)
  END METHOD

INTERFACE MyInterface
  INHERIT IUNKNOWN
  METHOD MyMethod()
    Result$ = ME.MyClassMethod(66)
  END METHOD
END INTERFACE
END CLASS
```

In the above example, MyClassMethod() is a CLASS METHOD, and is always accessed using the pseudo-object ME (in this case ME.MyClassMethod). Class methods are never accessible from outside a class, nor are they ever described or published in a type library. By definition, there is no reason to have a private [PROPERTY](#), so Classic PowerBASIC does not offer a CLASS PROPERTY structure.

See Also

[What is an object, anyway?](#)

[What does a Class look like?](#)

[Just what is COM?](#)

[What are Constructors and Destructors?](#)

What are Constructors and Destructors?

[Top](#) [Previous](#)
[Next](#)

There are two special [class methods](#) which you may optionally add to a [class](#). They meet a very specific need: automatic initialization when an [object](#) is created, and cleanup when an object is destroyed. Technically, they are known as constructor and destructor [methods](#), and can perform almost any functionality needed by your object: initialization of variables, reading/writing data to/from disk, etc. You do not call these methods directly from your code. If they are present in your class, Classic PowerBASIC automatically calls them each time an object of that class is created or destroyed. If you choose to use them, these special class methods must be named CREATE and DESTROY. They may take no parameters, and may not return a result. They are defined at the class level, so they may never appear within an [INTERFACE](#) definition.

```
CLASS MyClass
  INSTANCE MyVar AS LONG

  CLASS METHOD CREATE()
    ' Do initialization
  END METHOD

  CLASS METHOD Destroy()
    ' Do cleanup
  END METHOD

  INTERFACE MyInterface
    INHERIT IUNKNOWN
    METHOD MyMethod()
      ' Do things
    END METHOD
  END INTERFACE
END CLASS
```

As displayed above, CREATE and DESTROY must be placed at the class level, before any INTERFACE definitions. You should note that it's not possible to name any standard method (one that's accessible through an interface) as CREATE or DESTROY. That's just to help you remember the rules for a constructor or destructor. However, you may use these names as needed to describe a method external to your program.

A very important caution: You must never [create an object](#) of the current class in a CREATE method. To do so will cause CREATE to be executed again and again until all available memory is consumed. This is a fatal error, from which recovery is impossible.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is a Class Method?](#)

[What is DISPATCH?](#)

What is DISPATCH?

[Top](#) [Previous](#) [Next](#)

The DISPATCH INTERFACE is a slower form of [interface](#), originally introduced as a part of Microsoft Visual Basic. An implementation of [COM](#) DISPATCH support was introduced in a prior version of Classic PowerBASIC. It has now been substantially improved to offer COM SERVER as well as client support, Dual Interfaces, relaxed typing, exception information, and much more.

When you use DISPATCH, the compiler actually passes the name of the [METHOD](#) you wish to execute as a text string. The parameters can also be passed in the same way. The [object](#) must then look up the names, and decide which METHOD to execute, and which parameters to use, based upon the text strings provided. As if that weren't enough, DISPATCH requires that all parameters and return values be passed as [VARIANT](#) variables, with all those conversions the responsibility of the programmer. That's right, you. This is a slow process. However, DISPATCH is flexible, convenient, and forgiving. Further, you'll find that many scripting languages and application use DISPATCH as their sole method of operation, so continued support is absolutely necessary.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[Late Binding](#)

[ID Binding](#)

The standard methodology of [DISPATCH](#) is called "Late Binding", because nothing is done in advance. No method definitions. No Interface signatures. You can pretty much just start writing code:

```
LOCAL DispVar AS IDISPATCH
LET DispVar = NEWCOM "DispProgID"
OBJECT CALL DispVar.Method1(x&, y$)
```

It's just that easy. The first line declares an [object](#) variable which assumes the [DISPATCH interface](#), while the second line creates an object and assigns a reference to DispVar. The third line just executes a [method](#) on the new object.

The [OBJECT](#) statement is always used to execute methods on a DISPATCH interface. This differentiates it from [direct](#) access, so Classic PowerBASIC can handle your request in the appropriate manner.

It's important to note that this version of Classic PowerBASIC relaxes the strict type checking of Dispatch parameters. While DISPATCH interfaces require that all parameters and return values be passed as a [VARIANT](#) variable, this version of Classic PowerBASIC relaxes that requirement for you. You may substitute any [COM](#)-compatible variable, and Classic PowerBASIC will convert them automatically to and from Variant variables as an integral part of the OBJECT statement. How could it get easier?

So, how does this work internally?

Well, each method name is assigned a positive integer number as its Dispatch ID (or DispID), to differentiate it from the other methods. In a similar fashion, each parameter is numbered from 0 - n to identify each of them uniquely. When you execute a statement like:

```
OBJECT CALL DispVar.Method1(x&, y$)
```

Classic PowerBASIC packages up the Method Name (*Method1*) and the names of any named parameters (none in this example - more about that later), and passes them to a special DISPATCH function. After a bit of time for lookup, the Dispatch ID (let's say the number 77) is returned. Classic PowerBASIC then converts the two parameters to Variants and packages the whole thing up to call another special Dispatch function. This tells the server to execute Method number 77 using the two enclosed parameters. Finally, it returns with an HRESULT code to indicate success or failure. That's classic "Late Binding" Dispatch.

"Late Binding" is flexible and easy to use because everything is resolved at run-time. That flexibility comes at a price -- it's the slowest form of COM.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is DISPATCH?](#)

[ID Binding](#)

So, how can we speed things up?

Well, the worst bottleneck is the name lookup, and that's something we can deal with! We usually know all the [METHOD](#) definitions at compile-time. If we can tell the compiler the DispID's and the parameter info at compile-time, one whole step can be eliminated! That's called ID-BINDING of the [Dispatch](#) Interface. We create a simple [IDBIND Interface](#), which is written like this:

```
INTERFACE IDBIND MyDispIfaceName
  MEMBER CALL Method1<77> (WideVal AS LONG, WideText AS STRING)
  MEMBER CALL Method2<78> ()
  MEMBER CALL MethodX<79> ()
END INTERFACE
```

Classic PowerBASIC can use this IDBIND Interface to create faster Dispatch execution. Just create this structure, and place it in your source code prior to any references. Then, when you create an object variable, just use the IDBIND Interface Name instead of DISPATCH:

```
LOCAL DispVar AS MyDispIfaceName
LET DispVar = NEWCOM "DispProgID"
OBJECT CALL DispVar.Method1(abc&, xyz$)
```

"ID Binding" is faster than "Late Binding", but you must supply interface definitions in your source code.

How do you get this information? Most likely from the [Classic PowerBASIC COM Browser](#)! At your convenience, it will scan your system registry, and find any COM objects available. It will create all of the Interface definitions for you with just a click.

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is DISPATCH?](#)

[Late Binding](#)

[Creating a DISPATCH Object](#)

Creating a DISPATCH Object

[Top](#) [Previous](#) [Next](#)

[DISPATCH](#) objects are easy to create. The technique is virtually identical to that for [direct interfaces](#). You must first declare the object variable -- if you wish to use "[Late Binding](#)", you'll use the generic name [IDISPATCH](#).

```
LOCAL DispVar AS IDISPATCH
LET DispVar = NEWCOM "DispProgID"
```

If you wish to use "[ID Binding](#)", you'll use the interface name from your Interface IDBIND structure.

```
LOCAL DispVar AS MyDispIfaceName
LET DispVar = NEWCOM "DispProgID"
```

If all went well, you now have an [object](#)! (And an [object reference](#) in your object variable). Of course, it's always a good idea to use the [ISOBJECT](#)(*DispVar*) function to be certain that the operation was a success. If it failed, an attempt to use the object variable could cause a fatal exception.

See Also

[What is an object, anyway?](#)

[What is DISPATCH?](#)

[Late Binding](#)

[ID Binding](#)

[How do you call a DISPATCH METHOD?](#)

How do you call a DISPATCH METHOD?

[Top](#) [Previous](#)
[Next](#)

To call a Method through the [DISPATCH](#) interface, you will use the [OBJECT](#) statement. This differentiates it from [direct access](#), so Classic PowerBASIC can handle your request in the appropriate manner.

There are five general forms of the OBJECT statement:

OBJECT GET	Retrieve the value of a PROPERTY . This is similar to retrieving the value of a variable.
OBJECT LET	Write a value to a PROPERTY. This is similar to assigning a value to a variable.
OBJECT SET	Write an object reference to a PROPERTY. This is similar to assigning to an object variable.
OBJECT CALL	Call a DISPATCH METHOD . This is equivalent to calling a standard Sub or Function .
OBJECT RAISEEVENT	Call an EVENT METHOD. (Event Methods are fully covered in a later section).

```
OBJECT GET DispVar.Prop1 TO ResultVar
OBJECT LET DispVar.Prop1 = NewValue
OBJECT SET DispVar.Prop1 = NewReference
OBJECT CALL DispVar.Meth1(param1, TEXT=MyStr$)
OBJECT RAISEEVENT EventMeth1
```

All parameters, return values, and assignment values must be in the form of [COM](#)-compatible variables. Literals and expressions are not allowed. COM-compatible variables include [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). You should use caution passing string data since COM Objects assume that Unicode format is used. When string data is contained in a VARIANT variable, conversion to/from Unicode is automatic, and no intervention is needed from the programmer. However, if you pass data in a dynamic string variable, you must use the [ACODES\\$\(\)](#) and [UCODES\\$\(\)](#) functions to convert the data to an appropriate format.

The OBJECT statement can use both positional and named parameters, but you should keep in mind that not all COM Dispatch Servers support named parameters. Positional parameters are universally supported.

A positional parameter is a variable containing an appropriate value. It is identified by its position in the parameter list, just as in a traditional SUB or FUNCTION. A named parameter consists of a parameter identifier (a name), an equal (=) sign, and a variable containing an appropriate value. Positional parameters must precede any and all named parameters, but named parameters may be specified in any sequence.

Each time you call a Method or Property using the OBJECT statement, a status code is

returned in a hidden parameter to indicate the success or failure of the operation. You can retrieve information about this status code with the [OBJRESULT](#) function, and also by using the [IDISPINFO](#) Dispatch Information Object. If the failure was severe, then a Classic PowerBASIC [error 99](#) (Object Error) is also generated and the [ERR](#) system variable is set. You can find more information about these items by referring to [OBJRESULT](#), [IDISPINFO](#), and [ERR](#).

See Also

[What is an object, anyway?](#)

[What is DISPATCH?](#)

[Late Binding](#)

[ID Binding](#)

[What are Connection Points?](#)

What are Connection Points?

[Top](#) [Previous](#) [Next](#)

Generally speaking, a client module calls a server module to perform specific operations as they are needed. However, in many situations, it's convenient and efficient for a server to notify its client of a condition or event immediately, without forcing the client to inquire about the status. At the appropriate time, the server calls back to a client [method](#), passing information via the method parameters. This is the exact opposite of normal communication, because the server module is now calling the client module. In effect, the client is acting as a server for the purpose of handling these events. In the world of [objects](#), a server which can call such "Event Methods" is said to offer a "Connection Point". A Connection Point can be used with [COM](#) objects or internal objects. Further, it may use either a [direct interface](#) or the [DISPATCH](#) interface. Event methods may take parameters, but may not return a result.

In COM terminology, a server which offers a Connection Point is known as an "Event Source". A client which can attach to a Connection Point and handle events is known as an "Event Sink" or "Event Handler". The terms source and sink are analogous to the electrical engineering terms source and sink.

Perhaps you have a server object which performs complex arithmetic, and may take quite some time to finish. You'd like to notify the client of your progress towards completion at regular intervals. In that way, the client can continue other work, or just notify the user of the status. If a server object offers a Connection Point, it must declare the event interface:

```
INTERFACE STATUS $StatusGuid AS EVENT
    INHERIT IUNKNOWN
    METHOD Progress(Percent AS LONG)
END INTERFACE
```

Finally, the server class must include a declaration of the event interfaces it supports via a Connection Point by adding one or more [EVENT SOURCE](#) statements within the [class](#) definition:

```
EVENT SOURCE STATUS
EVENT SOURCE DISPATCH
```

Each server class created by Classic PowerBASIC may offer up to four event interfaces. A client module may subscribe to any or all of these event interfaces. When it's time for the server object to notify the client of an event, the [RAISEEVENT](#) statement is used. For the Dispatch interface, [OBJECT RAISEEVENT](#) is used instead. RAISEEVENT may only appear within a class which declares the Event Source interface. The concept of RAISEEVENT is very similar to the [CALL](#) statement, but it may only be used to execute event methods:

```
RaiseEvent Status.Progress(10) ' advise the code is 10% done
```

It should be noted that RaiseEvent does not reference an object variable at all, because it calls any and all Event Methods which are currently attached to the Connection Point. Instead, it references the interface name (in this case "Status"), followed by the name of the Event Method to be executed (in this case "Progress").

The client may choose to support the event by creating the appropriate event code (it must

precisely match the declaration in the server), or the client could just ignore the event completely. If supported, the client must have an event method to handle the event, and create an event object to do so. In effect, the client actually becomes an object server for this one purpose. The client code might be something like:

```
CLASS EventClass AS EVENT
    INTERFACE STATUS AS EVENT
        INHERIT IUNKNOWN
        METHOD Progress(Percent AS LONG)
            CALL DisplayIt(Percent)
        END METHOD
    END INTERFACE
END CLASS
```

In addition, the client must initiate a connection to the server with [EVENTS FROM](#), and disconnect when done with [EVENTS END](#):

```
DIM oEvent AS STATUS
oEvent = CLASS "EventClass"
EVENTS FROM MyObject CALL oEvent

' execute some code here...

EVENTS END oEvent
```

A Connection Point may be attached to one Event Method, multiple Event Methods, or no Event Method at all. Whenever a RAISEEVENT statement is executed, all Event Methods attached to the source object are called, one after another. There is no guarantee of the sequence of the calls, and you must consider the possibility that RAISEEVENT with a ByRef parameter could change the value of a parameter variable before any particular Event Method is executed.

Here is a complete program which demonstrates the execution of a Connection Point in a single, self-contained application. It uses only internal objects. Since the objects are all internal, it is not necessary to assign a [GUID](#) to each class and interface.

```
#COMPILE EXE

CLASS EvClass AS EVENT
    INTERFACE Status AS EVENT
        INHERIT IUNKNOWN
        METHOD Done
            MSGBOX "Done!"
        END METHOD
    END INTERFACE
END CLASS

CLASS MyClass
    INTERFACE MyMath
        INHERIT IUNKNOWN
        METHOD DoMath
            MSGBOX "Calculating..." ' Do some math calculations here
            RAISEEVENT Status.Done()
        END METHOD
    END INTERFACE

    EVENT SOURCE Status

END CLASS
```

```

FUNCTION PBMAIN()
    DIM oMath AS MyMath, oStatus AS Status
    LET oMath = CLASS "MyClass"
    LET oStatus = CLASS "EvClass"

    EVENTS FROM oMath CALL oStatus
    oMath.DoMath
    EVENTS END oStatus
END FUNCTION

```

Here is a set of programs which demonstrate the execution of a Connection Point using a COM SERVER and a COM CLIENT. It uses an in-process COM server ([DLL](#) created with PB/Win 9), and a COM CLIENT as an executable program. First the COM SERVER:

```

#COMPILE DLL "EvServer.dll"

$EvIFaceGuid = GUID$("{00000098-0000-0000-0000-000000000002}")
$MyClassGuid = GUID$("{00000098-0000-0000-0000-000000000003}")
$MyIFaceGuid = GUID$("{00000098-0000-0000-0000-000000000004}")

INTERFACE Status $EvIFaceGuid AS EVENT
    INHERIT IUNKNOWN
    METHOD Done
END INTERFACE

CLASS MyClass $MyClassGuid AS COM
    INTERFACE MyMath $MyIFaceGuid
        INHERIT IUNKNOWN
        METHOD DoMath
            MSGBOX "Calculating..." ' Do some math calculations here
            RAISEEVENT Status.Done()
        END METHOD
    END INTERFACE

    EVENT SOURCE Status

END CLASS

```

Next the COM CLIENT:

```

#COMPILE EXE "EvClient.exe"

$EvClassGuid = GUID$("{00000098-0000-0000-0000-000000000001}")
$EvIFaceGuid = GUID$("{00000098-0000-0000-0000-000000000002}")
$MyIFaceGuid = GUID$("{00000098-0000-0000-0000-000000000004}")

CLASS EvClass $EvClassGuid AS EVENT
    INTERFACE STATUS $EvIFaceGuid AS EVENT
        INHERIT IUNKNOWN
        METHOD Done
            MSGBOX "Done!"
        END METHOD
    END INTERFACE
END CLASS

INTERFACE MyMath $MyIFaceGuid
    INHERIT IUNKNOWN
    METHOD DoMath
END INTERFACE

FUNCTION PBMAIN()

```

```
DIM oMath AS MyMath
DIM oStatus AS STATUS

LET oMath = NEWCOM "MyClass"
LET oStatus = CLASS "EvClass"

EVENTS FROM oMath CALL oStatus
oMath.DoMath
EVENTS END oStatus
END FUNCTION
```

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[Enumerating Collections](#)

[What are Type Libraries?](#)

Enumerating Collections

[Top](#) [Previous](#) [Next](#)

A collection is simply a set or group of items, where each can be accessed through its own [Interface](#). For example, Microsoft Word can have multiple documents open at the same time, and it can provide an Interface reference for each open document.

Therefore, enumerating a collection is simply a matter of determining the number of items in the collection, looping through and retrieving the appropriate information for one or more Interface members of the collection.

We'll start off with the Visual Basic syntax and show how to perform the same kind of task with Classic PowerBASIC.

Visual Basic syntax for enumerating a collection looks something like this:

```
Dim Item As InterfaceItem
Dim Items As InterfaceItemsCollection
[statements]
For Each Item In Items
    'do something with the Item.member Method/Property, e.g.,
    var$ = Item.StringProp
Next
```

In Classic PowerBASIC, we can perform the same enumeration. For example:

```
DIM oItem AS InterfaceItem
DIM oItems AS InterfaceItemsCollection
[statements]
OBJECT GET oItems.Count TO c&
FOR Index& = 1 TO c&
    OBJECT GET oItems.Item(Index&) TO oItem
    'do something with the Item.member Method/Property, e.g.,
    OBJECT GET oItem.StringProp TO var$
NEXT
```

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

What are Type Libraries?

[Top](#) [Previous](#) [Next](#)

A Type Library is a block of data which describes one or more [COM](#) Object Classes. The internal format of the data is not important, because it is seldom accessed by application programs. Typically, it is only accessed by COM Browsers such as [PBROW.EXE](#) (supplied with Classic PowerBASIC), TypeLib Browser from Jose Roca, or OLEVIEW.EXE from Microsoft. In the unusual circumstance that you must access this data directly, the Windows API provides numerous functions for just that purpose.

A Type Library is usually supplied by the author of the COM server. It's frequently supplied as a standalone data file with a file name extension of TLB. The data can also be embedded as a [resource](#) in the associated [DLL](#) or EXE. In practice, you would generally use a COM Browser to extract enough information about a COM [Object](#) to allow you to use these [classes](#) in your program. Generally speaking, a Type Library usually supplies specific details about every [METHOD](#) and [PROPERTY](#) (function), and the parameters of each of them. This would include the names, data types, return values, and more. The Type Library may also offer information about related [equates](#), [User-Defined-Types](#), and more. To include a [numeric equate](#) in your type library, just append the words AS COM to the equate definition:

```
%ABCD = 99 AS COM
```

Traditionally, it was common to use Interface Definition Language (IDL) to create the source code for the definitions you wish to describe in a Type Library. IDL was created specifically for this purpose and resembles C++ syntax. Once the source code was written, you would use Microsoft's MIDL Compiler to create the final Type Library. That's a fairly cumbersome process.

With Classic PowerBASIC, it's a bit simpler than that. Whenever you create a COM server, simply add the [#COM TLIB ON](#) metastatement to your source and your Type Library will be created automatically. You can prevent a Type Library from being created by using the [#COM TLIB OFF](#) metastatement. A Type Library is created with the same primary name as your COM server, and a file extension of TLB. That is, if you create a COM server named XX.DLL, Classic PowerBASIC will name the Type Library as XX.TLB. The Type Library offers a description of every published class on the server. You can then use any COM Browser to display the type information in a format that meets your needs. The Classic PowerBASIC COM Browser converts it directly to Classic PowerBASIC source code declarations which can then be dropped into your COM client program. If any of your Methods or Properties use data types not supported by Type Libraries, you will receive a [Error 581 - Type Library creation error](#). If you wish to create a Type Library for you COM server, then only use data types that are compatible with Type Libraries, which are [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#).

As mentioned earlier, you can consolidate your distribution files by embedding your Type Library right into your DLL or EXE as a resource. A utility program named PBTYP.EXE is provided for just that purpose. PBTYP.EXE is executed with one or two command line

parameters used to specify the files to be used in the embedding process. The syntax is:

```
PBTYP.EXE TargetFile [ResourceFile]
```

The PBTYP.EXE utility requires that you supply two or three files: the Target File (the DLL or EXE which receives the resource), the TypeLib File (the Type Library to be embedded), and optionally a resource file to be used. Since it's assumed that the Target File and the TypeLib file share the same primary name, only the Target file name is needed. If an extension is not supplied, the default of ".DLL" is used. When executed, PBTYP.EXE scans the original resource file (such as ABC.RC), and replaces any references to a Type Library with a reference to the new Type Library. It then compiles it to a resource object file (such as ABC.RES), and then creates a final Classic PowerBASIC version (such as ABC.PBR). Finally, it removes any prior resource from the target file, and replaces it with the newly created resource. It should be noted that RC.EXE and PBRES.EXE must be present in your path for the process to complete.

See Also

[What is an object, anyway?](#)

[Where are objects located?](#)

[Why should I use objects?](#)

[How do you publish an object?](#)

How are GUID's used with objects?

[Top](#) [Previous](#) [Next](#)

A [GUID](#) is a "Globally Unique Identifier", a very large number which is used to uniquely identify every [interface](#), every [class](#), and every [COM](#) application or library which exists anywhere in the world. GUID's identify the specific components, wherever and whenever they may be used. A GUID is a 16-byte (128-bit) value, which could be represented as an integer or a string. This item is large enough to represent all the possible values needed.

The Classic PowerBASIC [GUID\\$\(\)](#) function (or a [hot-key](#) in the Classic PowerBASIC IDE) can generate a random GUID which is statistically guaranteed to be unique from any other generated GUID.

When a GUID is written in text, it takes the form:

```
{00CC0098-0000-0000-0000-0000000000FF}
```

When a GUID is used in a Classic PowerBASIC program, it is typically assigned to a [string](#) [equate](#), as that makes it easier to reference.

```
$MyLibGuid    = GUID$("{00000099-0000-0000-0000-000000000007}")
$MyClassGuid  = GUID$("{00000099-0000-0000-0000-000000000008}")
$MyIfaceGuid  = GUID$("{00000099-0000-0000-0000-000000000009}")
```

Every [COM COMPONENT](#), every CLASS, and every INTERFACE is assigned a GUID to uniquely identify it, and set it apart from another similar item. As the programmer, you can assign each of these identifiers, or they will be randomly assigned by the Classic PowerBASIC compiler.

When you create [objects](#) just for internal use within your programs, it's common to ignore the GUID's completely. Classic PowerBASIC will assign them for you automatically, so you don't need to give it a thought. However, if you plan to publish an object for any external use through COM services, it's very important that you assign an explicit identifier to each item in your code. Otherwise, the compiler will assign new identifiers randomly, every time you compile the source. No other application could possibly keep track of the changes.

The APPID or LIBID identifies the entire application or library. You specify this item with the [#COM GUID](#) metastatement:

```
#COM GUID $MyLibGuid
```

The CLSID identifies each CLASS. You specify this item in the [CLASS](#) statement:

```
CLASS MyClass $MyClassGuid AS COM
[statements]
END CLASS
```

The IID identifies each INTERFACE. You specify this item in the INTERFACE statement:

```
INTERFACE MyInterface $MyIfaceGuid
[statements]
END INTERFACE
```

See Also

[What is an object, anyway?](#)

[Just what is COM?](#)

What is inheritance?

How do you create an object?

The compiler provides a set of built-in Interfaces, including:

ICLASSFACTORY
ICONNECTIONPOINTCONTAINER
ICONNECTIONPOINT
IDISPATCH
IUNKNOWN

See Also

[What are the parts of an object?](#)

[Are there other important "Buzz-Words"?](#)

[What does an Interface look like?](#)

[Built-in numeric equates](#)

[Built-in string equates](#)

[Built-in User Defined Types](#)

[Built-in RGB Color Equates](#)

What is the Classic PowerBASIC COM Browser

[Top](#) [Previous](#)
[Next](#)

The Classic PowerBASIC COM Browser is an application that exposes the data stored in a [type library](#) and generates Classic PowerBASICCompatible source code for this data. A Type Library is a block of data which describes one or more [COM](#) Object Classes.

If you are unfamiliar with COM programming, you may wish to review the COM Programming section in the Classic PowerBASIC For Windows Help file to gain an insight into COM programming concepts before reading this topic.

A Type Library is usually supplied by the author of the COM server. It's frequently supplied as a standalone data file with a file name extension of TLB. The data can also be embedded as a resource in the associated DLL, EXE, OCX, etc. The Classic PowerBASIC COM Browser is used to extract information about a COM [Object](#) to allow you to use these classes in your program. Generally speaking, a Type Library usually supplies specific details about every [Method](#) and [Property](#), and the parameters of each of them. This would include the names, data types, return values, and more. The Type Library may also offer information about related equates, [User-Defined-Types](#), and more.

The Classic PowerBASIC COM Browser can be launched from the Tools menu in the [Classic PowerBASIC IDE](#), launched as a stand-alone application by double-clicking PBROW.EXE in the \PB\BIN\ folder, or run from the command-line by typing PBROW.EXE (and then press ENTER).

When launched, the Classic PowerBASIC COM Browser offers a straightforward user interface, with which you open specific type-library files or choose from a list of registered libraries.

Before we start, we should first clarify a few terms so avoid confusion:

COM Object An instance of an initialized COM library or application. COM Objects usually come in EXE (out-of-process), and DLL, or OCX formats (in-process). These discussions pertain to COM libraries that act as COM Servers, regardless of whether they are in-process or out-of-process Servers.

Type-Library A type-library is a file that contains a database or data dictionary describing the Interfaces and Interface members exposed by a COM Object.

See Also

[The Classic PowerBASIC COM Browser user interface](#)

[The Classic PowerBASIC COM Browser Tutorial](#)

[What is an object, anyway?](#)

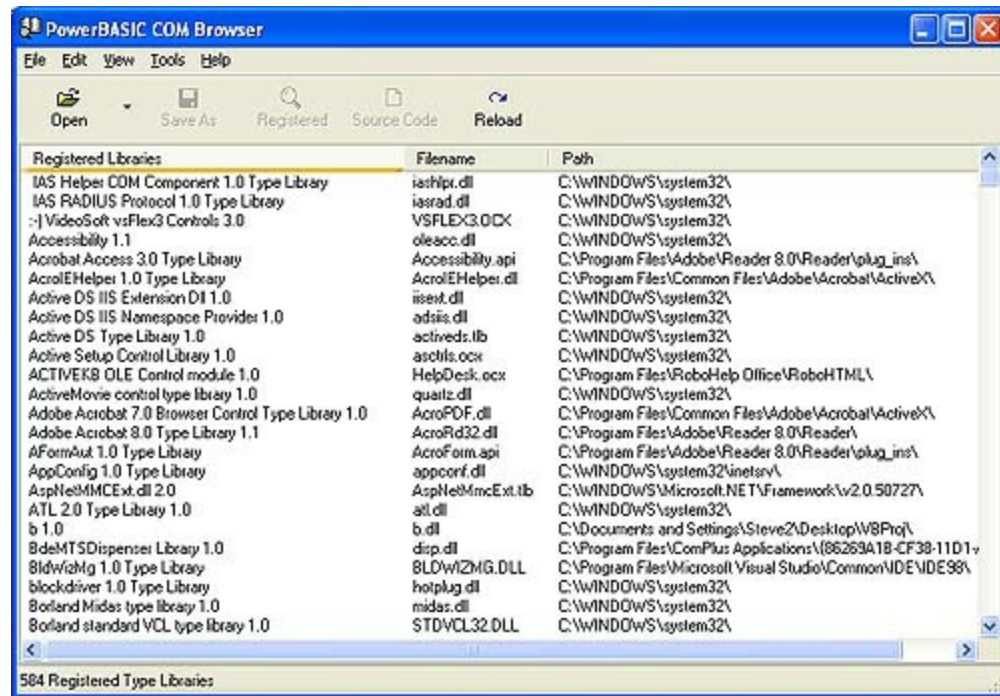
[Just what is COM?](#)

[What are Type Libraries?](#)

The Classic PowerBASIC COM Browser

[Top](#) [Previous](#)
[Next](#)

The Classic PowerBASIC COM Browser has two views, a list of all registered [type libraries](#) installed on the system and a source code view which displays the Classic PowerBASIC declarations for this type library. The Classic PowerBASIC COM Browser opens up with a list of all the registered type libraries installed on the users system.



See Also

[The Classic PowerBASIC COM Browser](#)

[The Classic PowerBASIC COM Browser Menu](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

The Classic PowerBASIC COM Browser Menu

[Top](#) [Previous](#)
[Next](#)

This topic briefly describes each menu option available from the Classic PowerBASIC COM Browser's menu.

File menu

- Open File** Open a type library file.
- Save File As** Save the currently loaded [source code](#) to disk.
- [recent files list]** A list of the most recently loaded type libraries.
- Exit** Exits the Classic PowerBASIC COM Browser.

Edit menu

- Select All** Select all the text in the current source code window.
- Copy** Copy the selected text in the source code window and place it in the Clipboard.

View menu

- Registered Libraries** Show a list of all the [Registered Libraries view](#).
- Source Code** Show the Source Code window view.
- Reload** If in Registered Library view, this options reloads the list of all Registered Libraries installed on the system. If in Source Code view, this option reloads the source code generated from the selected type library.

Tools menu

- Options** Display the [Options dialog](#), for configuring the Classic PowerBASIC COM Browser.

Help menu

- Help For PBRow** Displays the Classic PowerBASIC COM Browser help file.
- Help For Library** Displays the help file for the currently loaded type library if one exists.
- About** Display version information for the Classic PowerBASIC COM Browser.

See Also

[The Classic PowerBASIC COM Browser](#)
[The Classic PowerBASIC COM Browser User Interface](#)

What is an object, anyway?

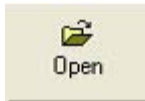
Just what is COM?

What are Type Libraries?

The Classic PowerBASIC COM Browser Toolbar

[Top](#) [Previous](#)
[Next](#)

This topic briefly describes each menu option available from the Classic PowerBASIC COM Browser's toolbar.



Open a type library file



Save the currently loaded [source code](#) to disk.



Show a list of all the [Registered Libraries view](#).



Show the Classic PowerBASIC compatible source code for the loaded type library.



If in Registered Library view, this options reloads the list of all Registered Libraries installed on the system. If in Source Code view, this option reloads the source code generated from the selected type library.

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Menu Items](#)

[Shortcut Keys](#)

[Registered Type Library View](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

Shortcut Keys

[Top](#) [Previous](#) [Next](#)

The following table summarizes the shortcut-keys available in the Classic PowerBASIC COM Browser.

Keystroke	Description
F1	Display the help file for the type library if available, otherwise display the Classic PowerBASIC COM Browser help file
CTRL+A	Select all the text in the Source Code View
CTRL+C	Copy the selected text in the Source Code window to the clipboard
CTRL+D	Display the Source Code View.
CTRL+L	Reload the Type library View or the Source Code View
CTRL+O	Open a Type Library file
CTRL+R	Display the Registered Type Library View.
CTRL+S	Save the current source code

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Registered Type Library View](#)

[Source Code View](#)

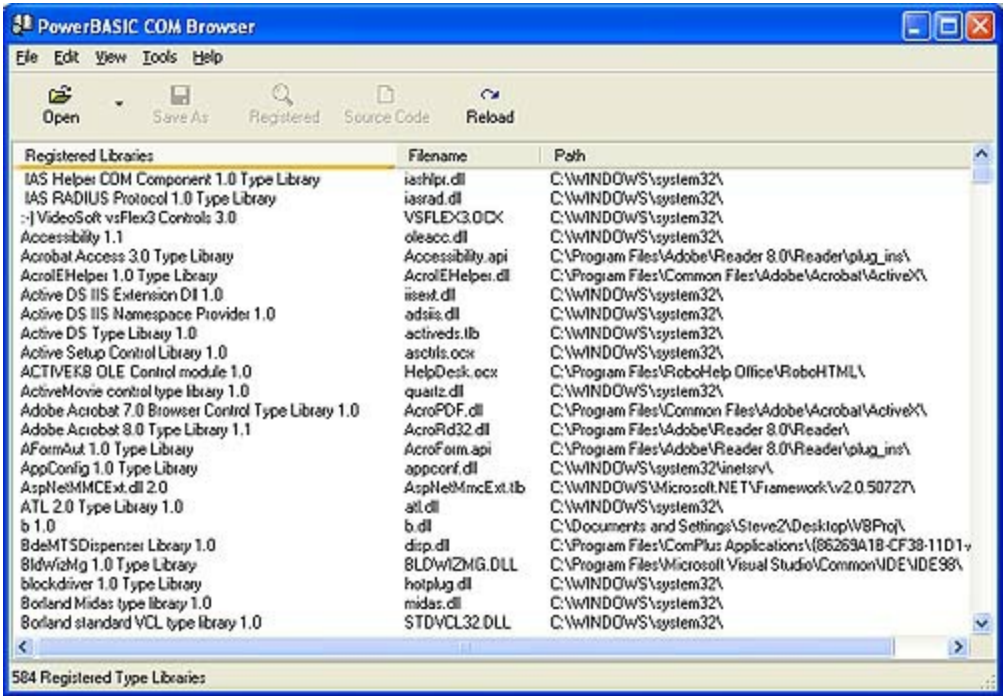
[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

Registered Type Library View

The Registered Library view displays a list of all the registered [type libraries](#) installed on the system. This Registered Library view is initially displayed when the Classic PowerBASIC COM Browser is started. When in the [Source Code view](#), you can switch to the Registered Library View by clicking the [Registered Button](#) or by selecting View | Registered Libraries from the [menu](#).



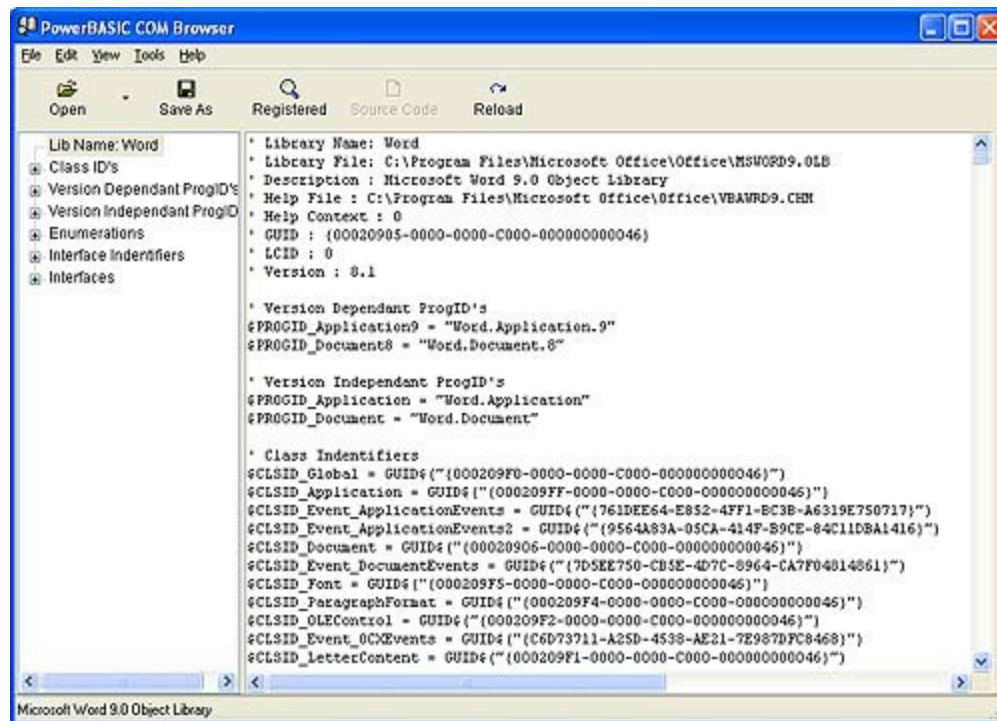
The Registered Libraries column, displays the descriptive name for the registered library. The Filename column displays the filename of the registered type library. The Path column displays the path to the registered type library. Each column header can be clicked to sort the column in ascending order, if you click the same column header again the column will be sorted in descending order. The Classic PowerBASIC COM Browser remembers the column and the sort order you last used and will display the list of registered type libraries using this information the next time you open the Classic PowerBASIC COM Browser.

To generate Classic PowerBASIC compatible declarations, double-click on the library name. The Classic PowerBASIC COM Browser will display the declarations in the Source Code view.

See Also

- [The Classic PowerBASIC COM Browser User Interface](#)
- [Source Code View](#)
- [Getting Help](#)
- [What is an object, anyway?](#)
- [Just what is COM?](#)
- [What are Type Libraries?](#)

The Source Code view displays a list of all the data included in the selected [type library](#) and the Classic PowerBASIC compatible declarations for this data.



Clicking on a type library data item on the left hand side will display only the Classic PowerBASIC compatible code for the item on the right hand side, clicking on the top level library name will display all the Classic PowerBASIC compatible code for the selected type library.

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Registered Type Library View](#)

[Getting Help](#)

[Saving the Source Code](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

If the type library has a help file installed on the system, you can press the F1 key while in the [Source Code view](#) to display the help file. If you have selected an item from the list of items available and press the [F1 key](#) the help topic for the selected item is displayed. If no help topic is available for the selected item the default topic for the [type libraries](#) help file is displayed. If there is no help file for the selected type library, then the Classic PowerBASIC COM Browsers help file (this file) is displayed.

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Opening a type-library](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

Opening a Type Library

[Top](#) [Previous](#) [Next](#)

The Classic PowerBASIC COM Browser supports opening both registered and unregistered [Type Libraries](#). A registered type library can be opened from the [Registered Type Library view](#) or by clicking the [Open button](#) and browsing to and selecting the type library file. An unregistered type library can only be opened by using the Open button and browsing to and selecting the type library file.

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Saving the Source Code](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

Saving the Source Code

[Top](#) [Previous](#) [Next](#)

To save the Classic PowerBASIC compatible source code, simply click the [Save As button](#) or select File -> Save As from the [menu](#) and the source code displayed in the [Source Code view](#) will be saved to disk. You may wish to save the entire Source Code for the [type library](#), in which case make sure you have selected the top level item on the right hand side, which is the type libraries name. If a type library item is selected on the left hand side of the Window, then only that portion of the displayed code will be saved to disk.

The selected text in the Source Code window can also be save to the Windows clipboard, by selecting Edit -> Copy.

See Also

[The Classic PowerBASIC COM Browser User Interface](#)

[Source Code View](#)

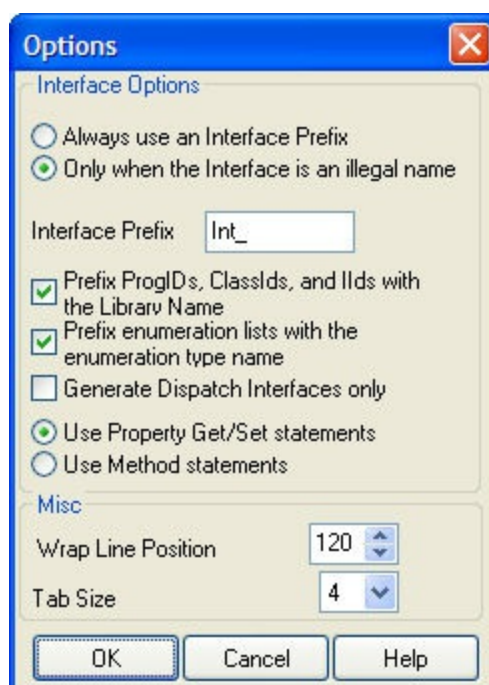
[Options Dialog](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What are Type Libraries?](#)

This topic describes the options available to customize the output of the source code generated by the Classic PowerBASIC COM Browser.



Always use an Interface Prefix

This option prefixes all Interface names with the text specified in the Interface Prefix textbox. This is useful when using multiple [type libraries](#) in a project and there are conflicts with duplicate Interface names.

Only When the Interface contains an illegal name

This option prefixes Interface names, only when they contain illegal characters or conflict with a reserved keyword.

Interface Prefix

This is the prefix used for Interface names.

Prefix ProgIDs,ClassIDs, and IIDs with Library Name

This option will prefix ClassIDs, ProgIDs, and IIDs with the library name. This option is used when you have multiple type libraries with conflicting names in one application.

Prefix enumeration lists with the enumeration type name

This option will prefix enumeration lists with the enumeration type name. This option is used when you have multiple type libraries with conflicting names in one application.

Generate Dispatch Interfaces only

This option generates [Dispatch](#) Interface's only for the purposes of [IDBinding](#) to a Dispatch COM Interface. Custom only Interfaces will be skipped when this option is used.

Use Property Get/Set statements

This option will generate Property Get/Set statements in the generated source code. A Property is a special type of Method, which is only used to set or retrieve data in an object. The use of Property Get/Set statements is the preferred syntax as it improved readability of the source code.

Use Method statements

This option will convert Property Get/Set statements to Method statements in the generated source code. This option is useful when the type library Property Get or Set definition contains an error and the use of a Method statement can usually resolve the type library error. This option is not available when you have the Generate Dispatch Interfaces only.

Wrap Line Position

When generating source code, The Classic PowerBASIC COM Browser wraps long lines of code (using standard Classic PowerBASIC line continuation characters) when they reach the wrap column indicated in this field.

Tab Size

The number of spaces that Tab characters are expanded to in the generated source code.

See Also

[The Classic PowerBASIC COM Browser](#)

[The Classic PowerBASIC COM Browser Tutorial](#)

[What is an object, anyway?](#)

[Just what is COM?](#)

[What is DISPATCH?](#)

[What are Type Libraries?](#)

The Classic PowerBASIC COM Browser Tutorial

[Top](#) [Previous](#)
[Next](#)

As described in the [What is the Classic PowerBASIC COM Browser](#) topic, the Classic PowerBASIC COM Browser generates Classic PowerBASIC compatible source code to be used in your application.

We will walk through an example of using the Classic PowerBASIC COM Browser to locate the Microsoft Agent Control 2.0 type library for a Windows application.

1. Start the Classic PowerBASIC COM Browser
2. Locate the Microsoft Agent Control 2.0 type library. This will be listed under the "Registered" type library installed; it can be downloaded for free from <http://www.microsoft.com/DOCS/olap/agents/agent20.htm> type libraries.
3. Double-click on the Microsoft Agent Control 2.0 type library listed in the list of Registered type libraries.
4. Click the "Save As..." button and save it with the name of "agent.inc"
5. Close the Classic PowerBASIC COM Browser
6. Start the Classic PowerBASIC For Windows 9 IDE
7. Click the Create New File button in the IDE
8. Paste the following code into the new file created in the IDE

```
#COMPILE EXE
#DIM ALL

%ID_START      = 1000
%ID_STOP       = 1001
%ID_EVENTLIST  = 1003

GLOBAL hDlg AS LONG
#INCLUDE "agent.inc"
#INCLUDE "Win32api.inc"

' Display an error message
MACRO DisplayError(txt)
    IF ISTRUE(ISOBJECT(AgentEvents)) THEN
        ' Detach the events handler
        EVENTS END AgentEvents
    END IF
    ' Print the error and then exit the callback routine
    MSGBOX txt, %MB_OK OR %MB_ICONERROR, "MS Agent Error"
    EXIT FUNCTION
END MACRO
```

CALLBACK FUNCTION DlgProc

STATIC AgentCtrlEx AS IAgentCtrlEx

STATIC AgentEvents AS Int__AgentEvents

LOCAL StartX AS LONG

LOCAL StartY AS LONG

LOCAL CharW AS LONG

LOCAL CharH AS LONG

SELECT CASE AS LONG CBMSG

CASE %WM_INITDIALOG

' Create the Agent Control Object

AgentCtrlEx = newcom \$PROGID_Agent2

IF ISFALSE(ISOBJECT(AgentCtrlEx)) THEN

DisplayError("The Microsoft Agent Control 2.0 is not in
"downloaded from <http://www.microsoft.com/>:"

END IF

' Create the Events handler interface

AgentEvents = CLASS "Class_Int__AgentEvents"

IF ISFALSE(ISOBJECT(AgentEvents)) THEN

DisplayError("Error creating the event interface.")

END IF

' Attach the Events handler interface to the Agent Contro

Events FROM AgentCtrlEx CALL AgentEvents

'Enable the Start button

CONTROL ENABLE CBHNDL, %ID_START

CASE %WM_COMMAND

SELECT CASE AS LONG CBCTL

CASE %ID_START

IF CBCTLMSG = %BN_CLICKED OR CBCTLMSG = 1 THEN

' Load the Merlin agent

AgentCtrlEx.Characters.Load("Merlin", "Merlin.acs")

IF OBJRESULT <> %S_OK THEN

DisplayError("The Microsoft Agent Control 2.0 Mer
"can be downloaded from <http://www.m>

END IF

```

    ' Show the Merlin agent on the screen
    AgentCtrlEx.Characters.Character("Merlin").Show
    ' Get the Width and Height of the Merlin agent
    CharW = AgentCtrlEx.Characters.Character("Merlin").
    CharH = AgentCtrlEx.Characters.Character("Merlin").
    ' Get the Width and Height of the Desktop
    DESKTOP GET CLIENT TO StartX, StartY
    ' Find the center of the desktop for Merlin agent
    StartX = (StartX - CharW)\2
    StartY = (StartY - CharH)\2
    ' Move the Merlin agent to the center of the deskto
    AgentCtrlEx.Characters.Character("Merlin").MoveTo(S
    ' Have the Merlin agent play the trumpet
    AgentCtrlEx.Characters.Character("Merlin").Play(UCO
    ' Make the Merlin agent speak
    AgentCtrlEx.Characters.Character("Merlin").Speak("W
    ' Disable the Start button and enable the Stop butt
    CONTROL DISABLE CBHNDL, %ID_START
    CONTROL ENABLE  CBHNDL, %ID_STOP
END IF
CASE %ID_STOP
    IF CBCTLMSG = %BN_CLICKED OR CBCTLMSG = 1 THEN
        ' Stop all actions by the Merlin agent and unload i
        AgentCtrlEx.Characters.Character("Merlin").STOP
        AgentCtrlEx.Characters.Unload("Merlin")
        ' Enable the Start button and disable the Stop butt
        CONTROL ENABLE  CBHNDL, %ID_START
        CONTROL DISABLE CBHNDL, %ID_STOP
    END IF
END SELECT
CASE %WM_DESTROY
    IF ISTRUE(ISOBJECT(AgentEvents)) THEN
        ' Detach the event handler interface
        Events END AgentEvents
    
```

```

        END IF
    END SELECT
END FUNCTION

FUNCTION PBMAIN () AS LONG
    DIALOG NEW 0, "COM Browser Tutorial", 201, 122, 198, 115, %WS_
        %WS_SYSMENU OR %WS_MINIMIZEBOX OR %WS_CLIPSIBLINGS OR %WS_V
        %DS_NOFAILCREATE OR %DS_SETFONT, %WS_EX_CONTROLPARENT OR %W
        %WS_EX_RIGHTSCROLLBAR, TO hDlg
    CONTROL ADD BUTTON, hDlg, %ID_START, "Start Agent", 5, 5, 50
        %WS_TABSTOP OR %BS_TEXT OR %BS_PUSHBUTTON OR %BS_CENTER OR
    CONTROL ADD BUTTON, hDlg, %ID_STOP, "Stop Agent", 5, 25, 50,
        %WS_TABSTOP OR %BS_TEXT OR %BS_PUSHBUTTON OR %BS_CENTER OR
    CONTROL ADD LISTBOX, hDlg, %ID_EVENTLIST, , 70, 0, 125, 110,
        %WS_EX_CLIENTEDGE OR %WS_EX_LEFT OR %WS_EX_LTRREADING OR %W
    DIALOG SHOW MODAL hDlg, CALL DlgProc
END FUNCTION

```

9. Click the Save All button and save this file as "agent.bas" in the same directory that you
10. Open the "agent.inc" file in the IDE
11. Search (CTRL+F) in the IDE for the text of "IAgentCtl event interface" (without the quote)

```

display the event that occurred in the dialogs listbox. Make the Int__AgentEvents, look
CLASS Class_Int__AgentEvents $CLSID_Event__AgentEvents AS EVENT
INTERFACE Int__AgentEvents $IID_Int__AgentEvents
    INHERIT IDISPATCH
    METHOD ActivateInput <1> (BYREF CharacterID AS STRING)
        LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Input Activated"
    END METHOD
    METHOD DeactivateInput <3> (BYREF CharacterID AS STRING)
        LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Input Deactivated"
    END METHOD
    METHOD Click <2> (BYVAL CharacterID AS STRING, BYVAL BUTTON
        LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Click at (" + FORM
    END METHOD
    METHOD DblClick <4> (BYVAL CharacterID AS STRING, BYVAL BUT
        LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Double Click at (
    END METHOD

```

```

METHOD DragStart <5> (BYVAL CharacterID AS STRING, BYVAL BU
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Drag Start at ("+
END METHOD

METHOD DragComplete <6> (BYVAL CharacterID AS STRING, BYVAL
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Drag Complete to
END METHOD

METHOD SHOW <15> (BYVAL CharacterID AS STRING, BYVAL Cause
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Character is show
END METHOD

METHOD Hide <7> (BYVAL CharacterID AS STRING, BYVAL Cause A
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Character is hidd
END METHOD

METHOD RequestStart <9> (BYVAL Request AS IDISPATCH)
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Request Start"
END METHOD

METHOD RequestComplete <11> (BYVAL Request AS IDISPATCH)
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Request Complete"
END METHOD

METHOD Restart <21> ()
    ' Insert your code here
END METHOD

METHOD Shutdown <12> ()
    ' Insert your code here
END METHOD

METHOD Bookmark <16> (BYVAL BookmarkID AS LONG)
    ' Insert your code here
END METHOD

METHOD Command <17> (BYVAL UserInput AS IDISPATCH)
    ' Insert your code here
END METHOD

METHOD IdleStart <19> (BYVAL CharacterID AS STRING)
    ' Insert your code here
END METHOD

METHOD IdleComplete <20> (BYVAL CharacterID AS STRING)
    ' Insert your code here

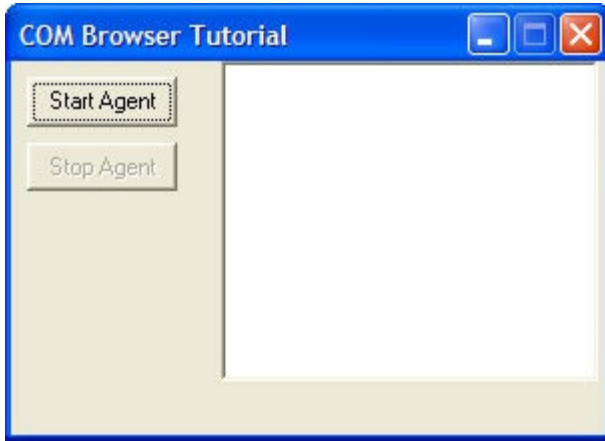
```

```

END METHOD
METHOD Move <22> (BYVAL CharacterID AS STRING, BYVAL x AS I
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Move to (" + FORMAT
END METHOD
METHOD SIZE <23> (BYVAL CharacterID AS STRING, BYVAL Param_
    ' Insert your code here
END METHOD
METHOD BalloonShow <24> (BYVAL CharacterID AS STRING)
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Showing balloon t
END METHOD
METHOD BalloonHide <25> (BYVAL CharacterID AS STRING)
    LISTBOX INSERT hDlg, %ID_EVENTLIST, 1, "Hiding balloon te
END METHOD
METHOD HelpComplete <26> (BYVAL CharacterID AS STRING, BYVA
    ' Insert your code here
END METHOD
METHOD ListenStart <27> (BYVAL CharacterID AS STRING)
    ' Insert your code here
END METHOD
METHOD ListenComplete <28> (BYVAL CharacterID AS STRING, BY
    ' Insert your code here
END METHOD
METHOD DefaultCharacterChange <30> (BYREF Param_GUID AS STR
    ' Insert your code here
END METHOD
METHOD AgentPropertyChange <31> ()
    ' Insert your code here
END METHOD
METHOD ActiveClientChange <32> (BYVAL CharacterID AS STRING
    ' Insert your code here
END METHOD
END INTERFACE
END CLASS

```

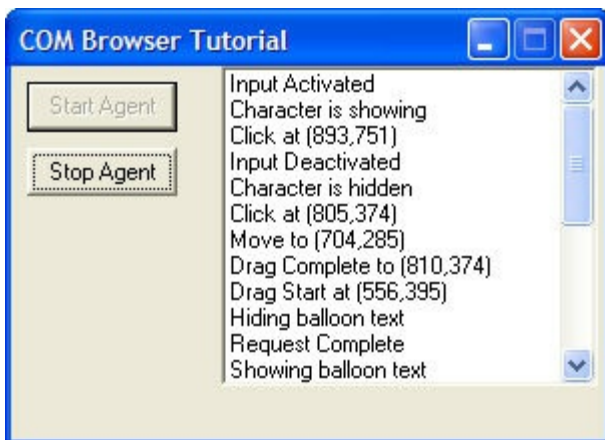
12. In the IDE, click the compile and run button. The application will be displayed as



13. Click the "Start Agent" button and the Merlin character will display in the top left corner. When Merlin is speaking, you will need to download and install the free SAPI 4.0 and a Text



14. You can click, double-click, drag and drop, hide (right-click on Merlin and select Hide),



15. Click the Stop Agent button to stop and unload the Merlin character.

See Also

[What is the Classic PowerBASIC COM Browser](#)

[The Classic PowerBASIC COM Browser User Interface](#)

Occasionally, you may run into a situation where the syntax and structure of the BASIC language is not the most suitable for a task at hand. Classic PowerBASIC addresses the need for optimal speed and flexibility with its built-in assembler. Inline assembly is the process of embedding assembly-language statements (opcodes) within the overall structure of your BASIC code. Those statements are compiled along with your BASIC code without the need for an external assembler.

This chapter discusses the different ways that Classic PowerBASIC lets you use assembly-language code in your BASIC programs. It also discusses some design philosophies and considerations, which you should keep in mind if you decide to write your own assembly-language procedures or functions.

The technique of interfacing with assembly-language is, by its very nature, somewhat complex. You should be reasonably familiar with assembly-language concepts before tackling the information in this chapter.

See Also

[Using assembly-language in your code](#)

[Flat memory model](#)

[Inline Assembler code syntax](#)

[Protected mode programming](#)

[Mnemonics and Operands](#)

[Opcodes and Mnemonics](#)

[Registers](#)

[Data types in registers](#)

[MMX registers](#)

[The stack](#)

[Balancing the stack](#)

[Tricks of the stack](#)

[Stack Overhead Reduction](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Intermixing ASM and BASIC code](#)

[Using ESP and EBP](#)

[Saving the FPU registers](#)

[Tricks in preserving registers](#)

[Addressing and pointers](#)

[Effective addressing](#)

[Passing parameters](#)

[Parameters passed by reference or by copy](#)

[Parameters passed by value](#)

[Passing dynamic strings](#)

[Passing arrays](#)

[Accessing Classic PowerBASIC variables by name](#)

[Commenting Assembly code](#)

Using assembly-language in your code

[Top](#) [Previous](#)
[Next](#)

Classic PowerBASIC provides a number of ways in which you can use assembly-language. You can write the whole program using the [Inline Assembler](#). You can write entire [Subs](#), [Functions](#), [Methods](#), and [Properties](#) in assembly-language, or, you can write individual lines of code in assembler, surrounded by BASIC statements. This ability to intermix BASIC and assembly-language, line by line, makes Classic PowerBASIC's Inline Assembler a very powerful tool when optimal performance is an essential issue.

To write good assembler code, you must be aware of certain items:

- The types of [variables](#) supported by Classic PowerBASIC
- How those variables are stored in memory
- How to use variable names in your Inline Assembler routines
- Which registers to save (and restore)
- How to pass arguments (to and from Inline Assembler routines)
- The need to pop everything you push
- The differences between near and far calls
- The rules to follow when writing assembly-language routines

See Also

[The Inline Assembler](#)

[Inline Assembler code syntax](#)

The [ASM](#) statement or, for a "shortcut", the exclamation point (!), is used to insert assembler instructions (or [opcodes](#)) into your BASIC program. They must appear at the beginning of each line that contains an assembler instruction. The [Inline Assembler](#) supports standard instructions and [registers](#), including 8086/8088, 80286, 80386, 80486, Pentium/MMX and 32-bit floating-point opcodes as defined in the Intel Reference Manuals, and can be downloaded from <http://developer.intel.com/>.

The machine code generated by ASM statements is placed directly in line with the code from your BASIC statements, so execution of your program will flow just as it appears in your source code. You should never, under any circumstances, attempt to exit a [Sub/Function/Method/Property](#) early by the use of a **RET** instruction, as that guarantees failure. If you need to terminate a routine at some point before the End Sub/Function/Method/Property statement, jump to a [label](#) at the end of the procedure instead.

See Also

[The Inline Assembler](#)

[Using assembly-language in your code](#)

[Flat memory model](#)

A program written in native 32 bit Windows format is created in what is called FLAT memory model that has a single segment, which contains both code and data. Such programs must be run on a 80386 or higher processor.

Differing from earlier 16-bit code that used combined segment and offset addressing with a 64 Kb segment limit, FLAT memory model works only in offsets and has a range of 4 Gigabytes. This makes [assembly code](#) easier to write and the compiled (assembled) code is generally a lot faster than the equivalent 16-bit code.

All segment [registers](#) are automatically set to the same value with the flat memory model. This means that segment / offset [addressing](#) must NOT be used in 32-bit programs that run in 32-bit Windows operating systems.

For programmers who have written code in DOS, a 32-bit Windows PE (executable) file is similar in some respects to a DOS COM file - they have a single segment that can contain both code and data and they both work directly in offsets. That is, neither uses segment / offset addressing.

Flat-model assembler code defaults to NEAR code addressing and NEAR data addressing within the range of 4 gigabytes.

The FS and GS segment registers are rarely (if ever) used in application programs but may be used in some instances by the operating system itself.

See Also

[The Inline Assembler](#)

[Protected mode programming](#)

[Mnemonics and Operands](#)

Protected mode programming

[Top](#) [Previous](#) [Next](#)

DLLs and EXEs generated by the Classic PowerBASIC 32-bit compilers require Microsoft Windows 95 or later, or Windows NT 3.10 or later. This includes Windows 98, Windows ME, Windows NT 4.0, Windows 2000, Windows XP, Windows Vista, Windows 7, and so on. All of these operating systems run your program in protected mode, using a 32-bit flat memory model.

In real-mode operating systems such as DOS, it was possible to overwrite sections of the operating system code if a program was not correctly written. This would crash the operating system and the computer would require a reboot before it would run again.

Protected-mode memory is designed to prevent this from happening. It uses a protected mode memory manager to control the address range that all applications can read and write to, and the memory manager terminates any application that tries to read or write to a memory address range that is outside of the allocated application memory space. The memory region assigned to an application is known as the *process address space*.

This style of memory management was available in 16-bit Windows but because of the method that 16-bit Windows used to simulate multitasking, it was possible to overwrite sections of memory that were owned by other applications or the operating system before the errant program crashed.

Depending on what portions of the operating system were overwritten, another (completely unrelated) program that had no errors in it could try to use a damaged operating system function, resulting in both a program and operating system crash.

The common symptom of this behavior was the infamous "blue screen of death" which told you something was wrong, but often reported misleading causes of the problem. If an application destroyed critical sections of the operating system, the result was often a "black screen of death" - an instant black screen accompanied by a solidly locked or frozen machine. This obviously gave no feedback as to the cause of the problem.

As improvements in hardware design occurred, the use of hardware based multitasking in 32-bit Windows made protected-mode memory managers increasingly reliable and resulted in new operating systems with ever increasing stability - more able to cope with preventing operating system crashes when an application crashes.

From this we can see that one of the fundamental "rules" of writing code for a protected mode operating system is to ensure the application code can read and write only within the process address space.

However, because [Inline Assembler](#) allows you to read and write to almost any memory address, this clearly places the onus on the programmer to take suitable precautions with all referenced memory addresses.

For example, if you allocate a 10 Kb buffer and subsequently try to read 20 Kb, the protected mode memory manager will trigger a Page Read fault (GPF) the instant the address goes outside the process address space - typically this would occur when the memory address advances beyond the original 10 Kb buffer.

An application can also get into similar trouble if it incorrectly dereferences a [register](#) (i.e., if it incorrectly treats the register content as a [pointer](#) or address, rather than a value). If the address points outside of the allocated process address space, a GPF is virtually guaranteed.

Page read and write faults are exceptions (GPFs) that are passed from the operating system to the application that makes the error. If the exception is not handled by the application, the operating system closes the application. Current versions of Classic PowerBASIC do not support native exception handling, but it is possible to configure an exception "trap" function using the Windows API.

Therefore, to create effective and stable assembler code, you should become familiar with protected mode programming concepts. As outlined above, you must never access memory not specifically assigned to your process, nor should you ever change the selector value in a segment register.

Likewise, all Calls, Jumps, and Returns should use "Near" addressing, as a full 32-bit offset is utilized. Should you violate any of the rules of protected mode programming, your code will likely fail catastrophically with a General Protection Fault (GPF).

See Also

[The Inline Assembler](#)

[Flat memory model](#)

Mnemonics and Operands

[Top](#) [Previous](#) [Next](#)

An assembly code instruction (statement) consists of a mnemonic (pronounced "nih-MON-ick"), and between zero and three operands. For a logical or arithmetic mnemonic with two operands, the right operand is the *source* and the left operand is the *destination*. In general, 80x86 assembly code instructions takes the following form:

```
[ASM|!] mnemonic destination, source
```

For example:

```
! MOV EAX, 1
ASM ADD EAX, EBX
```

In the examples above, the keywords MOV and ADD are the mnemonics, EAX is the destination operand, and 1 and EBX are the source operands.

In the first line, the value 1 is "moved" into the EAX operand (register). In BASIC code, this works similarly to the statement $A = 1$. The second example adds the value in EBX to the value in EAX and stores the result in EAX. In BASIC code, this works similarly to $A = A + B$.

See Also

[The Inline Assembler](#)

[Using assembly-language in your code](#)

[Opcodes and Mnemonics](#)

[Registers](#)

Opcodes and Mnemonics

[Top](#) [Previous](#) [Next](#)

At the hardware level in an Intel or compatible processor, *instructions* are built directly into the CPU circuitry and these are represented by *opcodes*.

While [assembly code](#) can be written at the Byte level, it is a particularly complex method of writing code since it involves memorizing a very large number of opcodes. Additionally, such program code must use the Intel numeric (*little-endian*) data format. With little-endian, multi-byte numeric values are stored in reverse order to usual human representation.

For example, to copy the 32-bit value &H56A700FE into the EAX [register](#) (MOV EAX, &H56A700FE), you must first find the opcode for the MOV EAX mnemonic (&HA1) followed by the data in reverse order (&HFE, &H00, &HA7, and &H56).

In hex format, the whole instruction would look like this:

```
A1 FE 00 A7 56
```

Obviously this becomes a very tedious and error prone way to write assembly code. As a result, a system was developed (many years ago) where groups of similar opcodes were given verbose names that made them a lot more convenient to use than raw numeric opcodes. These names are referred to as *mnemonics*, and this is the system used in Classic PowerBASIC's 32-bit [Inline Assembler](#).

Each mnemonic represents a reserved name that represents a family of opcodes that perform similar tasks in the processor. The actual numeric opcodes are different depending on the size and type of [operands](#) being used. For example, with the MOV mnemonic:

```
! MOV EAX, VAR1    ' opcode = &HA1
! MOV VAR1, EAX    ' opcode = &HA3
```

The use of mnemonics provides a far more intuitive way of representing opcodes; however, there is no exact correlation between what you write using mnemonics and what you get as finished opcodes. This is because the opcode you actually get for a given mnemonic can depend on whether it is using near or far addressing, the operand data types (registers or pointers or constants), etc. However, Classic PowerBASIC takes care of these details automatically and transparently for you, leaving you to get on with writing your actual program.

See Also

[The Inline Assembler](#)

[Using assembly-language in your code](#)

[Mnemonics and Operands](#)

[Registers](#)

Registers

Registers are a special working area within the processor. Registers are faster than memory [operands](#), and are designed to work with the processor's [opcodes](#).

Registers in an Intel or compatible processor are a very limited resource when writing assembler. In essence, there are eight general-purpose registers, EAX, EBX, ECX, EDX, ESI, EDI, [ESP](#), and [EBP](#). In most instances, ESP and EBP should remain unused as Classic PowerBASIC uses them for entry and exit of procedures.

This means that you have six 32-bit registers to use in your assembly code, plus any other memory locations that are useful in the procedure. ESI and EDI can be used in the normal manner in most instances but neither can be accessed at a Byte level. You can read the low WORD of ESI as SI and the low WORD of EDI as DI.

Understanding the size of registers and the data that you can place in them is very important when using assembler. A 32-bit Intel or compatible processor has three native data sizes that can be used by the normal integer instructions, [BYTE](#), [WORD](#), and [DWORD](#) corresponding to 8-bit, 16-bit and 32-bit.

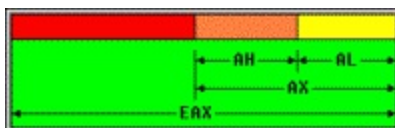
This can be shown in HEX notation.

BYTE	00
WORD	00 00
DWORD	00 00 00 00

In terms of registers, this corresponds to the three sizes that can be addressed with the normal integer registers. Intel and compatible processors are backwards compatible with older code that uses 8 and 16-bit registers, and it is done by accessing any of the general purpose registers in three different ways. Using the EAX register as an example:

AL or AH	= 8 bit
AX	= 16 bit
EAX	= 32 bit

This is the schematic of a general purpose 32-bit register:



This schematic is easier to understand at a bit level. Reading from the right side, you have 32 bits in the register, bits 0 to 31. Because of the bit positions for each piece of data that can be accessed in a 32-bit register, AL is called the Low (low-order) byte, AH is called the High (high-order) byte and AX is called the Low word.

To read the first BYTE in the register (bits 0 to 7) you use:

```
! MOV byteval, AL ; Copy the low-order byte into variable
```

Likewise, to read the second byte in the register (bits 8 to 15) you use:

```
! MOV byteval, AH ; Copy the high-order byte into bytevar
```

If you want to read the first WORD in the register (bits 0 to 15) you use:

```
! MOV wordval, AX ; Copy the low-order Word into a variable.
```

To get at bits 16 to 31, you must rotate the bits in the register so they can be accessed by

the previous instructions. Rotating a 32-bit register in either direction by 16 bits move the low-order 16-bits into the high-order 16-bit positions, and the high-order 16 bits into the low-order 16-bit positions of the register.

```
! ROL EAX, 16      ; Rotate EAX left by 16-bits
```

or:

```
! ROR EAX, 16      ; Rotate EAX right by 16-bits
```

You cannot put a different size piece of data into a register than its correct size and you cannot mix different register sizes:

```
! MOV EAX, CL      ; This fails as EAX is 32-bit, CL 8-bit
```

If you need to put the value in CL into a 32-bit register, you must first convert it using one of a number of different techniques:

```
! MOVZX EAX, CL    ; Zero extend unsigned Integer
```

```
! MOVSX EAX, CL    ; Sign extend signed Integer
```

In some instances you can use:

```
! XOR EAX, EAX     ; Clear EAX
```

```
! MOV AL, CL       ; Copy CL into AL
```

In addition, there are also some "older" mnemonics that will do the conversions too:.

```
! MOV AL, CL       ; Copy CL into AL
```

```
! CBW              ; Convert Byte in AL to Word in AX
```

```
! CWDE             ; Convert Word in AX to DWORD in EAX
```

See Also

[The Inline Assembler](#)

[Data types in registers](#)

[MMX registers](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Tricks in preserving registers](#)

[Saving the FPU registers](#)

[Using ESP and EBP](#)

Data types in registers

[Top](#) [Previous](#) [Next](#)

There are three basic types of operands that can be placed in a [register](#): immediate, memory or another register.

An *immediate* operand is usually a [numeric literal](#) (number) but it can also be a [string literal](#) in the form "a" which is converted by Classic PowerBASIC to its ASCII equivalent code.

```
! MOV AL, "a"           ; String literal
! MOV EDX, 0            ; Numeric immediate/literal
```

A *memory operand* is an [address](#) in memory of some form of data:

```
! MOV AL, [ESI]         ; Copy byte at address in ESI into AL
! MOV EDX, lpMemvar     ; Copy variable address into EDX
```

A *register* operand is a register with a value in it:

```
! MOV ECX, EDX          ; Copy EDX into ECX
```

The actions that can be performed are determined by the available [opcodes](#). For example, trying to move one memory operand directly into another does not work because there is no opcode in the 80x86 processor to do it.

```
! MOV mVar, lpMem       ; This fails as there is no opcode
```

However, if you have a "spare" register, you make an indirect copy through that register:

```
! MOV EAX, lpMem        ; Copy memory value into register
! MOV mVar, EAX          ; Copy register into memory variable
```

If you don't have a "spare" register, it can be done another way but it is slightly slower:

```
! PUSH lpMem            ; Push memory value onto the stack
! POP mVar               ; Pop it off as another memory value
```

See Also

[The Inline Assembler](#)

[Registers](#)

[MMX registers](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Using ESP and EBP](#)

[Saving the FPU registers](#)

MMX registers

[Top](#) [Previous](#) [Next](#)

According to the Intel documentation, all MMX registers require parentheses around the register number. These were compulsory in early versions of Classic PowerBASIC, but the parentheses are now optional.

```
!  PXOR  mm(7), mm(7)      ' PB/DLL 5.0, PB/CC 1.0 format
!  PXOR  mm7, mm7          ' PB/DLL/WIN 6.0+, PB/CC 2.0+ format
```

See Also

[The Inline Assembler](#)

[Registers](#)

[Data types in registers](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Tricks in preserving registers](#)

The stack is a range of memory addresses that can be used for temporary storage of data from either within a procedure or as the normal method of passing parameters to a procedure.

The stack is normally accessed in code by the [mnemonics](#) PUSH and POP. The stack is accessed on a last on, first off basis which means that the last value pushed onto the stack is the first one to be popped back off the stack.

The next position that can be written on the stack is called the top of the stack. When a piece of data is pushed onto the stack, the processor decrements the stack pointer ESP then writes the data to the top of the stack. When a piece of data is popped back off the stack, the processor reads the data from the top of the stack then increments the stack pointer. Therefore, the stack address decreases as more data is pushed onto it, and the address increases as data is popped back off the stack.

In the following images, each square represents 1 byte on the stack, and the different colors are intended to demonstrate the different data sizes being pushed and popped. The *top* of the stack is the left side of each image.

The following sequence demonstrates the stack layout as one 32-bit and two 16-bit values are pushed and popped from the stack.

Existing stack layout:



Pushes

Push a 32 bit value (PUSH EAX)



Push a 16 bit value (PUSH CX)



Push another 16 bit value (PUSH DX)



Pops

Pop a 16 bit value (POP DX)



Pop second 16 bit value (POP CX)



Pop the 32-bit value (POP EAX)



If you wrote the following code:

```
! MOV EDX, 100
! MOV ECX, 500

' push the 2 values onto the stack
! PUSH EDX ; EDX has the value 100
! PUSH ECX ; ECX has the value 500

' pop the 2 values off the stack
! POP EAX ; EAX has the value 500
! POP ECX ; ECX has the value 100
```

Classic PowerBASIC conforms to the 32-bit Windows convention that specifies that certain registers must be preserved around blocks of assembly code, namely EBX, ESI, and EDI. While EAX, ECX, and EDX can be modified freely within a procedure, some conditions apply to their use too. See [Saving registers](#) for more detailed information.

See Also

[The Inline Assembler](#)

[Balancing the stack](#)

[Tricks of the stack](#)

[Stack Overhead Reduction](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

Balancing the stack

[Top](#) [Previous](#) [Next](#)

An important consideration when using the stack is to be symmetrical in the byte count of what is pushed and what is popped.

If the stack is not balanced on exit from an assembly code block (i.e., you POP too few or too many [registers](#)), your routine will return to the wrong location in your code. This is because Classic PowerBASIC must assume that the last item on the stack is the address to which it should return.

In other words, if the stack is not *balanced* on exit from an assembly code block, program execution is likely to resume at the wrong address and instantly crash the program. In most instances, if you PUSH a given data size onto the stack, you must POP the same data size.

See Also

[The Inline Assembler](#)

[The stack](#)

[Tricks of the stack](#)

The [stack](#) is very flexible in what can be pushed and popped. There are a few tricks that are very useful when using the stack, you can push a 32-bit value and then pop two 16-bit values.

```
' Push a 32 bit value onto the stack
! PUSH EDX

' Now pop two 16 bit values off the stack
! POP AX
! POP CX
```

Even though the pushed data size is different to the popped data size, four bytes have been pushed onto the stack and four bytes have been popped back off the stack so the [stack is balanced](#).

The stack can be used for many different things, you can push a [register](#) and pop it later when you need it so that you do not need to allocate a memory [variable](#) to put it in. You can use the stack to move a piece of data between memory [operands](#) and registers.

```
! PUSH ECX
! POP memVar
```

or:

```
! PUSH memVar
! POP EDX
```

or between two memory variables:

```
! PUSH memVar1
! POP memVar2
```

instead of using a register:

```
! MOV EDX, memvar1
! MOV memvar2, EDX
```

A collection of small tricks of this type free up the number of registers that you can use in your procedures, provided the stack is managed carefully.

Before the end of your routine, you should make sure that all the registers you have pushed onto the stack have also been popped from the stack. It is easy to make a mistake in this area, especially if the routine conditionally PUSHes and POPs any registers.

See Also

[The Inline Assembler](#)

[The stack](#)

[Balancing the stack](#)

[Stack Overhead Reduction](#)

Stack Overhead Reduction

[Top](#) [Previous](#) [Next](#)

There may be some instances where you wish to repeatedly call a very small [SUB](#) and this may produce a situation where the normally modest [stack](#) overhead may actually become a factor in the speed of the entire algorithm.

To help boost performance in such cases, Classic PowerBASIC offers the often-overlooked [GOSUB/RETURN](#) statements, which can be used in place of a call to a [SUB/END SUB block](#). Where stack overhead reduction is critical, you can create a [Label](#) in the code below the end of your normal code (but still within the current Sub/[Function/Method/Property](#) block).

You may then take the [Inline Assembler](#) code from the target SUB, and place it right after the Label. Finally, a RETURN statement is added so that execution resumes at the next instruction after the GOSUB. Such code would look something like this:

```
FUNCTION MyFunc() AS LONG
  ' Inline Assembler code
  GOSUB LABEL
  ' More Inline Assembler code
  EXIT FUNCTION ' or EXIT SUB

LABEL:
  ' Your Inline Assembler code
  RETURN
END FUNCTION ' or END SUB
```

This technique is very efficient because the variables used in the Inline Assembler Sub/Function/Method/Property (that have been moved from a SUB, back into the calling code) are maintained within the same scope as the calling code, and can therefore be used without having to pass them on the stack. The result is that we have eliminated virtually all the stack overhead involved in repeatedly calling a SUB.

Finally, you can even use the standard Intel assembler notation !CALL and !RET within the Sub/Function/Method/Property, to jump to the Label and return from it to the next instruction. For example:

```
FUNCTION MyFunc() AS LONG
  ' Inline Assembler code
  ! CALL LABEL
  ' More Inline Assembler code
  EXIT FUNCTION ' or EXIT SUB

LABEL:
  ' Your Inline Assembler code
  ! RET
END FUNCTION ' or END SUB
```

Finally, it is very important to note that you must **NEVER** exit a Classic PowerBASIC procedure with the RET instruction. Classic PowerBASIC procedures automatically perform their own stack cleanup (of local variables, etc) when an END SUB/FUNCTION/METHOD/PROPERTY or EXIT SUB/FUNCTION/METHOD/PROPERTY statement is executed, whereas an RET instruction would try to force a procedure exit without the internal stack cleanup being performed. A RET instruction cannot ever be used

as a substitute for these BASIC statements.

In summary, your program will fail with a spectacular Stack Fault (GPF) if you attempt to terminate a Classic PowerBASIC procedure with RET mnemonic.

See Also

[The Inline Assembler](#)

[The stack](#)

[Balancing the stack](#)

[Tricks of the stack](#)

[Saving registers](#)

When writing assembler code in 32-bit Windows, there is a convention that governs the use of [registers](#) so programmers can interact with the Windows API functions in a predictable and completely reliable way.

However, the registers available with an 80x86 processor are a very limited resource, and they are used by every application (process) running and also by the operating system itself. Therefore, a reliable method of using registers is very important to the process of writing reliable assembler code.

An 80x86 processor has eight general-purpose integer registers, EAX, EBX, ECX, EDX, ESI, EDI, [ESP, and EBP](#). Of these, ESP and EBP are almost exclusively used to manage the entry and exit to a procedure, so there are effectively just six general-purpose registers available for application level programming.

Following on, the Windows convention splits the remaining registers so that 3 can be freely modified (EAX, ECX, and EDX) within the [Sub/Function/Method/Property](#) that uses them, while the other 3 must be preserved (EBX, ESI, and EDI) by the procedure. For the sake of this discussion, we'll refer to these two sets as *scratch* and *volatile* registers respectively.

In summary, Classic PowerBASIC automatically preserves EBX, ESI, and EDI at the procedure level, but the programmer is responsible for preserving both the scratch and volatile registers within the procedure.

"Preserving the registers" does not necessarily mean that you must push all the registers on the stack, though that is the usual way of ensuring their safety. Simple routines might not modify any of the registers; in which case, you may not need to take any precautions. We use may because it's best to avoid making assumptions, especially with assembler programming. It is better to be safe than sorry. When in doubt, preserve (save and restore) all of the registers.

See Also

[The Inline Assembler](#)

[Registers](#)

[Saving Registers at the procedure level](#)

[Saving the FPU registers](#)

[Tricks in preserving registers](#)

Saving Registers at the Procedure level

[Top](#) [Previous](#)
[Next](#)

To conform to the Windows programming conventions, Classic PowerBASIC must provide a "safe" environment for the range of functions that are available. This is achieved by transparently preserving the three volatile [registers](#) at the start of each [Sub/Function/Method/Property](#), and restoring these same registers before exit from the procedure. The following example shows approximately how Classic PowerBASIC constructs the entry and exit of a procedure to preserve these registers:

```
SUB MySub(Params)
! PUSH EBX ' Automatically added by Classic PowerBASIC
! PUSH ESI ' ---"---
! PUSH EDI ' ---"---

' the actual SUB code is placed here

! POP  EDI ' Automatically added by Classic PowerBASIC
! POP  ESI ' ---"---
! POP  EBX ' ---"---
END SUB
```

When writing a procedure in Classic PowerBASIC, we can safely predict that the EBX, ESI, and EDI registers will be *automatically* saved upon entry and restored upon exit from a procedure.

The virtue of code that observes these conventions is that it allows the programmer to safely assume that a call to any other procedure or API function is certain to follow the same register preservation rules for the EBX, ESI, and EDI registers. This helps ensure that writing Inline Assembler code in Classic PowerBASIC will result in reliable and completely predictable code execution in terms of register use when calling Classic PowerBASIC and API procedures.

The Classic PowerBASIC compiler is also very efficient in the way it calls API system functions. For example, the following BASIC statement which calls the *SendMessage* API:

```
CALL SendMessage(hWnd&, %WM_COMMAND, 50, 100)
```

is translated into assembly code in the compiled program, to resemble something like this:

```
PUSH 100
PUSH 50
PUSH %WM_COMMAND
PUSH hWnd&
CALL SendMessage
```

This direct low level translation is one of the main reasons why Classic PowerBASIC programmers can easily mix API code and assembler code. However, when it comes to intermixing assembler and BASIC code within a procedure, the programmer must take additional care.

See Also

[The Inline Assembler](#)

[Saving registers](#)

[Using ESP and EBP](#)

[Saving the FPU registers](#)

[Tricks in preserving registers](#)

Intermixing ASM and BASIC code

[Top](#) [Previous](#) [Next](#)

There are special conditions with [register](#) preservation that apply when writing mixed assembler and BASIC code. Classic PowerBASIC is a highly optimized compiler and among its optimizations are reductions in the [stack](#) overhead between BASIC code statements. Therefore, compiled Classic PowerBASIC code is designed to expect that the EBX, ESI, and EDI registers will remain unchanged between lines of BASIC code.

This means that if your assembler algorithm uses any of the EBX, ESI, or EDI registers, you must preserve their original state from the last line of BASIC code that precedes the [Inline Assembler](#) code. This is, you must PUSH them before your ASM code, and POP them again right before the BASIC code commences.

This may appear to be more code than is necessary but it must be remembered that the internal structure of Classic PowerBASIC does not duplicate the stack preservation that the application programmer must apply, so in terms of the stack overhead, the code is actually very efficient.

It should be noted that if your ASM code uses the EAX, ECX, or EDX registers, you should also preserve these as the internal execution of BASIC statements can also freely modify any of these three registers too.

The overall approach to preserving the registers around intermixed ASM and BASIC code is demonstrated in the following listing:

```
SUB TestProc(var1&, var2&)
  #REGISTER NONE      ' Ensure there is no conflict with
                      ' Classic PowerBASIC Register variables

  ' Code that uses EAX ECX and EDX goes here
  [statements]
  ! PUSH EAX          ' Save the scratch registers
  ! PUSH ECX
  ! PUSH EDX
  [statements]
  ' Call an API function here
  [statements]
  ! POP EDX           ' Restore the scratch registers
  ! POP ECX
  ! POP EAX
  [statements]
  ' Other ASM code that uses EAX ECX and EDX goes here
  [statements]
  ! PUSH EBX          ' Save the volatile registers
  ! PUSH ESI
  ! PUSH EDI
  [statements]
  ' Other BASIC statements here, for example:
  var1 = var2 + 2^8 - COS(var2)
  [statements]
  ! POP EDI           ' Restore from the stack
  ! POP ESI
  ! POP EBX
  [statements]
  ' More ASM code that relies on EBX, etc
```

END SUB

Using this format ensures that you are writing "safe" code and that all of the utilized registers are preserved, because:

- The EBX, ESI, and EDI registers are preserved by the Classic PowerBASIC compiler at the [Sub/Function/Method/Property](#) level.
- The EAX, ECX, and EDX registers are preserved by the application programmer around the API function call and the BASIC statements. This strategy ensures that EAX, ECX, and EDX registers are not overwritten (destroyed) by the function that is called.

With those points in mind, if there are no BASIC statements or API calls **after** the assembler code, preserving these registers is of no consequence. In this case, the automatic preservation code will take care of EBX, ESI, and EDI registers before the procedure terminates, and we can be sure that the calling code will also preserve the EAX, ECX, and EDX registers using the same conventions.

PROGRAMMING TIP: As described above, the EBX, ESI, and EDI registers are automatically preserved at the start and exit of a procedure. Therefore, if you need to use registers for counters or to store other values in your Inline Assembler code, you may use any of the EBX, ESI, or EDI registers for this purpose as they are restored when the procedure terminates. This helps ensure efficiency and can result in even slightly faster code since we do not have to preserve extra registers each time the procedure is executed.

See Also

[The Inline Assembler](#)

[Registers](#)

[Data types in registers](#)

[MMX registers](#)

[Using ESP and EBP](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Saving the FPU registers](#)

[Tricks in preserving registers](#)

Using ESP and EBP

[Top](#) [Previous](#) [Next](#)

It is possible in Classic PowerBASIC to write your own procedure within an existing [Sub](#), [Function](#), [Method](#), or [Property](#) by manually coding the [stack](#) entry and exit. This is a complicated area of [assembler](#) coding where it is very easy to crash the entire operating system if the code is not written correctly. For example:

```
! CALL procname
' Other Classic PowerBASIC code here
! JMP label      ; Jump over the procedure
```

procname:

```
! PUSH EBP      ; Preserve base pointer
! MOV EBP, ESP   ; Stack pointer into EBP

' Write your assembler code here

! MOV ESP, EBP   ; Restore stack pointer
! POP EBP        ; Restore base pointer
! RET
```

label:

```
' Other Classic PowerBASIC code here
```

There are other methods of preserving both ESP and EBP depending on personal taste and calling conventions, but you must save and restore the states of both [registers](#) if you choose to use a procedure of this type. It is very important to note that ESP and EBP must always be preserved if they are to be altered, regardless of relative position of Inline Assembler code to BASIC statements.

See Also

[Registers](#)

[MMX registers](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Data types in registers](#)

Saving the FPU registers

[Top](#) [Previous](#) [Next](#)

Whereas the CPU has [registers](#) with fixed names (EAX, etc), the FPU (Floating-Point Unit or co-processor) has [stack](#)-like registers which are numbered according to their position in the stack: ST(0) {top}, ST(1), ST(2), , ST(7) {bottom}. You deal with the FPU by loading a value onto the top of the FPU stack, or by storing the value held at the top of the stack. The latter operation may or may not involve removing the value from the stack.

Note the term loading is used to describe placing a value on the FPU stack, yet it operates more like a PUSH operation. In Classic PowerBASIC, it is not a question of which FPU registers are available, but that four registers (or stack slots) are usually available for use by the programmer. If more are required, the stack should be saved and restored accordingly.

FSAVE/FRSTOR

To preserve the entire FPU stack, the mnemonics FSAVE and FRSTOR take care of preserving, and restoring, the FPU stack (respectively). These work in a similar way to the PUSHFD and POPFD CPU, but are notoriously slow to execute and FPU programmers often avoid their use unless necessary. However, they can be useful when starting to write FPU code since they guarantee the preservation and restoration of the FPU stack.

See Also

[The Inline Assembler](#)

[Saving registers](#)

[Saving Registers at the Procedure level](#)

[Using ESP and EBP](#)

[Tricks in preserving registers](#)

When you are developing mixed [assembler](#)/API code, and you do not know what registers are used in the API functions, you can draw upon two pairs of assembler instructions that preserve *all* of the usual [registers](#) and the CPU flags as well: PUSHAD, POPAD, PUSHFD, and POPFD.

PUSHAD/POPAD

The first pair of mnemonics, PUSHAD and POPAD, save and restore the registers in a block. These [mnemonics](#) allow you to do things like display the value of a register in the middle of assembler code with a *MessageBox* API call.

```
' ...Assembler code
! PUSHAD
var& = 0
! MOV var&, EAX
MessageBox hWnd&,BYCOPY STR$(var&),"Test Value",%MB_OK
! POPAD
' ...More assembler code
```

It should be noted that the use of PUSHAD and POPAD in release code is less-than-optimal code design. That is, it does more work than is usually needed, but in the development stage, these two instructions can be very convenient.

PUSHFD/POPFD

If the code being tested has certain instructions, such as conditional jumps that depend on *flag* states within the processor, the other pair of block instructions to utilize is likely to be PUSHFD and POPFD. These preserve the state of the processor flags while code that may modify the flags is executed.

PROGRAMMING TIP: If the STD instruction is used to set the CPU direction flag, a CLD instruction must be executed before releasing control to a Windows API function or a BASIC statement.

See Also

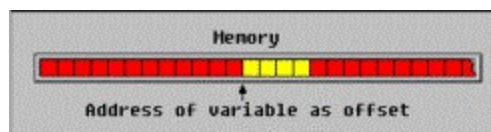
[ASM statement](#)

[The stack](#)

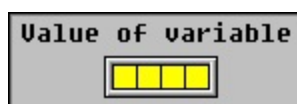
Addressing and pointers

An important distinction in assembler is the difference between the *address* of a variable and the *value* of a variable. The *address* of a variable is where it is located in memory; the *value* of a variable is what is stored at that address.

This is the ADDRESS of the variable in memory:



This is the VALUE at that address:



The method used in assembler to get the value at an address is a technique called *dereferencing*.

```
! MOV EAX, lpvar ; Copy address into EAX
! MOV EAX, [EAX] ; Dereference it
! MOV nuvar, EAX ; Copy EAX into new variable
```

Using square brackets around EAX gives access to the information at the *address* in EAX. This is the case with any 32-bit [register](#). A register enclosed in square brackets is effectively a memory operand. The size of the data accessed at the address is determined by the size of the register used to receive it. In the above example, it would be a 32-bit value since it uses a 32 bit register for the destination operand. Naturally, it can be done with 8 and 16-bit values as well using the correct size register.

Pointers

Pointers are a useful high-level language abstraction for passing addresses between procedures and performing other types of complex data manipulation.

In assembler, when you use an instruction like:

```
! LEA EAX, MyVar
```

you have put the address of a variable into the EAX register. When you take the next step and put that address into a variable of its own, you will have a POINTER to the address:

```
! LEA EAX, MyVar
! MOV lpMyVar, EAX
```

The mechanics of this process are worth understanding as it can generate errors that are hard to track down when the technique is used incorrectly.

You can pass a pointer to another procedure either by its value:

```
! MOV EAX, lpMyVar ; Copy the value into EAX
! PUSH EAX ; Push it as a parameter
! CALL MyProcedure ; Call the procedure
```

or you can pass it by reference:

```
! LEA EAX, lpMyVar ; Load the address into EAX
! PUSH EAX ; Push it as a parameter
```

```
! CALL MyProcedure    ; Call the procedure
```

When you pass an address in this manner, you have added an extra level of indirection so at the procedure end, you have a reference to a reference to an address. To get the address in the procedure, you need to dereference the variable to get back the original address:

```
! MOV EAX, lpMyVar
! MOV EAX, [EAX]
```

The original address is now stored in EAX.

See Also

[The Inline Assembler](#)

[Effective addressing](#)

The notation to calculate the effective address of data in memory can look complicated but it is in fact very clear and precise notation. In the range of allowable notation for Intel 80x86 assembler, an address in memory can be placed in a [register](#) and treated directly as a memory operand by enclosing it in square brackets.

```
! MOV EAX, lpArray ; Copy address into EAX
! MOV ECX, [EAX]   ; Copy 1st item in array into ECX
! ADD EAX, 4       ; Increment the array location by 4 bytes
! MOV ECX, [EAX]   ; Copy 2nd item in array into ECX
```

This works fine in simple situations where the register that has the address is manually incremented or decremented by the data size each time it is accessed, but there is a much more powerful and flexible technique available by using the standard Intel notation that is available.

The Intel 80x86 allows the following format to calculate the effective address of a value in memory:

[Base Address + Index * Scale + Displacement]

Base Address The register that has the starting address or *base address* of the array (or buffer) in memory.

Index The register used to determine the offset from the base address.

Scale The data size based multiplier for the index.

Displacement The additional offset adjustment from the base address.

For example:

[EBX + ECX * 4 + 8]

EBX is the *Base Address*

ECX is the *Index*

4 is the *Scale* based on the data size

8 is the *Displacement* in BYTES

Not all of the additional notation has to be used. For example, in a [Byte array](#), you can just use the *base address* and the *index*:

```
! MOV AL, [EBX + ECX]
```

The advantage of this technique is that you set the base address once and vary the index. In the case above, ECX is the index. In terms of flexibility, you have the choice of varying the base address, the index, and the displacement so that you can access data in memory by a number of different methods that best suit your code.

The only difference when using data sizes larger than Byte is that you multiply the "index" by the "scale" of the data size:

```
! MOV EAX, [EBX + ECX * 4]
```

To make a practical example let us assume we have an array of 64 items that were each 32-bits in size, and we wanted to read the 16th member of that 32-bit array. In this case, we would copy the 16th member of the zero-based index into the register that we are using as the *index*. Next, copy the address of the array into the register that you are using

as the *base address*, and finally read the value of the array member into another register.

```
! MOV ESI, lpArray          ; Base address register
! MOV ECX, 15                ; Zero-based index register
! MOV EAX, [ESI + ECX * 4] ; Copy the value into EAX
```

These three lines of code read the target value from the array into the EAX register.

If we wanted to compare the 16th and 17th members of the array and not have to use an additional register, we can add the required displacement so that we only have an extra line of code:

```
! MOV EAX, [ESI + ECX * 4]
! CMP EAX, [ESI + ECX * 4 + 4]
```

To compare the 17th and 18th members of the array, all we need to do is increment the index:

```
! INC ECX
```

Writing to the array is simply the reverse of reading it. With the same code as above:

```
! MOV ESI, lpArray          ; Base address register
! MOV ECX, 15                ; Zero-based index register
! MOV EAX, 1234
! MOV [ESI + ECX * 4], EAX
```

We can also write an *immediate* (literal) number to the array but it takes a slightly different notation:

```
! MOV DWORD PTR [ESI + ECX * 4], 1234
```

The extra notation "DWORD PTR" is because there is no way for the assembler to determine the data size from either the memory operand for the array or the immediate number. Specifying the size tells Classic PowerBASIC what data size should be written to the address contained in the memory operand.

A very similar notation is used when an array is placed on the stack by creating a [LOCAL](#) variable. With a [stack](#) variable *MyArray*, Classic PowerBASIC resolves this variable to an address on the stack, which will be something like this:

```
x& = VARPTR(Myarray(0)) ' first element
! mov edx, x&
! mov ecx, 3
! mov eax, [edx][ecx*4] ' assuming 32-bit integer

' eax = MyArray(3) ' 4th element of MyArray
```

See Also

[The Inline Assembler](#)

[Addressing and pointers](#)

[Registers](#)

[Passing parameters](#)

Classic PowerBASIC 32-bit compilers pass all parameters to procedures by pushing them in sequence from right to left. This is always the case when a procedure uses the default calling conventions of SDECL (and its synonym STDCALL), or the C calling conventions of CDECL. However, if the optional calling conventions are specified, parameters are pushed from left to right, and the called code is responsible for cleaning up the [stack](#) frame before returning. Classic PowerBASIC [Subs](#) and [Functions](#) that use the BDECL convention automatically clean up the stack before returning execution to the calling code.

By default, Classic PowerBASIC passes parameters by reference: a 32-bit pointer to the data. You can also pass most [parameters by value](#), by declaring with the optional BYVAL keyword. When a parameter is passed by value, the actual value of the parameter is pushed on the stack.

[Fixed-length strings](#), [ASCIIZ strings](#), and [User-Defined Types/Unions](#) may also be passed as BYVAL or OPTIONAL parameters, now. Try to avoid passing large items BYVAL, as its terribly inefficient, and there is a maximum size limit of 64 Kb for a given parameter list. [Arrays](#) cannot be passed BYVAL.

Classic PowerBASIC automatically sets up a local "stack frame" at the beginning of each procedure in your program. As per standard conventions, the [EBP register](#) is used to address the parameters. The lowest parameter can be found at EBP+8, and subsequent parameters will be found in adjacent locations on the stack.

In [assembler](#) routines, it is easier and safer to access parameters by name rather than calculating their locations on the stack. However, it is important to remember the difference in accessing parameters passed by value and [parameters passed by reference](#).

See Also

[The Inline Assembler](#)

[Parameters passed by reference or by copy](#)

[Parameters passed by value](#)

[Passing arrays](#)

[Passing dynamic strings](#)

[Accessing Classic PowerBASIC variables by name](#)

Parameters passed by reference or by copy

[Top](#) [Previous](#)
[Next](#)

When a parameter is passed by reference (the default method), Classic PowerBASIC passes a 32-bit [pointer](#) on the [stack](#). That pointer is the actual 32-bit offset, or memory location, of the variable to be utilized as a parameter. A 32-bit pointer occupies exactly four bytes of stack space. A parameter passed by reference is typically accessed in this way:

```
SUB MyProc(xyz&)      ' This will increment the
                      ' parameter variable by one
! PUSH EBX
! MOV  EBX, xyz&
! INC  DWORD PTR [EBX]
! POP  EBX
END SUB
```

Parameters passed by copy (such as expressions or constants) also take precisely 4 bytes on the stack. In this case, just as in parameters by reference, the item on the stack is not the value of the parameter. Rather, it is the address of a temporary location in memory where the value is stored. This may seem roundabout, but it has two distinct advantages. First, assembler routines can handle parameters BYREF and BYCOPY in precisely the same way. Second, routines can modify the value of a parameter without altering the original [variable](#) in the main program.

Suppose the first and only parameter is a [Long-integer](#). In that case, you can put the integer value into the ECX [register](#) by writing:

```
SUB MySub(xyz&)
! PUSH EBX
! MOV  EBX, xyz&    ; EBX is a pointer to xyz&
! MOV  ECX, [EBX]   ; ECX now contains xyz&
' ...more code would go in here
! POP  EBX
END SUB
```

In these cases, you must use the correct and complete address to access the value. But regardless of whether the parameter represents a variable, an expression, or a [literal constant](#), or whether it was passed by reference or by copy, the routine will always work correctly.

See Also

[The Inline Assembler](#)

[Passing parameters](#)

[Parameters passed by value](#)

[Passing arrays](#)

[Passing dynamic strings](#)

Passing arrays

[Top](#) [Previous](#) [Next](#)

Each [array](#) in your program has an associated array *descriptor*. This descriptor is saved in a proprietary format, which may change from version to version of PowerBASIC. Since most of the information in the descriptor is not relevant to assembler code, it is usually best to simply pass a [pointer](#) to the first element of the array instead. You can use the [VARPTR](#) function to retrieve that address. Subsequent elements of the array will immediately follow the first in memory, while multi-dimensional arrays are stored in column-major order.

In addition to the [LBOUND](#) and [UBOUND](#) functions, the [ARRAYATTR](#) function can be used to obtain array attributes and information on a given array.

See Also

[The Inline Assembler](#)

[Passing parameters](#)

[Parameters passed by reference or by copy](#)

[Parameters passed by value](#)

[Passing dynamic strings](#)

[Accessing PowerBASIC variables by name](#)

Passing dynamic strings

[Top](#) [Previous](#) [Next](#)

A [dynamic string](#) variable is defined as a 32-bit data item, which contains a [pointer](#) (or offset) to the string characters. When [passed by value](#), the parameter is actually a 32-bit offset of the data. When [passed by reference or by copy](#), the parameter is a pointer to another pointer that contains the offset of the actual string data. A dynamic string passed by reference is usually accessed in this way:

```
SUB MyProc(abc$)
! PUSH EBX
! MOV EBX, abc$ ; EBX is a pointer to the string handle
! MOV EBX, [EBX] ; EBX is now a pointer to string data
! MOV AL, [EBX] ; AL contains the 1st char of the string
' more code could go here
! POP EBX
END SUB
```

If you need to determine the current length of a dynamic string, there are two ways to do so. The end of string is always followed by a nul, [CHR\\$\(0\)](#), so it is possible to scan the string for the first occurrence. Of course, this will only work if there are no embedded nul bytes in the string data. An alternative method is to read the 32-bit [Long-integer](#) that immediately precedes the start of the string data, as the current length is always stored there.

PowerBASIC also calculates [string literals](#) in reverse order, in keeping with standard [assembler](#) operation. For example:

```
FUNCTION ab(x???) AS DWORD
! PUSH EBX
! MOV EBX, x???
! MOV DWORD PTR [EBX], "ABCD"
! POP EBX
END FUNCTION
```

The above code stores the value &H41424344 in the [DWORD](#) variable x, passed BYREF from the calling code. However, since the Intel platform uses little-endian numeric data format, the actual bytes are written to memory in the reverse order. For example, if we were to call the code above, and examine the actual memory locations of the passed parameter after the function call, we can see the effect of the reverse memory storage:

```
DIM a AS STRING, x AS DWORD
CALL ab(x)
a = HEX$(x,8) ' a = "41424344"
a = PEEK$(VARPTR(x),4) ' a = "DCBA"
```

See Also

[The Inline Assembler](#)

[Passing parameters](#)

[Parameters passed by reference or by copy](#)

[Parameters passed by value](#)

[Passing dynamic strings](#)

[Accessing PowerBASIC variables by name](#)

Accessing PowerBASIC variables by name

[Top](#) [Previous](#)
[Next](#)

Most variables in a PowerBASIC module are visible to Inline Assembler code created with the [ASM](#) statement. You can reference [LOCAL](#), [STATIC](#), and [GLOBAL](#) variables by name by simply using the name as an operand of the assembler opcode. That isn't possible with [INSTANCE](#) and [THREADED](#) variables, as their access requires multiple operations best handled by higher level PowerBASIC code. You can also reference procedure parameters by name, though you must differentiate between parameters passed by reference (BYREF), and those passed by value (BYVAL). Any variable referenced in an assembly-language statement must be defined prior to use.

```
SUB DoStuff (BYVAL c&)  
  LOCAL a%, b$  
  a% = 7                ' Local variable a%  
  ! PUSH EBX  
  ! MOV  AX, a%          ; Move value to AX  
  ! ADD  a%, AX          ; Add value back to a%  
  b$ = "LINDA"          ' Local variable b$  
  ! MOV  EBX, b$         ' Address of b$  
  ! MOV  [EBX], "l"      ' Put lowercase "l" in first position  
  ! MOV  EAX, c&         ' Put c& into EAX  
  ! INC  EAX             ' Increment its value  
  ! MOV  c&, EAX        ' Put it back  
  ! POP  EBX  
END SUB
```

See Also

[The Inline Assembler](#)

Commenting Assembly code

[Top](#) [Previous](#) [Next](#)

On assembly code lines, a semi-colon (;) is typically used for comments, although an apostrophe (') is still valid. For example:

```
SUB KerPlunk
  ASM PUSH EBX           ; Save EBX
  ASM MOV  EAX, 5         ; Put 5 into EAX
    ! MOV  EBX, &HFF      ; Put FFh into EBX
    ! ADD  EAX, EBX       ' EAX = EAX + EBX
    ! POP  EBX           ' Restore EBX
END SUB
```

See Also

[The Inline Assembler](#)

What is a Resource File?

[Top](#) [Previous](#) [Next](#)

A resource file may contain a collection of icons, menus, dialog boxes, strings tables, user-defined binary data and other types of items.

Once compiled into a suitable format, a resource file can be embedded directly into an executable or [DLL](#) file, **producing a single EXE or DLL containing both code and resources**. At run-time, the application can use the resource items in the embedded file. The process of creating a resource is straightforward, and is similar to compiling a PowerBASIC program.

The following sections describe the (manual) techniques involved in compiling resource scripts into a usable format, and describe the Resource Script.

See Also

[Resource Editors](#)

[Resource Compiling](#)

[Resource Scripts](#)

[Converting a .RC to a .RES](#)

[Converting a .RES to a .PBR](#)

The most popular technique is to use a Resource Editor. A Resource Editor is a tool that lets you design and test dialog boxes visually, instead of defining individual dialog statements in a resource script by hand. Using a Resource Editor, you can add, modify, rearrange, and delete controls and resources in a [resource script](#) file.

Resource Editors such as Microsoft's "Visual Studio" and Borland's "Resource Workshop" also make it easy to place string tables, version info tables, bitmaps, icons, and other types of resources into a resource script.

See Also

[What is a Resource File?](#)

[Resource Compiling](#)

[Resource Scripts](#)

We begin with a plain text file and compile it into a binary format that can be utilized by PowerBASIC. The plain text file is termed a [Resource Script](#) and these are stored with a .RC file extension. A Resource Compiler is then used to create a binary (.RES) file, and finally, the binary file is converted into PowerBASIC resource file format (a .PBR file).

FILE.RC >>> FILE.RES >>> FILE.PBR

The [PowerBASIC IDE](#) can be loaded with a Resource Script (.RC file) and compile the script into PowerBASIC resource file format. This is performed using the regular **Compile Current File button or the **RUN | Compile File** menu item.**

Once a .PBR file has been created, it can be embedded into an application EXE or DLL simply by using a [#RESOURCE](#) statement. During compilation, PowerBASIC automatically embeds the resource file to create a single file that contains compiled code and resources.

```
#RESOURCE "DIALOGS.PBR"
```

See Also

[What is a Resource File?](#)

[Resource Editors](#)

[Resource Scripts](#)

A resource script (.RC) file contains statements that define all of the items that will be included in the compiled binary resource file. Each statement describes a resource item, along with an identifier (*ID*) and any additional parameters (which vary according to the type of resource). A resource script can even reference a resource item that is stored in a separate file, such as a bitmap or icon.

A resource identifier can be numeric in the range 1 to 65535, or alphanumeric. When a PowerBASIC application needs to use a resource from the embedded resource file, it uses the resource's ID to identify it.

Hand-written scripts

There are several ways to create a resource script (.RC) file. The first technique is to write the file by hand, using a text editor like Notepad. This method is quite suitable for creating small resource scripts containing only a handful of statements.

Here is an example of a small handwritten resource script containing an icon and a version information block:

```
#include "resource.h"
ICON1 ICON "MYICON.ICO"
VS_VERSION_INFO VERSIONINFO
FILEVERSION 1, 5, 0, 0
PRODUCTVERSION 1, 5, 0, 0
FILEOS VOS_WINDOWS32
FILETYPE VFT_APP
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904E4"
        BEGIN
            VALUE "CompanyName",        "PowerBASIC, Inc.\000"
            VALUE "FileDescription",     "Program description\000"
            VALUE "FileVersion",         "01.50.0000\000"
            VALUE "InternalName",        "MYPROG\000"
            VALUE "OriginalFilename",    "MYPROG.EXE\000"
            VALUE "LegalCopyright",      "Copyright (c) 2008 PowerBASIC, Inc.\000"
            VALUE "LegalTrademarks",     "PowerBASIC is a trademark of PowerBASIC, _
                                         Inc.\000"
            VALUE "ProductName",         "MYPROG\000"
            VALUE "ProductVersion",      "01.50.0000\000"
            VALUE "Comments",            "Example for Windows 95/98/NT/XP/Vista.\000"
        END
    END
END
```

This script defines two resource items, whose alphanumeric IDs are ICON1 and VS_VERSION_INFO respectively. In this case, the actual icon binary data is stored in a separate file (MYICON.ICO). During compilation, the resource compiler takes the necessary information from the ICO file and includes it in the binary resource file it creates.

Complete details of the resource script language syntax and features can be found at <http://msdn.microsoft.com/en-us/library/aa381042.aspx>

See Also

[What is a Resource File?](#)

[Resource Editors](#)

[Resource Compiling](#)

Using the IDE

Firstly, ensure that the PowerBASIC IDE's [OPTIONS dialog](#) is configured to correctly point to the RC.EXE and PBRES.EXE files. Once configured, the [IDE](#) can automatically compile a .RC into a .RES file, and then produce a .PBR file from the .RES file. This is achieved in one simple step: simply load the .RC file into the IDE and select Compile.

Using the command-line Resource Compiler

To compile the .RC file, we need to run the Resource Compiler from a DOS box (command-line) to create the binary (.RES) resource file.

The resource compiler takes the filename of your modified (.RC) resource script file as a parameter, and produces a new 32-bit .RES file. For example:

```
C:\ClasPB9\BIN\RC.EXE MYAPP.RC
```

Note that you may need to change the path name to suit your individual settings. At this point, you should have a compiled binary resource file (i.e., MYAPP.RES), ready to be [converted to PowerBASIC's .PBR format](#).

See Also

[What is a Resource File?](#)

[Resource Editors](#)

[Resource Compiling](#)

[Resource Scripts](#)

[Converting a .RES to a .PBR](#)

Converting a .RES to a .PBR

[Top](#) [Previous](#) [Next](#)

The PowerBASIC Resource Converter (PBRES.EXE) converts binary resource files (.RES) into a compatible PowerBASIC format (.PBR). To use PBRES, open a DOS box and use the following syntax:

```
PBRES filename[.res]
```

where *filename* specifies a binary resource file. If a file extension is not specified, .RES is assumed.

Once PBRES converts a .RES file into .PBR format, the .PBR file can be linked into your EXE or DLL file through the [#RESOURCE](#) metastatement.

Complete details of the resource script language syntax and features can be found at <http://msdn.microsoft.com/en-us/library/aa381042.aspx>

See Also

[What is a Resource File?](#)

[Resource Editors](#)

[Resource Compiling](#)

[Resource Scripts](#)

[Converting a .RC to a .RES](#)

Visual BASIC Data Types

[Top](#) [Previous](#) [Next](#)

Both PowerBASIC and Visual Basic support the Currency data type - these are internally stored as a Quad-integer (occupying 8 bytes) with an implied decimal point at 4 digits. However, PowerBASIC also supports an Extended-currency data type with only two digits after the decimal point.

This data type is much more suited to dollar values, and uses the IEEE standard for calculations and rounding. The Extended-currency data format is signified by using two "at-signs" (@@) as the type-specifier, or using the CUX keyword in a [DIM](#) statement.

In 32-bit Windows, both PowerBASIC and Visual Basic use the OLE string engine for allocating, storing and releasing dynamic strings. The OLE API SysAllocStringLen call is made to allocate a string and the SysFreeString call is made to release a string.

Internally, however, Visual Basic and PowerBASIC store string data differently. Visual Basic stores string data using the Unicode (two bytes per character) format. PowerBASIC stores string data using the ASCII (one byte per character) format.

Typically, this will not be a problem, because Visual Basic always converts Unicode strings to ANSI (compatible with ASCII) format when passing them to a DLL or API call. On return from the call, Visual Basic will convert the string back to the Unicode format it uses internally. For example in the following code:

```
' VB code
MyString$ = "the quick brown fox jumped over the lazy dog."
Call PowerBasicDll(MyString$)
```

VB will convert the Unicode data to ANSI, call the function *PowerBasicDll*, and when the function returns, convert the string back to Unicode format. In PowerBASIC we can access the string data directly and modify it without any special API calls:

```
' PB code
SUB PowerBasicDll ALIAS "PowerBasicDll" (MyString$) EXPORT
  REPLACE "the" WITH "The" IN MyString$
END SUB
```

Many API calls require ASCIIZ, or nul-terminated strings. Visual Basic does not natively support this data-type, so it relies on a "kludge" to pass the correct data. To pass an ASCIIZ string, you add the BYVAL keyword to the [DECLARE](#) statement or [CALL](#):

```
' VB code
Call PowerBasicDll(ByVal MyString$)
```

Visual Basic converts the data in *MyString\$* to ANSI format with a Nul ([CHR\\$\(0\)](#) or [\\$NUL](#)) appended to the end of it, and passes a memory address to the location of the string data. PowerBASIC does natively support ASCIIZ string, so the BYVAL keyword is not used, as follows:

```
' PB Code
SUB PowerBasicDll ALIAS "PowerBasicDll" (MyString AS ASCIIZ) EXPORT
  REPLACE "the" WITH "The" in MyString
END SUB
```

If the called API or DLL modifies the data, it cannot extend the length past that of the data it received, or a General Protection Fault (GPF) may occur. If the modified data is shorter than the original string passed, the Visual Basic code must "fix" the string's length, by extracting all characters up to the [CHR\\$\(0\)](#) embedded in the string data.

It is also important to note that a string BYVAL in PowerBASIC is not the same thing as a

string BYVAL in Visual Basic. That is, since PowerBASIC natively supports ASCIIZ strings, it does not convert a dynamic string into an ASCIIZ string, as does Visual Basic.

PowerBASIC does exactly what you tell it: it makes a copy of the string, and passes the new string handle to the API or DLL. For example, if you have the following [DECLARE](#) statement in a Visual Basic module:

```
Declare Sub Hello Lib "HELLO.DLL" Alias "Hello" (Byval Greetings$)
```

then in PowerBASIC, you would create an almost identical exported [Sub](#), but with the BYVAL clause removed, and the string parameter changed to ASCIIZ. For example:

```
' PB code
#COMPILE DLL
DECLARE SUB Hello LIB "HELLO.DLL" ALIAS "Hello" (Greetings AS ASCIIZ)
```

```
SUB Hello ALIAS "Hello" (Greetings AS ASCIIZ) EXPORT
MSGBOX Greetings
END SUB
```

Both Visual Basic and PowerBASIC support User-Defined Types (UDT), also known as Structures. However, Visual Basic and PowerBASIC natively store the data for UDTs differently in memory.

In essence, Visual Basic uses "natural alignment" which is a technique that adds "padding" between members of UDT structures so that they are stored on boundaries that are a multiple of the member data size. Conversely, PowerBASIC does not pad UDT structures at all, and this is known as "byte alignment". First, we'll look at how Visual Basic pads a simple UDT:

```
' VB code
Type MyType1
    a As Integer
    b As Long
End Type
```

The memory address for each member of the UDT must begin at a location evenly divisible by 4 (as there are 4 bytes in a DWORD). So, the member "a" begins at offset zero and is two bytes long. In byte alignment, the next member, "b" would start at offset 2 (zero plus two equals 2). However, member "b" is 4 bytes wide, so Visual Basic inserts a "hidden" two-byte member, so that "b" can start at offset 4. This brings the total size of the UDT to 8 bytes, as opposed to 6 bytes that byte alignment would offer. This method of "padding" occurs with other data types as well:

```
' VB code
Type MyType2
    a As String * 5
    b As Long
End Type
```

If Visual Basic used simple byte alignment, the member "b" would start at offset 5. However, since this is not evenly divisible by four, three bytes are inserted to push "b" to offset 8.

In order to duplicate the storage technique used by Visual Basic, PowerBASIC offers DWORD FILL alignment, and this precisely follows Visual Basic's alignment method. For example:

```
' PB code
TYPE MyType1 DWORD FILL
    a AS INTEGER
```

```
b AS LONG
END TYPE
```

```
TYPE MyType2 DWORD FILL
  a AS STRING * 5
  b AS LONG
END TYPE
```

The DWORD FILL option tells PowerBASIC to insert any necessary "padding" between members so that the structure will be padded exactly as Visual Basic would pad it. Compared with the task of manually inserting dummy members to pad a UDT structure, the DWORD FILL alignment options offer a rapid solution to structure alignment issues between the two languages.

Finally, it should be noted that PowerBASIC does not support dynamic strings as members of UDT structures, since the size of a UDT structure must be fixed at compile-time. PowerBASIC now features native support for Variants for use with Visual Basic and when creating COM client/controller applications. In order to determine the type of data being passed in the Variant, the [VARIANTVT](#) function can be used. Strings and numeric data can be extracted with the [VARIANT#](#) and [VARIANT\\$](#) functions. See the [Variant Data Type](#) topic for more information on using Variants in PowerBASIC.

Like Visual Basic, PowerBASIC supports arrays of all supported data types. However, there are some differences on how each language deals with arrays internally.

Like Visual Basic, each element of an array follows the previous one consecutively in memory. So if element `A%(0)` is at offset zero, then `A%(1)` is at offset 4, and so on (since Long-integers are four bytes long).

Internally, however, Visual Basic and PowerBASIC reference arrays differently. Visual Basic uses a SafeArray handle created using the OLE API. This four-byte handle is then passed to the OLE API to find the first element location in memory, get the size of the array, etc. PowerBASIC, on the other hand, uses a 64-byte array descriptor, which holds all of the information about the array. The overhead of calling the OLE API to get information about the array is not necessary.

This means that you cannot simply pass a Visual Basic array to PowerBASIC and access it. There are several different methods for accessing a Visual Basic array, each with their own advantages.

If you need to access a Visual Basic array, and not change its size, the easiest way is to pass the first element of the array as one parameter, and pass the total number of elements as another parameter.

```
' VB code
Declare Sub SortInt Lib "MYDLL.DLL" Alias "SortInt" (Byref Arr As Integer, Byval iCount As Long)
...
Dim A%(1 TO 50)
A%(1) = 123
A%(2) = 456
...
Call SortLong(A%(1), 50)
```

In PowerBASIC, you would dimension a PowerBASIC array at the same memory location used by the passed Visual Basic array. To accomplish this, the DLL code expects to

receive the address of the array element, and the array can be created at that address:

```
' PB code
#COMPILE DLL "MYDLL.DLL"
SUB SortInt ALIAS "SortInt" (BYVAL FirstElem AS DWORD, BYVAL Total AS LONG) EXPORT
    DIM A(1 TO Total) AS INTEGER AT FirstElem
    ARRAY SORT A%()
END SUB
```

This allows you to read values from the array, and write new values to the array, as well as using the special ARRAY functions built-in to the PowerBASIC language ([ARRAY SORT](#), [ARRAY SCAN](#), [ARRAY INSERT](#), [ARRAY DELETE](#), etc.). The only thing you cannot do with this technique is resize the array.

The problem with the above method is when you need to access a string or UDT array passed from Visual Basic. As noted in the previous sections, Visual Basic stores string data in Unicode format. When you pass a string from Visual Basic to PowerBASIC, VB will convert the string to ANSI, so if you pass only the first element of the string array to PowerBASIC, only that particular element will be converted. All of the remaining elements in the VB array are still in Unicode format.

To access a string array or UDT array that has strings, you need to have Visual Basic pass the SafeArray handle to the array. By passing the array handle rather than the array element(s), you force Visual Basic to convert all of the strings from Unicode to ANSI format before making the call to the DLL.

```
' VB code
Declare Sub SortString Lib "MYDLL.DLL" Alias "SortString" (A$())
' more code here
Call SortString(A$())
```

Since PowerBASIC cannot access SafeArrays directly, you will need to call the OLE API to get the location of the first element in memory, as well as the size of the array. The VBAPI32.INC file in the Samples directory contains the necessary code to help you do this. Also included in the VBAPI32.INC file is a routine that lets you resize a Visual Basic array.

```
' PB code
#COMPILE DLL "MYDLL.DLL"
#include "VBAPI32.INC"
SUB SortString ALIAS "SortString" (pSA AS DWORD) EXPORT
    LOCAL l AS LONG
    LOCAL u AS LONG
    LOCAL vb AS DWORD

    l = vbArrayLBound(psa, 1)
    u = vbArrayUBound(psa, 1)
    vb = vbArrayFirstElem(psa)

    DIM A(l TO u) AS STRING AT vb
    ARRAY SORT A$()
END SUB
```

Also included in the VBAPI32.INC file is a routine that lets you resize a Visual Basic array from a PowerBASIC DLL. See the ReDim VB Array example project in the PB\SAMPLES\VB32\REDIM VB ARRAY folder.

Both Visual Basic and PowerBASIC support the following data types: [Byte](#), [Integer](#), [Long-integer](#), [Single-precision float](#), [Double-precision float](#), [Currency](#), [User-Defined Type](#), [Fixed-length string](#), [Dynamic string](#), and [Variant](#). Both products also support [arrays](#) of all data

types.

PowerBASIC also supports the following data types, which Visual Basic does not: [Word](#), [Double-word](#), [Quad-integer](#), [Extended-currency](#), [ASCIIZ string](#), [Unions](#), and [Pointers](#).

See Also

[Comparative Data Types To Visual Basic 6](#)

VB Run-time errors when calling a PowerBASIC DLL

[Top](#) [Previous](#)
[Next](#)

There is one common and avoidable error that may be encountered when first attempting to use a PowerBASIC [DLL](#) with a Visual Basic application: Error 48 "Error in loading DLL" or "DLL not found".

In almost all circumstances involving this VB error, the problem is not that VB cannot find the DLL, rather, that VB is not able to locate the specified [Sub/Function](#) inside the DLL. When this occurs, the problem is very likely to be due to mismatching capitalization of the Sub/Function and the VB Declare statement, or the Sub/Function has not been EXPORTed from the DLL.

To remedy these situations, either add an explicit ALIAS clause to the Exported PowerBASIC Subs and Functions to ensure that the exported name matches the VB Declare, or capitalize the VB Declare Sub/Function name. The latter solution works because PowerBASIC capitalizes all exported procedure (Sub, Function, Method, and Property) names that do not have an explicit ALIAS clause. For more information, please refer to the [FUNCTION](#), [SUB](#), [METHOD](#), and [PROPERTY](#) topics.

In addition, VB Errors 53 and 453 may sometimes be resolved by the addition of an ALIAS clause.

In the design environment, it is common practice to provide an explicit path to the DLL in the LIB clause of the VB Declare statement. In the final "distribution" version, such explicit paths should be removed from the VB Declare statements. When the paths are omitted, Visual Basic use the following strategy to try to locate the DLL:

1. Directory containing the calling EXE
2. Current directory
3. Windows 32-bit system directory
4. Windows 16-bit system directory
5. Windows directory
6. Folders specified in the PATH environmental variable

Therefore, it is also possible that certain VB run-time errors (especially in the design environment) may be attributed to VB failing to locate the DLL, or that VB may be loading the wrong version/copy of the DLL. When debugging such issues, place the DLL in the appropriate VB project directory, and all rename or delete any other copies.

Problems calling DLLs, or General Protection Faults (GPFs) when the application runs/closes can often be attributable to errors in the Visual Basic declarations. Visual basic declarations should generally be placed in the declarations section of a Visual Basic module, rather than elsewhere in the project to avoid scoping issues. Declarations in a

module should not use the Private Declare syntax.

General Protection Faults (GPFs) may also occur when incorrect parameters or passing methods are used with the DLL. Another source of GPF problems can occur if passed arrays are referenced beyond their boundaries from within the DLL code.

See Also

[Visual Basic Data Types](#)

Internally, the DOS and 32-bit Windows operating systems are very different. DOS applications run in 16-bit "Real Mode", which means that the largest single data object is 64 Kilobytes (the largest 16-bit value is 65535). And because of the way "memory segmentation" works, the total [addressing](#) space available in "Real Mode" is a little over 1 Megabyte. Since the CPU is running in 16-bit mode (Real mode), numeric operations are fastest when variables are 16-bits (Integers and Words).

In contrast, 32-bit Windows runs in "[Protected Mode](#)", and the largest single data object is two Gigabytes (the largest 32-bit value is actually four Gigabytes, but the operating system reserves half of that for itself). Because the CPU is running in 32-bit mode (Protected mode), numeric operations are fastest when variables are 32-bits (Long-integers and Double-words).

Use 32-bit Variables

As you move your DOS code into PowerBASIC, you should replace all "Integer" and "Word" variable types with [Long-integers](#) and [Double-words](#) respectively - particularly in [FOR/NEXT](#) loops and integer-class math calculations. It actually takes the CPU longer to perform a calculation on a 2-byte Integer than it does with a 4-byte Long-integer, and it takes even longer with single byte variables.

Use Register Variables

[Register](#) variables are variables that are stored directly in specific CPU registers, rather than in application memory. Since data in a CPU register can be accessed much faster, and with less code, Register variables are valuable optimization tools.

Register variables are always local to the [Sub](#), [Function](#), [Method](#), or [Property](#) where they appear. In the current version of PowerBASIC, there may be up to two integer-class Register variables (Word/Dword/Integer/Long), and up to four Extended-precision floats. It is possible that future versions of the compiler will change these limits, so you may declare an unlimited number of them. Any "extra" Register variables are automatically reclassified as locals.

The [REGISTER](#) statement allows you to choose which variables will be classified as Register variables. If you do not make the choice in a particular procedure, the compiler will attempt to choose for you. By default, the compiler will always assign the first two integer-class local variables available. Extended-precision float variables will be automatically assigned only in functions that contain no external function calls.

Integer class Register variables are most efficient for variables that are updated or used often, such as For/Next loop counter variables, and variables that are used repeatedly as array indexes. Floating-point Register variables should generally be chosen with a bit more caution, since the compiler must generate code to save and restore them to conventional memory around each call to a procedure. In some rather rare cases, it is possible that floating-point Register variables could actually reduce execution speed. However, they are

extremely valuable with intensive floating-point calculations and in functions that have few references to other procedures.

Due to the design of FPUs (floating point units), and the instruction sets available, the first float register variable declared in your program has far more optimization possibilities than the others do. Use care in choosing the variable which is used most within floating-point expressions (that is, on the right side of the '=' assignment operator), in order to gain the greatest advantage in execution speed. Also, remember it is typically valuable to assign floating-point numeric constants to Register variables when they are used in repetitive or intensive calculations.

You must use care with [Inline Assembler](#) floating-point [opcodes](#) in procedures that enable Register variables. Floating-point Register variables may occupy up to four of the FPU registers, so you must limit your use of the x87 registers to the remaining four. Further, floating-point Register variables may never be referenced by name from Inline Assembler code, as the compiler cannot always track the register locations with absolute certainty.

Register variables are preserved when a call to an external DLL or API function is made. Register variables are automatically thread-safe too.

Because Register variables are stored within the CPU, it is not possible to use [VARPTR](#) on a register variable. When passing a register variable to a procedure BYREF, the compiler temporarily converts the register variable into a memory variable, and reloads the register variable upon return from the procedure call. The overhead that this adds is insignificant.

See Also

[The Inline Assembler](#)

% # A B C D E F G H I J K L M N O P R S T U V W X

%

[%DEF operator](#)

#

- [#ALIGN metastatement](#)
- [#BLOAT metastatement](#)
- [#COM metastatement](#)
- [#COMPILE metastatement](#)
- [#COMPILER metastatement](#)
- [#DEBUG CODE metastatement](#)
- [#DEBUG DISPLAY metastatement](#)
- [#DEBUG ERROR metastatement](#)
- [#DEBUG PRINT metastatement](#)
- [#DIM metastatement](#)
- [#IF/#ELSEIF/#ELSE/#ENDIF metastatements](#)
- [#INCLUDE metastatement](#)
- [#MESSAGES metastatement](#)
- [#OPTIMIZE metastatement](#)
- [#OPTION metastatement](#)
- [#PBFORMS metastatement](#)
- [#REGISTER metastatement](#)
- [#RESOURCE metastatement](#)
- [#STACK metastatement](#)
- [#TOOLS metastatement](#)
- [#UTILITY metastatement](#)

A

- [ABS function](#)
- [ACCEL ATTACH statement](#)
- [ACODE\\$ function](#)
- [AND operator](#)

[ARRAY ASSIGN statement](#)
[ARRAY DELETE statement](#)
[ARRAY INSERT statement](#)
[ARRAY SCAN statement](#)
[ARRAY SORT statement](#)
[ARRAYATTR function](#)
[ASC function](#)
[ASC statement](#)
[ASM statement](#)
[ATN function](#)

B

[BEEP statement](#)
[BGR function](#)
[BIN\\$ function](#)
[BIT CALC statement](#)
[BIT function](#)
[BIT statement](#)
[BITS function](#)
[BITS functions](#)
[BITSE function](#)
[BUILD\\$ function](#)

C

[CALL statement](#)
[CALL DWORD statement](#)
[CALLSTK statement](#)
[CALLSTK\\$ function](#)
[CALLSTKCOUNT function](#)
[CB.CTL function](#)
[CB.CTLMSG function](#)
[CB.HNDL function](#)
[CB.LPARAM function](#)
[CB.MSG function](#)
[CB.WPARAM function](#)

[CB.NMCODE function](#)

[CB.NMHDR function](#)

[CB.NMHDR\\$ function](#)

[CB.NMHWND function](#)

[CB.NMID function](#)

[CBCTL function](#)

[CBCTLMSG function](#)

[CBHNDL function](#)

[CBLPARAM function](#)

[CBMSG function](#)

[CBWPARAM function](#)

[CBYT function](#)

[CCUR function](#)

[CCUX function](#)

[CDBL function](#)

[CDWD function](#)

[CEIL function](#)

[CEXT function](#)

[CHDIR statement](#)

[CHDRIVE statement](#)

[CHOOSE function](#)

[CHR\\$ function](#)

[CINT function](#)

[CLASS/END CLASS block](#)

[CLIPBOARD GET BITMAP](#)

[CLIPBOARD GET OEMTEXT](#)

[CLIPBOARD GET TEXT](#)

[CLIPBOARD GET UNICODE](#)

[CLIPBOARD RESET](#)

[CLIPBOARD SET BITMAP](#)

[CLIPBOARD SET OEMTEXT](#)

[CLIPBOARD SET TEXT](#)

[CLIPBOARD SET UNICODE](#)

[CLNG function](#)

[CLOSE statement](#)

[CLSID\\$ function](#)

[CODEPTR function](#)

[COMBOBOX ADD *hDlg* statement](#)

[COMBOBOX DELETE\[/****/\]statement](#)

[COMBOBOX FIND\[/****/\]statement](#)

[COMBOBOX FIND EXACT\[/****/\]statement](#)

[COMBOBOX GET COUNT\[/****/\]statement](#)

[COMBOBOX GET SELCOUNT\[/****/\]statement](#)

[COMBOBOX GET SELECT\[/****/\]statement](#)

[COMBOBOX GET STATE\[/****/\]statement](#)

[COMBOBOX GET TEXT\[/****/\]statement](#)

[COMBOBOX GET USER\[/****/\]statement](#)

[COMBOBOX INSERT\[/****/\]statement](#)

[COMBOBOX RESET\[/****/\]statement](#)

[COMBOBOX SELECT\[/****/\]statement](#)

[COMBOBOX SET TEXT\[/****/\]statement](#)

[COMBOBOX SET USER\[/****/\]statement](#)

[COMBOBOX UNSELECT\[/****/\]statement](#)

[COMM CLOSE statement](#)

[COMM function](#)

[COMM LINE statement](#)

[COMM OPEN statement](#)

[COMM PRINT statement](#)

[COMM RECV statement](#)

[COMM RESET statement](#)

[COMM SEND statement](#)

[COMM SET statement](#)

[COMMAND\\$ function](#)

[CONTROL ADD statement](#)

[CONTROL ADD BUTTON statement](#)

[CONTROL ADD CHECK3STATE statement](#)

[CONTROL ADD CHECKBOX statement](#)

[CONTROL ADD COMBOBOX statement](#)

[CONTROL ADD FRAME statement](#)

[CONTROL ADD GRAPHIC statement](#)

[CONTROL ADD IMAGE statement](#)

[CONTROL ADD IMAGEX statement](#)
[CONTROL ADD IMGBUTTON statement](#)
[CONTROL ADD IMGBUTTONX statement](#)
[CONTROL ADD LABEL statement](#)
[CONTROL ADD LINE statement](#)
[CONTROL ADD LISTBOX statement](#)
[CONTROL ADD LISTVIEW statement](#)
[CONTROL ADD OPTION statement](#)
[CONTROL ADD PROGRESSBAR statement](#)
[CONTROL ADD SCROLLBAR statement](#)
[CONTROL ADD STATUSBAR statement](#)
[CONTROL ADD TAB statement](#)
[CONTROL ADD TEXTBOX statement](#)
[CONTROL ADD TOOLBAR statement](#)
[CONTROL ADD TREEVIEW statement](#)
[CONTROL DISABLE statement](#)
[CONTROL ENABLE statement](#)
[CONTROL GET CHECK statement](#)
[CONTROL GET CLIENT statement](#)
[CONTROL GET LOC statement](#)
[CONTROL GET SIZE statement](#)
[CONTROL GET TEXT statement](#)
[CONTROL GET USER statement](#)
[CONTROL HANDLE statement](#)
[CONTROL KILL statement](#)
[CONTROL POST statement](#)
[CONTROL REDRAW statement](#)
[CONTROL SEND statement](#)
[CONTROL SET CHECK statement](#)
[CONTROL SET CLIENT statement](#)
[CONTROL SET COLOR statement](#)
[CONTROL SET FOCUS statement](#)
[CONTROL SET FONT statement](#)
[CONTROL SET IMAGE statement](#)
[CONTROL SET IMAGEX statement](#)
[CONTROL SET IMGBUTTON statement](#)

[CONTROL SET IMGBUTTONX statement](#)

[CONTROL SET LOC statement](#)

[CONTROL SET OPTION statement](#)

[CONTROL SET SIZE statement](#)

[CONTROL SET TEXT statement](#)

[CONTROL SET USER statement](#)

[CONTROL SHOW STATE statement](#)

[COS function](#)

[CQUD function](#)

[CSET statement](#)

[CSET\\$ function](#)

[CSNG function](#)

[CURDIR\\$ function](#)

[CVBYT function](#)

[CVCUR function](#)

[CVCUX function](#)

[CVD function](#)

[CVDWD function](#)

[CVE function](#)

[CVI function](#)

[CVL function](#)

[CVQ function](#)

[CVS function](#)

[CVWRD function](#)

[CWRD function](#)

D

[DATA statement](#)

[DATACOUNT function](#)

[DATE\\$ system variable](#)

[DECLARE statement](#)

[DECR statement](#)

[DEFBYT statement](#)

[DEFCUR statement](#)

[DEFCUX statement](#)

[DEFDBL statement](#)
[DEFDWD statement](#)
[DEFEXT statement](#)
[DEFINT statement](#)
[DEFLNG statement](#)
[DEFQUD statement](#)
[DEFSNG statement](#)
[DEFSTR statement](#)
[DEFWRD statement](#)
[DESKTOP GET CLIENT statement](#)
[DESKTOP GET LOC statement](#)
[DESKTOP GET SIZE statement](#)
[DIALOG DISABLE statement](#)
[DIALOG DOEVENTS statement](#)
[DIALOG ENABLE statement](#)
[DIALOG END statement](#)
[DIALOG FONT statement](#)
[DIALOG GET CLIENT statement](#)
[DIALOG GET LOC statetement](#)
[DIALOG GET SIZE statement](#)
[DIALOG GET TEXT statement](#)
[DIALOG GET USER statement](#)
[DIALOG NEW statement](#)
[DIALOG PIXELS statement](#)
[DIALOG POST statement](#)
[DIALOG REDRAW statement](#)
[DIALOG SEND statement](#)
[DIALOG SET CLIENT Statement](#)
[DIALOG SET COLOR statement](#)
[DIALOG SET ICON statement](#)
[DIALOG SET LOC statement](#)
[DIALOG SET SIZE statement](#)
[DIALOG SET TEXT statement](#)
[DIALOG SET USER statement](#)
[DIALOG SHOW MODAL statement](#)
[DIALOG SHOW MODELESS statement](#)

[DIALOG SHOW STATE statement](#)

[DIALOG UNITS statement](#)

[DIM statement](#)

[DIR\\$ function](#)

[DIR\\$ CLOSE statement](#)

[DISKFREE function](#)

[DISKSIZE function](#)

[DISPLAY BROWSE statement](#)

[DISPLAY COLOR statement](#)

[DISPLAY FONT statement](#)

[DISPLAY OPENFILE statement](#)

[DISPLAY SAVEFILE statement](#)

[DLLMAIN function](#)

[DO/LOOP statements](#)

E

[ENVIRON statement](#)

[ENVIRON\\$ function](#)

[EOF function](#)

[EQV operator](#)

[ERASE statement](#)

[ERL system variable](#)

[ERL\\$ function](#)

[ERR system variable](#)

[ERRCLEAR system variable](#)

[ERROR statement](#)

[ERROR\\$ function](#)

[EVENT SOURCE statement](#)

[EVENTS statement](#)

[EXE read-only user defined type](#)

[EXIT statement](#)

[EXP function](#)

[EXP2 function](#)

[EXP10 function](#)

[EXTRACT\\$ function](#)

F

[FIELD statement](#)
[FIELD RESET statement](#)
[FIELD STRING statement](#)
[FILEATTR function](#)
[FILECOPY statement](#)
[FILENAME\\$ function](#)
[FILESCAN statement](#)
[FIX function](#)
[FLUSH statement](#)
[FONT END statement](#)
[FONT NEW statement](#)
[FOR/NEXT statements](#)
[FORMAT\\$ function](#)
[FRAC function](#)
[FREEFILE function](#)
[FUNCNAME\\$ function](#)
[FUNCTION/END FUNCTION statements](#)

G

[GET statement](#)
[GET\\$ statement](#)
[GETATTR function](#)
[GLOBAL statement](#)
[GLOBALMEM ALLOC statement](#)
[GLOBALMEM FREE statement](#)
[GLOBALMEM LOCK statement](#)
[GLOBALMEM SIZE statement](#)
[GLOBALMEM UNLOCK statement](#)
[GOSUB statement](#)
[GOSUB DWORD statement](#)
[GOTO statement](#)
[GOTO DWORD statement](#)
[GRAPHIC ARC statement](#)

[GRAPHIC ATTACH statement](#)
[GRAPHIC BITMAP END statement](#)
[GRAPHIC BITMAP LOAD statement](#)
[GRAPHIC BITMAP NEW statement](#)
[GRAPHIC BOX statement](#)
[GRAPHIC CHR SIZE statement](#)
[GRAPHIC CLEAR statement](#)
[GRAPHIC COLOR statement](#)
[GRAPHIC COPY statement](#)
[GRAPHIC DETACH statement](#)
[GRAPHIC ELLIPSE statement](#)
[GRAPHIC FONT statement](#)
[GRAPHIC GET BITS statement](#)
[GRAPHIC GET CLIENT statement](#)
[GRAPHIC GET DC statement](#)
[GRAPHIC GET LINES statement](#)
[GRAPHIC GET LOC statement](#)
[GRAPHIC GET MIX statement](#)
[GRAPHIC GET PIXEL statement](#)
[GRAPHIC GET POS statement](#)
[GRAPHIC GET PPI statement](#)
[GRAPHIC GET SCALE statement](#)
[GRAPHIC IMAGELIST statement](#)
[GRAPHIC INKEY\\$ statement](#)
[GRAPHIC INPUT statement](#)
[GRAPHIC INSTAT statement](#)
[GRAPHIC LINE statement](#)
[GRAPHIC LINE INPUT statement](#)
[GRAPHIC PAINT statement](#)
[GRAPHIC PIE statement](#)
[GRAPHIC POLYGON statement](#)
[GRAPHIC POLYLINE statement](#)
[GRAPHIC PRINT statement](#)
[GRAPHIC REDRAW statement](#)
[GRAPHIC RENDER statement](#)
[GRAPHIC SAVE statement](#)

[GRAPHIC SCALE statement](#)

[GRAPHIC SET BITS statement](#)

[GRAPHIC SET FOCUS statement](#)

[GRAPHIC SET FONT statement](#)

[GRAPHIC SET LOC statement](#)

[GRAPHIC SET MIX statement](#)

[GRAPHIC SET PIXEL statement](#)

[GRAPHIC SET POS statement](#)

[GRAPHIC STRETCH statement](#)

[GRAPHIC STYLE statement](#)

[GRAPHIC TEXT SIZE statement](#)

[GRAPHIC WAITKEY\\$ statement](#)

[GRAPHIC WIDTH statement](#)

[GRAPHIC WINDOW statement](#)

[GRAPHIC WINDOW CLICK statement](#)

[GRAPHIC WINDOW END statement](#)

[GUID\\$ function](#)

[GUIDTXT\\$ function](#)

H

[HEX\\$ function](#)

[HI function](#)

[HIBYT function](#)

[HIINT function](#)

[HIWRD function](#)

[HOST ADDR statement](#)

[HOST NAME statement](#)

I

[IDISPINFO pseudo-object](#)

[IF statement](#)

[IF/END IF block](#)

[IIF function](#)

[IMAGELIST ADD BITMAP statement](#)

[IMAGELIST ADD BITMAP statement](#)

[IMAGELIST ADD ICON statement](#)
[IMAGELIST ADD ICON statement](#)
[IMAGELIST ADD MASKED statement](#)
[IMAGELIST ADD MASKED statement](#)
[IMAGELIST GET COUNT statement](#)
[IMAGELIST KILL statement](#)
[IMAGELIST NEW BITMAP statement](#)
[IMAGELIST NEW ICON statement](#)
[IMAGELIST SET OVERLAY statement](#)
[IMP operator](#)
[INCR statement](#)
[INPUT# statement](#)
[INPUTBOX\\$ function](#)
[INSTANCE statement](#)
[INSTR function](#)
[INT function](#)
[INTERFACE / END INTERFACE Block \(Direct\)](#)
[INTERFACE/END INTERFACE block \(IDBind\)](#)
[ISFALSE operator](#)
[ISFILE Function](#)
[ISFOLDER Function](#)
[ISINTERFACE Function](#)
[ISMISSING function](#)
[ISNOTHING function](#)
[ISOBJECT function](#)
[ISTRUE operator](#)
[ISWIN function](#)
[ITERATE statement](#)

J

[JOIN\\$ function](#)

K

[KILL statement](#)

L

- [LBOUND function](#)
- [LCASE\\$ function](#)
- [LEFT\\$ function](#)
- [LEN function](#)
- [LET statement](#)
- [LET statement \(with Objects\)](#)
- [LET statement \(with Types\)](#)
- [LET statement \(with Variants\)](#)
- [LIBMAIN function](#)
- [LINE INPUT# statement](#)
- [LISTBOX ADD statement](#)
- [LISTBOX DELETE statement](#)
- [LISTBOX FIND statement](#)
- [LISTBOX FIND EXACT statement](#)
- [LISTBOX GET COUNT statement](#)
- [LISTBOX GET SELCOUNT statement](#)
- [LISTBOX GET SELECT statement](#)
- [LISTBOX GET STATE statement](#)
- [LISTBOX GET TEXT statement](#)
- [LISTBOX GET USER statement](#)
- [LISTBOX INSERT statement](#)
- [LISTBOX RESET statement](#)
- [LISTBOX SELECT statement](#)
- [LISTBOX SET TEXT statement](#)
- [LISTBOX SET USER statement](#)
- [LISTBOX UNSELECT statement](#)
- [LISTVIEW DELETE COLUMN statement](#)
- [LISTVIEW DELETE ITEM statement](#)
- [LISTVIEW FIND statement](#)
- [LISTVIEW FIND EXACT statement](#)
- [LISTVIEW FIT CONTENT statement](#)
- [LISTVIEW FIT HEADER statement](#)
- [LISTVIEW GET COLUMN statement](#)
- [LISTVIEW GET COUNT statement](#)

[LISTVIEW GET HEADER statement](#)
[LISTVIEW GET MODE statement](#)
[LISTVIEW GET SELCOUNT statement](#)
[LISTVIEW GET SELECT statement](#)
[LISTVIEW GET STATE statement](#)
[LISTVIEW GET STYLEXX statement](#)
[LISTVIEW GET TEXT statement](#)
[LISTVIEW GET USER statement](#)
[LISTVIEW INSERT COLUMN statement](#)
[LISTVIEW INSERT ITEM statement](#)
[LISTVIEW RESET statement](#)
[LISTVIEW SELECT statement](#)
[LISTVIEW SET COLUMN statement](#)
[LISTVIEW SET HEADER statement](#)
[LISTVIEW SET IMAGE statement](#)
[LISTVIEW SET IMAGE2 statement](#)
[LISTVIEW SET IMAGELIST statement](#)
[LISTVIEW SET MODE statement](#)
[LISTVIEW SET OVERLAY statement](#)
[LISTVIEW SET STYLEXX statement](#)
[LISTVIEW SET TEXT statement](#)
[LISTVIEW SET USER statement](#)
[LISTVIEW SORT statement](#)
[LISTVIEW UNSELECT statement](#)
[LISTVIEW VISIBLE statement](#)
[LO function](#)
[LOBYT function](#)
[LOC function](#)
[LOCAL statement](#)
[LOCK statement](#)
[LOF function](#)
[LOG function](#)
[LOG2 function](#)
[LOG10 function](#)
[LOINT function](#)
[LOWRD function](#)

[LPRINT statement](#)

[LPRINT ATTACH statement](#)

[LPRINT CLOSE statement](#)

[LPRINT FLUSH statement](#)

[LPRINT FORMFEED statement](#)

[LPRINT\\$ function](#)

[LSET statement](#)

[LSET\\$ function](#)

[LTRIM\\$ function](#)

M

[MACRO/END MACRO block](#)

[MAK function](#)

[MAKDWD function](#)

[MAKINT function](#)

[MAKPTR function](#)

[MAKLNG function](#)

[MAKWRD function](#)

[MAT statement](#)

[MAX function](#)

[MCASE\\$ function](#)

[ME pseudo-variable](#)

[MENU ADD POPUP statement](#)

[MENU ADD STRING statement](#)

[MENU ATTACH statement](#)

[MENU DELETE statement](#)

[MENU DRAW BAR statement](#)

[MENU GET STATE statement](#)

[MENU GET TEXT statement](#)

[MENU NEW BAR statement](#)

[MENU NEW POPUP statement](#)

[MENU SET STATE statement](#)

[MENU SET TEXT statement](#)

[METHOD / END METHOD statements](#)

[MID\\$ function](#)

[MID\\$ statement](#)
[MIN function](#)
[MKBYT\\$ function](#)
[MKCUR\\$ function](#)
[MKCUX\\$ function](#)
[MKD\\$ function](#)
[MKDIR statement](#)
[MKDWD\\$ function](#)
[MKE\\$ function](#)
[MKI\\$ function](#)
[MKL\\$ function](#)
[MKQ\\$ function](#)
[MKS\\$ function](#)
[MKWRD\\$ function](#)
[MOD operator](#)
[MOUSEPTR statement](#)
[MSGBOX function](#)
[MSGBOX statement](#)
[MYBASE pseudo-variable](#)

N

[NAME statement](#)
[NEXT statement](#)
[NOT operator](#)
[NUL\\$ function](#)

O

[OBJACTIVE function](#)
[OBJECT GET statement](#)
[OBJECT LET statement](#)
[OBJECT SET statement](#)
[OBJECT CALL statement](#)
[OBJECT RAISEEVENT statement](#)
[OBJPTR function](#)
[OBJRESULT function](#)

[OBJRESULT\\$ function](#)

[OCT\\$ function](#)

[ON ERROR statement](#)

[ON GOSUB statement](#)

[ON GOTO statement](#)

[OPEN statement](#)

[OPTION EXPLICIT statement](#)

[OR operator](#)

P

[PARSE statement](#)

[PARSE\\$ function](#)

[PARSECOUNT function](#)

[PATHNAME\\$ function](#)

[PATHSCAN\\$ function](#)

[PBLIBMAIN function](#)

[PBMAIN function](#)

[PEEK function](#)

[PEEK\\$ function](#)

[POKE statement](#)

[POKE\\$ statement](#)

[PRINT# statement](#)

[PRINTER\\$ function](#)

[PRINTERCOUNT function](#)

[PROCESS GET PRIORITY statement](#)

[PROCESS SET PRIORITY statement](#)

[PROFILE statement](#)

[PROGID\\$ function](#)

[PROGRESSBAR GET POS statement](#)

[PROGRESSBAR GET RANGE statement](#)

[PROGRESSBAR SET POS statement](#)

[PROGRESSBAR SET RANGE statement](#)

[PROGRESSBAR SET STEP statement](#)

[PROGRESSBAR STEP statement](#)

[PROPERTY GET statement](#)

[PROPERTY SET statement](#)

[PUT statement](#)

[PUT\\$ statement](#)

R

[RAISEEVENT statement](#)

[RANDOMIZE statement](#)

[READ\\$ function](#)

[REDIM statement](#)

[REGEXPR statement](#)

[REGISTER statement](#)

[REGREPL statement](#)

[REM statement](#)

[REMAIN\\$ function](#)

[REMOVE\\$ function](#)

[REPEAT\\$ function](#)

[REPLACE statement](#)

[RESET statement](#)

[RESUME statement](#)

[RETAIN\\$ function](#)

[RETURN statement](#)

[RGB function](#)

[RIGHT\\$ function](#)

[RMDIR statement](#)

[RND function](#)

[ROTATE statement](#)

[ROUND function](#)

[RSET statement](#)

[RSET\\$ function](#)

[RTRIM\\$ function](#)

S

[SCROLLBAR GET PAGE SIZE statement](#)

[SCROLLBAR GET POS statement](#)

[SCROLLBAR GET RANGE statement](#)

[SCROLLBAR GET TRACKPOS statement](#)

[SCROLLBAR SET PAGESIZE statement](#)

[SCROLLBAR SET POS statement](#)

[SCROLLBAR SET RANGE statement](#)

[SEEK function](#)

[SEEK statement](#)

[SELECT CASE/END SELECT block](#)

[SETATTR statement](#)

[SETEOF statement](#)

[SGN function](#)

[SHELL function](#)

[SHELL statement](#)

[SHIFT statement](#)

[SIN function](#)

[SIZEOF function](#)

[SLEEP statement](#)

[SPACE\\$ function](#)

[SQR function](#)

[STATIC statement](#)

[STATUSBAR SET PARTS statement](#)

[STATUSBAR SET TEXT statement](#)

[STR\\$ function](#)

[STRDELETE\\$ function](#)

[STRING\\$ function](#)

[STRINSERT\\$ function](#)

[STRPTR function](#)

[STRREVERSE\\$ function](#)

[SUB/END SUB statements](#)

[SWAP statement](#)

[SWITCH function](#)

T

[TAB DELETE statement](#)

[TAB GET COUNT statement](#)

[TAB GET DIALOG statement](#)

[TAB GET SELECT statement](#)
[TAB INSERT PAGE statement](#)
[TAB RESET statement](#)
[TAB SELECT statement](#)
[TAB SET IMAGELIST statement](#)
[TAB\\$ function](#)
[TALLY function](#)
[TAN function](#)
[TCP ACCEPT statement](#)
[TCP CLOSE statement](#)
[TCP LINE INPUT statement](#)
[TCP NOTIFY statement](#)
[TCP OPEN statement](#)
[TCP PRINT statement](#)
[TCP RECV statement](#)
[TCP SEND statement](#)
[THREAD CLOSE statement](#)
[THREAD CREATE statement](#)
[THREAD FUNCTION statement](#)
[THREAD GET PRIORITY statement](#)
[THREAD RESUME statement](#)
[THREAD SET PRIORITY statement](#)
[THREAD STATUS statement](#)
[THREAD SUSPEND statement](#)
[THREADCOUNT function](#)
[THREADED statement](#)
[THREADID function](#)
[TIME\\$ system variable](#)
[TIMER function](#)
[TIX statement](#)
[TOOLBAR ADD BUTTON statement](#)
[TOOLBAR ADD SEPARATOR statement](#)
[TOOLBAR DELETE BUTTON statement](#)
[TOOLBAR GET STATE statement](#)
[TOOLBAR GET COUNT statement](#)
[TOOLBAR SET IMAGELIST statement](#)

[TOOLBAR SET STATE statement](#)
[TRACE statement](#)
[TREEVIEW DELETE statement](#)
[TREEVIEW GET BOLD statement](#)
[TREEVIEW GET CHECK statement](#)
[TREEVIEW GET CHILD statement](#)
[TREEVIEW GET COUNT statement](#)
[TREEVIEW GET EXPANDED statement](#)
[TREEVIEW GET NEXT statement](#)
[TREEVIEW GET PARENT statement](#)
[TREEVIEW GET PREVIOUS statement](#)
[TREEVIEW GET ROOT statement](#)
[TREEVIEW GET SELECT statement](#)
[TREEVIEW GET TEXT statement](#)
[TREEVIEW GET USER statement](#)
[TREEVIEW INSERT ITEM statement](#)
[TREEVIEW RESET statement](#)
[TREEVIEW SELECT statement](#)
[TREEVIEW SET BOLD statement](#)
[TREEVIEW SET CHECK statement](#)
[TREEVIEW SET EXPANDED statement](#)
[TREEVIEW SET IMAGELIST statement](#)
[TREEVIEW SET TEXT statement](#)
[TREEVIEW SET USER statement](#)
[TREEVIEW UNSELECT statement](#)
[TRIM\\$ function](#)
[TRY/END TRY block](#)
[TYPE SET statement](#)
[TYPE/END TYPE block](#)

U

[UBOUND function](#)
[UCASE\\$ function](#)
[UCODE\\$ function](#)
[UCODEPAGE statement](#)

[UDP CLOSE statement](#)

[UDP NOTIFY statement](#)

[UDP OPEN statement](#)

[UDP RECV statement](#)

[UDP SEND statement](#)

[UNION/END UNION block](#)

[UNLOCK statement](#)

[USING\\$ function](#)

V

[VAL function](#)

[VARIANT# function](#)

[VARIANT\\$ function](#)

[VARIANTVT function](#)

[VARPTR function](#)

[VERIFY function](#)

W

[WHILE/WEND statements](#)

[WINDOW GET ID statement](#)

[WINDOW GET PARENT statement](#)

[WINMAIN function](#)

[WRITE# statement](#)

X

[XOR operator](#)

[XPRINT\\$ function](#)

[XPRINT statement](#)

[XPRINT ARC statement](#)

[XPRINT ATTACH statement](#)

[XPRINT BOX statement](#)

[XPRINT CANCEL statement](#)

[XPRINT CHR SIZE statement](#)

[XPRINT CLOSE statement](#)

[XPRINT COLOR statement](#)

[XPRINT COPY statement](#)
[XPRINT ELLIPSE statement](#)
[XPRINT FONT statement](#)
[XPRINT FORMFEED statement](#)
[XPRINT GET CLIENT statement](#)
[XPRINT GET COLRATE statement](#)
[XPRINT GET COLORMODE statement](#)
[XPRINT GET COPIES statement](#)
[XPRINT GET DC statement](#)
[XPRINT GET DUPLEX statement](#)
[XPRINT GET LINES statement](#)
[XPRINT GET MARGIN statement](#)
[XPRINT GET MIX statement](#)
[XPRINT GET ORIENTATION statement](#)
[XPRINT GET PAPER statement](#)
[XPRINT GET PAPERS statement](#)
[XPRINT GET PIXEL statement](#)
[XPRINT GET POS statement](#)
[XPRINT GET PPI statement](#)
[XPRINT GET QUALITY statement](#)
[XPRINT GET SCALE statement](#)
[XPRINT GET SIZE statement](#)
[XPRINT GET TRAY statement](#)
[XPRINT GET TRAYS statement](#)
[XPRINT IMAGELIST statement](#)
[XPRINT LINE statement](#)
[XPRINT PIE statement](#)
[XPRINT POLYGON statement](#)
[XPRINT POLYLINE statement](#)
[XPRINT RENDER statement](#)
[XPRINT SCALE statement](#)
[XPRINT SCALE PIXELS statement](#)
[XPRINT SET COLRATE statement](#)
[XPRINT SET COLORMODE statement](#)
[XPRINT SET COPIES statement](#)
[XPRINT SET DUPLEX statement](#)

[XPRINT SET FONT](#)

[XPRINT SET MIX statement](#)

[XPRINT SET ORIENTATION statement](#)

[XPRINT SET PAPER statement](#)

[XPRINT SET PIXEL statement](#)

[XPRINT SET POS statement](#)

[XPRINT SET QUALITY statement](#)

[XPRINT SET TRAY statement](#)

[XPRINT STRETCH statement](#)

[XPRINT STYLE statement](#)

[XPRINT TEXT SIZE statement](#)

[XPRINT WIDTH statement](#)

This section contains an alphabetical listing of all of the PowerBASIC keywords. Each entry goes into specific detail about each command, and is cross-references to other relevant commands. The Programming Reference topics in this help file describe theory and example usage of a selection of essential commands.

The commands can be classified into four primary categories, according to their syntactic class: functions, statements, system variables, and metastatements:

Functions

These are predefined PowerBASIC functions, as opposed to user-defined functions. Functions generally return either a numeric or a string value, and these can be used within a more complex expression. Most functions require the program pass one or more arguments to them; these arguments being numeric or string, or combinations thereof, depending on the function. For example:

```
T = COS(3.14)
sResult = FORMAT$(T)
A$ = CHR$(123, "hello", 65, 66, 67, 65 TO 97)
```

Statements

Statements are building blocks that make up programs. They instruct the compiler to perform specific actions, such as opening a file, setting the date, sending data to a device, etc. Statements do not return a value, but often take one or more arguments. Each statement must appear on a line by itself; or be separated from other program elements with a delimiting colon (:) character. For example:

```
A& = A& + 10& : B$ = "PowerBASIC"
OPEN "A Long Filename.txt" FOR BINARY AS #1
Count& = 100
```

System variables

System variables allow a program to interact with the system (in this sense, "system" means the computer, the operating system, the internal run-time code, etc). System variables are predefined by PowerBASIC, and can be used to access and control certain information maintained by the system. For example:

```
A$ = DATE$
DATE$ = "03-03-2003"
ErrVal = ERRCLEAR
B$ = TIME$
TIME$ = "03:00"
```

Metastatements

Metastatements are instructions that control the action of the PowerBASIC compiler. Strictly speaking, metastatements are not part of the BASIC language because they do not operate at run-time (when the program is executing). Like compiler option-switches, metastatements can be used to determine how the compiler will operate during the compilation of program code (compile-time).

Metastatements are prefixed with a number (#) symbol to differentiate them from normal statements. Metastatements may take one or more arguments. For example:

```
#COMPILE EXE "The target filename.exe"  
#OPTION VERSION4  
#DIM ALL  
#INCLUDE "WIN32API.INC"
```

Please note that PowerBASIC supports both the dollar (\$) symbol and a the pound (#) symbol as a metastatement prefixes.

See Also

[Format and typefaces](#)

[Command Summary](#)

Every PowerBASIC command is listed alphabetically, as a separate topic. Each entry contains a brief explanation of what the command does, a description of its syntax, clarifying remarks and restrictions, plus examples of use. The examples are designed to be indicative of syntax and usage only.

The *syntax* section of each entry describes the available options and format each command may use, as follows:

Italic Indicates areas within commands that you need to fill in with application-specific information, such as variable names, procedure names, numeric or string values, etc. For example:

```
y = VAL(string_expression)
```

UPPERCASE Indicates part of the command must be entered exactly as shown. For example:

```
OPTION EXPLICIT
```

Brackets [] Indicates the information they enclose is optional. For example:

```
SEEK [#] filenum&, position&&
```

Braces { } Indicates a choice of two or more options, one of which **MUST** be used. For example:

```
#DIM {ALL | NONE}
```

Ellipses Indicates that part of the command can be repeated as many times as required. For example:

```
MACRO macroname [(prm1, prm2, ... )] = replacementtext
```

See Also

[Keyword Reference](#)

[Command Summary](#)

The following is a list of the commands built into the compiler and separated into 18 groups of related commands, which can assist with identifying the best command for the task at hand. Some commands may appear in more than one group.

Command List

[Array Operations](#)

[COM Commands](#)

[Communication Control](#)

[Compiler Operations](#)

[Debugging and Error Control](#)

[File Commands](#)

[Flow Control](#)

[Graphic Commands](#)

[Input Commands](#)

[Memory Management](#)

[Metastatements](#)

[Numeric Operations](#)

[Operating System](#)

[Printing Commands](#)

[String Operations](#)

[Thread Control](#)

[Time Commands](#)

[Misc Operations](#)

See Also

[Keyword Reference](#)

[Format and typefaces](#)

The following functions can be used to create and manage COM clients:

#COM DOC	Specifies a help string which usually provides a general description of the COM server
#COM HELP	Specifies the name of the associated help file and the help context code.
#COM NAME	Specifies the name of the server and the version number.
#COM GUID	Specifies the GUID which identifies the entire application or library (APPID or LIBID).
ACODE\$	Translate a Unicode string into an ANSI string
CLASS/END CLASS	Create the code and data for an object.
EVENT SOURCE	Declare an event interface within a Class definition.
EVENTS	Attach or detach an event handler to/from an event source
CLSID\$	Return a 16-byte (128-bit) GUID string containing a CLSID
GUID\$	Return a 16-byte (128-bit) Globally Unique Identifier GUID
GUIDTXT\$	Return a 38-byte human-readable GUID/UUID string
IDISPINFO	Sets and returns additional information about certain Dispatch Status Codes for the OBJRESULT function.
INSTANCE	Declare INSTANCE variables which are unique to each object
INTERFACE / END INTERFACE	Declare a direct object interface and its member Methods/Properties.
Block (Direct)	
INTERFACE/END INTERFACE	Declare a dispatch interface and its member Methods/Properties for the purposes of IDBinding to a Dispatch COM interface.
block (IDBind)	
ISINTERFACE	Determine whether an object supports a particular interface
ISNOTHING	Determine the current status of a given object variable
ISOBJECT	Determine the current status of a given object variable
LET (with	Assign an object reference to an object variable.

Objects)	
LET (with Variants)	Assign a value to a variable or Variant
ME	A pseudo object variable to reference the current object
METHOD / END METHOD	Define a METHOD procedure within a class
MYBASE	A pseudo object variable to reference the inherited parent object.
OBJECTIVE	Return True/False of the running state of a COM EXE object
OBJECT GET	Retrieve or read the value of an Dispatch Interface member Property
OBJECT LET	Assign or write a value to an Dispatch Interface member Property
OBJECT SET	Assign or write a value to an Dispatch Interface member Property that contains a reference to an object
OBJECT CALL	Call or execute a member Method of an Dispatch Interface
OBJECT RAISEEVENT	Call or execute a member Method of an event Dispatch Interface
OBJPTR	Return an object pointer of a specified object variable
OBJRESULT	Return the execution result of the most recent OBJECT statement
OBJRESULT\$	Returns a string which describes an OBJRESULT (hResult) code
PROGID\$	Return the alphanumeric PROGID string (text) of a given CLSID
PROPERTY GET	Retrieve a data value from the object
PROPERTY SET	Assign a data value to an object
RAISEEVENT	Call Event Handler code
RESET	Clear a Variant to empty (%VT_EMPTY)
UCODE\$	Translate an ANSI string into a Unicode string
VARIANT#	Return the numeric value contained in a Variant variable
VARIANT\$	Return the dynamic string value contained in a Variant variable

VARIANTVT

Determine the internal data type of the data stored in a Variant

The following functions can be used for external communications:

COMM	Retrieve the value or status of a communications parameter
COMM CLOSE	Close an open serial port
COMM LINE	Receive a CR/LF terminated "line" of data from a serial port
COMM OPEN	Open a serial port
COMM PRINT	Send a "line" of binary data through a serial port
COMM RECV	Receive binary data from a serial port
COMM RESET	Disable flow control for a given serial port
COMM SEND	Send a string of binary data through a serial port
COMM SET	Set communication options for a serial port
EOF	Return end-of-file status of a file, serial or TCP/UDP transmission
FREEFILE	Return the next available Classic PowerBASIC file number
HOST ADDR	Translate a host name into a corresponding IP address
HOST NAME	Translate an IP address into a corresponding host name
OPEN	Prepare a file or device for reading or writing
TCP ACCEPT	Accept an incoming request for TCP communication
TCP CLOSE	Close a previously opened TCP/IP port
TCP LINE INPUT	Receive a line of text from a specified TCP/IP port
TCP NOTIFY	Designate which TCP/IP events generate notification messages
TCP OPEN	Enable an app to communicate with a TCP/IP server or client
TCP PRINT	Write a string to a nominated TCP/IP
TCP RECV	Receive data from a specified TCP/IP port
TCP SEND	Write a string to a nominated TCP/IP port
UDP CLOSE	Close a previously opened UDP socket
UDP NOTIFY	Designate which TCP/IP events generate notification messages
UDP OPEN	Create a socket to communicate with a UDP server or client
UDP RECV	Receive data from a previously opened UDP port
UDP SEND	Send a string of data through a previously opened UDP socket

The following functions manipulate the compiler's operation:

#ALIGN	Align the next instruction to a boundary.
%DEF	Determine if an equate has been previously defined
#BLOAT	Artificially inflate the disk image size of a compiled program
#COMPILE	Determine which type of file will be created by the compiler
#DEBUG CODE	Compiler directive to suppress generation of debugging code
#DEBUG DISPLAY	Display a message when an untrapped run-time error occurs.
#DIM	Specify if variables must be declared before use
#IF	Define sections of source code to be compiled or ignored
#MESSAGES	Specify which messages should be sent to a Control Callback Function
#OPTIMIZE	Choose between faster execution or smaller code size
#REGISTER	Control automatic allocation of Register variables
#STACK	Set the maximum potential stack size
#TOOLS	Enable/disable integrated development tools in compiled code
DECLARE	Explicitly declare a Sub or Function
DEFBYT	Declare the default variable type to be Byte
DEFCUR	Declare the default variable type to be Currency
DEFCUX	Declare the default variable type to be Extended Currency
DEFDBL	Declare the default variable type to be Double-precision
DEFDWD	Declare the default variable type to be Double-word
DEFEXT	Declare the default variable type to be Extended-precision
DEFINT	Declare the default variable type to be Integer
DEFLNG	Declare the default variable type to be Long-integer
DEFQUD	Declare the default variable type to be Quad-integer
DEFSNG	Declare the default variable type to be Single-precision
DEFSTR	Declare the default variable type to be String
DEFWRD	Declare the default variable type to be Word
DIM	Declare and dimension arrays , scalar variables, and pointers
DLLMAIN	Function called by Windows each time a DLL is loaded into, and unloaded from, memory

<u>ERASE</u>	Deallocate <u>array</u> memory
<u>GLOBAL</u>	Declare global (shared) variables between Subs, Functions, <u>Classes</u> , <u>Methods</u> , and <u>Properties</u>
<u>INSTANCE</u>	Declare Instance variables which are unique to each object
<u>LIBMAIN</u>	Function called by Windows each time a DLL is loaded into, and unloaded from, memory
<u>LOCAL</u>	Declare local variables in a Sub, Function, Method or Property
<u>MACRO</u>	Define a single or multi-line text substitution block
<u>OPTION EXPLICIT</u>	Force explicit declaration of all variables
<u>PBLIBMAIN</u>	Function called by Windows each time a DLL is loaded into, and unloaded from, memory
<u>PBMAIN</u>	Define the initial entry-point Function for an application
<u>PROFILE</u>	Capture an execution time profile of the Subs, Functions, Methods, and Properties
<u>REDIM</u>	Declare dynamic arrays, allocate, reallocate, deallocate memory
<u>REGISTER</u>	Define local Register variables within a Sub, Function, Method, or Property
<u>STATIC</u>	Declare static variables inside of a Sub, Function, Method, or Property
<u>STRPTR</u>	Return the address of the data held by a <u>variable length string</u>
<u>VARPTR</u>	Return the 32-bit address of a variable or string handle
<u>WINMAIN</u>	Define the initial entry-point Function for an application

Debugging and Error Control

[Top](#) [Previous](#) [Next](#)

The following functions can be used to trap and manage [error](#) conditions:

#DEBUG CODE	Compiler directive to suppress generation of debugging code.
#DEBUG DISPLAY	Display a message when an untrapped run-time error occurs.
#DEBUG ERROR	Control generation of error checking code
#DEBUG PRINT	Display information in the IDE's Debug Window
#DIM	Specify if variables must be declared before use
#STACK	Set the maximum potential stack size
#TOOLS	Enable/disable integrated development tools in compiled code
CALLSTK	Capture a representation of the stack frames in the call stack
CALLSTK\$	Retrieve the details of a specific stack frame
CALLSTKCOUNT	Retrieve the number of stack frames in the call stack
ERL	Return the line number of the most recent run-time error
ERL\$	Return the last label, line number, or procedure name executed prior to the most recent error
ERR	Return the error code of the most recent run-time error
ERRCLEAR	Return and clear the error code of the most recent run-time error
ERROR	Cause a specific run-time error to be generated and set ERR
ERROR\$	Return a string containing the descriptive name of an error
FILENAME\$	Return the file-system name of an open file
FUNCNAME\$	Return the name of the current Sub/Function/Method/Property
ON ERROR	Specify an error handling routine; enable/disable trapping
OPTION EXPLICIT	Force explicit declaration of all variables
PROFILE	Capture an execution time profile of the Subs, Functions, Methods, and Properties
RESUME	Continue execution after error handling with ON ERROR GOTO
TRACE	Capture the precise flow of execution in a module
TRY/END TRY	A structured method of trapping and responding to errors

The following functions can be used to create GUI application interfaces:

ACCEL ATTACH	Attach a table of keyboard accelerators to a DDT dialog
CALLBACK FUNCTION	Define a Dialog /Control Callback Function block
CB.CTL	Return the numeric ID of the control sending a callback message
CB.CTLMSG	Return the numeric notification message parameter
CB.HNDL	Return the window handle of the parent dialog receiving the message
CB.LPARAM	Return the numeric value of the lParam& parameter of the message
CB.MSG	Return the numeric value of the message sent by the caller
CB.WPARAM	Return the numeric value of the wParam& parameter of the message
CB.NMCODE	Return the numeric value of the notification message describing the event which occurred
CB.NMHDR	Returns the address (a pointer) to the NMHDR UDT for this notification message
CB.NMHDR\$	Returns the contents of the NMHDR UDT as a dynamic string
CB.NMHWND	Returns the handle of the control which sent this message
CB.NMID	Returns the ID number assigned to the control
CBCTL	Return the numeric ID of the control sending a callback message
CBCTLMSG	Return the numeric notification message parameter
CBHNDL	Return the window handle of the parent dialog receiving the message
CBLPARAM	Return the numeric value of the lParam& parameter of the message
CBMSG	Return the numeric value of the message sent by the caller
CBWPARAM	Return the numeric value of the wParam& parameter of the message
CLIPBOARD GET ITEM	A LONG or DWORD is retrieved from the CLIPBOARD
CLIPBOARD GET TEXT	A text string is retrieved from the CLIPBOARD

<u>CLIPBOARD RESET</u>	The contents of the CLIPBOARD are deleted
<u>CLIPBOARD SET ITEM</u>	A new LONG or DWORD is stored in the CLIPBOARD.
<u>CLIPBOARD SET TEXT</u>	A new text string is stored in the CLIPBOARD.
<u>COMBOBOX ADD</u>	Add a string value to a combo box control
<u>COMBOBOX DELETE</u>	Remove a string from a <u>combo box</u> control
<u>COMBOBOX FIND</u>	Strings in the COMBOBOX are searched to find the first string which begins with the specified characters
<u>COMBOBOX FIND EXACT</u>	Strings in the COMBOBOX are searched to find the first string which exactly matches the specified characters
<u>COMBOBOX GET COUNT</u>	The number of items in the list box of the COMBOBOX is retrieved
<u>COMBOBOX GET SELCOUNT</u>	The number of selected items in the list box of the COMBOBOX is retrieved
<u>COMBOBOX GET SELECT</u>	The index of the currently selected item in the list box of the COMBOBOX is retrieved
<u>COMBOBOX GET STATE</u>	A data item is checked to see if it is currently selected
<u>COMBOBOX GET TEXT</u>	Retrieve the default text from a combo box
<u>COMBOBOX GET USER</u>	Retrieve the value in the user data area of the COMBOBOX
<u>COMBOBOX INSERT</u>	Insert a new data item at a specified location
<u>COMBOBOX RESET</u>	Remove all strings from a combo box
<u>COMBOBOX SELECT</u>	Select a string in a combo box and make it the default selection
<u>COMBOBOX SET TEXT</u>	Replace the string for a specific data item with a new string
<u>COMBOBOX SET USER</u>	Set a value in the user data area of the COMBOBOX
<u>COMBOBOX UNSELECT</u>	All items in a COMBOBOX control are set to an unselected state
<u>CONTROL ADD</u>	Add a custom control to a DDT dialog
<u>CONTROL ADD BUTTON</u>	Add a command button to a dialog
<u>CONTROL ADD CHECK3STATE</u>	Add an auto 3-state checkbox to a dialog

<u>CONTROL ADD CHECKBOX</u>	Add an checkbox to a dialog
<u>CONTROL ADD COMBOBOX</u>	Add a combo box to a dialog
<u>CONTROL ADD FRAME</u>	Add a frame control to a dialog
<u>CONTROL ADD GRAPHIC</u>	Add a <u>graphic</u> control to a dialog
<u>CONTROL ADD IMAGE</u>	Add a non-resizing image control to a dialog
<u>CONTROL ADD IMAGEX</u>	Add an image control to a dialog
<u>CONTROL ADD IMGBUTTON</u>	Add a non-resizing image button to a dialog
<u>CONTROL ADD IMGBUTTONX</u>	Add an image button to a dialog
<u>CONTROL ADD LABEL</u>	Add a text label to a dialog
<u>CONTROL ADD LINE</u>	Add a line control to a dialog
<u>CONTROL ADD LISTBOX</u>	Add a list box control to a dialog
<u>CONTROL ADD LISTVIEW</u>	Add a ListView control to a dialog
<u>CONTROL ADD OPTION</u>	Add an option button to a dialog
<u>CONTROL ADD PROGRESSBAR</u>	Add a ProgressBar control to a dialog
<u>CONTROL ADD SCROLLBAR</u>	Add a scroll bar control to a dialog
<u>CONTROL ADD STATUSBAR</u>	Add a StatusBar control to a dialog
<u>CONTROL ADD TAB</u>	Add a Tab Control to a dialog
<u>CONTROL ADD TEXTBOX</u>	Add a text box control to a dialog
<u>CONTROL ADD TOOLBAR</u>	Add a ToolBar control to a dialog
<u>CONTROL ADD TREEVIEW</u>	Add a TreeView control to a dialog
<u>CONTROL DISABLE</u>	Disable a control so that it no longer accepts user

	interaction
<u>CONTROL ENABLE</u>	Enable a control so that it can receive user interaction
<u>CONTROL GET CHECK</u>	Get the Check State of a 3-state, checkbox, or option button
<u>CONTROL GET CLIENT</u>	Get the client area dimensions of a control
<u>CONTROL GET LOC</u>	Get the location of the specified control in a dialog
<u>CONTROL GET SIZE</u>	Get the size of a control in the specified dialog
<u>CONTROL GET TEXT</u>	Get the text from a control
<u>CONTROL GET USER</u>	Retrieve a value from the user data area of a DDT control
<u>CONTROL HANDLE</u>	Return a window handle for a given control ID
<u>CONTROL KILL</u>	Remove a control from a dialog
<u>CONTROL POST</u>	Place a message into the message queue of a control (non-blocking)
<u>CONTROL REDRAW</u>	Schedule a control to be redrawn
<u>CONTROL SEND</u>	Send a message to a control and wait for it to be processed
<u>CONTROL SET CHECK</u>	Set the Check State for a 3-state or checkbox control
<u>CONTROL SET CLIENT</u>	Change the size of a control to a specific client area size
<u>CONTROL SET COLOR</u>	Set the foreground and background color of a control
<u>CONTROL SET FOCUS</u>	Set the keyboard focus to the specified control
<u>CONTROL SET FONT</u>	Select a font to be used for a particular Windows Control
<u>CONTROL SET IMAGE</u>	Change the icon or bitmap displayed in an IMAGE control
<u>CONTROL SET IMAGEX</u>	Change the icon or bitmap displayed in an IMAGEX control
<u>CONTROL SET IMGBUTTON</u>	Change the icon or bitmap displayed in an IMGBUTTON control
<u>CONTROL SET IMGBUTTONX</u>	Change the icon or bitmap displayed in an IMGBUTTONX control
<u>CONTROL SET LOC</u>	Move the control to a new location in the dialog
<u>CONTROL SET OPTION</u>	Set the Check State for an option (radio) control

<u>CONTROL SET SIZE</u>	Change the size of a control
<u>CONTROL SET TEXT</u>	Change the text in a control
<u>CONTROL SET USER</u>	Set a value in the user data area of a DDT control
<u>CONTROL SHOW STATE</u>	Change the visible state of a control
<u>DESKTOP GET CLIENT</u>	Retrieve the size of the client area of the desktop, in pixels
<u>DESKTOP GET LOC</u>	Retrieve the location of the top, left corner of the client area of the desktop, in pixels
<u>DESKTOP GET SIZE</u>	Return the size of the specified dialog
<u>DIALOG DISABLE</u>	Disable a dialog so that it no longer responds to user interaction
<u>DIALOG DOEVENTS</u>	Process pending window or dialog messages for modeless dialogs
<u>DIALOG ENABLE</u>	Enable a dialog so that it responds to user interaction
<u>DIALOG END</u>	Close and destroy the specified dialog
<u>DIALOG FONT</u>	Specify the default DDT font and point size
<u>DIALOG GET CLIENT</u>	Return the client size of the specified dialog
<u>DIALOG GET LOC</u>	Return the location of the specified dialog
<u>DIALOG GET SIZE</u>	Return the size of the specified dialog
<u>DIALOG GET TEXT</u>	Retrieve the text in a dialog or window caption
<u>DIALOG GET USER</u>	Retrieve a value from the user data area of a DDT dialog
<u>DIALOG NEW</u>	Create a new dialog in memory, ready for display
<u>DIALOG PIXELS</u>	Convert pixels (device units) into dialog units
<u>DIALOG POST</u>	Place a message in the dialog message queue (non-blocking)
<u>DIALOG REDRAW</u>	Force a dialog and all child controls to be redrawn immediately
<u>DIALOG SEND</u>	Send a message to a dialog and wait for it to be processed
<u>DIALOG SET CLIENT</u>	Change the size of a dialog to a specific client area size.
<u>DIALOG SET COLOR</u>	Set the background color of a dialog to a specific <u>RGB</u> color
<u>DIALOG SET ICON</u>	Change both the dialog icon in the caption, and the icon shown in the ALT+TAB task list
<u>DIALOG SET LOC</u>	Change the position of a dialog
<u>DIALOG SET SIZE</u>	Change the size of a dialog

<u>DIALOG SET TEXT</u>	Set the text in a dialog or window caption
<u>DIALOG SET USER</u>	Set a value in the user data area of a DDT dialog
<u>DIALOG SHOW MODAL</u>	Display and activate a modal_dialog
<u>DIALOG SHOW MODELESS</u>	Display and activate a modeless dialog
<u>DIALOG SHOW STATE</u>	Change the visible state of a dialog
<u>DIALOG UNITS</u>	Convert dialog units into pixels
<u>DISPLAY BROWSE</u>	Display a folder selection dialog to return the user's choice
<u>DISPLAY COLOR</u>	Display a color selection dialog to return the user's choice
<u>DISPLAY FONT</u>	Display a selection dialog to return user choices
<u>DISPLAY OPENFILE</u>	Display an OpenFile selection dialog to return user choices
<u>DISPLAY SAVEFILE</u>	Display a SaveFile selection dialog to return user choices
<u>FONT END</u>	Destroy a font when it is no longer needed
<u>FONT NEW</u>	Create a new font for use with <u>GRAPHIC PRINT</u> , <u>XPRINT</u> , etc
<u>FUNCTION/END FUNCTION</u>	Define a Function block
<u>IMAGELIST ADD BITMAP</u>	An bitmap image is added to the IMAGELIST
<u>IMAGELIST ADD ICON</u>	An icon image is added to the IMAGELIST
<u>IMAGELIST ADD MASKED</u>	A bitmap is added to the icon IMAGELIST
<u>IMAGELIST GET COUNT</u>	The number of images in the IMAGELIST is retrieved
<u>IMAGELIST KILL</u>	The specified IMAGELIST is destroyed
<u>IMAGELIST NEW BITMAP</u>	A new bitmap IMAGELIST structure is created
<u>IMAGELIST NEW ICON</u>	A new icon IMAGELIST structure is created
<u>IMAGELIST SET OVERLAY</u>	Specify an image to be used as an overlay
<u>INPUTBOX\$</u>	Displays a dialog box containing a prompt
<u>ISMISSING</u>	Determine whether an optional parameter was passed by the calling code
<u>ISWIN</u>	Determine whether a Control/Dialog/Window currently exists

<u>LISTBOX ADD</u>	Add a string value to a <u>LISTBOX</u> control
<u>LISTBOX DELETE</u>	Remove a string from a LISTBOX control
<u>LISTBOX FIND</u>	Strings in the LISTBOX are searched to find the first string which begins with the specified characters
<u>LISTBOX FIND EXACT</u>	Strings in the LISTBOX are searched to find the first string which exactly matches the specified characters
<u>LISTBOX GET COUNT</u>	The number of items in the LISTBOX is retrieved
<u>LISTBOX GET SELCOUNT</u>	The number of selected items in the LISTBOX is retrieved
<u>LISTBOX GET SELECT</u>	The LISTBOX is searched to find the first selected item
<u>LISTBOX GET STATE</u>	A data item is checked to see if it is currently selected
<u>LISTBOX GET TEXT</u>	Retrieve the default text from a LISTBOX control
<u>LISTBOX GET USER</u>	Retrieve the value in the user data area of the LISTBOX
<u>LISTBOX INSERT</u>	Insert a new data item at a specified location
<u>LISTBOX RESET</u>	Remove all strings from a list box
<u>LISTBOX SELECT</u>	Select a string in a list box and make it the default selection
<u>LISTBOX SET TEXT</u>	Replace the string for a specific data item with a new string
<u>LISTBOX SET USER</u>	Set a value in the user data area of the LISTBOX
<u>LISTBOX UNSELECT</u>	A specified data item in the LISTBOX control is set to an unselected state
<u>LISTVIEW DELETE COLUMN</u>	Delete a column, including its associated header text (if any) from the <u>LISTVIEW</u> control
<u>LISTVIEW DELETE ITEM</u>	The specified data item is deleted from the LISTVIEW control
<u>LISTVIEW FIND</u>	Strings in the LISTVIEW are searched to find the first string which begins with the specified characters
<u>LISTVIEW FIND EXACT</u>	Strings in the LISTVIEW are searched to find the first string which exactly matches the specified characters
<u>LISTVIEW FIT CONTENT</u>	The width of the specified column is adjusted to fit the width of the data items displayed in that column
<u>LISTVIEW FIT HEADER</u>	The width of the specified column is adjusted to fit the width of the data items displayed in that column, and the header text at the top of that column
<u>LISTVIEW GET COLUMN</u>	The width of the designated column is retrieved from the LISTVIEW
<u>LISTVIEW GET</u>	The number of data items in the LISTVIEW is retrieved

<u>COUNT</u>	
<u>LISTVIEW GET HEADER</u>	Column header text is retrieved from the LISTVIEW
<u>LISTVIEW GET MODE</u>	The display mode of the specified LISTVIEW control is retrieved
<u>LISTVIEW GET SELCOUNT</u>	The number of selected items in the LISTVIEW is retrieved
<u>LISTVIEW GET STATE</u>	A data item is tested to see if it is currently selected
<u>LISTVIEW GET STYLE</u>	Retrieves the current setting of the LISTVIEW controls extended style
<u>LISTVIEW GET TEXT</u>	A string data item is retrieved from the LISTVIEW control
<u>LISTVIEW GET USER</u>	Retrieve the value in the user data area of the LISTVIEW
<u>LISTVIEW INSERT COLUMN</u>	A new vertical column is defined for Report Mode of the LISTVIEW
<u>LISTVIEW INSERT ITEM</u>	A new data item is added to this LISTVIEW control
<u>LISTVIEW RESET</u>	All data items are deleted from the specified LISTVIEW control
<u>LISTVIEW SELECT</u>	The specified string data item is chosen as selected text for the LISTVIEW
<u>LISTVIEW SET COLUMN</u>	Change the width of a LISTVIEW column
<u>LISTVIEW SET HEADER</u>	New column header text is displayed above the specified column on the LISTVIEW control
<u>LISTVIEW SET IMAGE</u>	The specified image is displayed next to the item specified
<u>LISTVIEW SET IMAGE2</u>	The specified image is displayed as a secondary "status" image next to the primary image
<u>LISTVIEW SET IMAGELIST</u>	Attach an <u>IMAGELIST</u> to the LISTVIEW control
<u>LISTVIEW SET MODE</u>	Change the display mode of the specified LISTVIEW control
<u>LISTVIEW SET OVERLAY</u>	The specified overlay image is displayed on top of the image specified
<u>LISTVIEW SET STYLE</u>	Alter the current settings of the LISTVIEW controls extended style
<u>LISTVIEW SET TEXT</u>	The text, if any, for the specified data item is replaced with new text
<u>LISTVIEW SET USER</u>	Set a value in the user data area of the LISTVIEW

LISTVIEW SORT	All of the items in a LISTVIEW are sorted
LISTVIEW UNSELECT	The specified data item is set to an unselected state
LISTVIEW VISIBLE	The specified data item is scrolled, if necessary, to ensure that the data item is visible
PROGRESSBAR GET POS	The current position of the PROGRESSBAR is retrieved
PROGRESSBAR GET RANGE	The current range of the PROGRESSBAR is retrieved
PROGRESSBAR SET POS	Set the current position of the PROGRESSBAR
PROGRESSBAR SET RANGE	Set the minimum and maximum ranges of the PROGRESSBAR
PROGRESSBAR SET STEP	Specify the default increment value to be used by PROGRESSBAR STEP
PROGRESSBAR STEP	Advance the current position of the PROGRESSBAR by the default increment value
MENU ADD POPUP	Add a popup child menu to an existing menu
MENU ADD STRING	Add a string or separator to an existing menu
MENU ATTACH	Attach a menu to a given dialog
MENU DELETE	Delete a menu item from an existing menu
MENU DRAW BAR	Redraw the menu bar for a given dialog
MENU GET STATE	Return the state of a specified menu item
MENU GET TEXT	Return the text associated with a given menu item
MENU NEW BAR	Create a new menu bar
MENU NEW POPUP	Create a new popup menu
MENU SET STATE	Set the state of a specified menu item
MENU SET TEXT	Set the text of a given menu item
MOUSEPTR	Change the mouse pointer (cursor) to a new shape
SCROLLBAR GET PAGESIZE	Retrieve the current page size
SCROLLBAR GET POS	Returns the current position of the SCROLLBAR
SCROLLBAR GET RANGE	Returns the current range of the SCROLLBAR
SCROLLBAR GET TRACKPOS	Retrieve the current position of the scroll box
SCROLLBAR SET	Set the current page size

PAGESIZE	
SCROLLBAR SET POS	Set the current position of the SCROLLBAR
SCROLLBAR SET RANGE	Set the range of the SCROLLBAR
STATUSBAR SET PARTS	Set the number of parts to be displayed in the STATUSBAR
STATUSBAR SET TEXT	Assign the text to be displayed in the specified part of the STATUSBAR
TAB DELETE	Delete a page from the TAB control
TAB GET COUNT	Return the number of pages in a TAB control
TAB GET DIALOG	Retrieve the handle of the dialog for a specific page in a TAB control
TAB GET SELECT	Returns the currently selected page in a TAB control
TAB INSERT PAGE	Add a new page to a TAB control
TAB RESET	Delete all pages in a TAB control
TAB SELECT	Select a specific page in a TAB control to be the active page
TAB SET IMAGELIST	Assign an IMAGELIST to be used in a TAB control
TOOLBAR ADD BUTTON	Add a button to a TOOLBAR control
TOOLBAR ADD SEPARATOR	Add a separator to a TOOLBAR control
TOOLBAR DELETE BUTTON	Delete a button from a TOOLBAR control
TOOLBAR GET STATE	Get the state of a button on a TOOLBAR control
TOOLBAR GET COUNT	Retrieve the number of buttons on a TOOLBAR control
TOOLBAR SET IMAGELIST	Attach an IMAGELIST to a TOOLBAR control
TOOLBAR SET STATE	Set the state of a button on a TOOLBAR control
TREEVIEW DELETE	Delete a data item from a TREEVIEW control
TREEVIEW GET BOLD	The bold attribute for a data item is retrieved
TREEVIEW GET CHECK	The checkmark attribute for a data item is retrieved
TREEVIEW GET	Return the handle of the first child item of a specified data

<u>CHILD</u>	item
<u>TREEVIEW GET COUNT</u>	The number of data items in the TREEVIEW is retrieved
<u>TREEVIEW GET EXPANDED</u>	The expanded attribute for the data item is retrieved
<u>TREEVIEW GET NEXT</u>	Return the handle of the next sibling data item
<u>TREEVIEW GET PARENT</u>	The handle of the parent for a specified data item is returned
<u>TREEVIEW GET PREVIOUS</u>	Return the handle of the previous sibling data item
<u>TREEVIEW GET ROOT</u>	The handle of the very first data item (topmost) in the TREEVIEW is retrieved
<u>TREEVIEW GET SELECT</u>	The handle of the currently selected data item is retrieved
<u>TREEVIEW GET TEXT</u>	The text of a specific data item is retrieved
<u>TREEVIEW GET USER</u>	Retrieve the value in the user data area for a specific data item of the TREEVIEW
<u>TREEVIEW INSERT ITEM</u>	Add a new data item to a TREEVIEW control
<u>TREEVIEW RESET</u>	All data items are deleted from the specified TREEVIEW control
<u>TREEVIEW SELECT</u>	Select a specific data item in the TREEVIEW control
<u>TREEVIEW SET BOLD</u>	Set the bold attribute for specific data item
<u>TREEVIEW SET CHECK</u>	Set the checkmark attribute for a specific data item
<u>TREEVIEW SET EXPANDED</u>	Set the expanded attribute for a specific data item
<u>TREEVIEW SET IMAGELIST</u>	Attach an IMAGELIST to a TREEVIEW control
<u>TREEVIEW SET TEXT</u>	The text, if any, for the specified data item is replaced with new text
<u>TREEVIEW SET USER</u>	Set the value in the user data area for a specific data item in the TREEVIEW control
<u>TREEVIEW UNSELECT</u>	All items in the TREEVIEW control are set to an unselected state
<u>WINDOW GET ID</u>	The integer ID for a Window (usually a CONTROL) is returned
<u>WINDOW GET</u>	The handle of the parent is retrieved

The following functions can be used to manipulate [files](#), standard I/O and disk services:

CHDIR	Change the current (default) directory on a given drive
CHDRIVE	Change the current default drive
CLOSE	Conclude I/O (input/output) to/from a file or device
CURDIR\$	Return the current directory for a given drive
DIR\$	Return a filename that matches the given mask
DIR\$	Force the release the operating system FindNext handle
CLOSE	
DISKFREE	Return the amount of available space of a disk, in bytes
DISKSIZE	Return the total amount of space on a disk, in bytes
EOF	Return end-of-file status of a file, serial or TCP/UDP transmission
EXE	Return the path and/or name of the executing program.
FIELD	Bind a field string variable to a particular sub-section of a random file buffer or a dynamic string variable
FIELD	Reset the FIELD string to a nul (zero-length) dynamic string
RESET	
FIELD	Change the FILED string to a dynamic string, but first assigns the current sub-section data to it
STRING	
FILEATTR	Return information about an open file
FILECOPY	Copy a file
FILENAMES\$	Return the file-system name of an open file
FILESCAN	Rapidly scan a INPUT or BINARY file to obtain string size info
FLUSH	Flush file buffers to disk to ensure the disk information is current
FREEFILE	Return the next available Classic PowerBASIC file number
GET	Read a record from a random-access file
GET\$	Read a string from a file opened in binary mode
GETATTR	Return the file-system attribute(s) of a disk file or directory
INPUT#	Load variables with data from a sequential file
ISFILE	Determine whether or not a file exists
ISFOLDER	Determine whether or not a folder exists
KILL	Delete a disk file
LINE	Read line(s) from a sequential file into a string variable or array
INPUT#	
LOC	Determine the current seek position in an open disk file

<u>LOCK</u>	Lock part or all of an <u>open file</u> for exclusive access
<u>LOF</u>	Return the length of an open disk file
<u>MKDIR</u>	Create a subdirectory/folder (like the DOS MKDIR command)
<u>NAME</u>	Rename a file or a directory (like the DOS REN command)
<u>OPEN</u>	Prepare a file or device for reading or writing
<u>PATHNAME\$</u>	Parse a path/file name to extract component parts
<u>PATHSCAN\$</u>	Find a file on disk and return the path and/or file name parts
<u>PRINT#</u>	Write data to a device or <u>sequential file</u>
<u>PROFILE</u>	Create a file containing the time profile of <u>Subs</u> , <u>Functions</u> , <u>Methods</u> , and <u>Properties</u>
<u>PUT</u>	Write a record to a random-access file or variable to a <u>binary file</u>
<u>PUT\$</u>	Write a string to a file opened in <u>binary</u> mode
<u>RMDIR</u>	Delete a disk directory (like the DOS RMDIR command)
<u>SEEK</u>	File location where the next I/O operation will take place
<u>SEEK</u>	Set the position in a file for the next input or output operation
<u>SETATTR</u>	Set the file system attribute(s) of a disk file or directory
<u>SETEOF</u>	Truncate/extend a file to its current file pointer position
<u>SHELL</u>	Run an executable program asynchronously
<u>SHELL</u>	Run an executable program synchronously
<u>UNLOCK</u>	Remove exclusive-access locks placed on a file
<u>WRITE#</u>	Output data to a sequential file in a delimited format

The following functions can be used to manage program execution/flow:

%DEF	Determine if an equate has been previously defined
#IF	Define sections of source code to be compiled or ignored
#TOOLS	Enable/disable integrated development tools in compiled code
CALL	Invoke a procedure (Sub , Function , Method , or Property)
CALL DWORD	Invoke a procedure (Sub, Function, Method, or Property) indirectly
CALLSTK	Capture a representation of the stack frames in the call stack
CALLSTK\$	Retrieve the details of a specific stack frame
CALLSTKCOUN	Retrieve the number of stack frames in the call stack
CHOOSE	Return one of several values, based upon the value of an index
CODEPTR	Obtain a 32-bit address of a label , Sub or Function
DLLMAIN	User-defined function called when a DLL the DLL is loaded/unloaded
DO/LOOP	Define a group of statements that are executed repetitively
EXIT	Transfer program execution out of a block structure
FOR/NEXT	Define a loop of program statements controlled by a counter
FUNCNAME\$	Return the name of the current Sub, Function, Method, or Property
FUNCTION/END FUNCTION	Define a Function block
GOSUB	Invoke a local subroutine
GOSUB DWORD	Invoke a local subroutine indirectly
GOTO	Transfer program execution to the statement identified by a label
GOTO DWORD	Transfer execution indirectly to a local label or line number
IF	Test a condition and execute one or more program statements
IF/END IF	Create a IF/THEN/ELSE block with multiple lines and conditions
IIF	Return one of two values based upon a True/False evaluation
ISFALSE	Return the logical falsity of a given expression
ISMISSING	Determine whether an optional parameter was passed by the

calling code.

[ISNOTHING](#)

Determine the current status of a given [object variable](#)

[ISOBJECT](#)

Determine the current status of a given object variable

[ISTRUE](#)

Return the logical truth of a given expression

[ITERATE](#)

Start an immediate iteration of a loop structure

[LIBMAIN](#)

User-defined function called when a DLL the DLL is loaded/unloaded

[MACRO](#)

Define a single or multi-line text substitution block

[METHOD/END](#)

Define a METHOD procedure within a [class](#)

[METHOD](#)

[ON ERROR](#)

Specify an [error handling](#) routine; enable/disable trapping

[ON GOSUB](#)

Call one of several subroutines based on a numeric expression

[ON GOTO](#)

Send program flow to one of several labels based on a value

[PBLIBMAIN](#)

User-defined function called when a DLL the DLL is loaded/unloaded

[PBMAIN](#)

Define the initial entry-point Function for an application

[PROFILE](#)

Capture an execution time profile of the Subs, Functions, Methods, and Properties

[PROPERTY/END](#)

Define a PROPERTY procedure within a class

[PROPERTY](#)

[RETURN](#)

Return from a ([GOSUB](#)) subroutine to its caller

[SELECT CASE](#)

Control program flow based on the value of an expression

[SLEEP](#)

Pause the current [thread](#) for a specified number of milliseconds

[SUB/END SUB](#)

Define a Sub (procedure) block

[TRY/END TRY](#)

A structured method of [trapping](#) and responding to [errors](#)

[WHILE/WEND](#)

Define a block of statements that are executed repeatedly

[WINMAIN](#)

Define the initial entry-point Function for an application

The following functions can be used to display graphics:

CONTROL ADD IMAGE	Add a (non-resizing) image control to a dialog
CONTROL ADD IMAGEX	Add a stretched image control to a dialog
CONTROL ADD IMGBUTTON	Add an image button to a dialog
CONTROL ADD IMGBUTTONX	Add a stretched image button to a dialog
CONTROL ADD GRAPHIC	Add a graphic control to a dialog
BGR	Convert an RGB color value to BGR format
GRAPHIC ARC	Draw an arc in the selected graphic window
GRAPHIC ATTACH	Select the graphic target (window , control , or bitmap) on which future drawing operations will take place
GRAPHIC BITMAP END	Close the selected graphic bitmap
GRAPHIC BITMAP LOAD	Create a memory bitmap and load an image into it
GRAPHIC BITMAP NEW	Create a new memory bitmap
GRAPHIC BOX	Draw a box with square or rounded corners in the selected graphic window
GRAPHIC CHR SIZE	Retrieve the character size for the current font in the selected graphic window
GRAPHIC CLEAR	Clear the entire selected graphic window, optionally using a specified color and fill style
GRAPHIC COLOR	Set the foreground color and optionally the background color for various graphic statements
GRAPHIC COPY	Copy a bitmap to the selected graphic target
GRAPHIC DETACH	Detaches a graphic target
GRAPHIC ELLIPSE	Draw an ellipse or a circle in the selected graphic target
GRAPHIC FONT	Select a font for the GRAPHIC PRINT statement
GRAPHIC GET BITS	Retrieve a copy of a bitmap, storing it as a device-

	independent bitmap in a dynamic string variable
GRAPHIC GET CLIENT	Retrieve the client size of the selected graphic target
GRAPHIC GET DC	Retrieve the handle of the DC (device context) for the selected graphic target
GRAPHIC GET LINES	Retrieve the number of lines that can be printed on the graphic target
GRAPHIC GET LOC	Retrieve the location of the selected graphic target on the screen
GRAPHIC GET MIX	Retrieve the color mix mode for the selected graphic target
GRAPHIC GET PIXEL	Retrieve the color of the pixel at the specified point in the selected graphic target
GRAPHIC GET POS	Retrieve the POS (last point referenced) by a graphic statement
GRAPHIC GET PPI	Retrieve the resolution of the display device, in points per inch
GRAPHIC GET SCALE	Retrieve the current coordinate limits for the graphic target
GRAPHIC IMAGELIST	Display an image from an IMAGELIST
GRAPHIC INKEY\$	Read a keyboard character if one is ready from the graphic target
GRAPHIC INPUT	Read data from the keyboard from within a graphic window
GRAPHIC INSTAT	Determine whether a keyboard character is ready
GRAPHIC INPUT FLUSH	Remove all buffered keyboard data
GRAPHIC LINE	Draw a line in the selected graphic target
GRAPHIC LINE INPUT	Read an entire line from the keyboard from graphic window
GRAPHIC PAINT	Fill an area with a solid color or a hatch pattern
GRAPHIC PIE	Draw a pie section on the selected graphic target
GRAPHIC POLYGON	Draw a polygon in the selected graphic target
GRAPHIC POLYLINE	Draw a series of connected line segments
GRAPHIC PRINT	Output text to the selected graphic target
GRAPHIC REDRAW	Update buffered graphical statements, drawing them to the selected graphic target

<u>GRAPHIC RENDER</u>	Render an image on the selected graphic target
<u>GRAPHIC SAVE</u>	Save an image to a bitmap (.BMP) file
<u>GRAPHIC SCALE</u>	Define a custom coordinate system for the graphic target
<u>GRAPHIC SET BITS</u>	Replace a copy of a bitmap that was retrieved as a device-independent bitmap
<u>GRAPHIC SET FOCUS</u>	Bring the selected graphic window to the foreground and direct focus to it
<u>GRAPHIC SET FONT</u>	Select a font for the GRAPHIC PRINT, <u>GRAPHIC INPUT</u> , and <u>GRAPHIC LINE INPUT</u> statements
<u>GRAPHIC SET LOC</u>	Change the location of the selected graphic window on the screen
<u>GRAPHIC SET MIX</u>	Set the color mix mode for the selected graphic target
<u>GRAPHIC SET PIXEL</u>	Draw a single pixel to the selected graphic window
<u>GRAPHIC SET POS</u>	Set the last point referenced (POS) for the selected graphic target
<u>GRAPHIC STRETCH</u>	Copy and resize a bitmap to the selected graphic target
<u>GRAPHIC STYLE</u>	Set the line style to be used by various graphical statements in the selected graphic target
<u>GRAPHIC TEXT SIZE</u>	Calculate the size of text to be printed
<u>GRAPHIC WAITKEY\$</u>	Read a keyboard character from the graphic window, waiting until one is ready.
<u>GRAPHIC WIDTH</u>	Set the line width to be used by various graphical statements in the selected graphic target
<u>GRAPHIC WINDOW</u>	Create a new graphic window
<u>GRAPHIC WINDOW CLICK</u>	Check whether a GRAPHIC WINDOW has been clicked with the mouse
<u>GRAPHIC WINDOW END</u>	Close and destroy the selected graphic window
<u>IMAGELIST ADD BITMAP</u>	An bitmap image is added to the IMAGELIST
<u>IMAGELIST ADD ICON</u>	An icon image is added to the IMAGELIST
<u>IMAGELIST ADD MASKED</u>	A bitmap is added to the icon IMAGELIST
<u>IMAGELIST GET</u>	The number of images in the IMAGELIST is retrieved

[COUNT](#)

[IMAGELIST KILL](#)

The specified IMAGELIST is destroyed

[IMAGELIST NEW
BITMAP](#)

A new bitmap IMAGELIST structure is created

[IMAGELIST NEW
ICON](#)

A new icon IMAGELIST structure is created

[IMAGELIST SET
OVERLAY](#)

Specify an image to be used as an overlay

[RGB](#)

Return an [RGB](#) color value for use with the Windows API palette and GDI functions

The following functions can be used to gather input data:

COMM	Retrieve the value or status of a communications parameter
COMM LINE	Receive a CR/LF terminated "line" of data from a serial port
COMM RECV	Receive binary data from a serial port
COMMAND\$	Return the command-line used to start the program
ENVIRON	Modify the current program's environment table.
ENVIRON\$	Retrieve string from the operating system's environment table
EOF	Return end-of-file status of a file , serial or TCP/UDP transmission
FIELD	Bind a field string to a file buffer or dynamic string variable
FILESCAN	Rapidly scan a INPUT or BINARY file to obtain string size info
FREEFILE	Return the next available Classic PowerBASIC file number
GET	Read a record from a random-access file
GET\$	Read a string from a file opened in binary mode
GRAPHIC INKEY\$	Read a keyboard character if one is ready from the graphic window
GRAPHIC INPUT	Read data from the keyboard from within a graphic window
GRAPHIC INPUT FLUSH	Remove all buffered keyboard data.
GRAPHIC INSTAT	Determine whether a keyboard character is ready.
GRAPHIC LINE INPUT	Read an entire line from the keyboard from graphic window
GRAPHIC WAITKEY\$	Read a keyboard character from the graphic window, waiting until one is ready.
GRAPHIC WINDOW CLICK	Check whether a graphic window has been clicked with the mouse
INPUT#	Load variables with data from a sequential file
INPUTBOX\$	INPUTBOX\$ displays a dialog box containing a

prompt

[LINE INPUT#](#)

Read line(s) from a sequential file into a string variable or [array](#)

[LOC](#)

Determine the current seek position in an open disk file

[LOF](#)

Return the length of an open disk file

[MSGBOX](#)

Display a message box and get the users Ok/Cancel selection

[MSGBOX](#)

Display an informational message box and discard the users selection

[PEEK](#)

Return the byte at a specific memory location

[PEEK\\$](#)

Return a sequence of bytes starting at a specific memory location

The following functions control compiler and [debugger](#) behavior:

#ALIGN	Align the next instruction to a boundary.
%DEF	Determine if an equate has been previously defined
#COM DOC	Specifies a help string which usually provides a general description of the COM server
#COM HELP	Specifies the name of the associated help file and the help context code.
#COM NAME	Specifies the name of the server and the version number.
#COM GUID	Specifies the GUID which identifies the entire application or library (APPID or LIBID).
#BLOAT	Artificially inflate the disk image size of a compiled program
#COMPILE	Determine which type of file will be created by the compiler
#COMPILER	Define the compiler for this program
#DEBUG CODE	Compiler directive to suppress generation of debugging code.
#DEBUG DISPLAY	Display a message when an untrapped run-time error occurs.
#DEBUG ERROR	Control generation of error checking code
#DEBUG PRINT	Display information in the IDE's Debug Window
#DIM	Specify if variables must be declared before use
#IF	Define sections of source code to be compiled or ignored
#INCLUDE	Instruct the compiler to read an additional source file from disk
#MESSAGES	Specify which messages should be sent to a Control Callback Function.
#OPTION	Mark the file as requiring NT3.5; Win95/NT4; or Win2000/XP
#PBFORMS	Classic PowerBASIC Forms visual designer directives
#REGISTER	Control automatic allocation of Register variables
#RESOURCE	Embed a Classic PowerBASIC Resource file into the executable file
#STACK	Set the maximum potential stack size
#TOOLS	Enable/disable integrated development tools in compiled code
#UTILITY	Compiler directive to allow external utility programs to read text inserted on the #UTILITY line.

The following functions manipulate and manage numeric data:

ABS	Return the absolute value of a numeric expression
AND	AND works as both a logical and a bitwise arithmetic operator
ARRAY ASSIGN	Assign a number of values to successive elements of an array
ARRAY DELETE	Delete a single item from a given array
ARRAY INSERT	Insert a single item into a given array
ARRAY SCAN	Scan all or part of an array for a given value
ARRAY SORT	Sort all or part of a given array
ASC	Return the ASCII code of the specified character in a
ASC	Place an ASCII byte at the specified position in a string
ATN	Return the arctangent of its argument
BIN\$	Return a string with the binary (base 2) representation of a value
BIT CALC	Set or reset a bit in an integer-class variable
BIT	Return the value of a particular bit in an integer-class variable
BIT	Manipulate individual bits of an integer-class variable
BITS	Return the least significant portion of an integer class value
BITS	Return the least significant 8, 16, or 32 bits of an argument
BITSE	Compare integer-class values for equivalent bits regardless of sign
CBYT	Convert a value to a Byte data type
CCUR	Convert a value to a Currency data type
CCUX	Convert a value to a Extended Currency data type
CDBL	Convert a value to a Double-precision data type
CDWD	Convert a value to a Double-word data type
CEIL	Return an integer that is greater than or equal to an argument
CEXT	Convert a value to a Extended-precision data type
CHOOSE	Return one of several values, based upon the value of an index
CINT	Convert a value to a Integer data type
CLNG	Convert a value to a Long-integer data type
COS	Return the cosine of an argument
CQUD	Convert a value to a Quad-integer data type
CSNG	Convert a value to a Single-precision data type
CVBYT	Convert binary encoded string data to a byte value

<u>CVCUR</u>	Convert binary encoded string data to a Currency value
<u>CVCUX</u>	Convert binary encoded string data to Extended Currency
<u>CVD</u>	Convert binary encoded string data to a Double-precision value
<u>CVDWD</u>	Convert binary encoded string data to a Double-word value
<u>CVE</u>	Convert binary encoded string data to Extended-precision
<u>CVI</u>	Convert binary encoded string data to an Integer value
<u>CVL</u>	Convert binary encoded string data to a Long-integer value
<u>CVQ</u>	Convert binary encoded string data to a Quad-integer value
<u>CVS</u>	Convert binary encoded string data to a Single-precision value
<u>CVWRD</u>	Convert binary encoded string data to a <u>Word</u> value
<u>CWRD</u>	Convert a value to a Word data type
<u>DECR</u>	Decrement a <u>variable</u> , pointer, or pointer target
<u>DEFBYT</u>	Declare the default variable type to be Byte
<u>DEFCUR</u>	Declare the default variable type to be Currency
<u>DEFCUX</u>	Declare the default variable type to be Extended Currency
<u>DEFDBL</u>	Declare the default variable type to be Double-precision
<u>DEFDWD</u>	Declare the default variable type to be Double-word
<u>DEFEXT</u>	Declare the default variable type to be Extended-precision
<u>DEFINT</u>	Declare the default variable type to be Integer
<u>DEFLNG</u>	Declare the default variable type to be Long-integer
<u>DEFQUD</u>	Declare the default variable type to be Quad-integer
<u>DEFSNG</u>	Declare the default variable type to be Single-precision
<u>DEFSTR</u>	Declare the default variable type to be String
<u>DEFWRD</u>	Declare the default variable type to be Word
<u>EQV</u>	Perform a logical or a bitwise Equivalence operation
<u>EXP</u>	Return a base number raised to a power, with a base of e
<u>EXP2</u>	Return a base number raised to a power, with a base of 2
<u>EXP10</u>	Return a base number raised to a power, with a base of 10
<u>FIX</u>	Truncate a floating point number to an integer
<u>FORMAT\$</u>	Format numeric data according to a string mask expression
<u>FRAC</u>	Return the fractional part of a floating-point number
<u>HEX\$</u>	Hexadecimal (base 16) string representation of an argument
<u>HI</u>	Extract the most significant (high-order) portion of an argument

HIBYT	Extract the most significant (high-order) Byte of an argument
HIINT	Extract the most significant (high-order) Integer of an argument
HIWRD	Extract the most significant (high-order) Word of an argument
IIF	Return one of two values based upon a True/False evaluation
IMP	Perform a logical or a bitwise Implication operation
INCR	Increment a variable, pointer, or pointer target
INT	Convert a numeric expression to an integer-class value
ISFALSE	Return the logical falsity of a given expression
ISNOTHING	Determine the current status of a given object variable
ISOBJECT	Determine the current status of a given object variable
ISTRUE	Return the logical truth of a given expression
LBOUND	Return the lowest subscript of an array's specific dimension
LEN	Return the logical length of a variable, UDT , or Union
LET	Assign a value to a variable
LET (with Variants)	Assign a value or an object reference to a variant variable
LO	Extract the least significant (low-order) portion of an argument
LOBYT	Extract the least significant (low-order) Byte of an argument
LOG	Return the natural (base e) logarithm of an argument
LOG2	Return the base 2 logarithm of an argument
LOG10	Return the base 10 logarithm of an argument
LOINT	Extract the least significant (low-order) Integer of an argument
LOWRD	Extract the least significant (low-order) Word of an argument
MAKDWD	Combine two 16-bit arguments to form a Double-word
MAKINT	Combine two 8-bit arguments to form an Integer
MAKLNG	Combine two 16-bit arguments to form a Long-integer
MAKPTR	Combine two 16-bit arguments to form a 32-bit pointer
MAKWRD	Combine two 8-bit arguments to form a Word
MAT	Matrix calculations on numeric arrays
MAX	Return the argument with the largest (maximum) value
MIN	Return the argument with the smallest (minimum) value
MOD	Return the remainder of the division between two numbers
NOT	The NOT operator works as a bitwise arithmetic operator
OCT\$	Return a string that is a octal (base 8) representation of a value

<u>OR</u>	Perform a logical or a bitwise OR arithmetic operation
<u>PEEK</u>	Return the byte at a specific memory location
<u>POKE</u>	Store a byte at a specific memory location
<u>RANDOMIZE</u>	Seed the random number generator
<u>RESET</u>	Set a variable, array subscript, or an entire array to zero
<u>RGB</u>	Return a composite RGB color value
<u>RND</u>	Return a random number
<u>ROTATE</u>	Rotate the bits in an integer-class variable
<u>ROUND</u>	Round a numeric value to a specified number of decimal places
<u>SGN</u>	Return the sign of a numeric expression
<u>SHIFT</u>	Shift the bits in an integer-class variable
<u>SIN</u>	Return the sine of an argument
<u>SQR</u>	Return the square root of an argument
<u>SWAP</u>	Exchange the values of two variables, pointers, or pointer targets
<u>SWITCH</u>	Return one item of a series based upon a True/False evaluation
<u>TAN</u>	Return the tangent of an argument
<u>UBOUND</u>	Return the highest subscript of an array's specific dimension
<u>USING\$</u>	Format string/numeric expressions using a mask string
<u>VAL</u>	Return the numeric equivalent of a string argument
<u>VARIANT#</u>	Return the numeric value contained in a Variant variable
<u>XOR</u>	Perform a logical or a bitwise Exclusive-OR operation

The following functions manipulate file and operating system features:

CHDIR	Change the current (default) directory on a given drive
CHDRIVE	Change the current default drive
CLIPBOARD GET ITEM	A LONG or DWORD is retrieved from the CLIPBOARD
CLIPBOARD GET TEXT	A text string is retrieved from the CLIPBOARD
CLIPBOARD RESET	The contents of the CLIPBOARD are deleted
CLIPBOARD SET ITEM	A new LONG or DWORD is stored in the CLIPBOARD.
CLIPBOARD SET TEXT	A new text string is stored in the CLIPBOARD.
COMMAND\$	Return the command-line used to start the program
CURDIR\$	Return the current directory for a given drive
DATE\$	Set and retrieve the system date
DESKTOP GET CLIENT	Retrieve the size of the client area of the desktop, in pixels
DESKTOP GET LOC	Retrieve the location of the top, left corner of the client area of the desktop, in pixels
DESKTOP GET SIZE	Retrieve the size of the entire desktop, in pixels
DIR\$	Return a filename that matches the given mask
DIR\$ CLOSE	Force the release the operating system FindNext handle
DISKFREE	Return the amount of available space of a disk, in bytes
DISKSIZE	Return the total amount of space on a disk, in bytes
DISPLAY BROWSE	Display a folder selection dialog to return the user's choice
DISPLAY COLOR	Display a color selection dialog to return the user's choice
DISPLAY FONT	Display a selection dialog to return user choices
DISPLAY OPENFILE	Display an OpenFile selection dialog to return user choices
DISPLAY SAVEFILE	Display a SaveFile selection dialog to return user choices
ENVIRON	Modify the current program's environment table
ENVIRON\$	Retrieve string from the operating system's environment table
EXE.Extn\$	Returns the extension of the program which is currently executing
EXE.Full\$	Returns the complete drive, path, and file name of the program

which is currently executing

[EXE.Name\\$](#)

Returns just the file name of the program which is currently executing

[EXE.Name\\$x\\$](#)

Returns the file name and the extension of the program which is currently executing

[EXE.Path\\$](#)

Returns the complete drive and path of the program which is currently executing

[FILEATTR](#)

Return information about an [open file](#)

[FILECOPY](#)

Copy a file

[FILENAME\\$](#)

Return the file-system name of an open file

[FLUSH](#)

Flush file buffers to disk to ensure the disk information is current

[GETATTR](#)

Return the file-system attribute(s) of a disk file or directory

[HOST ADDR](#)

Translate a host name into a corresponding [IP](#) address

[HOST NAME](#)

Translate an IP address into a corresponding host name

[ISFILE](#)

Determine whether or not a file exists

[KILL](#)

Delete a disk file

[MKDIR](#)

Create a subdirectory/folder (like the DOS MKDIR command)

[NAME](#)

Rename a file or a directory (like the DOS REN command)

[OPEN](#)

Prepare a file or device for reading or writing

[PATHNAME\\$](#)

Parse a path/file name to extract component parts

[PATHSCAN\\$](#)

Find a file on disk and return the path and/or file name parts

[RGB](#)

Return a composite RGB color value

[RMDIR](#)

Delete a disk directory (like the DOS RMDIR command)

[SETATTR](#)

Set the file system attribute(s) of a disk file or directory

[SETEOF](#)

Truncate/extend a file to its current file pointer position

[SHELL](#)

Launch an executable program asynchronously

[SHELL](#)

Launch an executable program synchronously

[SLEEP](#)

Pause the current thread for a specified number of milliseconds

Printing Commands

[Top](#) [Previous](#) [Next](#)

The following functions are used to send data to a printer:

LPRINT	Output text and data to a printer device
LPRINT ATTACH	Connect directly to a line printer device
LPRINT CLOSE	Disconnect the current printer device
LPRINT FLUSH	Flush any remaining print data to the printer device
LPRINT FORMFEED	Send a formfeed (page eject) character to the printer
LPRINT\$	Return the current printer device used for LPRINT operations
PRINTER\$	Retrieve printer names and printer port names
PRINTERCOUNT	Retrieve the number of available (installed) printers
XPRINT	Output text to a host-printer device
XPRINT ARC	Draw an arc on a host printer page
XPRINT ATTACH	Connect a host-based (GDI) printer for use with XPRINT
XPRINT BOX	Draw a box with square or rounded corners on a host printer page
XPRINT CANCEL	Cancel a print job on the host printer
XPRINT CHR SIZE	Retrieve the character size for the current font on a host printer page
XPRINT CLOSE	Detach a host printer so printing may begin
XPRINT COLOR	Set the foreground color (and, optionally, the background color) for various XPRINT statements
XPRINT COPY	Copy a bitmap to a host printer page
XPRINT ELLIPSE	Draw an ellipse or a circle on a host printer page
XPRINT FONT	Select a font to be used by the XPRINT statement
XPRINT FORMFEED	Start a new page for the host printer
XPRINT GET CLIENT	Retrieve the size of the client area (printable area) on the host printer page
XPRINT GET COLLATE	Retrieve the XPRINT collate status
XPRINT GET COLORMODE	Retrieve the XPRINT colormode status
XPRINT GET COPIES	Retrieve the XPRINT copy count
XPRINT GET DC	Retrieve the handle of the device context (DC) for the host printer page
XPRINT GET DUPLEX	Retrieve the XPRINT duplex status

XPRINT GET LINES	Retrieve the number of lines that can be printed
XPRINT GET MARGIN	Retrieve the margin sizes for the host printer
XPRINT GET MIX	Retrieve the color mix mode for a host printer page
XPRINT GET ORIENTATION	Retrieve the paper orientation for a host printer page
XPRINT GET PAPER	Retrieve the current paper size/type
XPRINT GET PAPERS	Retrieve a list of supported paper types
XPRINT GET PIXEL	Retrieve the color of a pixel on a host printer page
XPRINT GET POS	Retrieve the last point referenced (POS) by an XPRINT statement
XPRINT GET PPI	Retrieve the resolution of the host printer page
XPRINT GET QUALITY	Retrieve the print quality setting for the host printer
XPRINT GET SCALE	Retrieve the current coordinate limits for the host printer page
XPRINT GET SIZE	Retrieve the total size of the host printer page
XPRINT GET TRAY	Retrieve the active printer tray
XPRINT GET TRAYS	Retrieve a list of supported paper trays
XPRINT IMAGELIST	Print an image from an IMAGELIST
XPRINT LINE	Draw a line on a host printer page
XPRINT PIE	Draw a pie section on a host printer page
XPRINT POLYGON	Draw a polygon on a host printer page
XPRINT POLYLINE	Draw a series of connected lines on a host printer page
XPRINT RENDER	Render an image on a host printer page
XPRINT SCALE	Define a custom world coordinate system for a host printer page
XPRINT SCALE PIXELS	Resets the coordinate system to the original default pixel coordinates
XPRINT SET COLLATE	Change the XPRINT collate status
XPRINT SET COLORMODE	Change the XPRINT colormode status
XPRINT SET COPIES	Change the XPRINT copy count
XPRINT SET DUPLEX	Change the XPRINT duplex status
XPRINT SET FONT	Select a font for the XPRINT statement
XPRINT SET MIX	Set the color mix mode for a host printer page
XPRINT SET ORIENTATION	Set the paper orientation for a host printer page

<u>XPRINT SET PAPER</u>	Set a new paper size/type
<u>XPRINT SET PIXEL</u>	Set the color of a pixel on a host printer page
<u>XPRINT SET POS</u>	Retrieve the last point referenced (POS) by an XPRINT statement
<u>XPRINT SET QUALITY</u>	Set the print quality for a host printer
<u>XPRINT SET TRAY</u>	Set a new active printer tray
<u>XPRINT STRETCH</u>	Copy and resize a bitmap to a host printer page
<u>XPRINT STYLE</u>	Set the line style to be used by various XPRINT statements
<u>XPRINT TEXT WIDTH</u>	Calculate the size of text to be printed on a host printer
<u>XPRINT WIDTH</u>	Set the graphic line width to be used by various XPRINT statements
<u>XPRINT\$</u>	Return the name of the attached host printer

The following functions are used to create and manage threads:

PROCESS GET PRIORITY	Retrieve the Priority Value for the current process.
PROCESS SET PRIORITY	Sets the Priority Value for the current process
THREADED	Declare Thread Local Storage (TLS) variables
THREAD CLOSE	Close a Windows thread
THREAD CREATE	Create a Windows thread
THREAD GET PRIORITY	Retrieve the Priority Value for a thread
THREAD FUNCTION	Declares a thread function
THREAD SET PRIORITY	Sets the Priority Value for a thread
THREAD RESUME	Resume execution of a suspended Windows thread
THREAD STATUS	Retrieve the Status of a Windows thread
THREAD SUSPEND	Suspend execution of a Windows thread
THREADCOUNT	Return the number of active threads that exist in a module
THREADID	Return a Long-integer thread identifier of the current thread

The following functions manipulate and manage time and the system date:

[DATE\\$](#) Set and retrieve the system date

[SLEEP](#) Pause the current thread for a specified number of milliseconds

[TIME\\$](#) Read and/or set the system time

[TIMER](#) Return the number of seconds that have elapsed since midnight

[TIX](#) Measures elapsed CPU cycles.

Miscellaneous functions:

- [ASM](#) Identify an [assembly-language](#) statement
- [BEEP](#) Play the default Windows sound through the computer speaker(s)
- [REM](#) Indicates the remainder of a line of source code is a remark or comment

Purpose	Align the next instruction to a boundary.
Syntax	<code>#ALIGN <i>boundary</i></code>
Remarks	<p>The #ALIGN metastatement is primarily used by advanced assembler programmers to gain ultimate efficiency from critical code sections.</p> <p>#ALIGN is used to round up the instruction location to a power of two address. The boundary parameter shown must be a power of two, in the range of 2 through 256.</p> <p>Classic PowerBASIC inserts NOP instructions into the code section to bring the instruction location up to the desired address. If the instruction location is already at a multiple of boundary, #ALIGN has no effect.</p>
See also	#OPTIMIZE , ASM , TIX

Purpose	Artificially inflate the disk image size of a compiled program.
Syntax	<code>#BLOAT <i>size_expression</i></code>
Remarks	#BLOAT allows the creation of artificially bloated program files on disk, in order to match or exceed that generated by competing " <i>BloatWare</i> " compilers. #BLOAT does not affect the memory image size (running size) of a compiled program.

size_expression The *size_expression* parameter is a simple [Long-integer](#) expression that specifies the total desired size of the compiled programs disk image, but is ignored if it is smaller than the actual program size. #BLOAT uses sections of the actual compiled code to fill and obfuscate the portion added to the file.

While #BLOAT adds no true merit to the technical efficiency of the compiled code, there are a number of reasons for its use, including:

1. To allow "*BloatWare*" programmers to feel more comfortable when using Classic PowerBASIC.
2. To impress project leaders/managers with the volume of executable code created.
3. To allay the fears of uninformed customers who may mistakenly infer that "such tiny programs couldn't possibly do everything that..."
4. To make certain versions of a program more readily identifiable simply by examining the size of the file on disk.
5. To improve convolution of the contents of the executable disk image, because the bloat region appears to contain executable code.

See also [#COMPILE EXE](#), [#OPTIMIZE](#)

Example `#BLOAT 1024 * 1024 * 4 ' Create a 4 MB EXE file`

Purpose	Declare information to be included in a COM Type Library .
Syntax	<pre>#COM DOC "This is specific information to be used in the Help String" #COM HELP "MyProg.chm"[, &H1E00] #COM NAME "LibName", 3.32 #COM GUID GUID\$("{20000000-2000-2000-2000000000000002}") #COM TLIB {ON + OFF -}</pre>
Remarks	<p>The #COM metastatement establishes information about the COM library or application which can be extracted by COM client programs.</p> <p>#COM DOC specifies a help string which usually provides a general description of the COM server.</p> <p>#COM HELP specifies the name of the associated help file and the help context code. The name must appear as a string literal, while the context code is an unsigned DWORD value greater than zero. The context code may be specified in decimal or radix format.</p> <p>#COM NAME specifies the name of the server and the version number. The name must consist of only letters, numbers, and underscore characters, and may contain no punctuation nor spaces. If no name is specified, Classic PowerBASIC substitutes the module name. If no version is specified, Classic PowerBASIC uses version number 0.0.</p> <p>#COM GUID specifies the GUID which identifies the entire application or library (APPID or LIBID). If no GUID is specified, Classic PowerBASIC substitutes a random GUID for this purpose.</p> <p>#COM TLIB ON specifies that the compiler should create a type library for the compiled EXE or DLL.</p> <p>#COM TLIB OFF (default) specifies that the compiler should not create a type library for the compiled EXE or DLL.</p>

Type Libraries only support the following data types: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). If any Methods or Properties use data types not supported by Type Libraries, you will receive a [Error 581 - Type Library creation error](#), when using the [#COM TLIB ON](#) metastatement.

See also [CLASS](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [Just what is COM?](#)

Purpose	Determine what type of file will be created by the compiler.
Syntax	<code>#COMPILE {EXE DLL} ["filename{.exe .dll}"]</code>
Remarks	<p>This metastatement is used to specify whether a module is to be compiled as an EXE or as a DLL file. The #COMPILE metastatement can only be used once per program, and must be placed before any executable code. You may, optionally, specify the target name and directory of the file. If this clause is omitted, the compiled file is given the name of the main source code file with an .EXE or .DLL extension.</p> <p>Some examples follow:</p> <pre>#COMPILE EXE ' Same name as source, i.e., ABC.EXE #COMPILE DLL ' Same name as source, i.e., ABC.DLL #COMPILE EXE "ABC" ' Compiles to ABC.EXE #COMPILE DLL "ABC" ' Compiles to ABC.DLL #COMPILE EXE "ABC.BAS" ' Compiles to ABC.EXE</pre> <p>If a path is included, the compiled file is placed in the named directory; otherwise, it is placed in the current directory.</p> <p>If the named directory does not exist, the filename is invalid or locked, if the EXE is still running, or if the file cannot be successfully stored in that location for some other reason, a compile-time Error 496 ("Destination file write error") occurs.</p>
Restrictions	<p>If #COMPILE is not specified, the default is #COMPILE EXE. The default value has changed since early versions of Classic PowerBASIC for Windows, so use an explicit #COMPILE metastatement to avoid ambiguity.</p> <p>When compiling a DLL, the module may not include a PBMAIN or WINMAIN function. See DLLMAIN, LIBMAIN, and PBLIBMAIN for further information.</p>
See also	#COMPILE , DLLMAIN , LIBMAIN , PBLIBMAIN , PBMAIN , WINMAIN

Purpose	Compiler directive to suppress generation of debugging code.
Syntax	<code>#DEBUG CODE {ON + OFF -}</code>
Remarks	<p>When a program is compiled for debugging in the Classic PowerBASIC IDE, the compiler must generate some additional code to facilitate setting of breakpoints and some other debug operations. In most cases, this does not affect the execution of your program. However, in the case of code repetition in a tight loop, or for certain assembler code or data which must not be altered, it may be very important that some debugging code be suppressed so that code will execute correctly, and at full speed.</p> <p>#DEBUG CODE OFF suppresses generation of debug code, from that line, until a subsequent #DEBUG CODE ON (or the end of the Sub/Function/Method/Property) is reached. Of course, when debug code is suppressed, it is not possible to set breakpoints on those lines.</p> <p>#DEBUG CODE metastatements are ignored if not compiling for debug.</p>
See also	Error Trapping , Errors , Debugging , #DEBUG DISPLAY , #DEBUG ERROR , #DEBUG PRINT

#DEBUG DISPLAY metastatement New! [Top](#) [Previous](#) [Next](#)

Purpose	Display a message when an untrapped run-time error occurs.
Syntax	<code>#DEBUG DISPLAY {ON + OFF -}</code>
Remarks	<p>#DEBUG DISPLAY ON enables error display mode within a compiled Classic PowerBASIC program. In this mode, whenever an untrapped error occurs (without the benefit of ON ERROR GOTO, TRY/CATCH, etc.), program execution is suspended, and a descriptive message is displayed. This message includes the error number, a brief description of the error, and a position descriptor word to help you find the location of the error. The position descriptor word is the first 8 characters of the name of the last (most recent) label, line number, or procedure that was executed. This mode should only be used during program development and debugging. It should never be used in a production program.</p> <p>When the descriptive message is displayed, it is accompanied by two buttons marked "OK" and "Cancel". If "OK" is selected, program execution continues despite the error condition. If "Cancel" is selected, program execution is stopped. However, if any child processes were started, it is possible they will continue running until ended normally.</p> <p>#DEBUG DISPLAY OFF suppresses display mode, and is the default condition.</p>
Restrictions	#DEBUG DISPLAY ON OFF can only be executed once and must precede all executable code. If #DEBUG DISPLAY is omitted, the default condition is #DEBUG DISPLAY OFF.
See also	Error Trapping , Errors , Debugging , #DEBUG CODE , #DEBUG ERROR , #DEBUG PRINT

Purpose	Control generation of error checking code.
Syntax	<code>#DEBUG ERROR {ON + OFF -}</code>
Remarks	<p>#DEBUG ERROR option specifies whether the compiler should generate code that checks for array boundary and null-pointer errors wherever they may occur. The default setting is OFF.</p> <p>When #DEBUG ERROR mode is ON, any attempt to access an array outside of its boundaries, or attempting to use a null-pointer will generate a run-time Error 9 ("Subscript/Pointer out of range"), and the statement itself is not executed.</p> <p>When OFF, all statements are executed "as-is" and no errors are generated. However, accessing an array outside its boundaries or using a null-pointer can cause a General Protection Fault (GPF) or Exception error.</p> <p>It is best to enable #DEBUG ERROR error checking when developing a program. Once all of the more obvious bugs have been eradicated, you will want to return to the default setting (OFF), as this will make your code smaller and faster. Depending on the type of application being developed, the final (production) version of a program may not need to contain any error-checking code.</p>
Restrictions	<p>#DEBUG ERROR is always enabled when code is running within the Debugger, regardless of any explicit #DEBUG ERROR metastatement. Disk I/O errors are always caught, regardless of the state of #DEBUG ERROR.</p> <p>#DEBUG ERROR ON does not trap array boundary errors of arrays within User-Defined Types and Unions. Pointers are only tested for null (zero) values. Non-zero pointer target addresses are not tested for readability or writeability.</p>
See also	Error Trapping , Errors , Debugging , #DEBUG CODE , #DEBUG DISPLAY , #DEBUG PRINT

#DEBUG PRINT metastatement

[Top](#) [Previous](#) [Next](#)

Purpose	Display information in the IDE 's Debugger Output Window
Syntax	<code>#DEBUG PRINT <i>string_expression</i></code>
Remarks	<p>The PRINT option allows the programmer to display arbitrary information in the IDE's Debug Output Window during a debugging session. The output window is provided by debugger to display status information about the state of the debugging session; however, #DEBUG PRINT provides a convenient way of creating a "process log" of a Sub/Function/Method/Property/Variable as the program runs. Combined with FUNCNAME\$, #DEBUG PRINT can be a useful tool for debugging application code. See the Example below.</p> <p>This is possible because the Debugger Output Window has a scrollable range somewhat like a console window, whereas the Watch Window shows only the instantaneous value of a variable.</p> <p>#DEBUG PRINT statements are ignored when code is compiled into a standalone (EXE/DLL) file; they are only included when using the Debugger. Control codes in the string are translated into hex format in the output window. For example, embedded CHR\$(0) or \$NUL bytes are displayed as "<00>".</p>
See also	Debugging , #DEBUG ERROR , FUNCNAME\$
Example	<pre>FUNCTION PBMAIN() AS LONG Arg1% = 10000 Arg2% = 20000 CALL MySub(Arg1%, Arg2%) CALL MySub(Arg2%, Arg1%) #DEBUG PRINT "Done!" END FUNCTION SUB MySub(Arg1%, Arg2%) #DEBUG PRINT "We're in " & FUNCNAME\$ #DEBUG PRINT "Arg2% is" & STR\$(Arg2%) END SUB</pre>
Result	<pre>We're in MYSUB Arg2% is 20000 We're in MYSUB Arg2% is 10000 Done!</pre>

%DEF operator

Purpose Determine if an equate has been previously defined.

Syntax `%DEF({%numeric_equate | $string_equate})`

Remarks The %DEF operator tests whether or not an equate has been defined. If the equate has been defined, %DEF returns TRUE (non-zero); or FALSE (zero) if it has not be defined.

Classic PowerBASIC automatically defines the equates in the following table according to the Classic PowerBASIC compiler being used. Please note the references to other Classic PowerBASIC compilers are included for those writing programs that may be compilable by more than one Classic PowerBASIC compiler.

Equate	Definition
%PB_CC32	Pre-defined as TRUE (non-zero) in PB/CC for Windows, but is not defined in other compilers.
%PB_DLL16	Pre-defined as TRUE (non-zero) in PB/DLL 16-bit, but is not defined in other compilers.
%PB_DLL32	Synonym of %PB_WIN32
%PB_WIN32	Pre-defined as TRUE (non-zero) in PB/Win 32-bit, but is not defined in other compilers.
%PB_REVISION	Pre-defined as the hex revision (8.00 = &H800).
%PB_REVLETTER	Pre-defined as the ASCII code of the revision letter (a = &H61), or &H20 if there is no revision letter.
%PB_EXE	Pre-defined as TRUE (non-zero) if compiling to EXE or CON format (CON valid in PB/CC only), or as FALSE (zero) if compiling to DLL format (PB/Win only). The equate %PB_EXE is always defined in Classic PowerBASIC, so %DEF(%PB_EXE) will always be evaluated as TRUE. The difference being the value assigned to the equate by the compiler. See the examples below.

These can be used in conjunction with #IF as a compiler directive to selectively include or exclude code from the compiled file.

See also [#IF](#), [Numeric Equates](#), [Built-in numeric equates](#), [String Equates](#), [Built-in string equates](#)

Example

```
' 1. Conditional compilation for PB/CC or PB/Win
#IF %DEF(%PB_CC32)
  'Assume PB/CC
```

```
#COMPILE EXE "\PBCC\APPS\MYPROG.EXE"
#ELSE
    'Assume PB/Win
    #COMPILE DLL "MYAPP.DLL"
#ENDIF

' 2. Conditional compilation for EXE/CON or DLL
#IF %PB_EXE
    ' we are compiling to an EXE (PB/CC or PB/Win)
    FUNCTION PBMAIN
        [statements]
    END FUNCTION
#ELSE
    ' we are compiling to a DLL (PB/Win)
    FUNCTION PBLIBMAIN
        [statements]
    END FUNCTION
#ENDIF
```

Purpose	Specify if variables must be declared before use.
Syntax	<code>#DIM {ALL NONE}</code>
Remarks	<p>#DIM NONE (the default), requires you to dimension arrays, but not other kinds of variables, before their use.</p> <p>Using #DIM ALL requires you to declare all variables before they are used in a program. This option makes Classic PowerBASIC behave a lot like languages like C++ and Pascal which require that all variables be declared before they can be used. Although this will require more work, as even simple variables must be declared with DIM, INSTANCE, LOCAL, GLOBAL, STATIC, or THREADED statements, it will protect you from subtle errors like misspelling a variable name. For example, if you are using a variable <i>NumRecords</i> in your program and write a line like:</p> <pre>INCR NumRecrods</pre> <p>Classic PowerBASIC will detect that you're trying to use a previously undeclared variable (since <i>NumRecrods</i> is misspelled) and give you a compile-time error 519 ("Missing declaration"). If you hadn't specified #DIM ALL, you wouldn't have gotten an error, but your program would now have a bug that could be difficult to diagnose.</p> <p>#DIM ALL means the same thing as OPTION EXPLICIT, and the two can be used interchangeably.</p>
Restrictions	When #DIM ALL is used, type-specifier symbols with variable names are not allowed in a DIM var statement. e.g. <code>Dim a\$(10)</code> will result in compile error 519. Instead variables or arrays defined with the DIM statement must use the AS vartype format. Additionally, DEFtype statements, such as DEFINT, DEFLNG, etc. will be ignored, resulting in an error 519 where any variable they would otherwise define is used.
See also	DEFtype , DIM , GLOBAL , INSTANCE , LOCAL , REDIM , STATIC , OPTION EXPLICIT
Example	<pre>#DIM ALL [statements] DIM ListName(1 TO 400) AS STRING [statements] FOR ix = 1 TO 10 ' Classic PowerBASIC flags this line ' since "ix" wasn't dimensioned ListName(ix) = "Test" NEXT</pre>

#IF/#ELSEIF/#ELSE/#ENDIF metastatements

- Purpose

Define sections of source code to be compiled or ignored, depending on a certain condition. This is often referred to as conditional compilation.
- Syntax

```
#IF [NOT] {%equate | %DEF({%numeric_equate | $string_equate}) |  
expression}  
[statements]  
[#ELSEIF [NOT] {%equate | %DEF({%numeric_equate | $string_equate}) |  
expression}  
[statements]]  
[#ELSE  
[statements]]  
#ENDIF
```
- Remarks

%equate is a named [constant](#) or constant value. The [%DEF](#) operator allows you to test whether an equate has been defined. [%DEF](#) returns [TRUE](#) or [FALSE](#). Typical usage: `#IF %DEF(%PB_DLL16)` or `#ELSEIF NOT %DEF(%PB_WIN32)`. *expression* may be a simple numeric expression using the arithmetic operators +, -, *, /, and \, and the relational operators >, <, >=, <=, <>, and =, and may also include the [CVQ](#) function.

Classic PowerBASIC automatically defines the equates in the following table according to the Classic PowerBASIC compiler being used. Please note the references to other Classic PowerBASIC compilers are included for those writing programs that may be compilable by more than one Classic PowerBASIC compiler.

Equate	Definition
%PB_CC32	Pre-defined as TRUE (non-zero) in PB/CC for Windows, but is not defined in other compilers.
%PB_DLL16	Pre-defined as TRUE (non-zero) in PB/DLL 16-bit, FALSE (zero) in other compilers.
%PB_DLL32	Synonym of %PB_WIN32
%PB_WIN32	Pre-defined as TRUE (non-zero) in PB/Win 32-bit, but is not defined in other compilers.
%PB_REVISION	Pre-defined as the hex revision (8.00 = &H800).
%PB_REVLETTER	Pre-defined as the ASCII code of the revision letter (a = &H61), or &H20 if there is no revision letter.
%PB_EXE	Pre-defined as TRUE (non-zero) if compiling to EXE or CON format (CON valid in PB/CC only), or as FALSE (zero) if compiling to DLL format (PB/Win only). The equate %PB_EXE is always defined in Classic

	PowerBASIC, so %DEF(%PB_EXE) will always be evaluated as TRUE. The difference being the value assigned to the equate by the compiler.
--	---

Examples of valid expressions can include:

```
#IF %DEBUG = -1&
#IF %DEBUG AND (NOT %RELEASE)
#IF NOT %DEBUG
#IF %VERSION <> CVQ("DemoMode")
```

Note that the [AND](#), [OR](#) and [NOT](#) operators work as bitwise operators, rather than logical operators, in #IF metastatements.

If the value of %equate or if %DEF(%equate|\$equate) is TRUE (non-zero) or if the result of *expression* is TRUE, the statements between #IF and #ELSE or #ELSEIF are compiled, and the statements between #ELSE or #ELSEIF and #ENDIF are ignored.

If the value of %equate or %DEF(%equate|\$equate) is FALSE (zero) or the result of *expression* is FALSE, the statements between #IF and #ELSE or #ELSEIF are ignored, and those between #ELSE or #ELSEIF and #ENDIF are compiled.

The #ELSE or #ELSEIF clause and associated statements are optional, but #ENDIF is required.

Conditional compilation statements can be nested up to 16 levels deep. A primary use of conditional compilation is to include test code in your programs that will be compiled during program development (but not in the final product), and to facilitate building special editions of an application from a single source code file.

It is possible to perform bitwise operations on equates to produce a TRUE/FALSE result. For example:

```
#IF (%PB_REVISION AND &H0FF00) - &H0700
    SoftwareVersion$ = "not 7.x"
#ELSE
    SoftwareVersion$ = "7.x"
#ENDIF
```

See also [%DEF operator](#), [IF statement](#), [IF block](#)

Example

```
Example ' 1. Conditional compilation by equate value
%DEBUG = -1          'set to 0 for no debugging
#IF %DEBUG
    CALL SubRoutine(Arg1, Arg2, Arg3, Answer)
    CALL DisplayDebugData(Answer)
#ELSE
    CALL SubRoutine(Arg1, Arg2, Arg3, Answer)
#ENDIF

' 2. Conditional compilation for EXE/CON or DLL
#IF %PB_EXE
' we are compiling to an EXE (PB/CC or PB/Win)
FUNCTION PBMAIN
    [statements]
```



```
    END FUNCTION
#ELSE
    ' we are compiling to a DLL (PB/Win)
    FUNCTION PBLIBMAIN
        [statements]
    END FUNCTION
#ENDIF
```

Purpose	Instruct the compiler to read a text file from disk and treat it as an integral part of the source code.
Syntax	<code>#INCLUDE [ONCE] "<i>filespec</i>"</code>
Remarks	<p>Use #INCLUDE to compile the text of another file along with the current file. <i>filespec</i> is a string constant that follows normal LFN file-naming conventions, and which names a Classic PowerBASIC source code file. If <i>filespec</i> does not include an extension, the compiler looks for that file name with the default extension of .BAS.</p> <p>If <i>filespec</i> does not include a <i>path</i>, the compiler scans the search path for each #INCLUDE file before checking the current (default) directory. For the IDE, the search path can be set in the Compiler Preferences tab in the Options dialog. The search path can also be specified when compiling from the command-line by using the /I Include option.</p> <p>When the compiler encounters an #INCLUDE metastatement, it reads <i>filespec</i> from disk and continues compilation with the source code in <i>filespec</i>. When the end of <i>filespec</i> is reached, compilation continues with the statement immediately following the #INCLUDE in the original source file. The result is the same as if the contents of the included file were physically present within the original text. This allows large source files to be broken into smaller "units" that are more manageable.</p> <p>#INCLUDE metastatements can be nested as many as twelve levels deep. That is, an included file can have #INCLUDE metastatements of its own, including files that also have #INCLUDE metastatements, and so on, for a total of twelve levels of files (including the primary file). Note that macros count as #include files for nesting purposes.</p> <p>If the optional keyword ONCE is included, the specified file is included only one time during compilation, regardless of how many times it appears in the program. This is particularly useful when including common declaration files like WIN32API.INC to avoid redundant code, and the resulting errors. To be effective, the ONCE option must appear on every #INCLUDE of a particular file. Effectively, #INCLUDE ONCE means: "Include this file only if it has not already been included."</p>
See also	WIN32API.INC Updates
Example	<pre>' MYHELLO.BAS #include ONCE "WIN32API.INC" 'include Windows API calls FUNCTION PBMAIN MessageBox 0, "Hello World!", "Classic PowerBASIC", %MB_OK END FUNCTION</pre>

Purpose	Specify which messages should be sent to a Control Callback Function.
Syntax	<code>#MESSAGES COMMAND</code> <code>#MESSAGES NOTIFY</code>
Remarks	<p>#MESSAGES COMMAND specifies that only %WM_COMMAND messages be sent to Control Callback Functions, just as in earlier versions of Classic PowerBASIC</p> <p>#MESSAGES NOTIFY specifies that %WM_NOTIFY messages (as well as %WM_COMMAND messages) be sent to Control Callback Functions. This is the default condition, and need not be explicitly stated.</p> <p>There are two general types of Callback Functions. The first is the DIALOG CALLBACK, which is specified with the CALL DLGPROC clause of the DIALOG SHOW statement. It receives all messages which are directed to the dialog, including certain messages regarding its child controls. Specifically, this would include both %WM_COMMAND and %WM_NOTIFY messages. The second is the CONTROL CALLBACK, which is specified with the CALL CTLPROC clause of the CONTROL ADD statement. If specified, it receives all %WM_COMMAND and %WM_NOTIFY messages sent to the parent dialog.</p> <p>Prior to version 9.0 of Classic PowerBASIC for Windows, Control Callback Functions received only %WM_COMMAND messages. Beginning with PB 9.0, %WM_NOTIFY messages are sent as well. There are many situations where these added messages will prove to be very important to you. If your existing callback functions are written with complete error checking (ensuring that CB.MSG = %WM_COMMAND), this minor addition will cause no problems. It just presents additional information which can be acted upon, or just ignored. However, if callbacks were written without complete error checking, some ambiguity is possible. In this case, you should either update your Control Callback code, or suppress %WM_NOTIFY messages with a #MESSAGES Command metastatement.</p>
See also	Callbacks , DIALOG SHOW MODAL , DIALOG SHOW MODELESS , FUNCTION/END FUNCTION

Purpose	Choose between faster execution or smaller code size.
Syntax	<code>#OPTIMIZE {SIZE SPEED}</code>
Remarks	<p>The #OPTIMIZE metastatement is used to tell the compiler your preferences in regards to the optimization of generated code. You can specify optimization for either execution speed or smaller code size.</p> <p>If you choose the SPEED option, one of the primary actions of the compiler is to align heavily used code sections on an address boundary which is most beneficial to the CPU/FPU.</p> <p>In some cases, the speed of loop mechanisms (FOR/NEXT, DO/UNTIL...) can be improved by as much as 100%, and occasionally even more.</p>
Restrictions	#OPTIMIZE SIZE SPEED can only be executed once and must precede all executable code. If #OPTIMIZE is omitted, the default condition is #OPTIMIZE SPEED.
See also	#ALIGN

Purpose	Set the minimum Windows version requirements for your program.
Syntax	<code>#OPTION {VERSION3 VERSION4 VERSION5}</code>
Remarks	<p>The #OPTION metastatement controls the "minimum Windows version" tag that is written into your compiled code. If the version you select is equal or lower to the version of Windows that is running, the application will be executed. In turn, Windows will tailor the messages it sends to your program according to this version number, so your program will not need to handle messages from a later Windows version. The version tag may also affect the appearance and behavior of Windows common dialogs.</p> <p>Conversely, if the version tag you select is higher than the version of Windows that is running, Windows will display an error message instead of running your application. For example, running a VERSION5 application on a VERSION4 platform would fail. It is your responsibility to make sure that your program only uses the Windows features that are present in the specified version of Windows. For example, don't call an API that's present only in Windows XP, if you want your program to run under Windows 98.</p> <p>Use #OPTION VERSION3 to make the compiled output file require a minimum of Windows 95 or NT 3.1. That includes Windows 95, 98, ME, Windows NT 3.1-4.0, Windows 2000, XP, Windows 2003 (and later). Use #OPTION VERSION4 (default) to make the compiled output file require a minimum of Windows 95 or NT4. That includes Windows 95, 98, ME, Windows NT 4.0, Windows 2000, XP, Windows 2003 (and later). Use #OPTION VERSION5 to make the compiled output file require a minimum of Windows 2000. That includes Windows 2000, XP, Windows 2003 (and later).</p>
Example	<code>#OPTION VERSION5</code>

Purpose	Compiler directive to mark named blocks of generated Classic PowerBASIC Forms code.
Syntax	<code>#PBFORMS <i>named_block_marker</i></code>
Remarks	<p>#PBFORMS metastatements are generated by the Classic PowerBASIC Forms visual design tool, and placed automatically into the generated source code. #PBFORMS metastatements identify named blocks of code that have special meaning to both the compiler and the Classic PowerBASIC Forms visual design tool.</p> <p>#PBFORMS metastatements should not be removed or utilized - they should only be created and positioned by Classic PowerBASIC Forms. For more information, please refer to the documentation supplied with Classic PowerBASIC Forms.</p>

Classic PowerBASIC Forms is a visual design environment that enables rapid visual design of GUI application dialogs. Classic PowerBASIC Forms generates compilable Dynamic Dialog Tools ([DDT](#)) source code, directly from the dialogs created in the designer. Classic PowerBASIC Forms product information can be found at <http://www.PowerBASIC.com/products/clfw>.

An example **Classic PowerBASIC Forms** template and completed project can be found in the PB\SAMPLES\DDT\INTERFACE EXPLORER folder installed with Classic PowerBASIC for Windows.

Example

```
#PBFORMS CREATED
#PBFORMS BEGIN INCLUDES
```

Purpose	Control automatic allocation of Register variables .
Syntax	<code>#REGISTER {ALL DEFAULT NONE}</code>
Remarks	<p>Register variables may be Extended-precision floating-point variables, or 16/32-bit integer-class variables (Word, Dword, Integer, or Long). The #REGISTER metastatement determines the method of automatic allocation of Register variables.</p> <p>The #REGISTER metastatement works at two levels - a "global" setting, and a "local" setting for each Sub/Function/Method/Property. To set the global default #REGISTER options, it must precede all executable code. To override the global register option for an individual routine, it must be placed between the FUNCTION/END FUNCTION, SUB/END SUB, METHOD/END METHOD, or PROPERTY/END PROPERTY pairs before any executable code.</p>
ALL	#REGISTER ALL requests automatic allocations of all possible Register variables, both integer-class and Extended-precision float variables.
DEFAULT	#REGISTER DEFAULT (default) requests automatic allocations of integer-class variables in all cases, and Extended-precision floating-point variables located in a routine which contains no reference to another procedure.
NONE	#REGISTER NONE disables automatic assignment of Register variables. You can still use the REGISTER statement to explicitly define Register variables in your code on an individual basis. This provides a way to hand-optimize your code to help obtain the utmost performance.
Restrictions	Classic PowerBASIC transparently prevents the automatic register conversion of the variable used in the TO clause of the DIALOG SHOW MODAL and DIALOG SHOW STATE statements. If the target variable is explicitly declared as a register variable, Classic PowerBASIC raises a compile-time Error 491 ("Invalid register variable"). This is necessary as the result values stored in such variables may be assigned from the context of other procedures, and this may only occur with a memory variable.
See also	REGISTER , Optimizing your code
Example	<pre>#REGISTER DEFAULT ' global register setting FUNCTION PBMAIN() AS LONG #REGISTER NONE ' No automatic register ' vars in this function REGISTER x& ' Explicitly declare x& END FUNCTION</pre>

Purpose Embed a Classic PowerBASIC [Resource](#) (.PBR) file into the compiled EXE or [DLL](#) file.

Syntax `#RESOURCE "filespec"`

Remarks This metastatement is used to include a Classic PowerBASIC resource file into your program or DLL. Classic PowerBASIC resource files are created using the PBRES.EXE utility.

To create a .PBR file, you must first create a resource script (.RC) which contains resource commands. A typical resource file to specify an Icon consists of just one line of text:

```
ICON1 ICON MYICON.ICO
```

This line of text would be saved as a .RC file, and then compiled into a .RES file with a resource compiler, such as the 32-bit Microsoft resource compiler (RC.EXE). Then, the .RES file would be converted into a Classic PowerBASIC resource (.PBR) with the PBRES.EXE.

Resource files can contain bitmaps, string data, custom binary data, version information, dialogs and a whole lot more. To create more complex resource files, a Resource Editor is highly recommended. Classic PowerBASIC's PB/Win compiler includes a copy of the Microsoft Resource Compiler (RC.EXE).

You can find examples of .RC files in the SAMPLES folder installed with Classic PowerBASIC, and in the Resource Files chapter.

The Classic PowerBASIC [IDE](#) can be used to compile a Resource Script (.RC file) into Classic PowerBASIC resource file format (a .PBR file) without resorting to using command-line tools to compile resource files.

Restrictions You cannot have more than one resource file in a module. If you need more than one, the usual way to accomplish this is to create a separate DLL for each additional resource file. Classic PowerBASIC does not permit more than one #RESOURCE statement per module.

Note: If you are adding an icon resource to a Console mode application (for example, a GUI app that has been "altered" to appear as a console app), please note that Windows 95's "Windows Explorer" will not display icons of console applications in file lists. This behavior was corrected in Windows 98, and later editions of Windows.

See also [Resource Files](#)

Example `#RESOURCE "HELLO.PBR"`

#STACK metastatement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the maximum potential stack size.
Syntax	<code>#STACK <i>num_expr</i></code>
Remarks	<p>The literal numeric expression is expressed in bytes, and is rounded up to the next 64 Kb boundary. The minimum allowable stack size is 128 Kb, and a typical stack size of at least 1 Megabyte (the default) is usually recommended.</p> <p>Upon program startup, an initial block of 128 Kb of physical memory is allocated to the stack. As the stack grows, additional memory is automatically added, as necessary, up to the specified maximum. Since physical memory is only committed as required, it is usually prudent to overestimate potential stack needs.</p>
Restrictions	#STACK is meaningful with EXE (executable) files only.

Purpose	Enable or disable integrated development tool code in compiled code.
Syntax	<code>#TOOLS [ON + OFF -]</code>
Remarks	The #TOOLS metastatement allows integrated development tools like TRACE , PROFILE , and CALLSTK to be readily disabled, ensuring that extra code and data is not compiled into the final (distribution) version of an application. #TOOLS defaults to ON, and may appear only once in the source code, before any statement that generates executable code.
See also	CALLSTK , CALLSTK\$, CALLSTKCOUNT , FUNCNAME\$, PROFILE , TRACE

#UTILITY metastatement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose	Compiler directive to allow external utility programs to read text inserted on the #UTILITY line.
Syntax	<code>#UTILITY <i>"any text for an external program"</i></code>
Remarks	The entire line is ignored by the Classic PowerBASIC compiler.

ABS function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the absolute value of a numeric expression.
Syntax	<i>y</i> = ABS(<i>numeric_expression</i>)
Remarks	The absolute value of a number is its non-negative value. For example, the absolute value of -3 is 3, and the absolute value of +3 is also 3. The absolute value of 0 is 0.
See also	SGN

Purpose	Attach a table of keyboard accelerators to a DDT dialog.
Syntax	<code>ACCEL ATTACH <i>hDlg</i>, <i>AccelTbl()</i> TO <i>hAccelHandle</i></code>
Remarks	<p>ACCEL ATTACH permits you to attach one table of accelerator key definitions to each DDT dialog in your application.</p> <p>The keyboard accelerator itself is a specific keystroke combination, which results in a %WM_COMMAND or %WM_SYSCOMMAND message being placed into the application's message queue.</p> <p>Keyboard accelerators are very similar to command accelerators, and often permit the same action selections as menus offer. Since keyboard accelerators can be used directly, they negate the need to navigate a menu in order to perform a specific action.</p> <p>Typically, application menu items inform users of available keyboard accelerators so expert users can work more efficiently without using the actual menus, but can still use the menu if required. The following image shows a menu showing command accelerator and keyboard accelerators for menu items (the command accelerators are underscored):</p>



On Windows XP and Windows 2000 you may need to press the ALT key before Command Accelerators are made visible. You can set if Command Accelerators are visible when using the ALT key or all the time in the Windows Display Settings.

For a command accelerator to operate, the specific menu item must be visible and enabled. Conversely, keyboard accelerators can be used without the menu being open. In the example above, the CTRL+X keystroke combination will perform the CUT action, but the accelerator letter t will only perform the Cut action if the EDIT menu is opened first.

AccelTbl() To utilize ACCEL ATTACH, you must first build the [array](#) *AccelTbl()* of

ACCELAPI [User-Defined Types](#) (UDTs). This *ACCELAPI* structure is a 6-byte structure with the following definition:

```
TYPE ACCELAPI WORD
  FVIRT AS BYTE ' Flags: One or more of %FVIRTKEY, %FSHIFT, %FALT and
%FCONTROL
  KEY AS WORD ' Accelerator key: ASCII code, or virtual key code
{%FVIRTKEY}
  CMD AS WORD ' Accelerator ID code gets passed in CB.CTL {LO(WORD,
WPARAM) }
END TYPE
```

You must build the array of *ACCELAPI* types yourself, then attach it to a dialog by executing an ACCEL ATTACH statement. There must be no empty elements in the array, so it must be sized accurately.

.FVIRT

The .FVIRT flags can be combined together with the [OR](#) operator to combine the actions of the individual flags, as follows:

- %FALT** The ALT key must be pressed along with the accelerator key.
- %FCONTROL** The CTRL key must be pressed along with the accelerator key.
- %FSHIFT** The SHIFT key must be pressed along with the accelerator key.
- %FVIRTKEY** The .KEY member specifies a virtual-key code. If this flag is not specified, the key member is assumed to specify an ASCII character code. %FVIRTKEY permits case-insensitive accelerator keystroke definitions - the Capslock state is ignored. For example, ALT+A and ALT+a (as determined by the Capslock key) produce the same accelerator event. If %FVIRTKEY is not used, the accelerator ALT+A would not trigger if Capslock were inactive.

.KEY

If the %FVIRTKEY flag is specified in the .FVIRT member, the .KEY field contains the virtual key code for the accelerator key. Virtual key equates are defined in the [WIN32API.INC](#) file, starting with the prefix %VK_.

If %FVIRTKEY is not specified, the accelerator key code in the .KEY member is the [ASCII](#) code of the accelerator key. In this case, alphanumeric keystrokes become case-sensitive and the state of the Capslock key state becomes important. For example, if an accelerator were defined for ALT+A, it would be activated only if the Capslock key was on. Conversely, if an accelerator were defined for ALT+a then it would only be activated Capslock was off.

.CMD

The .CMD member should contain the user-defined numeric ID code of the accelerator. When an accelerator keystroke occurs, a

WM_COMMAND message is sent to the dialog [Callback](#) Function, with the accelerator identifier returned by the [CB.CTL](#) function.

It is usual practice to use the ID of a control that is to be activated by an accelerator. Accelerator notification codes sent to the Callback Function have [CB.CTLMSG](#) set to 1 (as opposed to button click events messages where CB.CTLMSG = %BN_CLICKED).

hDlg The handle of the dialog to attach the accelerator table to.

hAccelHandle [Double-word](#) or [Long-integer](#) variable where the handle of the attached accelerator table will be stored, or zero if the attach operation was unsuccessful.

Restrictions If a previous table was attached to the target dialog, the table is automatically destroyed when the new table is attached in its place. The accelerator table is also destroyed automatically when the dialog is closed.

You can destroy the current accelerator table by executing ACCEL ATTACH with an array which is not dimensioned, but there is little or no reason to ever perform this action.

Accelerator tables can only run correctly when they are created in the same module that creates the dialog to which each table is attached.

See also [DIALOG NEW](#), [MENU ADD STRING](#), [MENU ATTACH](#)

Example

```
DIM ac(0 TO 8) AS ACCELAPI
LOCAL hAccelHandle AS DWORD

FOR x& = 0 TO 8
  ac(x&).fvirt = %FCONTROL OR %FSHIFT OR %FVIRTKEY
  ac(x&).key = %VK_1 + x& ' CTRL+SHIFT+1 to 9
  ac(x&).cmd = %BTN1 + x& ' %BTN1 to %BTN9
NEXT x&

ACCEL ATTACH hDlg, ac() TO hAccelHandle
```

ACODE\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose Translate a [Unicode](#) string into an ANSI string.

Syntax `a$ = ACODE$(UnicodeStrExpression [,CodePage&])`

Remarks ACODE\$ returns the ANSI, multi-byte equivalent string of Unicode contained in *UnicodeStrExpression*. To convert an ANSI string into Unicode, use the [UCODE\\$](#) function.

If the optional parameter *CodePage&* is present, it represents the code page to be used for the conversion process. If not given, the default code page for the locale of the executing computer is used.

Unicode strings require two [bytes](#) to represent a Unicode character, whereas ANSI strings (the native Classic PowerBASICstring format) use one byte to represent a character. Therefore, ACODE\$ returns a string that has half of the byte count of the Unicode string, yet represents the same number of characters.

See also [UCODE\\$](#), [UCODEPAGE](#)

Purpose

The AND operator works as both a logical and a bitwise arithmetic operator.

Syntax

$p \text{ AND } q$

Using AND as a logical operator

AND returns [TRUE](#) (non-zero) if (and only if) both its operands are TRUE. The AND truth table looks like this:

Truth Table

x	y	x AND y
T	T	T
T	F	F
F	T	F
F	F	F

Using AND as a bitwise arithmetic operator

AND masks clear selected bits of an integer-class value without affecting the other bits. For example, to clear the most-significant (leftmost) 2 bits in the integer value &H9700, AND it with &H3FFF. That is, the mask contains all 1s, except for the bit positions you want to force to 0:

	1001 0111 0000 0000	= &H09700	
AND	0011 1111 1111 1111	= &H03FFF (the mask)	
	0001 0111 0000 0000	= &H01700 (result)	
MSB	↑	↑	LSB (bit 0)

See also

[Arithmetic Operators](#), [EQV](#), [IMP](#), [ISFALSE](#), [ISTRUE](#), [NOT](#), [OR](#), [XOR](#)

ARRAY ASSIGN statement

[Top](#) [Previous](#) [Next](#)

Purpose	Allow the assignment of a number of values to successive elements of an array .
Syntax	<code>ARRAY ASSIGN <i>array</i>() = <i>param1</i> [,<i>param2</i>] [,]</code>
Remarks	ARRAY ASSIGN allows the assignment of a number of values to successive elements of an array. The assignment always starts with the first array element, and continues sequentially as the elements appear in memory. The values to be assigned must match the array type, and may be literals , variables , or expressions. ARRAY ASSIGN cannot be used on an array of Interfaces .
See also	ARRAY DELETE , ARRAY INSERT , ARRAY SCAN , ARRAY SORT , DIM , LBOUND , REDIM , UBOUND , Array Data Types
Example	<code>ARRAY ASSIGN x&() = 1,2,3,4,5,6,7,8,9,10</code>

ARRAY DELETE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Delete a single item from a given array .
Syntax	<code>ARRAY DELETE array([<i>index</i>]) [FOR <i>count</i>] [, <i>expression</i>]</code>
Remarks	<p>ARRAY DELETE deletes the data stored at the nominated element in <i>array</i>, an n-dimensional array. You can specify the <i>index</i> of the element which is to have its data deleted, how many elements (<i>count</i>) are to be automatically shifted down by one position, and what data value to give the last element after the rest of the elements have been shifted (<i>expression</i>).</p> <p>All of these parameters are optional. If <i>index</i> is not specified, the data stored in the element at the beginning of the array is deleted. If <i>expression</i> is not present, the last element that the data is shifted out of will contain zero if <i>array</i> is a numeric array, or an empty string if <i>array</i> is a string array. If a shift count is given, when shifting the rest of the array to eliminate the element, only count elements will be shifted.</p> <p>By default, ARRAY DELETE throws away the data at the element <i>index</i> of <i>array</i>, shifting the data in the appropriate portion of the array to cover the old element:</p>

```
DIM A(1 TO 4) AS LONG
ARRAY DELETE A(2), 17&
```

makes $A(2)=A(3)$, $A(3)=A(4)$, and $A(4)=17$. The original value of $A(1)$ remains in place. Use *count* to "protect" a portion of the array from the shift:

```
DIM A(1 TO 4) AS LONG
ARRAY DELETE A(2) FOR 2, 17&
```

makes $A(2)=A(3)$ and $A(3)=17$ because you told it to shift only 2 elements. The original values of $A(4)$ and $A(1)$ remain in place.

DELETE with multi-dimensional arrays

count can also be used with a [multi-dimensional array](#) (stored in linear column-major order; see [ARRAY SORT](#)), to prevent shifting element data from one dimension into another dimension, thus preserving the organization of the array. For example:

```
DIM A(0 TO 1,0 TO 1) AS INTEGER
A(0,0)=0
A(1,0)=100
A(0,1)=200
A(1,1)=300
ARRAY DELETE A(0,0) FOR 2, 17%
```

makes $A(0,0)=100$ and $A(1,0)=17$. The original values of $A(0,1)$ and $A(1,1)$ remain in place since you told it to shift only 2 elements. Without *count*:

```
ARRAY DELETE A(0,0), 17%
```

makes $A(0,0)=100$, $A(1,0)=200$, $A(0,1)=300$, and $A(1,1)=17$. The original value of $A(0,0)$ is lost.

Restrictions ARRAY DELETE cannot be used on arrays within [UDT](#) structures or on an array of [Interfaces](#). However, ARRAY DELETE can be used with arrays of UDT structures - simply treat them as if they were an array of [fixed-length strings](#).

To use ARRAY DELETE on an embedded UDT array, use [DIM..AT](#) to dimension a regular array (of the same type) directly "over the top" of the UDT array, and use ARRAY DELETE on that array. For example:

```
TYPE SalesType
  OrderNum AS LONG
  PartNumber(1 TO 20) AS STRING * 20
END TYPE
[statements]
DIM Sales AS SalesType
[statements]
DIM Temp(1 TO 20) AS STRING * 20 AT VARPTR(Sales.Partnumber(1))
ARRAY DELETE Temp(5), "string"
ERASE Temp()
```

See also [ARRAY ASSIGN](#), [ARRAY INSERT](#), [ARRAY SCAN](#), [ARRAY SORT](#), [DIM](#), [LBOUND](#), [REDIM](#), [UBOUND](#), [Array Data Types](#)

Example Makes $A(2)=3$ and $A(3)=2.5$. $A(0)$ and $A(1)$ remain in place:

```
DIM A(0 TO 3) AS CUX
A(0)=0
A(1)=1
A(2)=2
A(3)=3
ARRAY DELETE A(2), 2.5@@
```

Makes $A(0)=2$, $A(1)=3$, and $A(2)=0$. The original value of $A(0)$ is lost:

```
DIM A(0 TO 2) AS EXT
A(0)=1
A(1)=2
A(2)=3
ARRAY DELETE A()
```

ARRAY INSERT statement

[Top](#) [Previous](#) [Next](#)

Purpose Insert a single item into a given [array](#).

Syntax `ARRAY INSERT array([index]) [FOR count] [, expression]`

Remarks ARRAY INSERT inserts a single data item into *array*, an n-dimensional array. You can specify the *index* at which the new element data is to be inserted, how many elements (*count*) are to be shifted up by one position to make room for the new element data, and what data value to give the new element (*expression*).

All of these parameters are optional. If *index* is not specified, the element data is inserted at the beginning of the array. If *expression* is not present, the new element will contain zero if *array* is a numeric array, or an empty string if *array* is a [string array](#).

If a shift *count* is given, when shifting the rest of the array to make way for the new element data, only *count* elements will be shifted.

By default, ARRAY INSERT throws away the data in last element of *array*, then shifts the appropriate portion of the array to make way for the new element data:

```
DIM A(1 TO 4) AS LONG
ARRAY INSERT A(2), 17
```

makes $A(4)=A(3)$, $A(3)=A(2)$, and $A(2)=17$. The original value of $A(4)$ is lost, while the original value of $A(1)$ remains in place. Use *count* to "protect" a portion of the array from the shift:

```
DIM A(1 TO 4) AS LONG
ARRAY INSERT A(2) FOR 2, 17
```

makes $A(3)=A(2)$ and $A(2)=17$ because you told it to shift only 2 elements. The original values of $A(4)$ and $A(1)$ remain in place.

INSERT with multi-dimensional arrays

count can also be used with a [multi-dimensional array](#) (stored in linear column-major order; see [ARRAY SORT](#)), to prevent shifting data from one dimension into another dimension, and thus preserving the organization of the array. For example:

```
DIM A(0 TO 1,0 TO 1) AS SINGLE
A(0,0)=0
A(1,0)=100
A(0,1)=200
A(1,1)=300
ARRAY INSERT A(0,0) FOR 2, 17
```

makes $A(0,0)=17$ and $A(1,0)=0$. The original values of $A(0,1)$ and $A(1,1)$ remain in place since you told it to shift only 2 elements. Without *count*:

```
ARRAY INSERT A(0,0), 17
```


makes $A(0,0)=17$, $A(1,0)=0$, $A(0,1)=100$, and $A(1,1)=200$. The original value of $A(1,1)$ is lost.

Restrictions ARRAY INSERT cannot be used on arrays within [UDT](#) structures or on an array of [Interfaces](#). However, ARRAY INSERT can be used with arrays of UDT structures - simply treat them as if they were an array of fixed-length strings.

To use ARRAY INSERT on an embedded UDT array, use [DIM..AT](#) to dimension a regular array (of the same type) directly "over the top" of the UDT array, and use ARRAY INSERT on that array. For example:

```
TYPE SalesType
  OrderNum AS LONG
  PartNumber(1 TO 20) AS STRING * 20
END TYPE
[statements]
DIM Sales AS SalesType
[statements]
DIM Temp(1 TO 20) AS STRING * 20 AT VARPTR(Sales.PartNumber(1))
ARRAY INSERT Temp(5), "string"
ERASE Temp()
```

See also [ARRAY ASSIGN](#), [ARRAY DELETE](#), [ARRAY SCAN](#), [ARRAY SORT](#), [DIM](#), [LBOUND](#), [REDIM](#), [UBOUND](#)

Example Makes $A(3)=2.5$ and $A(4)=3$. $A(0)$, $A(1)$, and $A(2)$ remain in place:

```
DIM A(0 TO 4) AS DOUBLE
A(0)=0
A(1)=1
A(2)=2
A(3)=3
ARRAY INSERT A(3), 2.5#
```

Makes $A(0)=0$, $A(1)=1$, and $A(2)=2$. The original value of $A(2)$ is lost:

```
DIM A(0 TO 2) AS QUAD
A(0)=1
A(1)=2
A(2)=3
ARRAY INSERT A()
```

Purpose Scan all or part of an [array](#) for a given value.

Syntax *Numeric array:*

```
ARRAY SCAN array([index]) [FOR count], expression, TO lvar&
```

String arrays:

```
ARRAY SCAN array ([index]) [FOR count] [, FROM start TO end] [, COLLATE  
{UCASE |  
  cstring}], expression, TO lvar&
```

Remarks ARRAY SCAN scans all or part of array, an n-dimension array, for the first element that satisfies *expression*. *expression* consists of a relational operator (=, >, <, <>, >=, <=, <>=, <><=) followed by an expression of the same data type as *array*. The relative index of the first match is stored in *lvar&*, which must be a [Long-integer](#) variable:

```
ARRAY SCAN A&(), > 5, TO I&
```

This line of code identifies the relative index of the first element of array *A&()* that is greater than 5, and stores the relative index in *I&*. The match index ranges from 1 to the last element of the scan + 1.

Since it is a relative index:

```
DIM A(1 TO 10) AS SINGLE  
ARRAY SCAN A(), > 17.42!, TO I&
```

will store 2 in *I&* if *A(2) > 17.42*, but:

```
DIM A(5 TO 20) AS SINGLE  
ARRAY SCAN A(), > 17.42!, TO lvar&
```

will store 2, not 6, in *lvar&* if *A(6) > 17.42*.

If none of the scanned elements satisfy *expression*, zero will be stored in *lvar&*.

Together, *index* and *count* specify the portion of array to be scanned. *index* specifies the element at which the scan is to begin, while *count* specifies the number of consecutive elements to be scanned. If *index* is not specified, the scan begins at the first element of *array*. If *count* is not specified, the array is scanned from element *index* to the last element of *array*. If neither is specified, the entire array is scanned:

```
DIM A&(1 TO 100)  
ARRAY SCAN A&(5), =1, TO I&           'scans 5..100  
ARRAY SCAN A&() FOR 10, =1, TO I&     'scans 1..10  
ARRAY SCAN A&(10) FOR 20, =1, TO I&   'scans 10..29  
ARRAY SCAN A&(), =1, TO I&           'scans 1..100
```

Scanning a string array

When scanning a [string array](#), COLLATE UCASE treats all lowercase letters as uppercase during the scan (for example, element "Bob" would satisfy the condition = "BOB"):

```

ARRAY SCAN A$( ), COLLATE UCASE, = "BOB", TO I&
' scans A$( ) for "BOB"; all letters treated as
' uppercase

```

COLLATE string is used to specify a non-standard scanning order. *cstring* must contain exactly 256 characters, in the order in which they should be compared, from lowest to highest. For example, the normal ascending ASCII scan order (where "A" is considered less than "B", etc.) would be described by a string containing ASCII codes 0 through 255 in order:

```

C$ = CHR$(0 TO 255)
ARRAY SCAN A$( ), COLLATE C$, > "BOB", TO I&

```

The normal descending ASCII scan order would be described by a string containing the reverse of the above:

```

C$ = STRREVERSE$(CHR$(0 TO 255))
ARRAY SCAN A$( ), COLLATE C$, > "BOB", TO I&

```

The COLLATE string option is provided as a flexible means with which to specify a descending scan, or to specify the scanning order for strings containing international characters or other special symbols.

See [ARRAY SORT](#) for more information on building collating strings.

When scanning a string array, all characters of each element of the array are normally considered when performing comparisons. To limit the comparison to a specific subset of characters, use FROM to specify the *start* position, and TO to specify the *end* position that ARRAY SCAN will consider within each array element. For example, you could scan based on the zip code contained in the last 5 characters of a 40-character address string:

```

ARRAY SCAN A$( ), = "90210", TO I&
' considers all characters when scanning for "90210"

ARRAY SCAN A$( ), FROM 36 TO 40, = "90210", TO I&
' considers positions 36..40 only when scanning

```

Scanning a multi-dimensional array

When scanning a [multi-dimensional array](#), the array is treated as a single-dimension array containing all of the elements of the multi-dimensional array, in linear column-major order. That is, all elements where all dimensions (except the first), are held at their minimum bounds, will come first in memory. These are immediately followed by the elements where the second dimension is set to its next consecutive index value, etc.

For example, the elements of a two-dimensional array (DIM A(0 TO n, 0 TO x)) would be stored in consecutive memory locations as follows:

```

A(0,0), A(1,0), , A(n,0) ' The first n+1 elements,
A(0,1), A(1,1), , A(n,1) ' The next n+1 elements,
[statements]             ' Subsequent elements,
A(0,x), A(1,x), , A(n,x) ' The last n+1 statements.

```

or more clearly:

```

A(0,0), A(1,0), A(n,0), A(0,1), A(1,1), A(n,1), A(0,x),

```

`A(1,x) , A(n,x)`

In this case, `ARRAY SCAN A(0,0) FOR n+1, >5, TO I&` would scan only elements `(0,0)...``(n,0)`, while `ARRAY SCAN A(0,0), >5, TO I&` would scan the entire array: elements `(0,0)...``(n,x)`. As mentioned earlier, since `ARRAY SCAN` records the relative index of the matched element, `ARRAY SCAN A(0,0), >5, TO I&` would store 2 in `I&` if `A(1,0) > 5`.

Options

The options for `ARRAY SCAN` can be specified in any order, as long as the `FOR` option, if present, directly follows the closing parenthesis of the name of *array*.

Restrictions

`ARRAY SCAN` cannot be used on arrays within [UDT](#) structures or on an array of [Interfaces](#). However, `ARRAY SCAN` can be used with arrays of UDT structures - simply treat them as if they were an array of fixed-length strings.

To use `ARRAY SCAN` on an embedded UDT array, use [DIM..AT](#) to dimension a regular array (of the same type) directly "over the top" of the UDT array, and use `ARRAY SCAN` on that array. For example:

```
TYPE SalesType
  OrderNum AS LONG
  PartNumber(1 TO 20) AS STRING * 20
END TYPE
[statements]
DIM Sales AS SalesType
[statements]
DIM Temp(1 TO 20) AS STRING * 20 AT VARPTR(Sales.Partnumber(1))
ARRAY SCAN Temp(), FROM 1 TO LEN(Search$), = Search$, TO lResult&
ERASE Temp()
```

See also

[ARRAY ASSIGN](#), [ARRAY DELETE](#), [ARRAY INSERT](#), [ARRAY SORT](#), [DIM](#), [LBOUND](#), [REDIM](#), [UBOUND](#), [Array Data Types](#)

Example

```
ARRAY SCAN A&(5) FOR 10, > 64000&, TO B&
```

Scans elements 5 through 14 of array `A&`, looking for the first element whose value is `> 64000`, and stores the relative index of that element in `B&`.

```
ARRAY SCAN A$(5) FOR 10, FROM 16 TO 25, COLLATE C$, = D$, TO B&
```

Scans elements 5 through 14 of array `A$`, looking only at characters 16 to 25 of each element, using the order specified by collating string `C$`, looking for the first element whose value is equal to `D$`, and stores the relative index of that element in `B&`.

Purpose	Sort all or part of a given array .
Syntax	<p>Numeric array:</p> <pre>ARRAY SORT <i>darray</i>([<i>index</i>]) [FOR <i>count</i>] [,TAGARRAY <i>tarray</i>()] [{ASCEND DESCEND}]</pre> <p>String array:</p> <pre>ARRAY SORT <i>dArray</i>([<i>index</i>]) [FOR <i>count</i>] [,FROM <i>start</i> TO <i>end</i>] [{COLLATE {UCASE cstring}}] [,TAGARRAY <i>tarray</i>()] [{ASCEND DESCEND}]</pre> <p>Custom sort array:</p> <pre>ARRAY SORT <i>darray</i>([<i>index</i>]) [FOR <i>count</i>] [,TAGARRAY <i>tarray</i>()] ,USING <i>custfunc</i>()</pre>
Remarks	<p>ARRAY SORT sorts all or part of <i>darray</i>, an n-dimensional array, in ascending or descending order. <i>tarray</i> is a tag-along array whose elements are swapped in the same order as those in <i>darray</i> as the sort proceeds (you could sort an array of names and have an array of corresponding addresses tag along, for example). <i>tarray</i> must have at least as many elements as <i>darray</i>, since corresponding elements of <i>tarray</i> will be swapped during the sort.</p> <p>Note that <i>tarray</i> does not have to be of the same type as <i>darray</i>. For example, you could have a numeric array containing account numbers tag along with a string array containing user names:</p> <pre>DIM Users\$(100 TO 500), AcctNum&(100 TO 500) ARRAY SORT Users\$(), TAGARRAY AcctNum&()</pre> <p>Together, <i>index</i> and <i>count</i> specify the portion of <i>darray</i> to be sorted. <i>index</i> specifies the element at which the sort is to begin, while <i>count</i> specifies the number of consecutive elements to be sorted. If <i>index</i> is omitted, the sort begins at the first element of <i>darray</i>. If <i>count</i> is omitted, the array is sorted from element <i>index</i> to the last element of <i>darray</i>. If both are omitted, the entire array is sorted:</p> <pre>DIM A&(1 TO 99) ARRAY SORT A&(5) 'sorts elements 5..99 of A& ARRAY SORT A&() FOR 10 'sorts elements 1..10 of A& ARRAY SORT A&(9) FOR 20 'sorts elements 9..28 of A& ARRAY SORT A&() 'sorts elements 1..99 of A&</pre>
	<p>Sorting numeric arrays</p> <p>By default, arrays are sorted in ascending order. To sort in descending order, include the DESCEND keyword:</p> <pre>ARRAY SORT A&(), DESCEND ' descending order ARRAY SORT A&(), ASCEND ' ascending order ARRAY SORT A&() ' ascending order</pre>
	<p>Sorting string arrays</p>

When sorting a string array, the sort is performed in ascending order by default. In addition to DESCEND, ARRAY SORT provides the COLLATE UCASE and COLLATE string options.

COLLATE UCASE treats all lowercase letters as equal to their uppercase counterparts during the sort (elements "Bob" and "BOB" would be considered equal, for example):

```
DIM A$(1 TO 5)
A$(1) = "Bob"
A$(2) = "Jan"
A$(3) = "Linda"
A$(4) = "Ann"
A$(5) = "Jerry"
ARRAY SORT A$(), COLLATE UCASE, DESCEND
'sorts A$() in descending order; case-insensitive
```

COLLATE *cstring* is used to specify an entirely new sorting order. This can be used for a variety of purposes, the most obvious of which is the case of international character sets. The collate string *cstring* must contain exactly 256 characters, one for each of the ASCII codes 0-255, in the order that they would be sorted (from lowest to highest, if an ascending sort were performed on them).

Each position in the string represents the ASCII code of that value. The contents of the [byte](#) at that position tell Classic PowerBASIC the "weight" or importance factor of that particular ASCII code. The default is that position 0 has a weight of 0, position 1 has a weight of 1, etc, so that [CHR\\$\(0\)](#) sorts first, [CHR\\$\(1\)](#) sorts next, and so on through [CHR\\$\(255\)](#).

Suppose you want the special character "ä" to have the same weight as the standard character "a". It's easy: construct a string of 256 characters, 0-255; then go to the position of "ä" (ASCII code 132), and change the contents of that byte so it is exactly equal to the code for "a" (97). The following code fragment constructs just such a collate string:

```
' Create a 256-character string:
FOR ix = 0 TO 255
  C$ = C$ + CHR$(ix)
NEXT
MID$(C$, 132 + 1) = CHR$(97)
```

We add one to the [ASC](#) value for [MID\\$](#) because string positions start at 1, not 0. We can also use the expanded [CHR\\$](#) function to create the same collating string using less code:

```
C$ = CHR$(0 TO 131, 97, 133 TO 255)
```

It is most important to remember the rule for creating a collating string, as it is easy to make an intuitive jump to the wrong conclusion. Each position in the string (1-256) represents the ASCII code with that value minus one ([CHR\\$\(0\)](#) to [CHR\\$\(255\)](#)). The contents of the byte at that position tell the ARRAY SORT procedure the new "weight" or importance factor for that

particular code. This is exactly the technique used by the 80x86-assembler opcode XLAT.

Suppose you want CHR\$(0) to sort at the very end of the sequence. To do that, you would set the byte at position 0+1 to CHR\$(255) and the bytes at positions 0+2 to 0+256 to the values 0 to 254. The ASCII sequence in the collating string would appear like this: 255,0,1,2,3,4254. Using the expanded CHR\$ function, this is straightforward:

```
C$ = CHR$(255, 0 TO 254)
```

To sort upper case and lower case alphabetic characters as exactly equal, just set positions 97 to 122 (a-z) to the values 65-90 (A-Z). This is precisely how COLLATE UCASE is handled. With the collating method implemented by this procedure in Classic PowerBASIC, it is possible for two or more ASCII codes to have equal "weight".

As mentioned earlier, many programmers make a common, fatal mistake by intuitively creating a collating string that is simply a list of ASCII codes, in the sequence they wish to sort. That is, they expect the byte which appears first in the string to sort first, the byte which appears next to sort second, so that creating a collate string from the BASIC code:

```
CHR$(65) + CHR$(66) + CHR$(67) + ...
```

might cause the characters "ABC..." to be sorted first. *This technique will never work with the ARRAY statement and must be carefully avoided.* We describe it here only because it is a common error. While it is arguably more intuitive than the technique implemented in Classic PowerBASIC, the reason it does not work is that it doesn't allow two or more ASCII codes to have the same "weight".

The following code builds a collating string compatible with the American OEM ASCII character set. For the fastest operation, this code should be run only once and the collating string should be made [global](#).

```
GLOBAL cu AS STRING
FOR x = 0 TO 255
  cu = cu + CHR$(x)
NEXT
MID$(cu, 97+1, 26) = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
MID$(cu, 129+1, 6) = "ueaaaa" ' üéâäã
MID$(cu, 136+1, 9) = "eeiiiAAE" ' èëîïËÄÊ
MID$(cu, 147+1, 8) = "ooouuyOU" ' ôöûüÿÖÜ
MID$(cu, 161+1, 5) = "iounN" ' ìòûñÑ
MID$(cu, 168+1, 1) = "?" ' ¿
[ your code goes here ]
ARRAY SORT MyArray$, COLLATE cu
```

An alternative arrangement using the expanded CHR\$ function may look like this:

```
cu = CHR$(0 TO 96, "ABCDEFGHIJKLMNOPQRSTUVWXYZ", _
      123 TO 128, "ueaaaa", _ ' üéâäã
      135, "eeiiiAAE", _ ' èëîïËÄÊ
      145 TO 146, "ooouuyOU", _ ' ôöûüÿÖÜ
```



```

155 TO 160, "iounN", _      ' ìòùñÑ
166 TO 167, "?", _        ' ¿
169 TO 255)

```

For example, the normal ascending ASCII sort order would be described by a string containing ASCII codes 0 through 255 in order:

```

C$ = CHR$(0 TO 255)
ARRAY SORT A$(), COLLATE C$

```

The normal descending ASCII sort order would be described by a collating string containing the reverse of the above:

```

C$ = STRREVERSE$(CHR$(0 TO 255))
ARRAY SORT A$(), COLLATE C$

```

COLLATE *string* can also be used with the ASCEND or DESCEND option. With ASCEND, the sort is performed in the order specified by COLLATE *string*; DESCEND sorts using the reverse of the order specified by COLLATE *string*:

```

ARRAY SORT A$(), COLLATE C$, DESCEND

```

The COLLATE string option is provided as a flexible means with which to specify the sorting order for strings containing international characters or other special symbols. Please keep in mind that the characters with ASCII code above CHR\$(127) may have different meanings in different countries. The examples here assume that the default American OEM ASCII code page is in use.

When sorting a string array, all characters of each element of the array are normally considered when performing comparisons. To limit the comparison to a specific subset of characters, use FROM to specify the *start* position, and TO to specify the *end* position that ARRAY SORT will consider within each array element. For example, you could sort based on the zip code contained in the last 5 characters of a 40-character address string:

```

ARRAY SORT A$()           ' sorts all chars
ARRAY SORT A$(), FROM 36 TO 40 ' sorts 36 - 40 only/p>

```

By using the FROM..TO keywords, it also becomes possible to sort an array of User-Defined Types. In this case, ARRAY SORT can sort the array as if it were an array of [fixed-length strings](#).

Sorting custom arrays:

In most cases, the standard numeric and string sorts should serve your needs very well. However, in the case of more complex data, it is frequently necessary to create multi-key sorts, or other unusual data sequences. Generally speaking, a multi-key sort is used when you wish to order data based upon multiple sections of a string or UDT. For example, you may wish to have customers sequenced by name -- but in the case of duplicate names, order each set of duplicates by ZIP code. With the custom array option, you can sort by any number of keys, in any sequence you may desire.

A custom array may be user-defined types, fixed-length strings, or [ASCIIZ](#) strings. With a custom array sort, you can write your own simple function to tell Classic PowerBASIC the correct sequence for any two array elements. In the following example, the array MyType() is sorted based upon the code you write in the user-written function named MyFunc().

```
ARRAY SORT MyType(), USING MyFunc()
```

As Classic PowerBASIC proceeds through the sort, each time it needs to compare two array elements, it calls your custom function (in this case named MyFunc) to determine the correct sequence of the two elements. The custom function you write must always have exactly two ByRef parameters with precisely the same data type as the sorted array, for ASCIIZ and FIELD strings, they must contain the length. These are the two variables which you must compare to determine the correct sequence. Your custom function must return a long integer to tell the correct sequence. It returns -1 if the first parameter should precede the second parameter. It returns +1 if the second parameter should precede the first. It returns 0 if the parameters are equal. This affords the Classic PowerBASIC programmer the ultimate tool in sorting capabilities. You can have any number of keys. You can sort ascending, descending, or some other special sequence. The conditions are now totally under your control. The following example show how easy it is to create a multi-key sort, even those based upon non-string members of a UDT.

```
Type TheType
  LastName   as String * 40
  FirstName  as String * 20
  BalanceDue as Currency
End Type
[statements]
Dim MyType(100) as TheType
[statements]
Array Sort MyType(), Using MyFunc()
[statements]
Function MyFunc(Param1 as TheType, Param2 as TheType) As Long
  If Param1.LastName < Param2.LastName Then
    Function = -1 : Exit Function
  End If
  If Param1.LastName > Param2.LastName Then
    Function = +1 : Exit Function
  End If
  If Param1.FirstName < Param2.FirstName Then
    Function = -1 : Exit Function
  End If
  If Param1.FirstName > Param2.FirstName Then
    Function = +1 : Exit Function
  End If
  If Param1.BalanceDue < Param2.BalanceDue Then
    Function = +1 : Exit Function
  End If
  If Param1.BalanceDue > Param2.BalanceDue Then
    Function = -1 : Exit Function
  End If
```

Notice that this function first sorts by last name in ascending sequence. If the last names are equal, it then sorts by first name in ascending sequence. If both names are equal, it then sorts by Balance Due in descending sequence so that the accounts with the highest balance appear first. This descending sequence is accomplished by switching the values -1/+1 in the final tests.

The array to be sorted, and the function parameters, must be fixed-length strings, ASCIIZ strings, or user-defined types. Classic PowerBASIC verifies that the size of the data and parameters are identical. However, to allow maximum flexibility, it does not require that the data types be the same. Therefore, for example, it's possible to sort an array of fixed-length strings using a function with UDT parameters as long as the data size is identical. It is the programmer's responsibility to ensure accuracy.

Sorting a multi-dimensional array

When sorting a [multi-dimensional array](#), the array is treated as a single-dimension array containing all of the elements of the multi-dimensional array, in linear column-major order. That is, all elements where all dimensions (except the first), are held at their minimum bounds, will come first in memory. These are immediately followed by the elements where the second dimension is set to its next consecutive index value, etc.

For example, the elements of a two-dimensional array (i.e., DIM A(n,x)) would be stored in consecutive memory locations like this:

(0,0) , , (n,0) , (0,1) , , (n,1) , , (0,x) , , (n,x)

In this case, ARRAY SORT A(0,0) FOR n+1 would sort only elements (0,0)...(n,0), while ARRAY SORT A(0,0) would sort the entire array: elements (0,0)(n,x).

Be very careful when using ARRAY SORT with multi-dimensional arrays so as not to disrupt the organization of the data in the arrays.

Options

The options for ARRAY SORT can be specified in any order, as long as the FOR option, if it is present, directly follows the closing parenthesis of the name of *darray*.

Restrictions

ARRAY SORT cannot be used on arrays *within* [UDT](#) structures or on an array of [Interfaces](#). However, ARRAY SORT can be used with arrays of UDT structures - simply treat them as if they were an array of fixed-length strings.

To use ARRAY SORT on an embedded UDT array, use [DIM..AT](#) to dimension a regular array (of the same type) directly "over the top" of the UDT array, and use ARRAY SORT on that array. For example:

```

TYPE SalesType
    OrderNum AS LONG
    PartNumber(1 TO 20) AS STRING * 20
END TYPE
[statements]
DIM Sales AS SalesType
[statements]
DIM Temp(1 TO 20) AS STRING * 20 AT VARPTR(Sales.Partnumber(1))
ARRAY SORT Temp()
ERASE Temp()

```

See also

[ARRAY ASSIGN](#), [ARRAY DELETE](#), [ARRAY INSERT](#), [ARRAY SCAN](#), [CHR\\$](#), [DIM](#), [LBOUND](#), [REDIM](#), [UBOUND](#), [Array Data Types](#)

Example

```
A&(5) FOR 10, TAGARRAY B$( ), DESCEND
```

Sorts elements 5 through 14 of array *A&* in descending order, tagging along elements 5 through 14 of array *B\$*.

```
ARRAY SORT A#()
```

Sorts all elements of array *A#* in ascending order, using no tag-along array.

```
ARRAY SORT A$(5) FOR 10, FROM 16 TO 25, COLLATE C$, TAGARRAY D()
```

Sorts elements 5 to 14 of array *A\$*, considering only characters 16 to 25 of each element, using the sort order specified by collating string *C\$*, tagging along elements 5 to 14 of array *D*.

```
ARRAY SORT A$()
```

Sorts all elements of array *A\$* in ascending order, considering all characters of each element, using no tag-along array.

```
ARRAY SORT MYTYPE(), USING MYFUNC()
```

Sorts all elements of the UDT array *MYTYPE*, using the custom UDT comparison function *MYFUNC()* to determine the sequence.

ARRAYATTR function

Purpose Return descriptive attributes of a given [array](#).

Syntax `y = ARRAYATTR(Arr(), AttrNum)`

Remarks ARRAYATTR returns various descriptive attributes of an array, depending upon the value of *AttrNum*

AttrNum	Definition			
0	Returns TRUE (-1) if the array is currently dimensioned, FALSE (0) if not.			
1	Returns the data <i>type</i> , as defined in the following table. Note that the numeric equates listed on the right of the table are built into Classic PowerBASIC, but the numeric values they represent are subject to change. Therefore, application code should always use the numeric equates rather than the numeric value, to ensure compatibility with future versions of Classic PowerBASIC. The current data type definitions are:			
	Type	Array type	Keyword	Equate
	0	Byte	BYTE	%VARCLASS_BYT
	1	Word	WORD	%VARCLASS_WRD
	2	Double-word	DWORD	%VARCLASS_DWD
	4	Integer	INTEGER	%VARCLASS_INT
	5	Long-integer	LONG	%VARCLASS_LNG
	8	Quad-integer	QUAD	%VARCLASS_QUD
	10	Single-precision	SINGLE	%VARCLASS_SNG
	11	Double-precision	DOUBLE	%VARCLASS_DBL
	12	Extended-precision	EXT	%VARCLASS_EXT
	13	Currency	CURRENCY	%VARCLASS_CUR
	14	Extended Currency	CURRENCYX	%VARCLASS_CUX
	17	Variant	VARIANT	%VARCLASS_VRNT
	18	Interface	INTERFACE	%VARCLASS_IFAC
	19	GUID	GUID	%VARCLASS_TYPE

	20	UDT or Union	TYPE/UNION	%VARCLASS_TYPE
	21	ASCIIZ string	ASCIIZ/ASCIZ	%VARCLASS_ASC
	22	Fixed-length string	STRING * <i>n</i>	%VARCLASS_FIX
	23	Dynamic string	STRING	%VARCLASS_STR
	24	Field string	FIELD	%VARCLASS_FLD
	2	Returns TRUE (-1) if it is an array of pointers, FALSE (0) if not.		
	3	Returns the number of dimensions of the array. The lower and upper boundaries of each dimension can be retrieved with the LBOUND and UBOUND functions respectively		
	4	Returns the total number of elements in the array. For example, the array DIM A(3,4,5) would comprise 120 elements (4 x 5 x 6 = 120).		
	5	Returns the array element size. For example, an array of Double-precision variables would be 8 bytes. For dynamic strings, the size of the string handle (4 bytes) is returned. For Object Data Types and pointer arrays, ARRAYATTR returns the size of a pointer variable (4 bytes).		

Use should note that a GUID is stored internally as a 16 byte User-defined type. Therefore, ARRAYATTR returns %VARCLASS_TYPE.

See also

[DIM](#), [LBOUND](#), [UBOUND](#), [REDIM](#)

Example

```
DIM z(3,4,5) AS CURRENCYX
DIM x AS LONG, Answer AS STRING
FOR x = 0 TO 5
    Answer = Answer + FORMAT$(x)
    Answer = Answer + $TAB
    Answer = Answer + FORMAT$(ARRAYATTR(z()),x)
    Answer = Answer + $CRLF
NEXT x
' The results are stored in Answer:
```

Result

```
0      -1
1      14
2      0
3      3
4      120
5      8
```

ASC function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the ASCII code of the specified character in a string.
Syntax	<code>y = ASC(<i>string_expression</i> [, <i>position&</i>])</code>
Remarks	<p>ASC returns the ASCII code (0 to 255) of the specified character of <i>string_expression</i>. The first character of the string is position one, followed by position two, etc. If <i>position&</i> is not specified then the first character is presumed. If <i>position&</i> is a negative number, ASC counts from the end of the string back toward the front. For example, if <i>position&</i> is -1 then the last character is returned. If <i>position&</i> is -2 then the second from last character is returned, and so on.</p> <p>The acronym ASCII stands for American Standard Code for Information Interchange, a standardized code in which the numbers between 0 and 127 are used to represent the upper and lowercase letters, the numerals 0 to 9, punctuation marks, and other symbols used in data communication. PCs employ an extended version of ASCII, which contains codes 0 through 255. The "upper" 128 codes (128-255) are not standard ASCII codes, and may differ from one country to another, and from code page to code page.</p> <p>CHR\$ does the reverse of ASC - it produces the one-character string corresponding to its ASCII code argument.</p>
Restrictions	If the string passed is null (zero-length) or the position is zero or greater than the length of the string, the value -1 is returned.
See also	ASC statement , CHR\$
Example	<pre>x\$ = "The ASCII value of A is" + STR\$(ASC("A"))</pre>
Result	The ASCII value of A is 65

ASC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Place an ASCII byte at the specified position in a string.
Syntax	<code>ASC(<i>string</i>, <i>position</i>&) = <i>byte_expression</i></code>
Remarks	<p>ASC places the ASCII code (0 to 255) of the specified character code <i>byte_expression</i> into <i>string</i> at <i>position</i>&. The first character of the string is position one, followed by position two, etc. If <i>position</i>& is negative, ASC counts from the end of the string. For example, if <i>position</i>& is -2, the second to last character in <i>string</i> is modified.</p> <p>See the ASC function for more information on ASCII codes.</p>
Restrictions	The ASC statement cannot be used to extend the length of a string. If <i>string</i> is null (zero-length), or <i>position</i> is zero or greater than the length of <i>string</i> , the operation is ignored.
See also	ASC function , CHR\$, MID\$ statement
Example	<pre>A\$ = "hello There" ASC(A\$,1) = 72 'replace 1st character with an "H"</pre>

Purpose	Identify an assembly-language statement. Classic PowerBASIC's Inline Assembler supports 8086/8088, 80286, 80386, 80486, Pentium, Floating-Point, SIMD and MMX instructions.
Syntax	<code>{! ASM} opcode</code>
Remarks	<p>This statement allows you to place assembly-language code within your Classic PowerBASIC source code. An exclamation mark (!) serves as a shortcut for the ASM keyword.</p> <p>Each group of ASM statements must preserve the following CPU registers if the assembler code causes them to change: EBX, ESI, EDI, ESP, EBP, and all segment registers. See Saving Registers for more information.</p> <p>No other statements may appear on the same line as an ASM statement; however, comments are acceptable.</p> <pre>! DB 1, 2, 3, 4, 5 ! DD &H45001222, &HFFFFFF4327</pre> <p>Any variable referenced in an assembly-language statement must be defined prior to use. For example:</p> <pre>x% = 10 ! MOV AX, x%</pre> <p>You cannot access the target of a pointer with a single ASM statement as you might do in BASIC source code. Instead, you must use the pointer address indirectly. To simulate the BASIC statement INCR @x, you would write:</p> <pre>DIM x AS INTEGER PTR ASM MOV EAX, x ; EAX holds a pointer to ; an Integer ASM INC WORD PTR [EAX] ; Add one to target value</pre> <p>String literals of up to four characters may be used in Inline Assembler code:</p> <pre>! MOV AL, "a" ; move char a into reg AL ! MOV AX, "ab" ; move chars ab into reg AX ! ; "a" into AL, "b" into AH ! MOV EAX, "abcd" ; move chars abcd into reg EAX</pre> <p>Classic PowerBASIC recognizes either an apostrophe (') or a semi-colon (;) to specify a comment after a line of assembler code:</p> <pre>! PUSH EAX ; save the EAX register ! PUSH EBX ' save the EBX register</pre>
Restrictions	Care should be exercised to ensure registers are appropriately preserved when Inline Assembler code is intermixed with BASIC statements. See Saving Registers for more information.

See also [The Inline Assembler](#)

Example

To add the values a&, b&, and c&, you would write:

```
LOCAL a&, b&, c&, z&
! MOV  EAX, a&
! ADD  EAX, b&
! ADD  EAX, c&
! MOV  z&, EAX
```

Notes

The follow lists outline the supported [mnemonics](#), data types, operators, and registers that can be used with the ASM statement.

The ASM statement supports the following mnemonics:

AAA, AAD, AAM, AAS, ADC, ADD, ADDPD, ADDPS, ADDSD, ADDSS, ADDSUBPD, ADDSUBPS, ANDNPD, ANDNPS, ANDPD, ANDPS, AND

BLENDPD, BLENDPS, BLENDVPD, BLENDVPS, BOUND, BSF, BSR, BSWAP, BT, BTC, BTR, BTS

CALL, CBW, CWD, CDQ, CLC, CLD, CLFLUSH, CLI, CMC, CMOVA, CMOVAE, CMOVB, CMOVBE, CMOV, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVNA, CMOVNAE, CMOVNB, CMOVNBE, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, CMOVNZ, CMOV, CMOVPE, CMOVPO, CMOV, CMOVZ, CMP, CMPPD, CMPPS, CMPSB, CMPSD, CMPS, CMPSW, CMPXCHG, CMPXCHG8B, COMISD, COMISS, CPUID, CRC32, CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPI2PS, CVTPS2DQ, CVTPS2PD, CVTPS2PI, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSI2SS, CVTSS2SD, CVTSS2SI, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTPS2PI, CVTTSD2SI, CVTTSS2SI, CWDE

DAA, DAS, DEC, DIV, DIVPD, DIVPS, DIVSD, DIVSS, DPPD, DPPS

EMMS, ENTER, EXTRACTPS

F2XM1, FABS, FADD, FADDP, FBLD, FBSTP, FCHS, FCLEX, FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, FCMOVU, FCOM, FCOMI, FCOMIP, FCOMP, FCOMPP, FCOS, FDECSTP, FDIV, FDIVP, FDIVR, FDIVRP, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, FINIT, FIST, FISTP, FISTTP, FISUB, FISUBR, FLD, FLD1, FLDCW, FLDENV, FLDL2E, FLDL2T, FLDLG2, FLDLN2, FLDPI, FLDZ, FMUL, FMULP, FNCLEX, FNINIT, FNLDCW, FNOP, FNSAVE, FNSTCW, FNSTENV, FNSTSW, FPATAN, FPREM, FPREM1, FPTAN, FRNDINT, FRSTOR, FSAVE, FSCALE, FSIN, FSINCOS, FSQRT, FST, FSTCW, FSTENV, FSTP, FSTSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FUCOM, FUCOMI, FUCOMIP, FUCOMP, FUCOMPP, FWAIT, FXAM, FXCH, FXRSTOR,

```
FXSAVE, FXTRACT, FYL2X, FYL2XP1  
HADDPD, HADDPS, HLT, HSUBPD, HSUBPS  
IDIV, IMUL, IN, INC, INSB, INSD, INSERTPS, INSW, INT,  
INTO, IRET, IRETD  
  
JA, JAE, JB, JBE, JC, JE, JECXZ, JG, JGE, JL, JLE,  
JMP, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL,  
JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ  
  
LAHF, LAR, LDDQU, LDMXCSR, LDS, LEA, LEAVE, LES,  
LFENCE, LFS, LGS, LOCK, LODSB, LODSD, LODSW, LOOP,  
LOOPE, LOOPNE, LOOPNZ, LOOPZ, LSL, LSS  
  
MASKMOVDQU, MASKMOVQ, MAXPD, MAXPS, MAXSD, MAXSS,  
MFENCE, MINPD, MINPS, MINSR, MINSS, MONITOR, MOV,  
MOVAPD, MOVAPS, MOVB, MOVDB, MOVDDUP, MOVQA, MOVQ,  
MOVEDI, MOVHLPS, MOVHPD, MOVHPS, MOVLHS, MOVLPL,  
MOVLPS, MOVMSKPD, MOVMSKPS, MOVNTDQA, MOVNTDQ, MOVNTI,  
MOVNTPD, MOVNTPS, MOVNTQ, MOVQB, MOVQS, MOVSB, MOVSD,  
MOVSHDUP, MOVSLDUP, MOVSS, MOVSW, MOVSR, MOVUPD,  
MOVUPS, MOVZX, MPSADBW, MUL, MULPD, MULPS, MULSD,  
MULSS, MWAIT  
  
NEG, NOP, NOT  
  
OR, ORPD, ORPS, OUT, OUTSB, OUTSD, OUTSW  
  
PABSB, PABSD, PABS, PACKSSDW, PACKSSWB, PACKUSDW,  
PACKUSWB, PADDB, PADDD, PADDQ, PADDSB, PADDSD,  
PADUSB, PADDUSW, PADDW, PALIGNR, PAND, PANDN, PAUSE,  
PAVGB, PAVGW, PBLENDB, BLENDW, PCMPEQB, PCMPEQD,  
PCMPEQW, PCMPEQQ, PCMPSTRI, PCMPSTRM, PCMPISTRI,  
PCMPISTRM, PCMPGTB, PCMPGTD, PCMPGTQ, PCMPGTW, PEXTRB,  
PEXTRD, PEXTRW, PHADDD, PHADDW, PHADDSW, PHMINPOSUW,  
PHSUBD, PHSUBSW, PHSUBW, PINSRB, PINSRD, PINSRW,  
PMADDUBSW, PMADDWD, PMASB, MASD, MASW, MAXUB,  
MAXUD, MAXUW, MINSB, MINSD, MINSW, MINUB,  
MINUD, MINUW, PMOVMSKB, PMOVSRBW, PMOVSRBD,  
PMOVSRBQ, PMOVSRWD, PMOVSRWQ, PMOVSRXDQ, PMOVZXBW,  
PMOVZBXD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ,  
PMULDQ, PMULHRW, PMULHUW, PMULHW, PMULLD, PMULLW,  
PMULUDQ, POP, POPA, POPAD, POPCNT, POPF, POPFD, POR,  
PREFETCHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA,  
PSADB, PSDFB, PSDFI, SHUFHW, SHUFLW, SHUFFW,  
PSGNB, PSGND, PSIGNW, PLLD, PLLDQ, PLLQ, PLWL,  
PSRAD, PSRAW, SRLD, SRLDQ, RL, RLLQ, RLWL,  
PSBB, SUBD, SUBQ, SUBW, SRSB, SRWS, SUBUSB,  
SUBUSW, PSTEST, PUNPKHBW, PUNPKHDQ, PUNPKHQDQ,  
PUNPKHWD, PUNPKLBW, PUNPKLDQ, PUNPKLQDQ,  
PUNPKLWD, PUSH, PSHA, PSHAD, PSHF, PSYED, XOR  
RCCL, RCRR, RCPPS, RCPS, RDPMC, RDTSC, REP, REPE,
```

REPNE, REPNZ, REPZ, RET, RETF, RETN, ROL, ROR,
 ROUNDPD, ROUNDPS, ROUNDSD, ROUNDSS, RSQRTPS, RSQRTSS
 SAHF, SAL, SAR, SBB, SCASB, SCASD, SCASW, SETA, SETAE,
 SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE,
 SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG,
 SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, SETNZ,
 SETO, SETP, SETPE, SETPO, SETS, SETZ, SFENCE, SHL,
 SHLD, SHR, SHRD, SHUFPD, SHUFPS, SQRTPD, SQRTPS,
 SQRTSD, SQRTSS, STC, STD, STI, STMXCSR, STOSB, STOSD,
 STOSW, SUB, SUBPD, SUBPS, SUBSD, SUBSS
 TEST
 UCOMISD, UCOMISS, UNPCKHPD, UNPCKHPS, UNPCKLPD,
 UNPCKLPS
 VERR, VERW
 WAIT
 XADD, XCHG, XGETBV, XLAT, XOR, XORPD, XORPS, XRSTOR,
 XSAVE, XSETBV

The ASM statement supports the following data types and operators:

BYTE
 DB, DD, DW, DWD, DWORD
 FAR
 NEAR
 POINTER, PTR
 QWD, QWORD
 SHORT
 TBY, TBYTE
 WORD, WRD

The ASM statement supports the following registers:

Integer

32-bit	Low 16-bit	High 8-bit	Low 8-bit
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
ESP	SP		

EBP	BP		
-----	----	--	--

Segments

CS, DS, ES, SS, FS, GS

MMX Registers

MM(0), MM(1), MM(2), MM(3), MM(4), MM(5), MM(6), MM(7)
MM0, MM1, MM2, MM3, MM4, MM5, MM6, MM7

Floating Point registers

ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7)

XMM registers

XMM(0), XMM(1), XMM(2), XMM(3), XMM(4), XMM(5),
XMM(6), XMM(7)
XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7

ASM supports these special words

Classic PowerBASIC supports three special reserved words, which are used to specify a return value from a procedure of the same type:

```
FUNCTION      ASM  mov FUNCTION, eax
METHOD       ASM  mov METHOD, 3
PROPERTY     ASM  mov PROPERTY, dx
```

The above examples are the functional equivalent of the comparable BASIC syntax:

```
FUNCTION = 3
METHOD   = x&
PROPERTY = z%
```

The exception is that the assembler syntax allows you to assign a return value directly from an appropriate CPU register. Of course, these special reserved words may only be referenced within a procedure of the same type (FUNCTION may only be used in a user-defined function, etc.)

See Also [#ALIGN](#)

Purpose Return the arctangent of an argument.

Syntax `y = ATN(numeric_expression)`

Remarks ATN returns the arctangent (Inverse Tangent) of *numeric_expression*; that is, the angle whose tangent is *numeric_expression*.

The result, as with all operations involving *angles* in Classic PowerBASIC, is in *radians* rather than *degrees*. Although it is common to specify angles in degrees, the radian is a more convenient measurement for mathematical operations. One radian is defined as the angle at the center of a circle that subtends an arc equal in length to one radius. Since for all circles, using the constant π :

$$\text{Circumference} / \text{radius} = 2 * \pi$$

the length of the circumference of a circle is equal to $2 * \pi * \text{radius}$, and the angle of a full circle (360 degrees) is equal to $2 * \pi$ radians.

To convert radians to degrees, just multiply the radian value by $180/\pi$, or 57.29577951308232###. For example, the arctangent of 0.23456 can be converted this way:

```
t = ATN(.23456!)           't = 0.230395 (radians)
t = 57.29577951308232## * ATN(.23456!) 't= 13.200 (degrees)
```

To convert degrees to radians, multiply by 0.0174532925199433###. For example:

$$14 \text{ degrees} = (0.0174532925199433## * 14) = 0.2443460952792062 \text{ radians}$$

Rather than memorizing the radians/degrees conversion factors, calculate them for yourself by remembering this relationship: 2π radians equals a full circle (360 degrees), so 1π radian is $180 / \pi$ degrees. Conversely, 1 degree equals $\pi / 180$ radians.

π is a transcendental constant, meaning that it has an infinite number of decimal places. To 15-place accuracy, adequate for most applications, $\pi = 3.141592653589793###$. This value can be closely approximated with the expression:

$$\text{pi}\#\# = 4 * \text{ATN}(1)$$

Degrees-to-radians and radians-to-degrees conversions are good applications for user-defined functions.

The ATN function always returns an [Extended-precision](#) result.

The Tangent (TAN) of a value can be easily calculated with the [TAN](#) function.

The Hyperbolic Tangent (TANH) can be calculated:

$$\text{TanH} = (\text{EXP}(2 * \text{Value}) - 1) / (\text{EXP}(2 * \text{Value}) + 1)$$

The Inverse Hyperbolic Tangent (ARCTANH) of a value can be calculated:

```
ArcTanH = LOG((1 + Value) / (1 - Value)) / 2
```

```
' Useful Macro functions
```

```
MACRO Pi = 3.141592653589793##
```

```
MACRO DegreesToRadians(dpDegrees) = (dpDegrees*0.0174532925199433##)
```

```
MACRO RadiansToDegrees(dpRadians) = (dpRadians*57.29577951308232##)
```

See also

[COS](#), [SIN](#), [TAN](#)

BEEP statement

[Top](#) [Previous](#) [Next](#)

Purpose	Sound a tone through the computer's speaker.
Syntax	<code>BEEP</code>
Remarks	BEEP plays the default Windows waveform sound, typically a ¼ second tone, through either the built-in speaker; or a sound card if installed (in which case the Windows "Default Beep" sound is played). The Default Beep can be configured in the Sounds section of Control Panel.
Restrictions	The physical aspects of the built-in speaker may have an effect on the quality and level of the resultant sound.

Purpose	Create a BGR color value from 3 primary color values or from an RGB value
Syntax	<pre>result& = BGR(red&, green&, blue&) result& = BGR(rgbexpr&)</pre>
Remarks	<p>An RGB value is a long integer value in the range of 0 to &H00FFFFFF. It is used to specify a very precise color to various Classic PowerBASIC functions and Windows API functions. The lowest three bytes of the value each specify the intensity of a primary color which combine to form the resultant color. Byte 1 (lowest) represents the red component, byte 2 the green, and byte 3 the blue. They can each take on a value in the range of 0 to 255. Byte 4 (highest) is always 0.</p> <p>Some Windows API functions, such as those which reference Device Independent Bitmaps (DIB), require that the colors be specified in the reverse sequence (Blue-Green-Red instead of Red-Green-Blue). In order to maximize performance and execution speed, Classic PowerBASIC statements and functions which reference these structures also use the BGR format. These include GRAPHIC GET BITS and GRAPHIC SET BITS. When used with 3 parameters, the BGR() function creates a BGR value from the three component values.</p> <p>When used with one parameter, this function translates an RGB value to its BGR equivalent by swapping the first byte with the third byte, and returning the result.</p> <p>For example, the RGB value of blue is &HFF0000. BGR() translates it to &H0000FF. Calling RGB() with that value converts it back to &HFF0000.</p>
See also	Built In RGB Color Equates , RGB

Purpose	Return a string that is the binary (base 2) representation of its argument.
Syntax	<code>s\$ = BIN\$(numeric_expression [, digits])</code>
Remarks	<p><i>numeric_expression</i> must be in the range -2,147,483,648 to +2,147,483,647. Any fractional part of <i>numeric_expression</i> is rounded before the string is created. If <i>digits</i> is specified, the resulting string will be of length <i>digits</i>. If the resulting string is longer than <i>digits</i>, the result will be truncated from the left as needed. If the resulting string is shorter than <i>digits</i>, leading zeros will be added to the left to make the final string <i>digits</i> long.</p>

Thus, for 12959:

```
0011 0010 1001 1111      ' +12959
1100 1101 0110 0000      ' reverse all bits
      +1                  ' add 1
1100 1101 0110 0001      ' -12959
```

Due to the use of two's-complement binary form to represent numbers, bit patterns with the most significant bit set to 1 may represent either of two possible values, depending on whether the value is regarded as signed or unsigned. See [Numeric Literals](#) for more information on explicitly casting numeric literal values to specific data types and distinguishing between signed and unsigned values. Also see the [HEX\\$](#) function.

For example, the value with 32 binary ones "11111111111111111111111111111111" may be either -1 (signed Long-integer) or 4,294,967,295 (unsigned Double-word).

Note that the binary representation for a negative number is different for numbers that have different bit lengths. For a regular [16-bit Integer](#), -1 is represented as 16 binary ones ("1111111111111111"). This corresponds to the unsigned value 65,535. If you store this value in a [32-bit Long-integer](#), you would end up with either -1 or 65,535 depending on how you store it.

Binary strings can be converted to numeric values with the [VAL](#) function by prefixing the binary string with "&B". If the string has a leading zero, the result is always unsigned. For example:

```
a$ = BIN$(65535)          ' = "1111111111111111"
x& = VAL("&B" + a$)      ' Signed result (-1)
y& = VAL("&B0" + a$)     ' Unsigned result (65535)
```

See also [FORMAT\\$](#), [HEX\\$](#), [OCT\\$](#), [STR\\$](#), [USING\\$](#), [VAL](#)

Purpose	Set or reset a bit in an integer class variable (or implied bit-array) based upon the result of an expression.
Syntax	<code>BIT CALC <i>intvar</i>, <i>bitnumber</i>, <i>calcexpr</i></code>
Remarks	BIT CALC performs like a combination of the BIT SET and BIT RESET statements, offering the choice between set (1) and reset (0) according to the result of a numeric expression.
<i>intvar</i>	An integer class variable (Byte , Word , Double-word , Integer , Long-integer or Quad-integer), or a variable forming the base of an implied bit-array.
<i>bitnumber</i>	An integer class expression or numeric literal that specifies the bit number to adjust. Bit numbers start from zero (0), and extend to the size of the target variable or bit-array. For example, a 16-bit integer variable uses the range 0 to 15. An implied bit-array comprised of a Long-integer array with 100 elements (4 bytes * 100 = 400 bytes = 3200 bits) covers the bit range 0 to 3199.
<i>calcexpr</i>	The value derived from bit zero of <i>calcexpr</i> determines the set or reset action. If bit zero contains a zero (0), the bit in <i>intvar</i> is reset; if bit zero in <i>calcexpr</i> contains a one (1), the bit in <i>intvar</i> is set. This action can be more easily remembered if we consider Classic PowerBASIC performs an implied bitwise AND operation (<i>calcexpr</i> AND 1) to derive the set or reset action.

Care must be exercised to ensure that the bit index number (*bitnumber*) does not exceed the number of bits that can be validly accessed. For example, reading the 17th bit of a 16-bit scalar variable may trigger a General Protection Fault (GPF). Similarly, adjusting the 4097th bit of a bit-array derived from a 128-element [DWORD](#) array may cause similar problems. *bitnumber* is always zero-based, so the 129th bit of an implied bit-array is referenced in the BIT statement with *bitnumber* equal to 128. For example: `x& = BIT(A?(1), 128)`.

See also [BIT function](#), [BIT statement](#), [BITS functions](#)

Example

```
DIM dwStatus1 AS DWORD
DIM dwStatus2 AS DWORD
DIM iBit      AS INTEGER
DIM sResult1  AS STRING
DIM sResult2  AS STRING

FOR iBit = 0 TO 31
  BIT CALC dwStatus1, iBit, RND(0,1)
  BIT CALC dwStatus2, iBit, iBit MOD 3
```

```
NEXT iBit  
sResult1 = BIN$(dwStatus1,32)  
sResult2 = BIN$(dwStatus2,32)
```

Result

```
sResult1 = "01001101001110101110111010010101"  
sResult2 = "10010010010010010010010010010010"
```

Purpose	Return the value of a particular bit in an variable (or in an implied bit-array)
Syntax	<i>flag</i> = BIT(<i>intvar</i> , <i>bitnumber</i>)
Remarks	The BIT function is used to determine the value of one particular bit in an integer-class variable or implied bit-array.
<i>intvar</i>	The parameter <i>intvar</i> must be a variable, not an expression. The BIT function returns either 0 or 1 to indicate the value of the specified bit.
<i>bitnumber</i>	<p>The bit in question. The allowable range for the parameter is the same as that of a Long-integer. This makes it possible to have implicit bit-arrays of more than 2 billion bits in size. For such arrays, bits 0 to 15 are in the first word starting at <i>intvar</i>, bits 16-31 are in the next word, and so forth.</p> <p>Implied bit-arrays are considered to start at the memory position of the variable <i>intvar</i>. For example, if <i>intvar</i> is itself an array variable, it is possible to access bits in any of the following elements of the array. See the array examples below.</p> <p>Care must be exercised to ensure that the bit index number (<i>bitnumber</i>) does not exceed the number of bits that can be validly accessed. For example, reading the 17th bit of a 16-bit scalar variable may trigger a General Protection Fault (GPF). Similarly, adjusting the 4097th bit of a bit-array derived from a 128-element DWORD array may cause similar problems.</p> <p><i>bitnumber</i> is always zero-based, so the 129th bit of an implied bit-array is referenced in the BIT statement with <i>bitnumber</i> equal to 128. For example: <code>x& = BIT(A?(1), 128)</code>.</p>

See also [BIT CALC statement](#), [BIT statement](#), [BITS functions](#)

Example

```
x% = 7
y% = BIT(x%, 2)
[statements]
DIM z%(1:2000000) ' 32 million element bit-array
y% = BIT(z%(1),16) ' bit 0 of 2nd word of z%()
y% = BIT(z%(2000000),15) ' MSB of last element
y% = BIT(z%(1), 31999999&) ' MSB of last element
```

Purpose	Manipulate individual bits of an variable (or in an implied bit-array), for storing values such as TRUE/FALSE (flag) settings quickly and efficiently.
Syntax	<code>BIT {SET RESET TOGGLE} <i>intvar</i>, <i>bitnumber</i></code>
Remarks	<p><i>intvar</i> must be one of the integer-class variable types: Byte, Word, Integer, Double-word, Long-integer, or Quad-integer.</p> <p>The allowable range for the parameter <i>bitnumber</i> is the same as that of a Long-integer, making it possible to have implicit bit-arrays of more than 2 billion bits in size. Bits 0 to 15 are in the first word starting at <i>intvar</i>, bits 16-31 are in the next word, and so forth.</p> <p>Implied bit-arrays are considered to start at the memory position of the variable <i>intvar</i>. For example, if <i>intvar</i> is itself an array variable, it is possible to access bits in any of the following elements of the array. See the array examples below.</p>

Care must be exercised to ensure that the bit index number (*bitnumber*) does not exceed the number of bits that can be validly accessed. For example, adjusting the 17th bit of a 16-bit scalar variable may cause a subtle memory corruption problem, and/or may trigger a General Protection Fault (GPF). Similarly, adjusting the 4097th bit of a bit-array derived from a 128-element [DWORD](#) array may cause similar problems. *bitnumber* is always zero-based, so the 129th bit of an implied bit-array is referenced in the BIT statement with *bitnumber* equal to 128. For example:

BIT SET A?(1), 128.

SET	Sets the indicated bit to one.
RESET	Sets the indicated bit to zero.
TOGGLE	Toggles the indicated bit: one becomes zero; zero becomes one.

See also [BIT CALC statement](#), [BIT function](#), [BITS functions](#)

Example

```
x% = 7
BIT SET x%, 2           ' Sets the 3rd bit (bit 2) to 1
BIT RESET x%, 10        ' Sets the 11th bit (bit 10) to 0
BIT TOGGLE x%, 5         ' Toggle bit 5
[statements]
DIM z%(1 TO 2000)       ' 32000 element bit-array
BIT SET z%(1), 37        ' Sets bit 5 of 3rd word to a 1
BIT TOGGLE z%(1), 0      ' Toggle lowest bit in 1st word
BIT RESET z%(2000), 15   ' Clear the MSB of integer array element
```

```
                                ' 2000 (bit 31999 of the implied bit array,  
                                ' numbered 0 to 31999)  
BIT RESET z%(1), 31999      ' Clear the MSB of element 2000  
                                ' (this is equivalent to the previous line)
```

Purpose	Converts an integer class value into another data type, based upon the bit pattern of the value. This is particularly helpful in converting between signed and unsigned representations.	
Syntax	<i>resultvar</i> = BITS(<i>datatype</i> , <i>expression</i>)	
<i>datatype</i>	The parameter <i>datatype</i> may be BYTE , WORD , DWORD , INTEGER , or LONG to specify the new data type which should be returned by the function.	
<i>expression</i>	An integer class variable, expression, or numeric literal , which designates the original value to be converted.	
Remarks	<p>Since the integer value -1 and word value 65535 have the identical bit pattern of 111111111111111, BITS(WORD,-1) would return the unsigned word value of 65535. Of course, BITS(INTEGER,65535) would then return the integer value -1. Other values and data types would follow the same pattern and rules.</p> <p>This newer form of BITS condenses the functionality of the older forms (BITS%, BITS&, BITS?, BITS?? and BITS???) into a single function. In particular, this provides for the addition of new data types in future version of Classic PowerBASIC, particularly those which may not have an associated type-specifier character.</p>	
Restrictions	The superseded syntax will continue to be supported for a limited period of time, although existing code should be converted to the new syntax as soon as possible.	
	<u>Superseded syntax</u>	<u>Replacement syntax</u>
	ByteVar = BITS?(expr)	ByteVar = BITS(BYTE, expr)
	WordVar = BITS??(expr)	WordVar = BITS(WORD, expr)
	DWordVar = BITS???(expr)	DWordVar = BITS(DWORD, expr)
	IntVar = BITS%(expr)	IntVar = BITS(INT, expr)
	Longvar = BITS&(expr)	Longvar = BITS(LONG, expr)
See also	BIT CALC statement , BIT function , BIT statement , BITS functions , BITSE	

Purpose Return the least significant [Byte](#), [Word](#), [Integer](#), [Double-word](#), or [Long-integer](#) of an expression. Allows easy conversion between signed and unsigned expressions.

The BITS functions has been superceded by the [BITS function](#), although the BITS functions remain supported for a limited period. Existing code should be converted to the new syntax as soon as possible.

Syntax

```
bytevar  = BITS?(expression)  
wordvar  = BITS??(expression)  
dwordvar = BITS???(expression)  
intvar   = BITS%(expression)  
longvar  = BITS&(expression)
```

Remarks The various BITS functions return the least-significant bits of *expression*, as shown in the following table:

Function	Accepts	Returns
BITS?	Long-integer	Byte (least-significant 8 bits, unsigned)
BITS??	Long-integer	Word (least-significant 16 bits, unsigned)
BITS???	Quad-integer	Double-word (least-significant 32 bits, unsigned)
BITS%	Long-integer	Integer (least-significant 16 bits, signed)
BITS&	Quad-integer	Long-integer (least-significant 32 bits, signed)

See also [BIT CALC statement](#), [BIT function](#), [BIT statement](#), [BITS function](#)

Purpose Compare integral class values for equivalent bits regardless of sign.

Syntax `x& = BITSE(nexp, nexp, bitsize)`

Remarks This function allows you to compare two integer class values for equivalent bit patterns, regardless of whether they are signed or unsigned values. The two numeric expressions (*nexp*) are the integer class values to be compared, The *bitsize* parameter specifies the number of bits to be compared, 8, 16, or 32.

For example, the integer value -1 and the word value 65535 both have the identical bit pattern: 1111111111111111. The difference is simply the way the bits are interpreted by a program.

```
x& = BITSE( -1, 65535, 16)
```

The above example would cause the lowest 16 bits of the expressions to be compared. Since they are equal, the value [TRUE](#) (-1) is returned.

See also [BITS](#)

Purpose Concatenate multiple [strings](#) with high efficiency

Syntax `x$ = BUILD$(a$,b$,c$,d$. . .)`

Remarks In some cases, string concatenation using the classic [string operators](#) can be a slow process. This is particularly true when there are many operands using longer strings. The BUILD\$ function passes all the typical bottlenecks to create a new string at the greatest possible speed. The following 2 lines are functionally identical, but the BUILD\$ version will execute substantially faster.

```
x$ = a$ + "bb" + c$ + y$(7) + y$(i&) + z$  
x$ = BUILD$(a$, "bb", c$, y$(7), y$(i&), z$)
```

In order to extract the utmost efficiency, BUILD\$() was designed to work with a very narrow definition. The component parameters must be [dynamic string](#) variables, either scalar or [array](#), [string literals](#), or [string equates](#). There is virtually no limit as to the number of parameters. Generally speaking, the greater the number of parameters, the greater the increase in execution speed.

See also [LET](#), [CHR\\$](#), [CSET](#), [CSET\\$](#), [JOIN\\$](#), [LSET](#), [LSET\\$](#), [REPEAT\\$](#), [RSET](#), [RSET\\$](#), [STRING\\$](#), [STRINSERT\\$](#)

Purpose	Invoke a procedure (Sub , Function , Method , or Property).
Syntax	<code>CALL ProcName [[([arguments])] [TO result_var]</code>
Remarks	The CALL statement has the following parts:
<i>ProcName</i>	The name of a Sub, Function, Method, or Property defined elsewhere in the program.
<i>arguments</i>	<p>An optional, comma-delimited list of variables, expressions, and constants to be passed to the procedure as parameters, for up to 32 parameters. If the CALL keyword is used, the <i>arguments</i> must be enclosed in parentheses.</p> <p>As long as <i>ProcName</i> is a Classic PowerBASIC procedure, or is an external procedure declared with the DECLARE statement, you can omit the CALL keyword. If you do so, you may omit the parentheses surrounding arguments. For example, the following lines are equivalent:</p> <pre>CALL MyProc(parm1, parm2) MyProc(parm1, Parm2) MyProc parm1, parm2</pre> <p>However, if the first parameter argument is enclosed in parentheses for any reason, the entire parameter list must be enclosed in parentheses. For example:</p> <pre>MyProc (3+z, b) ' Valid syntax MyProc ((3+z), b) ' Valid syntax MyProc (3+z), b ' Invalid syntax</pre> <p>This updated syntax now permits macros to be called using the SUB-style convention if/when the macros expand directly to Function calls. For example:</p> <pre>MACRO sm(Msg) = SendMessage(a, Msg, b, c)</pre> <p>can be called like this (when the return value is not required):</p> <pre>sm(x)</pre> <p>In all cases, the number and type of parameters passed must agree with the <i>arguments</i> in procedure definition.</p>

Passing parameters

In a procedure definition, every parameter is described by type and according to the way it is passed to the procedure. The type may be any normal variable type, such as integer, floating point, string, [User-Defined Type](#), etc. The passing method describes how the value is presented to the procedure: by reference (BYREF), by value (BYVAL), or by reference to a copy (BYCOPY).

BYREF	When a parameter is passed by reference, it consists of a 4-byte address
-------	--

of the data. In this case, the original data can be modified by the procedure.

BYVAL

When a parameter is passed by value, it consists of an actual copy of the data. Since the parameter is a copy, the original data cannot be modified by the procedure.

When you pass parameters from the calling code with an explicit BYVAL, you effectively switch off the compilers type-checking for that parameter. This can be useful in cases where the called code is expecting a BYREF parameter, and you wish to pass an address of another data type that would trigger a [compile-time error](#) without the BYVAL method. For example:

```
SUB TheSub(x AS ASCIIZ) ' Address of x expected
    [statements]
END SUB
[statements]
DIM a$
a$ = "Dynamic string data"
CALL TheSub(BYVAL STRPTR(a$)) ' Pass data address
```

BYCOPY

A parameter passed by copy is a special case; somewhat of a hybrid of the other two methods. When a procedure expects a parameter to be passed by reference, it expects to see a [pointer](#) to the data. In some cases, such as when the parameter is a calculated expression, it is not precisely possible to pass a pointer, since an expression result is a temporary value that does not exist in a permanent memory location. On the other hand, if you wish to ensure that the original data is not modified by the procedure, you can place a BYCOPY override in the *arguments* list. In both cases, a copy of the data is stored in a temporary memory location, and the parameter consists of a 4-byte address of this temporary location. Another way to force BYCOPY is to enclose a [variable](#) name in parentheses, so it will appear to the compiler as an expression, rather than just a single variable.

Unless declared otherwise, parameters default to BYREF passing method. [Expressions and constants](#) are always passed BYCOPY. [Fixed length strings](#), [User-Defined Types](#), and full [arrays](#) are always passed BYREF.

```
CALL MySub (i&)           ' i& is passed by reference
CALL MySub (BYREF i&)     ' i& is passed by reference
CALL MySub (BYCOPY i&)    ' i& is passed by copy
CALL MySub ((i&))         ' i& is passed by copy
```

Entire arrays are specified by using an empty set of parentheses after the array, while individual array elements are specified by [subscript](#) index number. For example:

```
CALL SumArray(a())        ' pass entire array 'a'
CALL SumArray(a(3))       ' pass element 3 of array 'a'
```

The CALL statement can be used to invoke functions, subs, methods, or

properties. In this case, the return value of the function is simply discarded, unless the TO keyword is used to specify a return variable.

If a procedure expects a parameter by reference, it is possible to substitute a pointer by value, for the identical result. This is particularly useful with Fixed-length strings and Types:

```
DECLARE SUB a(z%)
DIM MyInt AS INTEGER, x AS INTEGER PTR
x = VARPTR(MyInt)
CALL a(MyInt)
' or
CALL a(BYVAL x)
' or
CALL a(BYVAL VARPTR(MyInt))
```

Of course, if the procedure is expecting a parameter by value, you may not pass the pointer, but rather the pointer target (i.e., CALL a(@x)).

Classic PowerBASIC compilers have a limit of 32 parameters per SUB, FUNCTION, METHOD, and PROPERTY. To pass more than 32 parameters, construct a User-Defined Type (UDT) and pass (the address of) the UDT by reference (BYREF) instead.

Fixed-length strings, [ASCIIZ](#) strings, and User-Defined Types/Unions may also be passed as BYVAL or OPTIONAL parameters, now. Try to avoid passing large items BYVAL, as it's terribly inefficient, and there is a maximum size limit of 64 Kb for a given parameter list. Arrays cannot be passed BYVAL.

When a procedure definition specifies either a BYREF parameter or a pointer variable parameter, the calling code may freely pass a BYVAL DWORD or a Pointer instead. While the use of the explicit BYVAL override in the calling code is optional, it is recommended for clarity. It is necessary to explicitly declare all pointer parameters as BYVAL (i.e., BYVAL X AS BYTE PTR). Failure to do so will generate a compile-time [Error 549](#) ("BYVAL required with pointers").

A procedure may also be imported and exported within the same module. That is, a function in the module may be stated as EXPORT, while a DECLARE in the same module specifies it as an imported function by the option LIB "XXX.DLL", provided that XXX.DLL is the name of the module. This may be particularly valuable when you wish to build an [#INCLUDE](#) file with all of the DECLARE statements for a project.

For information on using OPTIONAL parameters, please see [DECLARE](#), [FUNCTION](#), [METHOD](#), [PROPERTY](#) and [SUB](#) topics.

NOTHING

The reserved word NOTHING can be used to replace any OBJECT variable parameter. In this case, the compiler passes a null object (or a pointer to a null object if BYREF) in place of a typical parameter. While this simplifies some programming issues, the technique must be used with

caution. If the target METHOD or FUNCTION is not expecting a null parameter, it could cause a fatal error condition.

TO *result_var* This offers an optional way to assign a function return value to *result_var*. For example, the following code assigns the return value to x% in two different ways:

```
x% = MyFunCall  
CALL MyFunCall TO x%
```

Restrictions A thread Function may not be directly called or executed, except by a [THREAD CREATE](#) statement.

See also [CALL DWORD](#), [DECLARE](#), [FUNCTION/END FUNCTION](#), [METHOD](#), [PROPERTY](#), [SUB/END SUB](#), [THREAD CREATE](#)

Purpose	Invoke a procedure (Sub or Function) indirectly.
Syntax	<pre>CALL DWORD <i>dwpointer</i> [{BDECL CDECL SDECL} ()] CALL DWORD <i>dwpointer</i> USING <i>abc</i>(<i>arguments</i>) [TO <i>result_var</i>]</pre>
Remarks	<p>CALL DWORD is an essential ingredient for implementing run-time (explicit) dynamic linking of DLLs, rather than the more common load-time (implicit) dynamic linking. This provides a way of constructing calls to APIs and DLLs that may not be present in all versions of Windows. This technique ensures that an application can start up successfully, even if Windows cannot resolve the API or DLL function. For more information on explicit linking, please review the <i>Restrictions</i> and <i>Example code</i> sections below.</p> <p>The CALL DWORD statement has the following parts:</p>
<i>dwpointer</i>	A Double-word , Long-integer , or pointer variable that contains the address of the entry point of a procedure (Sub or Function).
BDECL	<p>Specifies that the declared procedure uses the BASIC/Pascal calling convention. When a procedure calls a BDECL procedure, it passes its parameters on the stack left to right.</p> <p>It is the responsibility of the called procedure to clean up the stack before returning to the calling procedure. Therefore, all Classic PowerBASIC Subs and Functions that specify the BDECL convention automatically clean up the stack before execution returns to the calling code.</p>
CDECL	<p>Specifies that the declared procedure uses the C calling convention. When a procedure calls a CDECL procedure, it passes its parameters on the stack right to left.</p> <p>In addition, the calling procedure cleans up the stack upon return from the called procedure. When Classic PowerBASIC code calls Subs and Functions using the CDECL convention, the stack is cleaned automatically after execution returns from the called code.</p> <p>In the event the called procedure is imported or exported, Classic PowerBASIC will automatically create a lowercase ALIAS, prefixed with an underscore. Any periods in the name are replaced with underscores at the same time.</p> <p>The following shows how the ALIAS name would be created by using a DECLARE statement as the example format:</p> <pre>DECLARE SUB C_Function CDECL () DECLARE SUB C_Function CDECL ALIAS "_c_function" ()</pre> <p>When declaring a CDECL SUB or FUNCTION, you can specify trailing</p>

parameters as [OPTIONAL](#), using a set of brackets [..]:

```
DECLARE SUB KerPlunk CDECL (x%, y% [, z%])
```

Note that the comma separating the y% parameter from the optional z% parameter is inside the brackets. The following calling sequences would then be valid:

```
CALL KerPlunk (x%, y%)
CALL KerPlunk (x%, y%, z%)
```

Optional parameters must be the last parameters designated in the list. The following is not valid:

```
DECLARE SUB KerPlunk CDECL ([x%,] y%, z%)
```

Because the SUB or FUNCTION being called does not know how many parameters are being passed at the time it is called, you should pass the number of parameters as one of the required parameters in the list.

SDECL

This is the default if neither BDECL nor CDECL are specified. SDECL (and its synonym STDCALL) specifies that the declared procedure uses the "Standard Calling Convention" as defined by [Microsoft](#). When calling an SDECL procedure, parameters are passed on the stack right to left.

Classic PowerBASIC Subs and Functions that use the SDECL/STDCALL convention automatically clean up the stack before execution returns to the calling code.

USING

This option is used to define a model procedure declaration which matches all of the calling conventions desired to be used to invoke the target Sub/Function. For example, the following two calls to the function *MySubCall* are equivalent:

```
DECLARE SUB MySubCall (Param1%, Param2%)
DIM PtrMySubCall AS DWORD
PtrMySubCall = CODEPTR (MySubCall)
[statements]
CALL MySubCall (x1%, x2%)
CALL DWORD PtrMySubCall USING MySubCall (x1%, x2%)
```

arguments

An optional, comma-delimited list of [variables](#), expressions, and [constants](#) to be passed to the procedure as parameters. In the CALL DWORD context, enclosing parentheses are required. The number and type of parameters passed must agree with the *arguments* of the procedure named by the USING clause. See [CALL](#) for more information on parameter passing methods.

For information on using optional parameters, please see [DECLARE](#), [FUNCTION](#) and [SUB](#) topics.

TO result_var

When calling a Function (which returns a value), the TO keyword offers a way to assign the function return value to *result_var*. TO cannot be used without a USING clause also being used (and hence a DECLARE statement too), since Classic PowerBASIC must be able to resolve the

return data type.

If you call a Function through CALL DWORD with the USING option (but without a TO clause), the compiler can automatically discard the function result for you. However, this is safe only if the return value is of integer-class ([Byte](#), [Word](#), [Integer](#), [Long](#), and [Dword](#)). If the return value is a string or a floating-point value, you must employ the USING clause (in conjunction with an appropriate DECLARE statement) to avoid memory leaks.

Restrictions A thread Function may not be directly called or executed, except by a [THREAD CREATE](#) statement.

DECLARE statements and the USING option

If you employ the USING option to specify a DECLARE statement that describes the parameter list, make sure the DECLARE statement does not include a LIB clause. Otherwise, the application will become linked to the DLL, just as if you had called the routine directly, rather than using CALL DWORD.

To solve the problem of accidental implicit-linking, remove the LIB and the ALIAS clauses from the DECLARE statement, and if necessary, rename the function to ensure it has a unique name (i.e., to avoid conflicts with DECLARE statements in [WIN32API.INC](#), etc). For example, consider the following "standard" DECLARE statement:

```
DECLARE FUNCTION GetDiskFreeSpaceEx LIB "KERNEL32.DLL" ALIAS _
    "GetDiskFreeSpaceExA" (lpPath AS ASCIIZ, lpFreeToCaller AS _
        QUAD, lpTotalBytes AS QUAD, lpTotalFreeBytes AS QUAD) AS LONG
```

This particular API is not available on early versions of Windows 95; therefore, it is standard practice to use explicit (run-time) linking to call this function, if it is available. This is achieved through the *LoadLibrary* API (to determine if the DLL is available), followed by a call to *GetProcAddress* to obtain a code pointer to the actual API function (if the DLL includes the function).

Once a valid pointer has been obtained, the API can be directly called with the CALL DWORD statement, using a modified DECLARE to form a template for the USING option. The modified DECLARE may look like this:

```
DECLARE FUNCTION MyDiskFreeSpaceEx(lpPath AS ASCIIZ, _
    lpFreeToCaller AS QUAD, lpTotalBytes AS QUAD, lpTotalFreeBytes _
    AS QUAD) AS LONG
```

Please review the Example code section below to see this how this modified DECLARE statement is used with the CALL DWORD statement.

See also [CALL](#), [CODEPTR](#), [DECLARE](#), [FUNCTION/END FUNCTION](#), [SUB/END SUB](#), [THREAD CREATE](#)

Example

```
' Uses the DECLARE statement shown above
DIM hLib AS DWORD
```

```
DIM pAddr    AS DWORD
DIM szDrv    AS ASCIIZ * %MAX_PATH
DIM lResult  AS LONG
szDrv = "C:\"
hLib = LoadLibrary("KERNEL32.DLL")
pAddr = GetProcAddress(hLib, "GetDiskFreeSpaceExA")
IF pAddr <> 0 THEN _
    CALL DWORD pAddr USING MyDiskFreeSpaceEx(szDrv, FreeToUserQuota&&,
    SizeOfDisk&&, TotalFree&&) TO lResult
FreeLibrary hLib
```

Purpose	Capture a complete representation of the stack frames in the call stack.
Syntax	<code>CALLSTK <i>diskfilename</i>\$</code>
Remarks	<p>Classic PowerBASIC creates a <i>stack frame</i> for each call to a Sub, Function, Method, or Property, and records each nested call in a <i>call stack</i>. The stack frame holds the parameters being passed to the routine, and providing space for local variable storage, etc. Since procedures can call other procedures to an almost limitless depth, there may be a substantial number of stack frames present at any given moment.</p> <p>The CALLSTK statement can help provide answers to the age-old "how did I get here?" question. When combined with other debugging statements such as CALLSTK\$, CALLSTKCOUNT, and TRACE, the programmer has a set of tools that can significantly reduce the amount of effort required to debug an application.</p> <p>Executing a CALLSTK statement captures a representation of all of the stack frames that exist above the one that includes the CALLSTK statement. When the CALLSTK statement is executed, a standard sequential file (of the specified file name in <i>diskfilename</i>\$) is created. The resulting disk file contains a list of every call to a procedure, and their associated parameter values, which are currently defined on the call stack. <i>diskfilename</i>\$ must be a legal file spec, may be a Long File Name (LFN), and may include a path. If the file cannot be created for any reason, the operation will be ignored and no run-time error will be generated. If present, CALLSTK overwrites the existing file.</p> <p>If PBMAIN calls the SUB aaa(x&) which then calls the SUB bbb(y&), the CALLSTK from within bbb(y&) might look like this:</p> <pre>PBMAIN () aaa (77) bbb (-1)</pre> <p>Later, if bbb(y&) exited, then aaa(x&) called ccc(z&), the updated CALLSTK from within ccc(z&) might then appear as:</p> <pre>PBMAIN () aaa (77) ccc (33)</pre> <p>Numeric parameters are displayed in decimal, while pointer and array parameters display a decimal representation of the offset of the target value.</p>
Restrictions	CALLSTK can be invaluable during debugging, but it generates substantial additional code that should be avoided in a final release version of an application. If the source code contains #TOOLS OFF , all CALLSTK

statements which remain in the program are ignored.

The CALLSTK statement is "thread-aware", displaying only stack frame details from the thread in which it was executed.

See also

[#TOOLS](#), [CALLSTK\\$](#), [CALLSTKCOUNT](#), [FUNCNAME\\$](#), [PROFILE](#), [TRACE](#)

Example

```
FUNCTION PBMAIN
  CALL Sb1(100)
END FUNCTION

SUB Sb1(x AS LONG)
  CALL Sb2(x + 1)
END SUB

SUB Sb2(y AS LONG)
  CALLSTK "Stack frame test.txt"
END SUB

PBMAIN()
SB1(100)
SB2(101)
```

Result

CALLSTK\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the details of a specific stack frame from the call stack.
Syntax	<i>sfname\$</i> = CALLSTK\$(<i>n</i>)
Remarks	<p>CALLSTK\$(1) returns the name of the current Sub, Function, Method, or Property, and the value of each of the parameters at the time it was called. CALLSTK\$(2) returns the name of the procedure which called the current one, as well as its parameters. Likewise, CALLSTK\$(3) returns the one above it, and so forth.</p> <p>If the CALLSTK\$(<i>n</i>) parameter is outside the range of one (1) through the number of stack frames identified by CALLSTKCOUNT, an empty string is returned. Numeric parameters are displayed in decimal, while pointer and array parameters display a decimal representation of the offset of the target value.</p>
Restrictions	<p>The CALLSTK\$ function can be invaluable during debugging, but it generates substantial extra code which should be avoided in a final release version of an application. If the source code contains #TOOLS OFF, all CALLSTK\$ functions which remain in the program return an empty string.</p> <p>The CALLSTK\$ function is "thread-aware", returning only stack frame details from the thread in which it was referenced.</p>
See also	#TOOLS , CALLSTK , CALLSTKCOUNT , FUNCNAME\$, PROFILE , TRACE
Example	<pre>FOR x& = CALLSTKCOUNT TO 1 STEP -1 A\$ = A\$ + CALLSTK\$(x&) NEXT x&</pre>

CALLSTKCOUNT function

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the number of stack frames in the call stack. Used in conjunction with the CALLSTK\$ function.
Syntax	<code>count& = CALLSTKCOUNT</code>
Remarks	<p>CALLSTKCOUNT returns a Long-integer value that represents the total number of stack frames that currently exist on the application call stack.</p> <p>Retrieve individual stack frame details with the CALLSTK\$ function, or write them all to a disk file with the CALLSTK statement.</p>
Restrictions	<p>The CALLSTKCOUNT function, when used in conjunction with the CALLSTK\$ function, can be invaluable during debugging, but its use generates substantial extra code which should be avoided in a final release version of an application. If the source code contains #TOOLS OFF, all CALLSTKCOUNT functions which remain in the program return zero.</p> <p>The CALLSTKCOUNT function is "thread-aware", returning only the stack frame count from the thread in which it was referenced.</p>
See also	#TOOLS , CALLSTK\$, CALLSTK , FUNCNAME\$, PROFILE , TRACE
Example	<pre>FOR x& = CALLSTKCOUNT TO 1 STEP 1 A\$ = A\$ + CALLSTK\$(x&) NEXT x&</pre>

Purpose	In a Callback Function , return information about a message.
Syntax	<pre>CtlID = CB.CTL CtlMsg = CB.CTLMSG WinHndl = CB.HNDL Value = CB.LPARAM Msg = CB.MSG Value = CB.WPARAM CodeMsg = CB.NMCODE NmPtr = CB.NMHDR NmStruc = CB.NMHDR\$ NmHndl = CB.NMHWND NmID = CB.NMID</pre>
Remarks	<p>When an event occurs (like a user clicking on a button, a character typed into a text box, etc.) Windows sends a message to the control Callback Function, or the Dialog Callback Function. The CB functions are used to easily retrieve information about the message. These CB functions can only be used within a callback function.</p> <p>Callback functions in Windows have a standard set of four parameters. For this reason, Classic PowerBASIC allows you to ignore them and save some typing in your source code. The implied parameters are:</p>

```
FUNCTION DlgCallback(BYVAL hDlg AS DWORD _
    BYVAL wParam AS LONG _
    BYVAL lParam AS LONG)
```

Generic Callback Functions

CB.HNDL	This function returns the window handle of the parent dialog. This is the value specified by the <i>hDlg</i> parameter above.
CB.MSG	Each type of message sent to your callback function has a unique numeric value, such as %WM_COMMAND, %WM_NOTIFY, etc. CB.MSG will return the actual numeric message value of the message being processed. The definitions of the numeric values in other CB functions (CB.LPARAM, CB.WPARAM, CB.CTL, etc.) can only be ascertained once CB.MSG is identified. Therefore, callback functions usually test the value of CB.MSG first.
CB.WPARAM	When Windows sends a message to a callback function, the wParam value contains different values, depending on the nature of the particular message (CB.MSG). In other words, CB.WPARAM returns a message-dependent value.
CB.LPARAM	When Windows sends a message to a callback function, the lParam value contains different values, depending on the nature

of the particular message (CB.MSG). In other words, CB.LPARAM returns a message-dependent value.

%WM_COMMAND Specific Callback Functions

CB.CTL

If CB.MSG = %WM_COMMAND, this function returns the ID number assigned to the control with the CONTROL ADD statement. For other values of CB.MSG, it returns message-dependent values. This value is sent as the low-order word of the wParam parameter. It's functionally equivalent to [LO](#)(WORD, wParam&) in a conventional function, or [LO](#)(WORD, CB.WPARAM) in a [DDT](#) Callback Function.

CB.CTLMSG

If CB.MSG = %WM_COMMAND, this function returns the specific control message describing the event which occurred. For example, CB.CTLMSG returns %BN_CLICKED when the user clicks a button. For other values of CB.MSG, it returns message-dependent values. This value is sent as the high-order word of the wParam parameter. It's functionally equivalent to [HI](#)(WORD, wParam&) in a conventional function, or [HI](#)(WORD, CB.WPARAM) in a DDT Callback Function.

%WM_NOTIFY Specific Callback Functions

CB.NMCODE

If CB.MSG = %WM_NOTIFY, this function returns the specific notification message describing the event which occurred. For example, CB.NMCODE returns %NM_SETFOCUS when the described control gains the focus. For other values of CB.MSG, the value returned is meaningless.

CB.NMHDR

If CB.MSG = %WM_NOTIFY, this function returns the address (a pointer) to the NMHDR [UDT](#) for this notification message. NMHDR is defined as:

Type NMHDR

```
hwndFrom as DWord ' Handle of the control sending the
message
idfrom as DWord ' Identifier of the control sending the
message
code as Long ' Notification code
End Type
```

Some notification messages (%NM_CHAR, %NM_CLICK, etc.) require an extended version of the NM structure. However, all NM structures begin with an NMHDR UDT, so the pointer returned here is always accurate. For other values of CB.MSG, the pointer returned by CB.NMHDR is meaningless.

CB.NMHDR\$

If CB.MSG = %WM_NOTIFY, this function returns the contents

of the NMHDR UDT as a [dynamic string](#). If the notification message is one which requires an extended version of the NM structure, the string returned contains all of the data for the extended UDT. However, in all cases, the first 12 bytes of the returned string will be the contents of NMHDR. You can use [TYPE SET](#) to assign the string data to an appropriate user-defined type. For other values of CB.MSG, the string returned by CB.NMHDR\$ is meaningless.

The following notification messages use the extended NM structures as listed, so an appropriately longer string is returned:

Message	UDT
%NM_CLICK	NM_MOUSE
%NM_RCLICK	NM_MOUSE
%NM_NCHITTEST	NM_MOUSE
%NM_KEYDOWN	NM_KEY
%NM_SETCURSOR	NM_MOUSE
%NM_CHAR	NM_CHAR
%NM_TOOLTIPSCREATED	NM_TOOLTIPSCREATED

Other special notify messages may use a different extended NM structure than those listed above. To ensure compatibility, you can include an optional numeric parameter to specify the size of the special UDT you are using:

```
TYPE SET NotifyUDT = CB.NMHDR$(sizeof(NotifyUDT))
```

CB.NMHWND

If CB.MSG = %WM_NOTIFY, this function returns the handle of the control which sent this message. For other values of CB.MSG, the value returned is meaningless.

CB.NMID

If CB.MSG = %WM_NOTIFY, this function returns the ID number assigned to this control. For other values of CB.MSG, the value returned is meaningless.

Restrictions These functions are only valid inside a Callback Function. The CB Callback functions replace [CBMSG](#), [CBHNDL](#), [CBLPARAM](#), [CBWPARAM](#), [CBCTL](#), and [CBCTLMSG](#). Note these functions may not be supported in future versions of Classic PowerBASIC, so update your code to use the new syntax.

See also [Callbacks](#), [Dynamic Dialog Tools](#)

Purpose	<p>In a Callback Function, return the numeric ID value of the control sending the callback message.</p> <p>CBCTL has been superceded by the CB Callback functions, although CBCTL remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre><i>Id&</i> = CBCTL</pre>
Remarks	<p>When a user clicks on a button, types into a text box, or generally interacts with any control in a dialog, Windows sends a message to the callback for the control (or the dialog if the control does not have a callback).</p> <p>CBCTL returns the ID number as assigned to the control with the CONTROL ADD statement. CBCTL will return the ID of a control if CBMSG = %WM_COMMAND. For other values of CBMSG, CBCTL will return message-dependent values.</p> <p>If we consider a conventional callback procedure:</p> <pre>FUNCTION DlgCallback(BYVAL hDlg???, BYVAL wParam, BYVAL lParam, BYVAL lParam) _ EXPORT AS LONG</pre> <p>The notification message is sent to the callback as the low-order word of the wParam parameter. Therefore, CBCTL is functionally equivalent to LO(WORD, wParam) in a conventional Callback Function, and equivalent to LO(WORD, CBWPARAM) in a DDT Callback Function.</p>
Restrictions	<p>This function is only valid inside a Callback Function.</p>
See also	<p>Dynamic Dialog Tools, CB Callback functions</p>
Example	<pre>CALLBACK FUNCTION ComboCallback() AS LONG SELECT CASE CBMSG CASE %WM_COMMAND IF CBCTL = %IDCOMBO AND CBCTLMSG = %CBN_DBLCLK THEN CALL UpdateMyStatus() FUNCTION = 1 EXIT FUNCTION END IF CASE Additional_Values [statements] END FUNCTION</pre>

Purpose In a [Callback](#) Function, return the numeric value of a notification message parameter sent to the callback from a control.

CBCTLMSG has been superceded by the [CB](#) Callback functions, although CBCTLMSG remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.

Syntax `CtlMsg& = CBCTLMSG`

Remarks When a user clicks on a button, types into a text box, or generally interacts with any control in a dialog, Windows sends a message to the callback for the control (or the dialog if the control does not have a callback).

CBCTLMSG will return the numeric value of the notification message if [CBMSG](#) = %WM_COMMAND. For other values of CBMSG, CBCTLMSG will return message-dependent values.

In other words, CBCTLMSG can only return a valid notification message value if CBMSG indicates that the message includes a notification message. That is, when CBMSG = %WM_COMMAND, CBCTLMSG will contain a notification message value sent by the originating control. For example, CBCTLMSG = %BN_CLICKED when the user clicks a button.

If we consider a conventional callback procedure:

```
FUNCTION DlgCallback(BYVAL hDlg???, BYVAL wParam, BYVAL lParam, BYVAL  
lParam) _  
EXPORT AS LONG
```

The notification message is sent to the callback as the high-order word of the wParam parameter. Therefore, CBCTLMSG is functionally equivalent to [HI](#)(WORD, wParam) in a conventional Callback Function, and equivalent to [HI](#)(WORD, [CBWPARAM](#)) in a DDT Callback Function.

Restrictions This function is only valid inside a Callback Function.

See also [Dynamic Dialog Tools](#), [CB Callback functions](#)

Example

```
CALLBACK FUNCTION ButtonCallback() AS LONG  
    SELECT CASE CBMSG  
        CASE %WM_COMMAND  
            IF CBCTL = %ID_OK AND CBCTLMSG = %BN_CLICKED THEN  
                MSGBOX "Button was clicked!", %MB_TASKMODAL  
                FUNCTION = 1  
            EXIT FUNCTION  
        END IF  
    END SELECT  
END FUNCTION
```

Purpose	<p>In a Callback Function, return the window handle of the caller.</p> <p>CBHNDL has been superceded by the CB Callback functions, although CBHNDL remains supported for a limited period.</p> <p>Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<i>WindowHandle??? = CBHNDL</i>
Remarks	<p>When a user clicks on a button, types into a text box, or generally interacts with any control in a dialog, Windows sends a %WM_COMMAND message to the control callback function. If no control callback function is defined, the message is sent to the dialog callback function. In either type of callback, CBHNDL returns the window handle of the parent dialog.</p> <p>CBHNDL will also return the window handle for all other types of messages that flow through dialog callback functions. For example, %WM_PAINT, %WM_INITDIALOG, etc. As it is possible to share a control callback function with multiple controls, it is also possible to share a dialog callback function with multiple dialogs. In such cases, CBHNDL can be used to make a clear distinction between each dialog.</p> <p>If we consider a conventional callback procedure definition:</p> <pre>FUNCTION DlgCallback(BYVAL hDlg???, BYVAL wParam, BYVAL lParam) _ EXPORT AS LONG</pre> <p>The handle is sent to the callback as the <i>hDlg???</i> parameter. <i>hDlg</i> may be a Long-integer or Double-word variable (i.e., hDlg& or hDlg???), but a Double-word variable is recommended.</p>
Restrictions	This function is only valid inside a Callback Function.
See also	Dynamic Dialog Tools , CB Callback functions
Example	<pre>CALLBACK FUNCTION DlgCallback() AS LONG SELECT CASE CBMSG ' CBMSG holds the message CASE %WM_SYSCOMMAND ' CBHNDL is dialog handle here CASE %WM_COMMAND ' CBCTL is the control ID here ' CBCTLMSG is the notification message [statements] END SELECT END FUNCTION</pre>

Purpose In a [Callback](#) Function, return the numeric value of the *lParam*& parameter of the message sent by the caller.

CBLPARAM has been superceded by the [CB](#) Callback functions, although CBLPARAM remains supported for a limited period.

Existing code should be converted to the new syntax as soon as possible.

Syntax *lParam*& = CBLPARAM

Remarks When Windows sends a message to the callback, the *lParam*& value contains different values depending on the value returned by *wMsg*&. In other words, CBLPARAM returns a message-dependent value. See [Dynamic Dialog Tools](#) for more information on callbacks and messages.

Restrictions This function is only valid inside a Callback Function.

See also [Dynamic Dialog Tools](#), [CB Callback functions](#)

Example

```
CALLBACK FUNCTION DlgCallback() AS LONG
  SELECT CASE CBMSG
    CASE %WM_INITDIALOG
      InitParam = CBLPARAM
    CASE %WM_USER
      x& = CBLPARAM ' user defined value
      [statements]
  END SELECT
END FUNCTION
```

Purpose

In a [Callback](#) Function, return the numeric value of the message sent by the caller.

CBMSG has been superceded by the [CB](#) Callback functions, although CBMSG remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.

Syntax

wMsg& = CBMSG

Remarks

Each type of message sent to your Callback Function has a unique numeric value. CBMSG will return the actual numeric message value of the message being processed. The definitions of the numeric values in other [DDT](#) functions ([CBLPARAM](#), [CBWPARAM](#), [CBCTL](#), etc) can only be ascertained once CBMSG is identified. Therefore, Callback Functions usually test the value of CBMSG first.

In the generally case, a dialog Callback Function should return TRUE (non-zero) for all %WM_COMMAND messages it processes. However, this rule cannot be equally applied to other types of messages, since the return value will be message-specific.

The CALLBACK FUNCTION automatically copies the FUNCTION return result value over to the DWL_MSGRESULT data area, but only if the return value is non-zero, and the existing DWL_MSGRESULT value is zero. This helps reduce callback code size when dealing with custom controls and many Windows Common Controls.

For more information on return values for messages, consult WIN32.HLP or MSDN at <http://msdn.microsoft.com>.

See [Dynamic Dialog Tools](#) for more information on callbacks and messages.

Restrictions

This function is only valid inside a Callback Function.

See also

[Dynamic Dialog Tools](#), [CB Callback functions](#)

Example

```
CALLBACK FUNCTION DlgCallback() AS LONG
  SELECT CASE CBMSG
    CASE %WM_INITDIALOG
      ' process message
      FUNCTION = 1
    CASE %WM_USER
      ' process message
      FUNCTION = 1
    CASE %WM_COMMAND
      ' process the control notification
      FUNCTION = 1
  END SELECT
END FUNCTION
```


Purpose	<p>Return the <i>wParam</i>& message in a Callback Function.</p> <p>CBWPARAM has been superceded by the CB Callback functions, although CBWPARAM remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre><i>wParam</i>& = CBWPARAM</pre>
Remarks	<p>When Windows sends a message to your Callback Function, the value of the <i>wParam</i>& parameter will depend on the <i>wMsg</i>& command sent. See Dynamic Dialog Tools for more information on callbacks and messages.</p>
Restrictions	<p>This function is only valid inside a Callback Function.</p>
See also	<p>Dynamic Dialog Tools, CB Callback functions</p>

[Top](#)
[Previous](#)
[Next](#)

Syntax

<i>bytevar?</i>	= CBYT (<i>numeric_expression</i>)
<i>currencyvar@</i>	= CCUR (<i>numeric_expression</i>)
<i>currencyextvar@@</i>	= CCUX (<i>numeric_expression</i>)
<i>doublevar#</i>	= CDBL (<i>numeric_expression</i>)
<i>doublewordvar???</i>	= CDWD (<i>numeric_expression</i>)
<i>extendedvar##</i>	= CEXT (<i>numeric_expression</i>)
<i>integervar%</i>	= CINT (<i>numeric_expression</i>)
<i>longintvar&</i>	= CLNG (<i>numeric_expression</i>)
<i>quadintvar&&</i>	= CQUD (<i>numeric_expression</i>)
<i>singlevar!</i>	= CSNG (<i>numeric_expression</i>)
<i>wordvar??</i>	= CWRD (<i>numeric_expression</i>)

Function	Result type
CBYT	<u>Byte</u>
CCUR	<u>Currency</u>
CCUX	<u>Extended-currency</u>
CDBL	<u>Double-precision floating-point</u>
CDWD	<u>Double-word</u>
CEXT	<u>Extended-precision floating-point</u>
CINT	<u>Integer</u>
CLNG	<u>Long-integer</u>
CQUD	<u>Quad-integer</u>
CSNG	<u>Single-precision floating-point</u>
CWRD	<u>Word</u>

These conversion functions are rarely needed as Classic PowerBASIC automatically performs any necessary conversions when executing an

assignment statement or passing parameters. For example:

```
e% = f#
```

is equivalent to:

```
e% = CINT(f#)
```

In the case of the functions that convert to integral values, the fractional part of the number is rounded. If the fractional part is exactly .5 then it rounds to the nearest even integer. For example, CINT(1.5) returns 2, CINT(.5) returns 0, and CLNG(-0.6) returns -1.

Restrictions CSNG limit string display to 7 significant digits.

See also [CEIL](#), [CVI and associated functions](#), [FIX](#), [INT](#), [MKI\\$ and associated functions](#)

Example

```
' Calculate CINT for a series of values
FOR I! = 2.4! TO 2.65! STEP 0.05!
  x$ = FORMAT$(I!, "0.00") + " is" + STR$(CINT(I!))
NEXT I!
```

Result

```
2.40 is 2
2.45 is 2
2.50 is 2
2.55 is 3
2.60 is 3
2.65 is 3
```

Purpose	Convert a floating-point variable or expression into an integral-class value, by returning the smallest integer value that is greater than or equal to its argument.
Syntax	<code>intvar = CEIL(numeric_expression)</code>
Remarks	The CEIL function rounds upward, returning the smallest integer value that is greater than or equal to <i>numeric_expression</i> . For example, <code>y = CEIL(1.5)</code> places the value 2 into <code>y</code> .
See also	CINT , FIX , FRAC , INT , ROUND
Example	<pre>' Display the ceiling for a series of values FOR W! = -1.5! TO 1.5! STEP 0.5! x\$ = "CEIL" + FORMAT\$(W!, "* 0.00") + _ " =" + FORMAT\$(CEIL(W!), "* 0.00") NEXT W!</pre>
Result	<pre>CEIL -1.50 = -1.00 CEIL -1.00 = -1.00 CEIL -0.50 = 0.00 CEIL 0.00 = 0.00 CEIL 0.50 = 1.00 CEIL 1.00 = 1.00 CEIL 1.50 = 2.00</pre>

Purpose	Change the current (default) directory on the default drive, or any other drive (similar to the DOS CHDIR command). CHDIR affects only the default drive for the current program.
Syntax	<code>CHDIR <i>path</i></code>
Remarks	<p><i>path</i> is a string expression containing either a relative or an explicit directory name. The directory name can be constructed from a (DOS-Style) Short File Name (SFN) directory name, a Long File Name (LFN) directory name, or a combination of the two. Also, <i>path</i> may be prefixed with a drive letter and colon (i.e., "D:") to change the current directory on a non-default drive.</p> <p>The current directory is the location where your program will perform file operations by default. Thus:</p> <pre>CHDIR "\DATA"</pre> <p>changes to the \DATA subdirectory on the current drive, and:<pre>CHDIR "..\DATA2"</pre><p>changes the current directory to a directory whose parent is also the parent to the original directory. The double-period implies the parent directory.</p><pre>CHDIR "J:\Program Files\Internet Explorer"</pre><p>changes the current directory of Drive J. Drive J need not be the current default drive.</p><p>If <i>path</i> does not specify a valid directory on the target drive, a run-time Error 76 occurs ("Path not found").</p><p>A program that changes the current directory on the default drive also changes its active directory.</p><p><i>path</i> may also be used with UNC names (i.e., \\server\share), but their use is subject to operating system restrictions.</p></p>
Restrictions	CHDIR is not intended to change the current default drive. Use CHDRIVE instead.
See also	CHDRIVE , CURDIR\$, MKDIR , RMDIR

CHDRIVE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the current default drive.
Syntax	<code>CHDRIVE <i>drive</i></code>
Remarks	<i>drive</i> is a string expression whose first character is a letter from A to the highest logical drive letter. The trailing colon (:) that DOS uses is optional in Classic PowerBASIC. If <i>drive</i> does not indicate a valid drive, a run-time Error 76 occurs ("Path not found").
See also	CHDIR , CURDIR\$, MKDIR , RMDIR
Example	<pre>SDrive\$ = "C" CHDRIVE SDrive\$ ' change to the C: drive</pre>

CHOOSE function

[Top](#) [Previous](#) [Next](#)

Purpose	Return one of several values, based upon the value of an index.
Syntax	<pre>y = CHOOSE(index&, choice1 [, choice2] ...) y& = CHOOSE&(index&, choice1 [, choice2] ...) y\$ = CHOOSE\$(index&, choice1 [, choice2] ...)</pre>
Remarks	<p>These functions take any number of choice arguments, and return the argument identified by <i>index&</i>. If <i>index&</i> evaluates to one, <i>choice1</i> is returned, if two, <i>choice2</i> is returned, etc. If <i>index&</i> is less than one or greater than the number of choices provided, zero or an empty string is returned accordingly.</p> <p>CHOOSE expects choices of any numeric type. CHOOSE& expects choices optimized for the Long-integer type. CHOOSE\$ expects choices of string type. CHOOSE% is recognized as a valid synonym for CHOOSE&.</p>
Restrictions	Classic PowerBASIC only evaluates the selected choice at run-time, not all of them. This ensures optimum execution speed, as well as the elimination of unanticipated side effects.
See also	IIF , IIF& , IIF\$, MAX , MAX& , MAX\$, MIN , MIN& , MIN\$, SWITCH , SWITCH& , SWITCH\$, SELECT
Example	<pre>y& = 4 a\$ = CHOOSE\$(y&, "Bill", "Bob", "Bruce", "Barry")</pre>
Result	<pre>a\$ = "Barry"</pre>

Purpose	Convert one or more ASCII codes, ranges, and/or strings into a single string containing the corresponding ASCII character(s).
Syntax	<pre>s\$ = CHR\$(expression [,expression] [...]) s\$ = CHR\$(string_expression [...]) s\$ = CHR\$(x& TO y& [...])</pre>
Remarks	<p>CHR\$ returns a string, each character of which represents the corresponding ASCII codes of the arguments. Single or multiple arguments may be specified, each of which may be a numeric character code expression, string expression, or numeric character code range. These argument types may be freely intermixed in a single CHR\$ function. Numeric ASCII arguments must be in the range -1 through 255.</p> <p>CHR\$(x& TO y&) returns a sequence of all characters from CHR\$(x&) through CHR\$(y&) inclusive, provided x& <= y&. If x& > y&, an empty string is returned. For example, CHR\$(65 TO 70) returns "ABCDEF".</p> <p>The expanded CHR\$ definition is intended to assist in the encoding of multi-byte strings, to avoid the need for concatenation operations. For example, the CHR\$ function can be used to create COLLATE strings for the ARRAY SORT and ARRAY SCAN statements at run-time, and can be used to create string equates at compile time:</p> <pre>\$colstring = CHR\$(0 TO 131, 97, 133 TO 255)</pre> <p>The following lines are functionally equivalent, and return the same string result:</p> <pre>a\$ = CHR\$("Line1", 13, 10, "Line2") a\$ = "Line1" & CHR\$(13) & CHR\$(10) & "Line2" a\$ = "Line1" & \$CRLF & "Line2"</pre> <p>CHR\$ complements the ASC function, which returns the ASCII code of a nominated character in a string. In addition, CHR\$ with a parameter value of -1 returns an empty string for that parameter. For example, CHR\$(65, -1, 66) returns "AB".</p> <p>CHR\$ is also handy for creating characters that are difficult to enter at the keyboard, such as international characters for text output, or control code sequences for printer output, etc.</p>
See also	ARRAY SCAN , ARRAY SORT , ASC function , ASC statement , NUL\$, SPACE\$, STRING\$
Example	<pre>H\$ = CHR\$("a\$=", \$DQ, 33, \$DQ+\$DQ, 35 TO 39, 40, \$DQ)</pre>
Result	<pre>a\$="!""#\$\$%&'("</pre>

Purpose

Create the code and data for an [object](#).

Syntax

```
CLASS name [$GUID] [AS COM | AS EVENT]
  INSTANCE ClassName AS STRING
  Class Method code blocks...
  INTERFACE name $GUID [AS EVENT]
  INHERIT IUNKNOWN
  Method and Property code blocks...
END INTERFACE
EVENT SOURCE interface-name
END CLASS
```

Remarks

CLASS / END CLASS statements enclose the Interface implementation(s) and Instance variable declarations of a [Class](#). [METHOD](#) and [PROPERTY](#) blocks contain the code to be executed on an object. [INSTANCE](#) statements define the [variables](#) which are unique to each instance of an object of this class.

The name and optional \$GUID are supplied by the programmer to identify the class. By default, a class is considered private, so that the methods are accessible only from within the EXE or DLL where it is defined. The AS COM attribute makes the class available externally, to virtually any process which is [COM](#)-aware.

With a private class, the \$GUID may be freely omitted, as Classic PowerBASIC can readily identify the class by name. With a [published](#) COM class, you should insert a specific [GUID](#) of your choice. If omitted, a random GUID will be created by the compiler, but it will change every time you compile the program. This will be difficult to synchronize with other programs which wish to identify and access your object.

If a class is an [Event](#) Source (it generates events rather than handling events), one or more [EVENT SOURCE](#) statements are included to name the event interfaces. The event interfaces must be declared and implemented separately. An event is generated by executing a [RAISEEVENT](#) statement or an [OBJECT RAISEEVENT](#) statement in the class. If a class is an Event Handler (it contains code to handle an event generated by an Event Source), the AS EVENT attribute must appear on the CLASS statement and each [INTERFACE](#) statement. An Event Handler is also known as an "Event Sink".

See also

[EVENT SOURCE](#), [EVENTS](#), [INSTANCE](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [Just what is COM?](#), [METHOD](#), [PROPERTY](#), [RAISEEVENT](#), [What is an object, anyway?](#)

Purpose Copy data to/from the Windows ClipBoard.

Syntax

```
CLIPBOARD GET BITMAP [TO] ClipVar [, ClipResult]
CLIPBOARD GET OEMTEXT [TO] StrgVar [, ClipResult]
CLIPBOARD GET TEXT [TO] StrgVar [, ClipResult]
CLIPBOARD GET UNICODE [TO] StrgVar [, ClipResult]
CLIPBOARD RESET [, ClipResult]
CLIPBOARD SET BITMAP ClipHndl [, ClipResult]
CLIPBOARD SET OEMTEXT StrgExpr [, ClipResult]
CLIPBOARD SET TEXT StrgExpr [, ClipResult]
CLIPBOARD SET UNICODE StrgExpr [, ClipResult]
```

ClipHndl A [Long Integer](#) or [Dword](#) value which specifies the 32-bit handle of a GRAPHIC BITMAP passed to the clipboard.

ClipResult A Long Integer or Dword variable which receives a [true](#) result (-1) if the operation was successful, or a [false](#) result (0) if it failed.

ClipVar A Long Integer or Dword variable which receives a 32-bit handle of a newly created GRAPHIC BITMAP.

StrgExpr A [string expression](#) which specifies data to be passed to the clipboard.

StrgVar A [Dynamic String](#) variable which receives string data from the clipboard.

Remarks

The Windows ClipBoard provides support for the transfer of various types of data between applications, or even different parts of a single application. The concept is simple -- save some data on the ClipBoard and retrieve it later. In most cases, it's just used to transfer plain text, so the Classic PowerBASIC CLIPBOARD statement concentrates on the common data formats. With text transfer, you can just read or write a string. With bitmaps, a GRAPHIC BITMAP is used for this purpose.

When you retrieve data using CLIPBOARD, the original copy always remains in the CLIPBOARD, so the operation can be repeated any number of times. When you store data on the CLIPBOARD, your original copy remains unchanged. The data is copied, with no change of ownership.

The clipboard can hold multiple data items, but only one of each data format at a time. Generally speaking, multiple data items are only used to store a single piece of data in multiple formats to ensure it can be retrieved successfully later. However, you should note that Windows automatically converts string data between TEXT, OEMTEXT, and UNICODE. When you store data in one of those forms, it's not necessary to repeat it with the others.

You must execute a CLIPBOARD RESET to empty the clipboard before storing new data items. Each form of the CLIPBOARD statement

offers an optional *ClipResult* variable. If the requested operation is deemed successful by Windows, this variable is assigned the value TRUE (-1). If it fails, the value FALSE (0) is assigned instead. You should note that the success test is not a comprehensive one. It tests only the operation, not the validity of the data.

There are nine general forms of the CLIPBOARD statement:

CLIPBOARD GET BITMAP [TO] *ClipVar* [, *ClipResult*]

A new GRAPHIC BITMAP is automatically created. A Bitmap is copied from the ClipBoard and stored in this newly created GRAPHIC BITMAP. The handle of the new GRAPHIC BITMAP is assigned to the *ClipVar*, a DWord or Long Integer variable. If the operation is not successful, the value zero (0) is assigned instead.

CLIPBOARD GET OEMTEXT [TO] *StrgVar* [, *ClipResult*]

A text string is retrieved from the CLIPBOARD, and assigned to the dynamic string variable specified by *StrgVar*. If necessary, it is converted to OEM Text format, the format used by the Windows Console. If no text can be retrieved, a nul (zero-length) string is assigned instead.

CLIPBOARD GET TEXT [TO] *StrgVar* [, *ClipResult*]

A text string is retrieved from the CLIPBOARD, and assigned to the dynamic string variable specified by *StrgVar*. If necessary, it is converted to standard ASCII format. If no text can be retrieved, a nul (zero-length) string is assigned instead.

CLIPBOARD GET UNICODE [TO] *StrgVar* [, *ClipResult*]

A text string is retrieved from the CLIPBOARD, and assigned to the dynamic string variable specified by *StrgVar*. If necessary, it is converted to Unicode Text format. If no text can be retrieved, a nul (zero-length) string is assigned instead.

CLIPBOARD RESET [, *ClipResult*]

The contents of the CLIPBOARD are deleted.

CLIPBOARD SET BITMAP *ClipHndl* [, *ClipResult*]

A GRAPHIC BITMAP, specified by *ClipHndl*, is stored on the CLIPBOARD. The GRAPHIC BITMAP may be a [GRAPHIC CONTROL](#), [GRAPHIC WINDOW](#), or GRAPHIC BITMAP. When passing a GRAPHIC CONTROL to the Clipboard, use [CONTROL HANDLE](#) to obtain the handle to the GRAPHIC CONTROL.

CLIPBOARD SET OEMTEXT *StrgExpr* [, *ClipResult*]

A text string, specified by *StrExpr*, is stored on the CLIPBOARD. The string data is assumed to use characters in the OEM character set.

CLIPBOARD SET TEXT *StrgExpr* [, *ClipResult*]

A text string, specified by *StrExpr*, is stored on the CLIPBOARD. The string data is assumed to be in standard ASCII format.

CLIPBOARD SET UNICODE *StrgExpr* [, *ClipResult*]

A text string, specified by *StrExpr*, is stored on the CLIPBOARD. The string data is assumed to be in Unicode format.

CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Conclude I/O (input/output) to / from a file or device .
Syntax	<code>CLOSE [[#] <i>filenum</i>& [, [#] <i>filenum</i>&] ...]</code>
Remarks	<p>CLOSE ends the relationship between a Classic PowerBASIC <i>file number</i> and the disk file or device that was associated with it by an OPEN statement. Any pending I/O operations on the file/device are concluded, buffers are flushed and released, and the disk directory information (if any) for that file is updated.</p> <p>If no file number is specified, CLOSE closes all open files.</p> <p>If the file was opened using OPEN HANDLE, the CLOSE statement is still needed, although it does not tell the operating system to close the file. In this special case, the file was already open when OPEN HANDLE provided access to it, and will remain open after CLOSE disassociates the file from Classic PowerBASIC.</p> <p>CLOSE works with all types of files and devices (disk files, devices, COMM, TCP, UDP, etc).</p> <p>The number symbols (#) are optional but recommended for clarity.</p>
See also	COMM CLOSE , FILEATTR , FLUSH , OPEN , TCP CLOSE , UDP CLOSE

CLSID\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a 16-byte GUID string (128-bit GUID format string) containing a CLSID associated with a unique ProgramID string of a COM object or component .
Syntax	<code>a\$ = CLSID\$(<i>ProgramID\$</i>)</code>
Remarks	<p>A CLSID string is a 128-bit (16-byte) binary string representing the GUID or UUID of a COM object/component. A CLSID string is not in a human-readable format.</p> <p>You can convert textual ID name of a COM object/component into a CLSID string with the CLSID\$ function. CLSID examines the system registry in order to determine the CLSID string associated with the <i>ProgramID\$</i> string.</p> <p>The <i>ProgramID\$</i> parameter is not case-sensitive, so "MSAGENT.CONTROL.2", "MSAgent.Control.2" and "msagent.control.2" all refer to the same COM object/component. If the <i>ProgramID\$</i> cannot be found, or if any error occurs during the lookup and conversion process, CLSID\$ will not set the ERR system variable, but will return an empty string.</p> <p>To convert the binary CLSID string into human-readable GUID/UUID format, use the GUIDTXT\$ function. CLSID\$ is the complement to the PROGID\$ function.</p> <p>Classic PowerBASIC programmers rarely, if ever, need to deal with CLSID strings in order to utilize a COM object or component.</p>
a\$	The return string may be assigned to a dynamic string , fixed-length or ASCIIZ string (at least 16 bytes long), or (typically) a GUID variable. See DIM for more information.
See also	DIM , GUID\$, GUIDTXT\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISNOTHING , ISOBJECT , LET (with Objects) , OBJECT , OBJACTIVE , OBJPTR , OBJRESULT , PROGID\$, What does a Class look like? , What is an object, anyway?
Example	<pre>MSWordClassID\$ = CLSID\$("word.application.8") IF LEN(MSWordClassID\$) = 16 THEN ' Success getting the CLSID\$ of MSWord a\$ = PROGID\$(MSWordClassID\$) 'a\$ holds "Word.Application.8" b\$ = GUIDTXT\$(MSWordClassID\$) 'b\$ holds "{000209FF-0000-0000-C000-000000000046}" END IF</pre>

Purpose	Obtain a 32-bit address of a label , Sub or Function .
Syntax	<i>address32</i> = CODEPTR({ <i>label</i> <i>functionname</i> <i>subname</i> })
Remarks	CODEPTR is particularly useful when it is necessary to pass the address of a SUB or FUNCTION to Windows for callbacks. <i>address32</i> must be a Long-integer (LONG) or Double-word (DWORD) variable.
Restrictions	Pointers may not be obtained for labels that are outside the scope of the current Sub, Function, Method , or Property . Labels have local scope in current versions of Classic PowerBASIC.
See also	STRPTR , VARPTR , CALL DWORD
Example	<pre>#COMPILE EXE SUB MySub() END SUB FUNCTION PBMAIN LOCAL MySubPtr AS LONG, X AS STRING MySubPtr = CODEPTR(MySub) ' Address of MySub() X = "MySub() is located at address " + FORMAT\$(MySubPtr) END FUNCTION</pre>

Purpose Manipulate a [COMBOBOX](#) control in order to set/retrieve data.

Syntax

```
COMBOBOX ADD hDlg, id&, StrExpr
COMBOBOX DELETE hDlg, id&, item&
COMBOBOX FIND hDlg, id&, item&, StrExpr TO datav&
COMBOBOX FIND EXACT hDlg, id&, item&, StrExpr TO datav&
COMBOBOX GET COUNT hDlg, id& TO datav&
COMBOBOX GET SELCOUNT hDlg, id& TO datav&
COMBOBOX GET SELECT hDlg, id& TO datav&
COMBOBOX GET STATE hDlg, id&, item& TO datav&
COMBOBOX GET TEXT hDlg, id& [, item&] TO txtv$
COMBOBOX GET USER hDlg, id&, item& TO datav&
COMBOBOX INSERT hDlg, id&, item&, StrExpr
COMBOBOX RESET hDlg, id&
COMBOBOX SELECT hDlg, id&, item&
COMBOBOX SET TEXT hDlg, id&, item&, StrExpr
COMBOBOX SET USER hDlg, id&, item&, NumExpr
COMBOBOX UNSELECT hDlg, id&
```

hDlg Handle of the [dialog](#) that owns the combobox.

id& The control identifier assigned with [CONTROL ADD COMBOBOX](#).

item& Position of data in the COMBOBOX. First string=1, second=2...

NumExpr A numeric expression passed as a parameter.

StrExpr A [string expression](#) passed as a parameter.

txtv\$ A string variable to which result text is assigned.

datav& A [long integer](#) variable to which result data is assigned.

Remarks In each of the following samples and descriptions, the COMBOBOX control which is the subject of the statement is identified by the handle of the dialog that owns the COMBOBOX (*hDlg*), and the unique control identifier you gave it upon creation in CONTROL ADD COMBOBOX.

COMBOBOX ADD *hDlg*, *id*&, *StrExpr*

The string value specified by *StrExpr* is added to the COMBOBOX control. If the COMBOBOX has the [%CBS_SORT](#) style, the new string is inserted in alphanumeric order; otherwise it is added to the end of the existing list.

COMBOBOX DELETE *hDlg*, *id*&, *item*&

The string at the position specified by *item*& is deleted from the COMBOBOX. The item number (*item*&) is indexed to one (1=first, 2=second, and so on).

COMBOBOX FIND *hDlg, id&, item&, StrExpr* TO *datav&*

Strings in the COMBOBOX are searched to find the first string which begins with the data in *StrExpr*, regardless of any characters which follow. Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *item&*, and ending with the last string in the COMBOBOX. Searching does not wrap to the beginning of the list. The item number (*item&*) is indexed to 1 (1=first, 2=second, etc.). To search the entire COMBOBOX starting with the first string, *item&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

COMBOBOX FIND EXACT *hDlg, id&, item&, StrExpr* TO *datav&*

Strings in the COMBOBOX are searched to find the first string which exactly matches the data in *StrExpr*. Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *item&*, and ending with the last string in the COMBOBOX. Searching does not wrap to the beginning of the list. The item number (*item&*) is indexed to 1 (1=first, 2=second, etc.). To search the entire COMBOBOX starting with the first string, *item&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

COMBOBOX GET COUNT *hDlg, id&* TO *datav&*

The number of items in the list box of the COMBOBOX is retrieved, and assigned to the long integer variable specified by *datav&*.

COMBOBOX GET SELCOUNT *hDlg, id&* TO *datav&*

The number of selected items in the list box of the COMBOBOX is retrieved, and assigned to the long integer variable specified by *datav&*. Since this is a single-selection list box, the retrieved value will always be either zero or one.

COMBOBOX GET SELECT *hDlg, id&* TO *datav&*

The index of the currently selected item in the list box of the COMBOBOX is retrieved, and assigned to the variable specified by *datav&*. The index is 1 for the first item, 2 for the second item, etc. If there is no current selection, the value zero (0) is assigned.

COMBOBOX GET STATE *hDlg, id&, item&* TO *datav&*

A data item is checked to see if it is currently selected. The numeric value

item& specifies which user value is to be checked, 1 for the first item, 2 for the second item, etc. If the item is selected, -1 (true) is assigned to the variable specified by *datav&*. Otherwise, 0 (false) is assigned to it.

COMBOBOX GET TEXT *hDlg, id& [,item&] TO txtv\$*

Text is retrieved from the COMBOBOX and assigned to the string variable specified by *txtv\$*. If the numeric expression *item&* is included, it determines which text string is returned, 1 for the first item, 2 for the second item, etc. If *item&* is missing, or contains the value zero, the selected text is returned (or an empty string if none is selected). If you wish to retrieve the text found in the edit box portion of the COMBOBOX (regardless of whether it was typed or selected), you should use the [CONTROL GET TEXT](#) statement instead.

COMBOBOX GET USER *hDlg, id&, item& TO datav&*

Each item in a COMBOBOX may have a long integer user value associated with it at the discretion of the programmer. This user value is retrieved with COMBOBOX GET USER. The numeric value *item&* specifies which user value is requested, 1 for the first item, 2 for the second item, etc. The returned user value is assigned to the long integer variable specified by *datav&*. COMBOBOX user values are assigned with the COMBOBOX SET USER statement. In addition to these COMBOBOX user values, every DDT control offers an additional eight user values which can be accessed with [CONTROL GET USER](#) and [CONTROL SET USER](#).

COMBOBOX INSERT *hDlg, id&, item&, StrExpr*

The text for a new data item, specified by *StrExpr*, is inserted at the location given by *item&*. The value of *item&* = 1 for the first item, 2 for the second item, etc. The list of data items is not re-sorted, even if the COMBOBOX was created with the style %CBS_SORT. If you wish to sort all of the items, use COMBOBOX ADD instead.

COMBOBOX RESET *hDlg, id&*

Delete all contents of the specified COMBOBOX.

COMBOBOX SELECT *hDlg, id&, item&*

The string value specified by *item&* is chosen as selected text for the COMBOBOX control, and the selected text is scrolled into a visible position. The value of *item&* = 1 for the first item, 2 for the second item, etc.

COMBOBOX SET TEXT *hDlg, id&, item&, StrExpr*

The text for the data item specified by *item&* is replaced with the new text in *StrExpr*. The value of *item&* = 1 for the first item, 2 for the second item, etc. The list of data items is not re-sorted, even if the COMBOBOX was created with the style %CBS_SORT. If you wish to sort all of the items, use COMBOBOX DELETE followed by COMBOBOX ADD instead.

COMBOBOX SET USER *hDlg, id&, item&, NumExpr*

Each item in a COMBOBOX may have a long integer user value associated with it at the discretion of the programmer. This user value is assigned with COMBOBOX SET USER, and retrieved with COMBOBOX GET USER. The numeric value *item&* specifies which user value is to be accessed, 1 for the first item, 2 for the second item, etc. The value specified by *NumExpr* is saved for later retrieval. In addition to these COMBOBOX user values, every DDT control offers an additional eight user values which can be accessed with CONTROL GET USER and CONTROL SET USER.

COMBOBOX UNSELECT *hDlg, id&*

All items in a COMBOBOX control are set to an unselected state.

Restrictions Under Windows 95/98/ME, a list box is limited to 32,767 items. In all versions of Windows, the actual string data contained by the list box is limited only by available memory.

See also [Dynamic Dialog Tools](#), [CONTROL ADD COMBOBOX](#), [CONTROL GET TEXT](#)

COMM CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Close an open serial port .
Syntax	<code>COMM CLOSE [#] <i>hComm</i> [, [#] <i>hComm</i> ...]</code>
Remarks	<p>Closes one or more communication ports, as specified by the Classic PowerBASIC file number held in each <i>hComm</i> parameter. COMM CLOSE ends the relationship between a Classic PowerBASIC file number, and the serial port device that was previously associated with it by the COMM OPEN statement.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p> <p>It is also recommended that you explicitly close any serial port that you have opened before your application terminates. Note that COMM CLOSE is a synonym for CLOSE.</p>
See also	Serial Communications , CLOSE , COMM function , COMM LINE , COMM OPEN , COMM PRINT , COMM RECV , COMM RESET , COMM SEND , COMM SET
Example	<code>COMM CLOSE #hComm, 5 ' Close <i>hComm</i> and file number 5</code>

Purpose

Syntax

Remarks

Retrieve the value or status of a [communications](#) parameter.

lResult& = COMM([#] *hComm*, *Comfunc*)

hComm is the Classic PowerBASIC file number as was used by the [COMM OPEN](#) statement to open the communications port. Select a *Comfunc* keyword from the following table to retrieve the associated setting.

Comfunc	value (TRUE <> 0, FALSE = 0)
BAUD	Port Baud Rate (9600, 14400, 19200, etc).
BREAK	TRUE/FALSE Break is asserted. Break is generally used to "get the attention" of the connected modem, terminal or system.
BYTE	Number of bits per byte (4, 5, 6, 7, or 8).
CD	TRUE/FALSE Carrier Detect state; synonym for RLSD (<i>READ-ONLY</i>). When CD is TRUE, the DCE (modem) has a suitable connection on the communications channel present. When CD is FALSE, there is no suitable connection.
CTS	TRUE/FALSE Clear-To-Send state is returned (<i>READ-ONLY</i>).
CTSFLOW	TRUE/FALSE Enable CTS output flow control (Input signal). When CTSFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the CTS signal is set to logic low by the DCE (modem). Transmission continues when the DCE (modem) sets the CTS signal back to logic high. The CTS signal is usually used in response to an RTS signal.
DSR	TRUE/FALSE Data-Set-Ready state is returned (<i>READ-ONLY</i>).
DSRFLOW	TRUE/FALSE Enable DSR output flow control (Input signal). When DSRFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the DSR signal is set to logic low by the DCE (modem). Transmission is enabled when the DSR signal returns to logic high. The DSR signal is often used in conjunction with CTS in response to a RTS signal.
DSRSENS	TRUE/FALSE Enable DSR sensitivity. When DSRSENS is enabled, data received by the DTE (computer) is placed into the receive buffer only if DSR is set to logic high. If DSR is set low, received data is discarded. Enabling DSRSENS

	allows DSR to enable or disable the DTE (the computer) to receive data from the DTE (the modem). DSRSENS is rarely used in practical communications situations.
DTRFLOW	TRUE/FALSE Enable DTR handshaking flow control (Output signal). When DTRFLOW is enabled, it signals that the DCE (modem) should prepare to connect to the communications channel. DTR is usually used for modem on-hook/off-hook control, but can also be used in conjunction with DSR for handshaking.
DTRLIN	TRUE/FALSE Enable DTR line. When enabled, DTRLIN leaves the DTR line active when the port is closed by the DTE (computer). This ensures that the DCE (modem) does not close the communications channel when the port is closed.
NULL	TRUE/FALSE Null (\$NUL) bytes are discarded when read.
PARITY	TRUE/FALSE Enable parity checking. This mode must be enabled for the other Parity options to be selected.
PARITYCHAR	Character to use for parity error replacement. PARITY must be enabled.
PARITYREPL	TRUE/FALSE Enable character replacement on parity error. PARITY must be enabled.
PARITYTYPE	0 = None, 1 = Odd, 2 = Even, 3 = Mark, 4 = Space. PARITY must be enabled. Default = 0.
RING	TRUE/FALSE Ring indicator is on (<i>READ-ONLY</i>). When RING returns TRUE, a ringing signal is being received on the communications channel (by the modem). RING approximates the state of the ringing signal; however, it may not be reported accurately on all Windows platforms.
RLSD	Receive-line-signal-detect (<i>READ-ONLY</i>). See CD/Carrier Detect above.
RTSFLOW	Ready To Send (Output signal). 0 = Disable, 1 = Enable, 2 = Handshake, 3 = Toggle. Toggle is used for half-duplex (2-wire) operations to "reverse" the line. While the DTE (computer) is busy sending data, it raises the RTS signal and the DCE (modem) blocks its data receive channel. When RTS signal reverts to logic low, the DCE (modem) reverts to transmit mode and the DTE (computer) switches to receive mode. Handshake mode causes the DTE (computer) to check the receive buffer (RXQUEUE) after each character is placed into

	the buffer. When the buffer is 5/6th full, the RTS signal is dropped. When the receive buffer drops to below 1/6th full, RTS is raised again
RXBUFFER	Size of the receive buffer in bytes.
RXQUE	Bytes currently in the receive buffer (<i>READ-ONLY</i>).
STOP	0 = 1 stop bits, 1 = 1.5 stop bits, 2 = 2 stop bits.
TXBUFFER	Size of the transmit buffer in bytes. In some cases, Windows may not be able to report the transmit size.
TXQUE	Bytes currently in the transmit buffer (<i>READ-ONLY</i>).
XINPFLOW	TRUE/FALSE Enable XON/XOFF input flow control. When the DTE (computer) receive buffer is full, an XOFF character is sent to the DCE (modem) to instruct it to halt transmission. When the DCE is ready to resume transmission, an XON character is sent to the DCE. Typically, XOFF is sent when the receive buffer has less than 1/16th remaining, and XON is sent when the receive buffer drops to less than 1/16th of its maximum size. Default = FALSE.
XOUTFLOW	TRUE/FALSE Enable XON/XOFF out flow control. When enabled, the DCE (modem) sends an XOFF to the DTE (computer) to halt data transmission to the DCE. When the DCE is ready to receive more data, an XON character is sent. XOUTFLOW typically uses the same 1/16th rules as XINPFLOW. Default = FALSE.

Common baud rates range from 110 to 256000. There are equates defined in the [WIN32API.INC](#) file, prefixed with %CBR_ to assist you with specifying a common baud rate, but you are not restricted to a limited set of rates.

Classic PowerBASIC sets the [ERR](#) system variable if an error occurs when using the COMM function.

The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.

Restrictions Due to differences between Win32 operating systems, parameters (such as the TXBUFFER and TXQUE) may not be *queried* successfully in all circumstances.

See also [Serial Communications](#), [COMM CLOSE](#), [COMM LINE](#), [COMM OPEN](#), [COMM PRINT](#), [COMM RECV](#), [COMM RESET](#), [COMM SEND](#), [COMM SET](#)

Example

```
Qty& = COMM(#hComm, RXQUE)
x$ = "The receive buffer contains " + _
    FORMAT$(Qty&) + " bytes of data."

Qty& = COMM(#hComm, TXBUFFER) - COMM(#hComm, TXQUE)
x$ = "There is room for " + FORMAT$(Qty&) + _
    " bytes in the transmit buffer."
```

COMM LINE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Receive a CR/LF (\$CRLF) terminated "line" of data from a serial port .
Syntax	<code>COMM LINE [INPUT] [#] hComm, string_var</code>
Remarks	<p>Read a delimited line of data from the receive buffer, where a "line" is defined as a stream of data that is terminated by a CR/LF (carriage return and linefeed, \$CRLF, or CHR\$(13,10)). COMM LINE INPUT is ideal for retrieving modem response strings in reply to "AT" commands sent to a modem.</p> <p>COMM LINE reads the receive buffer up to the next \$CRLF character pair. The \$CRLF bytes are removed from the buffer but do not form part of the string data returned by COMM LINE. Note that if there is no \$CRLF pair in the receive buffer, the statement will wait indefinitely for a complete \$CRLF terminated line of data. In this sense, COMM LINE is a blocking statement. The <i>SetCommTimeouts</i> API can be used to specify COMM timeouts.</p> <p>The EOF function may also be used with COMM LINE (and TCP LINE) to detect that an incomplete line was received. Normally, the COMM LINE statement reads data until a \$CRLF character pair is found, and in that case, EOF will return false (zero). However, if a timeout does occur, COMM LINE will return whatever data has been accumulated, and set EOF to logical TRUE (non-zero).</p> <p>In many cases, it would be prudent to test EOF after every COMM LINE statement to verify that a full line has been received. In some cases, you may wish to execute the statement one or more additional times, combining the data, in order to obtain a full line of text.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p>
See also	Serial Communications , COMM CLOSE , COMM function , COMM OPEN , COMM PRINT , COMM RECV , COMM RESET , COMM SEND , COMM SET , EOF
Example	<pre>COMM PRINT #hComm, "AT" SLEEP 1000 ' delay for modem to respond DO COMM LINE INPUT #hComm, a\$ CALL DisplayResponse(a\$) ' display the modem echo LOOP UNTIL LEN(a\$)</pre>

Purpose	Open a serial port .
Syntax	<code>COMM OPEN "COMn" AS [#] hComm</code>
Remarks	<p>Opens a serial port to begin communications, creating a relationship between a file number and a specific serial port device.</p>
COMn	Identifies the serial port number, for example, COM1, COM4, etc. A colon must not follow the port specification. See Restrictions below.
hComm	<p>A valid integer-class variable containing a free Classic PowerBASIC file number, typically provided by the FREEFILE function. Although not recommended, hComm can be specified as a numeric literal; however, using hard-coded values can be more difficult to manage successfully in large programs, and such code is unlikely to be thread-safe.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p> <p>If the port was not opened successfully, the ERR system variable will contain the error code. Before actual communications through the port can commence, you must configure the communication parameters by using a COMM SET statement for each parameter.</p>
Restrictions	<p>COMM OPEN cannot use an operating system <i>file handle</i>, nor open a port that is already in use. When opening ports <u>above</u> COM9, Windows requires the port name to be specified using the following syntax:</p> <pre>COMM OPEN "\\.\COM15" AS #hComm</pre>
See also	Serial Communications , COMM CLOSE , COMM function , COMM LINE , COMM PRINT , COMM RECV , COMM RESET , COMM SEND , COMM SET , FREEFILE , OPEN
Example	<pre>DIM hComm AS LONG hComm = FREEFILE COMM OPEN "COM1" AS #hComm COMM OPEN "COM2" AS #5</pre>

Purpose	Send a "line" of binary data through a serial port with optional CR/LF (\$CRLF) termination.
Syntax	<code>COMM PRINT [#] <i>hComm</i>, <i>string_expression</i> [;]</code>
Remarks	<p>Send <i>string_expression</i> to the serial port, followed by a carriage return and linefeed pair (CHR\$(13,10) or \$CRLF), unless a trailing semicolon is employed. This is a variation of COMM SEND and is intended as a convenience for transmitting text data.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p> <p>COMM PRINT is ideal for sending "AT" commands to a modem. Omit the trailing semicolon for this purpose, since you would want the CR/LF to be send along with the data.</p>
Restrictions	Avoid using ASCIIZ strings to hold binary data, as any embedded CHR\$(0) or \$NUL characters will be misunderstood as indicating the end of the string.
See also	Serial Communications , COMM CLOSE , COMM function , COMM LINE , COMM OPEN , COMM RECV , COMM RESET , COMM SEND , COMM SET

Purpose	Receive binary data from a serial port .
Syntax	<code>COMM RECV [#] <i>hComm</i>, <i>count</i>&, <i>string_var</i></code>
Remarks	<p>Retrieve the <i>count</i>& number of bytes from the receive buffer, placing the results in <i>string_var</i>. Program execution will halt until <i>count</i>& bytes are available, so it is wise to check how many bytes are available before making a COMM RECV request. You can do this by checking the RXQUE value with the COMM function, as shown in the example below.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p>
Restrictions	Avoid using ASCIIZ string to hold binary data, as any embedded CHR\$(0) or \$NUL characters will be misunderstood as indicating the end of the string.
See also	Serial Communications , COMM CLOSE , COMM function , COMM LINE , COMM OPEN , COMM PRINT , COMM RESET , COMM SEND , COMM SET
Example	<pre>Qty& = COMM(#hComm, RXQUE) COMM RECV #hComm, Qty&, a\$</pre>

COMM RESET statement

[Top](#) [Previous](#) [Next](#)

Purpose	Disable flow control for a given serial port .
Syntax	<code>COMM RESET [#] <i>hComm</i>, FLOW</code>
Remarks	<p>Switches off all flow control to the serial port as specified by the file number stored in <i>hComm</i>.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p>
See also	Serial Communications , COMM CLOSE , COMM function , COMM LINE , COMM OPEN , COMM PRINT , COMM RECV , COMM SEND , COMM SET

Purpose	Send a string of binary data through a serial port .
Syntax	<code>COMM SEND [#] hComm, string_expression</code>
Remarks	<p>Send <i>string_expression</i> to the serial port. The data is appended to any data that may already be waiting in the transmit buffer. If <i>string_expression</i> is empty, no data is added to the transmit buffer.</p> <p>The data is sent without trailing CR/LF (\$CRLF or CHR\$(13,10)) bytes appended to the data. If you are sending lines of text, you can either append CR (\$CR or CHR\$(13)) and/or LF (\$LF or CHR\$(10)) bytes to the text, or use the COMM PRINT statement instead.</p> <p>The Number symbol (#) prefix is optional, but recommended for the purposes of clarity.</p>
Restrictions	It is not always appropriate to use fixed length strings in <i>string_expression</i> , as the entire length of the string is added to the transmit buffer. Avoid using ASCIIZ strings to hold binary data, as any embedded CHR\$(0) or \$NUL characters will be misunderstood as indicating the end of the string.
See also	Serial Communications , COMM CLOSE , COMM function , COMM LINE , COMM OPEN , COMM PRINT , COMM RECV , COMM RESET , COMM SET
Example	<pre>A\$ = "ATDT1,555-1234;" COMM SEND #hComm, a\$</pre>

Purpose	Set communication options for a serial port .
Syntax	<code>COMM SET [#] hComm, Comfunc = value</code>
Remarks	<p>Set the parameters needed to communicate with a serial port. This must always be done before you can send and receive data through the port.</p> <p>To configure the communication parameters, use keywords from the following table to specify the <i>Comfunc</i> as well as a suitable <i>value</i> chosen from the range applicable to the <i>Comfunc</i> parameter you want to set. If an error occurs when attempting to set a parameter, Classic PowerBASIC sets the ERR system variable to indicate the error number. While each parameter must be set individually, it is also possible to change certain parameters without the need to close and re-establish communications.</p>

COMM SET keywords table

<i>Comfunc</i>	<i>value</i> (TRUE <> 0, FALSE = 0)
BAUD	Port Baud Rate (9600, 14400, 19200, etc). See notes below.
BREAK	TRUE/FALSE Break is asserted. Break is generally used to "get the attention" of the connected modem, terminal or system.
BYTE	Number of bits per byte (4, 5, 6, 7, or 8).
CD	TRUE/FALSE Carrier Detect state; synonym for RLSD (<i>READ-ONLY</i>). When CD is TRUE, the DCE (modem) has a suitable connection on the communications channel present. When CD is FALSE, there is no suitable connection.
CTSFLOW	TRUE/FALSE Enable CTS output flow control (Input signal). When CTSFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the CTS signal is set to logic low by the DCE (modem). Transmission continues when the DCE (modem) sets the CTS signal back to logic high. The CTS signal is usually used in response to an RTS signal.
DSRFLOW	TRUE/FALSE Enable DSR output flow control (Input signal). When DSRFLOW is enabled, it causes the DTE (computer) to stop sending data whenever the DSR signal is set to logic low by the DCE (modem). Transmission is enabled when the DSR signal returns to logic high. The DSR signal is often used in conjunction with CTS in response to a RTS signal.

DSRSENS	TRUE/FALSE Enable DSR sensitivity. When DSRSENS is enabled, data received by the DTE (computer) is placed into the receive buffer only if DSR is set to logic high. If DSR is set low, received data is discarded. Enabling DSRSENS allows DSR to enable or disable the DTE (the computer) to receive data from the DTE (the modem). DSRSENS is rarely used in practical communications situations.
DTRFLOW	TRUE/FALSE Enable DTR handshaking flow control (Output signal). When DTRFLOW is enabled, it signals that the DCE (modem) should prepare to connect to the communications channel. DTR is usually used for modem on-hook/off-hook control, but can also be used in conjunction with DSR for handshaking.
DTRLIN	TRUE/FALSE Enable DTR line. When enabled, DTRLIN leaves the DTR line active when the port is closed by the DTE (computer). This ensures that the DCE (modem) does not close the communications channel when the port is closed.
NULL	TRUE/FALSE Null (\$NUL) bytes are discarded when read.
PARITY	TRUE/FALSE Enable parity checking. This mode must be enabled for the other Parity options to be selected.
PARITYCHAR	Character to use for parity error replacement. PARITY must be enabled.
PARITYREPL	TRUE/FALSE Enable character replacement on parity error. PARITY must be enabled.
PARITYTYPE	0 = None, 1 = Odd, 2 = Even, 3 = Mark, 4 = Space. PARITY must be enabled. Default = 0.
RING	TRUE/FALSE Ring indicator is on (<i>READ-ONLY</i>). When RING returns TRUE, a ringing signal is being received on the communications channel (by the modem). RING approximates the state of the ringing signal; however, it may not be reported accurately on all Windows platforms.
RLSD	Receive-line-signal-detect (<i>READ-ONLY</i>). See CD/Carrier Detect above.
RTSFLOW	Ready To Send (Output signal). 0 = Disable, 1 = Enable, 2 = Handshake, 3 = Toggle. Toggle is used for half-duplex (2-wire) operations to "reverse" the line. While the DTE

	<p>(computer) is busy sending data, it raises the RTS signal and the DCE (modem) blocks its data receive channel. When RTS signal reverts to logic low, the DCE (modem) reverts to transmit mode and the DTE (computer) switches to receive mode.</p> <p>Handshake mode causes the DTE (computer) to check the receive buffer (RXQUE) after each character is placed into the buffer. When the buffer is 5/6th full, the RTS signal is dropped. When the receive buffer drops to below 1/6th full, RTS is raised again.</p>
RXBUFFER	Size of the receive buffer in bytes.
RXQUE	Bytes currently in the receive buffer (<i>READ-ONLY</i>).
STOP	0 = 1 stop bits, 1 = 1.5 stop bits, 2 = 2 stop bits.
TXBUFFER	Size of the transmit buffer in bytes. In some cases, Windows may not be able to report the transmit size.
TXQUE	Bytes currently in the transmit buffer (<i>READ-ONLY</i>).
XINPFLOW	<p>TRUE/FALSE Enable XON/XOFF input flow control. When the DTE (computer) receive buffer is full, an XOFF character is sent to the DCE (modem) to instruct it to halt transmission. When the DCE is ready to resume transmission, an XON character is sent to the DCE.</p> <p>Typically, XOFF is sent when the receive buffer has less than 1/16th remaining, and XON is sent when the receive buffer drops to less than 1/16th of its maximum size. Default = FALSE.</p>
XOUTFLOW	<p>TRUE/FALSE Enable XON/XOFF out flow control. When enabled, the DCE (modem) sends an XOFF to the DTE (computer) to halt data transmission to the DCE. When the DCE is ready to receive more data, an XON character is sent. XOUTFLOW typically uses the same 1/16th rules as XINPFLOW. Default = FALSE.</p>

Common baud rates range from 110 to 256000. There are equates defined in the [WIN32API.INC](#) file, prefixed with %CBR_ to assist you with specifying a common baud rate, but you are not restricted to a limited set of rates.

Attempting to set a READ-ONLY attribute will result in a compile-time [Error 542](#) ("May not be altered").

The Number symbol (#) prefix is optional, but recommended for the

Purpose Return the command-line arguments used to start the program.

Syntax `s$ = COMMAND$`
`s$ = COMMAND$ (ArgNum)`

Remarks COMMAND\$ returns everything that was typed following the program name. Some operating system manuals refer to this text as the trailer or command tail. You can use COMMAND\$ to collect run-time arguments, like filenames, and program options.

Depending upon the optional argument number, COMMAND\$ will return either the complete trailer, or just one of the arguments. If the *ArgNum* is zero (0), or not present, the complete trailer is returned. If the *ArgNum* is greater than zero, the trailer is parsed to return an individual argument (1 = first argument. 2 = second argument, etc.). If the *ArgNum* is greater than the number of arguments, a null string (zero-length) is returned.

Arguments are delimited by one or more blank spaces. If blank spaces are significant, you should enclose the argument in double quotes ("). Any such double-quotes are stripped from the return value by COMMAND\$. If a zero-length quoted string (") is found, it is ignored entirely.

For example, consider a program named FASTSORT.EXE that reads data from one file, sorts it, and puts the result in a new file. Using COMMAND\$ lets you specify the input and output file names when the program is invoked:

```
FASTSORT.EXE cust.dta cust.new
```

When FASTSORT begins execution, COMMAND\$ or COMMAND\$(0) would return:

```
cust.dta cust.new
```

COMMAND\$(1) would return:

```
cust.dta
```

COMMAND\$(2) would return:

```
cust.new
```

Restrictions In some recent versions of Windows, file association and drag-drop file operations cause filenames to be enclosed with double-quote marks when they are passed in COMMAND\$. It would be wise to ensure that your applications are prepared for this possibility. Some operating systems automatically enclose the command-line in double-quote marks.

Classic PowerBASIC imposes no arbitrary limits on the length of the string returned by COMMAND\$ but, the operating system may impose limits. Such limits may become evident, for example, when attempting to Drag and Drop a large number of files onto an EXE within Windows Explorer.

Usually, attempting to drop more files than the operating system permits will result in an operating system warning message.

Within the [IDE](#), a COMMAND\$ [command-line parameter](#) can be specified for the purposes of testing in both Compile and Execute and Compile and [Debug](#) modes.

See also

[JOIN\\$](#), [PARSE](#), [PARSE\\$](#), [PARSECOUNT](#), [PATHNAME\\$](#), [PATHSCAN\\$](#), [WINMAIN](#)

Example

```
#COMPILE EXE
FUNCTION PBMAIN
  IF TRIM$(COMMAND$) = "" THEN
    EXIT FUNCTION ' No command-line params given, just quit
  ELSEIF INSTR(COMMAND$, "/Q") THEN
    ' Process the /Q option
  ELSEIF INSTR(COMMAND$, "/W") THEN
    ' Process the /W option
  END IF
END FUNCTION
```

CONTROL ADD "custom-control" statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a custom control to a DDT dialog.
Syntax	<code>CONTROL ADD classname\$, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>classname\$</i>	A registered custom control or common control class name, for example, "MSCTLS_STATUSBAR32", etc. <i>classname\$</i> may be a string expression , quoted string literal , or a string equate .
<i>hDlg</i>	Handle of the dialog in which the control will be created.
<i>id&</i>	Unique identifier for the control. Equates are recommended for clarity of the source code.
<i>txt\$</i>	Text to be displayed in the control, if any. <i>txt\$</i> may be a string expression, string, or string constant, and may be zero length.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog.
<i>style&</i>	Primary style of the custom control. There are no default style values for a custom control. Many standard Windows common controls require the %WS_CHILD and %WS_VISIBLE styles to be explicitly specified, or the control may not be visible or function correctly. Please consult the control's documentation for information on its primary and extended styles.
<i>exstyle&</i>	Extended style of the custom control. As with <i>style&</i> above, there are no default extended style values for a custom control - the statement should explicitly include all required primary and extended styles for the control.
<i>callback</i>	Optional name of a Callback Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the custom control. See the #MESSAGES metastatement to choose which messages will be received.

If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the messages are handled by the DDT engine.

Remarks

When the user interacts with the control, a message is sent to the designated Callback Function. If there is no Callback Function designated, the message is sent to the callback for the dialog.

The *style* and *exstyle* values are dependent on the type of custom control or common control being used. The notification messages sent to your callback are also dependent on the type of custom control or common control being used.

When the Callback Function receives a %WM_COMMAND message, the identity of the control sending the message can be found with the [CB.CTL](#) function. Use the [CB.CTLMSG](#) function to retrieve the notification message value in your callback. However, many Windows common controls send %WM_NOTIFY messages (to the parent dialog's callback, not the control callback) rather than the more conventional %WM_COMMAND messages. In such cases, the meaning of the message parameters [CB.WPARAM](#) and [CB.LPARAM](#) will vary according to the type of notification message being processed.

See also

[#MESSAGES](#), [Dynamic Dialog Tools](#), [CONTROL HANDLE](#), [CONTROL SEND](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

CONTROL ADD BUTTON statement

[Top](#) [Previous](#) [Next](#)

Purpose	Add a command button to a dialog. A command button is a button that causes an action to occur when the button is clicked. A common example of a command button is the "OK" button on a message box dialog.
Syntax	<pre>CONTROL ADD BUTTON, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the button will be created. The dialog will become the parent of the command button.
<i>id&</i>	<p>Unique identifier for the button in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%NewAccount</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.</p> <p>However, it is typical for a dialog to include an OK and/or a Cancel button, represented by the predefined equates <code>%IDOK</code> and <code>%IDCANCEL</code> respectively. A button with an ID of <code>%IDOK</code> is triggered (clicked) when the ENTER key is pressed by the user, and a button with the ID of <code>%IDCANCEL</code> is triggered when the ESCAPE key is pressed. These and other predefined "standard" equates can be found in the WIN32API.INC and <code>DDT.INC</code> files.</p>
<i>txt\$</i>	Text to be displayed in the button. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below. OK and Cancel/Close buttons do not usually contain accelerators, since such buttons usually respond to the ENTER and ESCAPE keystrokes, respectively.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the button. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 50 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the button. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.

style&

Primary [style](#) of the button. The default button style comprises %BS_CENTER, %BS_VCENTER, and %WS_TABSTOP. The default style is used if both the primary and extended style parameters are omitted from the statement. For example:

```
CONTROL ADD BUTTON, hDlg, id&, txt$, 100, 100, 150, 200, , , _  
CALL ButtonCallback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary button style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%BS_BOTTOM	Place the text at the bottom of the button.
%BS_CENTER	Center the text horizontally in the button. (default)
%BS_DEFAULT	Create a button with a heavy black border. The user can select this button by pressing the ENTER key. This style is useful for enabling the user to quickly select the most likely option. You can only have one Default button per dialog. It is recommended to make <i>id& = 1</i> , or <i>id& = %IDOK</i> for this control. Synonym of %BS_DEFPUSHBUTTON.
%BS_DEFPUSHBUTTON	Synonym of %BS_DEFAULT.
%BS_FLAT	Create a flat button (without the raised 3D look).
%BS_LEFT	Place the text on the left side of the button.
%BS_MULTILINE	Wrap the caption text across multiple lines, if the text string is too long to fit on a single line. To force a wrap, insert a \$CR (or \$CRLF) into the caption text at the desired wrap position.
%BS_NOTIFY	Enable a button to send the %BN_KILLFOCUS and %BN_SETFOCUS notification messages to the button Callback Function.
%BS_PUSHLIKE	Button state alternates (toggles) between normal (raised) and depressed (sunken) modes.
%BS_RIGHT	Place the text on the right side of the button.
%BS_TOP	Place the text at the top edge of the button.
%BS_VCENTER	Center the text vertically in the button. (default)
%WS_BORDER	Add a thin line border around the control.
%WS_DISABLED	Create a control that is initially disabled. A

disabled control cannot receive input from the user. Use the [CONTROL ENABLE](#) statement to re-enable the button.

%WS_GROUP

Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.

%WS_TABSTOP

Allow button control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

exstyle&

Extended style of the button control. The default extended button style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended style parameters are omitted from the CONTROL ADD BUTTON statement, in the same manner as *style&* above.

The extended button style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_LEFT

The button has generic "left-aligned" properties. (default)

%WS_EX_RIGHT

The button has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_TRANSPARENT

Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly "click" a control, simply by pressing and holding the ALT key while pressing the specified hot-key. For example, "E&xit" makes ALT+x the hot-key.

On Windows XP and Windows 2000 you may need to press the ALT key before Control Accelerators are made visible. You can set if Command Accelerators are visible when using the ALT key or all the time in the Windows Display Settings.

Unless the %BS_FLAT style is used, the button is drawn on the dialog using a 3-dimensional look. When the user clicks a button, a message is sent to the Callback Function designated for the button. If there is no Callback Function designated, the message is sent to the callback for the dialog.

In general, if the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE, if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Notification messages are sent to the Callback Function, with [CB.MSG](#) = %WM_COMMAND, [CB.CTL](#) holding the ID (id&) of the control, and [CB.CTLMSG](#) holding the following values:

%BN_CLICKED Sent when the user clicks a mouse button, or activates the button with the hot-key (unless the button has been disabled).

%BN_DISABLE Sent when a button is disabled.

%BN_KILLFOCUS Sent when a button loses the keyboard focus. The button must include the %BS_NOTIFY style.

%BN_SETFOCUS Sent when a button receives the keyboard focus. The button must include the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL GET TEXT](#), [CONTROL SET FONT](#), [CONTROL SET TEXT](#)

CONTROL ADD CHECK3STATE statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add an auto 3-state checkbox to a dialog. This is commonly used to indicate a selection that may be True (set or checked), False (unset or cleared) or Indeterminate (grayed), and is often found in dialogs that provide "multiple choice" options.
Syntax	<code>CONTROL ADD CHECK3STATE, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>hDlg</i>	Handle of the dialog in which the 3-state checkbox will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the <code>%AutoLogoff</code> equate is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to be displayed in the 3-state checkbox. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.
<i>style&</i>	Primary style of the 3-state checkbox control. The default 3-state checkbox style comprises <code>%BS_LEFT</code> , <code>%BS_VCENTER</code> , and <code>%WS_TABSTOP</code> . The default style is used only if both the primary and extended style parameters are omitted from the statement. For example: <pre>CONTROL ADD CHECK3STATE, hDlg, id&, txt\$, 100, 100, 40, 14, , , _ CALL Check3Callback() ' Use default styles</pre> Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all

necessary primary and extended style parameters.

The primary 3-state checkbox style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%BS_BOTTOM	Place the text at the bottom of the control.
%BS_CENTER	Center the text horizontally in the control.
%BS_FLAT	Create a flat control (without the raised 3D look).
%BS_LEFT	Place the text on the left side of the checkbox. Also see %BS_LEFTTEXT. (default)
%BS_LEFTTEXT	Place the checkbox to the right of the text portion of the control. Combine with %BS_RIGHT to right-align text against the left side of the checkbox control.
%BS_MULTILINE	Wrap the caption text across multiple lines, if the text string is too long to fit on a single line. To force a wrap, insert a \$CR (or \$CRLF) into the caption text at the desired wrap position.
%BS_NOTIFY	Enable a control to send the %BN_KILLFOCUS and %BN_SETFOCUS messages to the callback .
%BS_PUSHLIKE	Button state alternates (toggles) between normal (raised) and depressed (sunken) modes.
%BS_RIGHT	Place the text on the right side of the checkbox. Also see %BS_LEFTTEXT.
%BS_TOP	Place the text at the top of the control.
%BS_VCENTER	Center the text vertically in the control. (default)
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.
%WS_TABSTOP	Allow the 3-state checkbox to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style,

and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

exstyle&

Extended style of the 3-state checkbox control. The default extended 3-state checkbox style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended style parameters are omitted from the CONTROL ADD CHECK3STATE statement, in the same manner as style& above.

The extended 3-state checkbox style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

In general, if the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Remarks

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly "click" a control, simply by pressing and holding the ALT key while pressing the specified hot-key. For example, "Set s&tate" makes ALT+t the hot-key.

When the user clicks a 3-state checkbox, a message is sent to the Callback Function designated for the control. If there is no Callback Function designated, the message is sent to the callback for the dialog.

If the control callback processes the notification message, it should return TRUE (non-zero) to prevent the message being passed needlessly to the dialog callback, and eventually to the DDT engine itself.

Notification messages are sent to the Callback Function, with [CB.MSG](#) = %WM_COMMAND, [CB.CTL](#) holding the ID (id&) of the control, and [CB.CTLMSG](#) holding the following values:

%BN_CLICKED Sent when the user clicks a mouse button, or activates the control with the hot-key (unless the control has been disabled).

%BN_DISABLE Sent when a control is disabled.

%BN_KILLFOCUS Sent when a control loses the keyboard focus. The control must include the %BS_NOTIFY style.

%BN_SETFOCUS Sent when a control receives the keyboard focus. The control must include the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD CHECKBOX](#), [CONTROL ADD OPTION](#), [CONTROL GET CHECK](#), [CONTROL SET CHECK](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

CONTROL ADD CHECKBOX statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add an auto-checkbox to a dialog. This is typically used to indicate a True/False or on/off selection, and is common in dialogs that offer choices of options to a user.
Syntax	<pre>CONTROL ADD CHECKBOX, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the checkbox will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <i>%DisableUser</i> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to be displayed next to the checkbox. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.
<i>style&</i>	Primary style of the checkbox control. The default checkbox style comprises %BS_LEFT, %BS_VCENTER, and %WS_TABSTOP. The default style is used only if both the primary and extended parameters are omitted from the statement. For example: <pre>CONTROL ADD CHECKBOX, hDlg, id&, txt\$, 100, 100, 40, 14, , , _ CALL CheckboxCallback() ' Use default styles</pre> Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary checkbox style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%BS_BOTTOM	Place the text at the bottom of the control.
%BS_CENTER	Center the text horizontally in the control.
%BS_LEFT	Place the text on the left side of the label portion of the control. Also see %BS_LEFTTEXT. (default)
%BS_LEFTTEXT	Place the checkbox to the right of the text portion of the control. Combine with %BS_RIGHT to right-align text against the left side of the checkbox control.
%BS_MULTILINE	Wrap the caption text across multiple lines, if the text string is too long to fit on a single line. To force a wrap, insert a \$CR (or \$CRLF) into the caption text at the desired wrap position.
%BS_NOTIFY	Enable a control to send the %BN_KILLFOCUS and %BN_SETFOCUS messages to the callback .
%BS_PUSHLIKE	Button state alternates (toggles) between normal (raised) and depressed (sunken) modes.
%BS_RIGHT	Place the text on the right side of the label portion of the control. Also see %BS_LEFTTEXT.
%BS_TOP	Place the text at the top of the control.
%BS_VCENTER	Center the text vertically in the control. (default)
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.
%WS_TABSTOP	Allow checkbox control to receive the keyboard focus when the user presses the TAB and SHIFT+TAB keys. Pressing the TAB key changes the keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB moves it to the previous control with %WS_TABSTOP. (default)

exstyle&	<p>Extended style of the checkbox control. The default extended checkbox style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD CHECKBOX statement, in the same manner as style& above.</p> <p>The extended checkbox style value can be a combination of any values below, combined together with the OR operator to form a bitmask:</p> <p>%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.</p> <p>%WS_EX_LEFT The control has generic "left-aligned" properties. (default)</p> <p>%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.</p> <p>%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).</p> <p>%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.</p> <p>%WS_EX_WINDOWEDGE Apply a raised edge border to the control.</p>
callback	<p>Optional name of a Callback Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the #MESSAGES metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.</p> <p>In general, when the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.</p>
Remarks	<p>If the ampersand (&) character appears in the <i>txt\$</i> parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly "click" a control, simply by pressing</p>

and holding the ALT key while pressing the specified hot-key. For example, "O&ption " makes ALT+p the hot-key.

When the user clicks a control, a message is sent to the Callback Function designated for the control. If there is no Callback Function designated, the message is sent to the callback for the dialog.

If the control callback processes the notification message, it should return TRUE (non-zero) to prevent the message being passed needlessly to the dialog callback, and eventually to the DDT engine itself.

Notification messages are sent to the Callback Function, with [CB.MSG](#) = %WM_COMMAND, [CB.CTL](#) holding the ID (id&) of the control, and [CB.CTLMSG](#) holding the following values:

[%BN_CLICKED](#) Sent when the user clicks a mouse button or activates the control with the hot-key (unless the control has been disabled).

[%BN_DISABLE](#) Sent when a control is disabled.

[%BN_KILLFOCUS](#) Sent when a control loses the keyboard focus. The control must include the %BS_NOTIFY style.

[%BN_SETFOCUS](#) Sent when a control receives the keyboard focus. The control must include the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD CHECK3STATE](#), [CONTROL ADD OPTION](#), [CONTROL GET CHECK](#), [CONTROL SET CHECK](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

CONTROL ADD COMBOBOX statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a combo box to a dialog. A combo box is often used to allow a user to select an item from a predefined list, or enter a fresh (unlisted) item. A combo box may contain only text strings. To put numbers in a combo box, convert them to strings with the FORMAT\$, USING\$, or STR\$ functions.
Syntax	<pre>CONTROL ADD COMBOBOX, hDlg, id&, [items\$()], x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the combo box will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%StockNumberList</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers
<i>items\$()</i>	<p>Optional dynamic (variable length) string array, containing the initial items to be displayed in the combo box. Items are copied from the array to the combo box, starting at the lowest subscript of the array (LBOUND), continuing on toward the end of the array, until an empty string is encountered, or the highest subscript is reached. If an array with an LBOUND of zero (the default) is specified, be sure that the 1st element (0) contains data.</p> <p>To create a combo box that is initially empty, either omit this parameter, or specify an array whose first element contains an empty string. If the combo box uses the <code>%CBS_SORT</code> style, the items are sorted alphanumerically as they are added to the combo box.</p>
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is around 100 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog

Editor and Visual Studio is 40 dialog units.

style&

Primary [style](#) of the control.

There are three types of combo boxes: **simple**, **dropdown**, and **dropdownlist**. A **simple** combo box consists of a text box control and a list box; the list box is always displayed. A **dropdown** combo box consists of a text box control and a list box; the list box is not displayed unless the user clicks an icon. A **dropdownlist** combo box consists of a label control (not editable) and a list box; the list box is not displayed unless the user clicks an icon.

Combo box style	List box control	Text box control
Simple	No	Yes
Dropdown (default)	Yes	Yes
Dropdownlist	Yes	No

Note that some styles of combo box are mutually exclusive. In other words, you cannot combine certain styles that may conflict with one another. For example, you cannot specify %CBS_SIMPLE and %CBS_DROPDOWN at the same time.

The default combo box style comprises %CBS_DROPDOWN, %CBS_SORT, and %WS_TABSTOP. The default style is used only if both the primary and extended style parameter values are omitted from the statement. For example:

```
CONTROL ADD COMBOBOX, hDlg, id&, txt$(), 100, 100, 100, 40, , , CALL  
ComboCallback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary combo box style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

- %CBS_AUTOHSCROLL

Automatically scroll the text in the text box to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.
- %CBS_DISABLENOSCROLL

Show a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items.
- %CBS_DROPDOWN

Similar to %CBS_SIMPLE, except that the

	list box is not displayed unless the user selects the icon next to the edit control. (default)
%CBS_DROPDOWNLIST	Similar to %CBS_DROPDOWN, except that the text box is replaced by a (non-editable) label item that displays the current selection in the list box.
%CBS_HASSTRINGS	The combo box will contain strings. (persistent)
%CBS_LOWERCASE	Convert to lowercase any uppercase characters entered into the text box control portion of the combo box.
%CBS_NOINTEGRALHEIGHT	Create the list box portion of the combo box with exactly the size specified by the CONTROL ADD COMBOBOX statement. Without this style, Windows reduces the height of the list box portion of the combo box so that it does not display any partial (clipped) items.
%CBS_SIMPLE	Display the list box at all times. The current selection in the list box is displayed in the text box.
%CBS_SORT	Automatically sorts strings added to the combo box. (default)
%CBS_UPPERCASE	Convert any characters entered into the text box of a combo box into uppercase.
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user. Use the CONTROL ENABLE statement to re-enable a disabled control.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group.
%WS_TABSTOP	Allow combo box control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and

SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

%WS_VSCROLL

Allow the control to display a vertical scroll bar if the list is longer than the height of the combo box. Use in conjunction with %CBS_DISABLENOSCROLL to make the scroll bar visible at all times.

Do not intermix list box styles with similarly named combo box styles as the numeric values of similar styles can produce unexpected results. For example, %LBS_SORT = &H2 and %CBS_SORT = &H100. Combo box styles are prefixed with %CBS.

exstyle&

Extended style of the combo box control. The default extended combo box style comprises %WS_EX_LEFT, and %WS_EX_CLIENTEDGE. The default extended style is only used if both the primary and extended parameters are omitted from the CONTROL ADD COMBOBOX statement, in the same manner as *style&* above.

The extended combo box style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control. (default)

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND

and `%WM_NOTIFY` messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

In general, when the control Callback Function processes a message, it should return `TRUE` (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return `TRUE` if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Remarks

When the user selects an item or edits the text of a combo box, a message is sent to the Callback Function designated for the combo box. If there is no Callback Function designated then the message is sent to the callback for the dialog.

If the control callback processes the notification message, it should return `TRUE` (non-zero) to prevent the message being passed needlessly to the dialog callback, and eventually to the [DDT](#) engine itself.

Notification messages are sent to the Callback Function, with [CB.MSG](#) = `%WM_COMMAND`, [CB.CTL](#) holding the ID (id&) of the control, and [CB.CTLMSG](#) holding the following values:

<code>%CBN_CLOSEUP</code>	Sent when the list box of a combo box has been closed.
<code>%CBN_DBLCLK</code>	Sent when the user double-clicks a string in the list box of a combo box.
<code>%CBN_DROPDOWN</code>	Sent when the list box of a combo box is about to be made visible.
<code>%CBN_EDITCHANGE</code>	Sent after the user has taken an action that may have altered the text in the text box portion of a combo box. Unlike the <code>%CBN_EDITUPDATE</code> notification message, this notification message is sent after Windows updates the screen.
<code>%CBN_EDITUPDATE</code>	Sent when the text box portion of a combo box is about to display altered text. This notification message is sent after the control has formatted the text, but before it displays the text.
<code>%CBN_ERRSPACE</code>	Sent when a combo box cannot allocate enough memory to meet a specific request.
<code>%CBN_KILLFOCUS</code>	Sent when a combo box loses the keyboard focus.
<code>%CBN_SELCHANGE</code>	Sent when the selection in the list box of a combo box is about to be changed, as a result of the user

either clicking in the list box or changing the selection by using the arrow keys.

- | | |
|----------------|--|
| %CBN_SELCANCEL | Sent when the user selects an item, but then selects another control or closes the dialog box. It indicates the user's initial selection is to be ignored. |
| %CBN_SELENDOK | Sent when the user selects a list item, or selects an item and then closes the list. It indicates that the user's selection is to be processed. |
| %CBN_SETFOCUS | Sent when a combo box receives the keyboard focus. |

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [COMBOBOX](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

Purpose	Add a frame to a dialog. This is also known as a "group" control, and is typically drawn around controls to indicate a visual association between such controls. A frame control is often used around related Option controls.
Syntax	<code>CONTROL ADD FRAME, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]]</code>
<i>hDlg</i>	Handle of the dialog in which the frame will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%RelatedItems</code> is more informative than a literal value such as 497. If you will not be changing the text in a frame control after it is created, you may use -1 for the <i>id&</i> ; however, best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to be displayed in the frame. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.
<i>style&</i>	<p>Primary style of the frame control. The default frame style comprises <code>%BS_LEFT</code>, and <code>%BS_TOP</code>. The default style is used only if both the primary and extended parameters are omitted from the statement. For example:</p> <pre>CONTROL ADD FRAME, hDlg, id&, txt\$, 100, 100, 40, 14, , ' Use default styles</pre> <p>Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary</p>

properties. (default)

`%WS_EX_RIGHT`

The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

`%WS_EX_STATICEDGE`

Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

`%WS_EX_TRANSPARENT`

Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

`%WS_EX_WINDOWEDGE`

Apply a raised edge border to the control.

Remarks

A frame control does not send messages to its parent dialog and does not require or support a [Callback](#).

See also

[Dynamic Dialog Tools](#), [CONTROL_GET_TEXT](#), [CONTROL_SET_COLOR](#), [CONTROL_SET_FONT](#), [CONTROL_SET_TEXT](#)

CONTROL ADD GRAPHIC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Add a static graphic control to a dialog for pictures, text, etc.		
Syntax	CONTROL ADD GRAPHIC, <i>hDlg</i> , <i>id&</i> , "", <i>x&</i> , <i>y&</i> , <i>wide&</i> , <i>high&</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]]] [[,] CALL <i>callback</i>]		
<i>hDlg</i>	Handle of the dialog in which the graphic control will be created. The dialog will become the parent of the control.		
<i>id&</i>	Unique identifier for the image in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%IDC_GRAPHIC1</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.		
""	This parameter is not currently used with graphic controls, but is reserved for future implementation. Any data placed at this position is ignored.		
<i>x</i> , <i>y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.		
<i>wide&</i>	Integer expression, variable, or numeric literal value, specifying the width of the image. The width is given in the same terms (pixels or dialog units) as the parent dialog.		
<i>high&</i>	Integer expression, variable, or numeric literal value, specifying the height of the image. The height is given in the same terms (pixels or dialog units) as the parent dialog.		
<i>style&</i>	Optional primary style of the image control. This value can be a combination of the values below, combined together with the OR operator to form a bitmask. If <i>style&</i> is omitted, the default combination is <code>%WS_CHILD OR %WS_VISIBLE OR %SS_OWNERDRAW</code> .		
	<code>%SS_NOTIFY</code>	Send <code>%STN_CLICKED</code> and <code>%STN_DBLCLK</code> notification messages to the Callback Function when the user clicks or double-clicks the control.	
	<code>%SS_SUNKEN</code>	Draw a half-sunken border around the graphic control.	
	<code>%WS_BORDER</code>	Add a thin line border around the graphic control.	
	<code>%WS_DLGFRAME</code>	Create a graphic control that has a border of the style typically used with dialog boxes.	
<i>exstyle&</i>	Optional extended style of the graphic control. This value can be a		

combination of the values below, combined together with the OR operator to form a bitmask. If `exstyle&` is omitted, there is no default extended style.

`%WS_EX_CLIENTEDGE` Apply a sunken edge border to the control.

`%WS_EX_STATICEDGE` Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

callback

Optional name of a Callback Function that receives all `%WM_COMMAND` and `%WM_NOTIFY` messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return `TRUE` (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return `TRUE` if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

A graphic control is typically used with graphic statements to draw graphs, pictures, text, etc. After you create a graphic control, you must use [GRAPHIC ATTACH](#) to choose it for use with other statements and functions.

A graphic control will only send notification messages to a callback if the `%SS_NOTIFY` style is used. Notification messages are sent to the callback function with [CB.MSG](#) = `%WM_COMMAND`, [CB.CTL](#) holding the ID (*id&*) of the control, and [CB.CTLMSG](#) holding one of the following values:

`%STN_CLICKED` Sent when the user clicks a mouse button on the graphic control (unless the image control has been disabled).

`%STN_DBLCLK` Sent when the user double-clicks on a graphic control (unless the control has been disabled).

`%STN_DISABLE` Sent when a graphic control has been disabled.

`%STN_ENABLE` Sent when a graphic control has been enabled.

When a callback function receives a `%WM_COMMAND` message, it should explicitly test the value of `CB.CTL` and `CB.CTLMSG` to guarantee it is responding appropriately to the notification message.

All Classic PowerBASIC graphical displays are persistent -- they will be automatically redrawn when altered or temporarily covered by another window. Due to the nature of a graphic control and its contents, it may not

be resized. If a new size is needed, the control should be destroyed and recreated.

See also

[Dynamic Dialog Tools](#), [GRAPHIC ATTACH](#), [GRAPHIC COLOR](#), [GRAPHIC SCALE](#), [GRAPHIC SET FONT](#), [GRAPHIC STYLE](#), [GRAPHIC WIDTH](#), [GRAPHIC WINDOW](#)

Purpose	Add a (non-resizing) image control to a dialog. This is typically used to display a bitmap or icon stored in a resource file (.PBR file).
Syntax	<pre>CONTROL ADD IMAGE, hDlg, id&, image\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the image will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the image in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%WizardBMP</code> is more informative than a literal value such as 497. If you will not be changing the image in the control after it is created, you may use -1 for the <code>id&</code> ; however, best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>image\$</i>	Name of the bitmap or icon in the resource file (.PBR file). If the image resource uses an integer identifier, <code>image\$</code> should begin with a Number symbol (#) followed by the identifier in an ASCII format, e.g., <code>"#998"</code> or <code>FORMAT\$(rcid&, "\###")</code> . Otherwise, use the text identifier name for the image.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <code>x</code> is the horizontal position, and <code>y</code> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the image. The width is given in the same terms (pixels or dialog units) as the parent dialog. This value is ignored unless the <code>%SS_CENTERIMAGE</code> style is specified.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the image. The height is given in the same terms (pixels or dialog units) as the parent dialog. This value is ignored unless the <code>%SS_CENTERIMAGE</code> style is specified.
<i>style&</i>	Primary style of the image control. This value can be a combination of the values below, combined together with the OR operator to form a bitmask. In addition, the initial image format may be specified explicitly as either <code>%SS_ICON</code> or <code>%SS_BITMAP</code> , or the image format may be omitted completely. If the image format is specified, it must match the format of the file specified in <i>image\$</i> . However, if the image format is not specified, Classic

PowerBASIC will examine the file to determine the correct image format to use.

%SS_BITMAP	Display only bitmap images. Also see %SS_ICON. (persistent)
%SS_CENTERIMAGE	If the image is smaller than the label, fill the rest of the label with the color of the pixel in the top left corner of the image.
%SS_ICON	Display only icon images. Also see %SS_ICON. (persistent)
%SS_NOTIFY	Send %STN_CLICKED and %STN_DBLCLK notification messages to the Callback Function when the user clicks or double-clicks the control.
%SS_SUNKEN	Draw a half-sunken border around the image control.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.

exstyle&

Extended style of the image control. The default extended image control style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD IMAGE statement, in the same manner as *style&* above.

The extended image control style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

callback

Optional name of a Callback Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

In general, when the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

The bitmap or icon used in the image is not resized to fit the control. If your control is 64 dialog units wide and your icon or bitmap is only 32, half of the image will be blank. For best results, icons should be 32x32 pixels.

Once an image control has been created, the images it displays can be changed with the [CONTROL SET IMAGE](#) statement, but only if the images are of the same format as the original. For example, if an image control was initially created showing a bitmap file, all subsequent image changes must also be bitmap images. However, if the image format must be changed at run-time, for example, because icons are to be displayed instead of bitmaps, there are a couple of options. For example, the application could use separate controls for each image format, or the existing control could be destroyed, and a new control created with an image of the opposite format.

An image control will only send notification messages to a callback if the %SS_NOTIFY style is used. Notification messages are sent to the Callback Function with [CB.MSG](#) = %WM_COMMAND, [CB.CTL](#) holding the ID (*id*&) of the control, and [CB.CTLMSG](#) holding the following values:

%STN_CLICKED Sent when the user clicks a mouse button on the image control (unless the image control has been disabled).

%STN_DBLCLK	Sent when the user double-clicks on an image control (unless the control has been disabled).
%STN_DISABLE	Sent when an image control has been disabled.
%STN_ENABLE	Sent when an image control has been enabled.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGEX](#), [CONTROL ADD IMGBUTTON](#), [CONTROL ADD IMGBUTTONX](#), [CONTROL SET IMAGE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTON](#), [CONTROL SET IMGBUTTONX](#)

Purpose	Add a stretched image control to a dialog. This is typically used to display bitmaps and icons, which are automatically stretched or shrunk to fill the controls client area.
Syntax	<pre>CONTROL ADD IMAGEX, hDlg, id&, image\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the image will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the image in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%BackgroundIMG</code> is more informative than a literal value such as 497. If you will not be changing the image in the control after it is created, you may use -1 for the <i>id&</i> ; however, best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>image\$</i>	Name of the bitmap or icon in the resource file (.PBR file). If the image resource uses an integer identifier, <i>image\$</i> should begin with a Number symbol (#) followed by the identifier in an ASCII format, e.g., <code>"#998"</code> or <code>FORMAT\$(rcid&, "\###")</code> . Otherwise, use the text identifier name for the image.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the image. The width is given in the same terms (pixels or dialog units) as the parent dialog.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the image. The height is given in the same terms (pixels or dialog units) as the parent dialog.
<i>style&</i>	Primary style of the stretched image control. In addition to the image control styles listed below, the initial image format may be specified explicitly as either <code>%SS_ICON</code> or <code>%SS_BITMAP</code> , or you may choose not to specify the image format at all. If the image format is specified, it must match the format of the file specified in <i>image\$</i> . However, if the image format is not specified, Classic PowerBASIC will examine the file to determine the correct image format to use.

This value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%SS_BITMAP	Display only bitmap images. Also see %SS_ICON. (persistent)
%SS_ICON	Display only icon images. Also see %SS_ICON. (persistent)
%SS_NOTIFY	Send %STN_CLICKED and %STN_DBLCLK notification messages to the Callback Function when the user clicks or double-clicks the control.
%SS_SUNKEN	Draw a half-sunken border around the image control.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.

exstyle&

Extended style of the stretched image control. The default extended image style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD IMAGEX statement, in the same manner as *style&* above. The extended stretched image style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_CLIENTEDGE	Apply a sunken edge border to the control.
%WS_EX_LEFT	The control has generic "left-aligned" properties. (default)
%WS_EX_RIGHT	The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment.
%WS_EX_STATICEDGE	Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).
%WS_EX_TRANSPARENT	Controls/windows beneath the control are drawn before the control is drawn. The control

is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

In general, when the control Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Remarks

The bitmap or icon used in the image is resized to fit the control. If your control is 64 dialog units wide and your icon or bitmap is only 32, it will be stretched to cover the entire control. For best results, icons should be 32x32 pixels.

An image control will only send notification messages to a callback if the %SS_NOTIFY style is used. Notification messages are sent to the Callback Function with [CB.MSG](#) = %WM_COMMAND, [CB.CTL](#) holding the ID (*id&*) of the control, and [CB.CTLMSG](#) holding the following values:

%STN_CLICKED	Sent when the user clicks a mouse button on the image control (unless the image control has been disabled).
%STN_DBLCLK	Sent when the user double-clicks on an image control (unless the control has been disabled).
%STN_DISABLE	Sent when an image control has been disabled.
%STN_ENABLE	Sent when an image control has been enabled.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

Restrictions

Under Windows 95/98/ME, an attempt to stretch an icon significantly above 64x64 may fail due to internal limits that vary between those particular versions of Windows. Bitmaps are not affected in this manner. Windows NT/2000/XP systems do not impose any comparable limitations on either icons or bitmaps.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGE](#), [CONTROL ADD IMGBUTTON](#), [CONTROL ADD IMGBUTTONX](#), [CONTROL SET IMAGE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTON](#), [CONTROL SET IMGBUTTONX](#)

CONTROL ADD IMGBUTTON statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add an image button to a dialog. Image buttons are often used to enhance the appearance of a dialog.
Syntax	<pre>CONTROL ADD IMGBUTTON, <i>hDlg</i>, <i>id&</i>, <i>image\$</i>, <i>x</i>, <i>y</i>, <i>xx</i>, <i>yy</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]]] [[,] CALL <i>callback</i>]</pre>
<i>hDlg</i>	Handle of the dialog in which the button will be created. The dialog will become the parent of the control.
<i>id&</i>	<p>Unique identifier for the button in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%IconButton1</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.</p> <p>However, it is typical for a dialog to include an OK and/or a Cancel button, represented by the predefined equates <code>%IDOK</code> and <code>%IDCANCEL</code> respectively. A button with an ID of <code>%IDOK</code> is triggered (clicked) when the ENTER key is pressed by the user, and a button with the ID of <code>%IDCANCEL</code> is triggered when the ESCAPE key is pressed. These and other predefined "standard" equates can be found in the WIN32API.INC and <code>DDT.INC</code> files.</p>
<i>image\$</i>	Name of the bitmap or icon in the resource file (.PBR file). If the image resource uses an integer identifier, <i>image\$</i> should begin with a Number symbol (#) followed by the identifier in an ASCII format, e.g., <code>"#998"</code> . Otherwise, use the text identifier name for the image.
<i>x</i> , <i>y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.

style&

Primary [style](#) of the image button control. The default image button style is %WS_TABSTOP. The default style is used only if both the primary and extended parameters are omitted from the statement. For example:

```
CONTROL ADD IMGBUTTON, hDlg, id&, txt$, 100, 100, 150, 200, , , _  
CALL ImgButtonCallback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary image button style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%BS_DEFAULT	Create the button with a heavy black border. The user can select this button by pressing the ENTER key. This style is useful for enabling the user to quickly select the most likely option. There may only be one Default button per dialog.
%BS_FLAT	Create a flat button (without the raised 3D look).
%BS_NOTIFY	Enable a button to send the %BN_KILLFOCUS and %BN_SETFOCUS notification messages to the button Callback Function.
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user. Use the CONTROL ENABLE statement to re-enable the button.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.
%WS_TABSTOP	Allow button control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

exstyle&

Extended style of the image button control. The default extended image

button style comprises %WS_EX_LEFT. The default extended style is only used if both the primary and extended parameters are omitted from the CONTROL ADD IMGBUTTON statement, in the same manner as *style*& above.

The extended image button style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_LEFT	The button has generic "left-aligned" properties. (default)
%WS_EX_RIGHT	The button has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.
%WS_EX_TRANSPARENT	Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

Generally speaking, if the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Remarks

The bitmap or icon used in the button is not resized to fit the button. If your button is 64 dialog units wide and your icon or bitmap is only 32, half of the button will be blank. For best results, icons should be 32x32 pixels.

An image button is drawn on the dialog using a 3-dimensional look, unless the %BS_FLAT style is specified. When the user clicks on the image button, a message is sent to the button's Callback Function. If there is no Callback Function designated, the message is sent to the callback for the dialog.

Notification messages are sent to the Callback Function with [CB.MSG](#) =

%WM_COMMAND, [CB.CTL](#) holding the ID (*id&*) of the control, and [CB.CTLMSG](#) holding the following values:

%BN_CLICKED	Sent when the user clicks a mouse button, or activates the button with the hot-key (unless the button has been disabled).
%BN_DISABLE	Sent when a button is disabled.
%BN_KILLFOCUS	Sent when a button loses the keyboard focus. The button must include the %BS_NOTIFY style.
%BN_SETFOCUS	Sent when a button receives the keyboard focus. The button must include the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGE](#), [CONTROL ADD IMAGEX](#), [CONTROL ADD IMGBUTTONX](#), [CONTROL SET IMAGE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTON](#), [CONTROL SET IMGBUTTONX](#)

CONTROL ADD IMGBUTTONX statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a stretched image button to a dialog. Stretched image buttons are often used to enhance the appearance of a dialog, with the image being automatically stretched or shrunk to fill the control.
Syntax	<pre>CONTROL ADD IMGBUTTONX, hDlg, id&, image\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the button will be created. The dialog will become the parent of the control.
<i>id&</i>	<p>Unique identifier for the button in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%IconButton2</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.</p> <p>However, it is typical for a dialog to include an OK and/or a Cancel button, represented by the predefined equates <code>%IDOK</code> and <code>%IDCANCEL</code> respectively. A button with an ID of <code>%IDOK</code> is triggered (clicked) when the ENTER key is pressed by the user, and a button with the ID of <code>%IDCANCEL</code> is triggered when the ESCAPE key is pressed. These and other predefined "standard" equates can be found in the WIN32API.INC and DDT.INC files.</p>
<i>image\$</i>	Name of the bitmap or icon in the resource file (.PBR). If the image resource uses an integer identifier, image\$ should begin with a Number symbol (#) followed by the identifier in an ASCII format, e.g., <code>"#998"</code> . Otherwise, use the text identifier name for the image.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.

style&

Primary [style](#) of the stretched image button. The default image button style is %WS_TABSTOP. The default style is used if both the primary and extended style parameters are omitted from the statement. For example:

```
CONTROL ADD IMGBUTTONX, hDlg, id&, txt$, 100, 100, 150, 200, , , _  
CALL ImgButtonxCallback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary stretched image button style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%BS_DEFAULT	Create the button with a heavy black border. The user can select this button by pressing the ENTER key. This style is useful for enabling the user to quickly select the most likely option. There may only be one Default button per dialog.
%BS_FLAT	Create a flat button (without the raised 3D look).
%BS_NOTIFY	Enable a button to send the %BN_KILLFOCUS and %BN_SETFOCUS notification messages to the button Callback Function.
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user. Use the CONTROL ENABLE statement to re-enable the button.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.
%WS_TABSTOP	Allow button control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

exstyle&

Extended style of the stretched image button control. The default extended

button style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD IMGBUTTONX statement, in the same manner as *style*& above.

The extended stretched image style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_LEFT	The button has generic "left-aligned" properties. (default)
%WS_EX_RIGHT	The button has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.
%WS_EX_TRANSPARENT	Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.

Remarks

The bitmap or icon used in the button is resized to fit the button. If your button is 64 dialog units wide and your icon or bitmap is only 32, it will be stretched to cover the entire button. For best results, icons should be 32x32 pixels.

The image button is drawn on the dialog using a 3-dimensional look, unless the %BS_FLAT style is specified. When the user clicks a button, a message is sent to the Callback Function designated for the button. If there is no Callback Function designated, the message is sent to the callback for the dialog.

Notification messages are sent to the Callback Function with [CB.MSG](#) =

%WM_COMMAND, [CB.CTL](#) holding the ID (*id&*) of the control, and [CB.CTLMSG](#) holding the following values:

%BN_CLICKED	Sent when the user clicks a mouse button, or activates the button with the hot-key (unless the button has been disabled).
%BN_DISABLE	Sent when a button is disabled.
%BN_KILLFOCUS	Sent when a button loses the keyboard focus. The button must include the %BS_NOTIFY style.
%BN_SETFOCUS	Sent when a button receives the keyboard focus. The button must include the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGE](#), [CONTROL ADD IMAGEX](#), [CONTROL ADD IMGBUTTON](#), [CONTROL SET IMAGE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTON](#), [CONTROL SET IMGBUTTONX](#)

Purpose	Add a text label to a dialog. A text label is similar to a conventional static control.
Syntax	<code>CONTROL ADD LABEL, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>hDlg</i>	Handle of the dialog in which the label will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%BlockTitle</code> is more informative than a literal value such as 497. If you will not be changing the text in a line control after it is created, you may use -1 for the <i>id&</i> ; however, best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to be displayed in text label. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 8 dialog units.
<i>style&</i>	<p>Primary style of the label control. The default label style is <code>%SS_LEFT</code>. The default style is used if both the primary and extended style parameters are omitted from the statement. For example:</p> <pre>CONTROL ADD LABEL, hDlg, id&, txt\$, 100, 100, 150, 200, , , _ CALL LabelCallback() ' Use default styles</pre> <p>Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary label style value can be a combination of any values below, combined together with the OR operator to form a bitmask:</p>

%SS_CENTER	Horizontally center the caption text. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next centered line.
%SS_CENTERIMAGE	Vertically center the caption text. The text is not wrapped even if it extends beyond the width of the control.
%SS_ENDELLIPSIS	Replace the end of the given with ellipsis as needed to fit the result in the specified rectangle. Windows NT/2000/XP only.
%SS_ETCHEDFRAME	Draw the frame of the control using an etched edge style.
%SS_ETCHEDHORZ	Draw the horizontal edges of the control using an etched edge style.
%SS_ETCHEDVERT	Draw the vertical edges of the control using an etched edge style.
%SS_LEFT	Left-align the given text. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next left-aligned line. (default)
%SS_NOPREFIX	Prevent interpretation of ampersand (&) characters in the label text as control accelerator prefix characters. These are normally displayed with the ampersand removed and the next character in the string underscored.
%SS_NOTIFY	Send %STN_CLICKED and %STN_DBLCLK notification messages to the Callback Function when the user clicks or double-clicks the control.
%SS_NOWORDWRAP	Left-align the given text. Tabs are expanded but words are not wrapped. Text that extends past the end of a line is clipped.
%SS_PATHELLIPSIS	Replace the file path portion of the given string with ellipsis as needed to fit the result in the specified rectangle. Windows 2000/XP only.
%SS_RIGHT	Right-align the given text. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to

	the beginning of the next right-aligned line.
<code>%SS_SIMPLE</code>	The caption text is left-aligned. If the control is colored, color is only applied to the region containing the caption text, and the remainder of the control is drawn in standard colors.
<code>%SS_SUNKEN</code>	Draw a half-sunken border around the label control.
<code>%SS_WORDELLIPSIS</code>	Truncate text that does not fit, adding ellipsis as needed. Windows NT/2000/XP only
<code>%WS_GROUP</code>	Define the start of a group of controls. The first control in each group should also use <code>%WS_TABSTOP</code> style. The next <code>%WS_GROUP</code> control in the tab order defines the end of this group and the start of a new group.

exstyle&

Extended style of the label control. The default extended label style comprises `%WS_EX_LEFT`. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD LABEL statement, in the same manner as *style&* above.

The extended label style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

<code>%WS_EX_CLIENTEDGE</code>	Apply a sunken edge border to the control.
<code>%WS_EX_LEFT</code>	The control has generic "left-aligned" properties. (default)
<code>%WS_EX_RIGHT</code>	The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.
<code>%WS_EX_STATICEDGE</code>	Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).
<code>%WS_EX_TRANSPARENT</code>	Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.
<code>%WS_EX_WINDOWEDGE</code>	Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly "click" the control that immediately follows in the Tab-Order after the Label control, simply by pressing and holding the ALT key while pressing the specified hot-key. For example, "Choose &Security Level" makes ALT+S the hot-key.

A label control will only send messages to a callback if the %SS_NOTIFY style is used. The following notifications are sent to the Callback Function:

%STN_CLICKED	Sent when the user clicks a mouse button, or activates the button with the hot-key (unless the button has been disabled).
%STN_DBLCLK	Sent when the user double-clicks on a label control (unless the control has been disabled).
%STN_DISABLE	Sent when a button is disabled.
%STN_ENABLE	Sent when a label control has been enabled.

Use the [CONTROL SET TEXT](#) statement to change the text in a label control and [CONTROL SET FONT](#) to change the font used in a label control. This is only possible if the label has a unique ID value (i.e., id& should not be -1).

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL GET TEXT](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#), [CONTROL SET TEXT](#)

Purpose	Add a line control to a dialog. A line control may also be a rectangle (empty or filled).
Syntax	<code>CONTROL ADD LINE, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>hDlg</i>	Handle of the dialog in which the line will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%SeparatorLeft</code> is more informative than a literal value such as 497. If you will not be changing the text in a line control after it is created, you may use -1 for the <i>id&</i> however, best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the line control. A line control does not display text, so it is possible to use this string for your own purposes; however, an ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 1 dialog unit.
<i>style&</i>	Primary style of the line control. The default line style is <code>%SS_ETCHEDFRAME</code> . The default style is used if both the primary and extended style parameters are omitted from the statement. For example: <pre>CONTROL ADD LINE, hDlg, id&, "", 100, 100, 150, 1, , , _ CALL LineCallback() ' Use default styles</pre> Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary

primary and extended style parameters.

The primary line style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%SS_BLACKFRAME	Draw a box with the frame drawn in the same color as the window frames. This color is black in the default Windows color scheme.
%SS_BLACKRECT	Draw a rectangle filled with the current window frame color. This color is black in the default Windows color scheme.
%SS_ETCHEDFRAME	Draw the frame of the control using an etched edge style. (default)
%SS_ETCHEDHORZ	Draw the horizontal edges of the control using an etched edge style.
%SS_ETCHEDVERT	Draw the vertical edges of the control using an etched edge style.
%SS_GRAYFRAME	Draw a box with the frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.
%SS_GRAYRECT	Draw a rectangle filled with the current screen background color. This color is gray in the default Windows color scheme.
%SS_NOPREFIX	Prevent interpretation of any ampersand (&) characters in the control's text as a control accelerator prefix characters. These normally are displayed with the ampersand removed and the next character in the string underscored.
%SS_NOTIFY	Sends %STN_CLICKED and %STN_DBLCLK notification messages to the line controls Callback Function when the user clicks or double-clicks the line control.
%SS_RIGHTJUST	Force the bottom-right corner of the control to remain fixed when the control is resized. Only the top and left sides are adjusted to accommodate a new image.
%SS_WHITEFRAME	Draw a box with the frame drawn with the same color as the window backgrounds. This color is white in the default Windows color scheme.
%SS_WHITERECT	Draw a rectangle filled with the current window background color. This color is white in the default Windows color scheme.

exstyle&

Extended style of the line control. The default extended line style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD LINE statement, in the same manner as *style&* above.

The extended line control style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control. The Callback Function will only receive messages if the %SS_NOTIFY style is used.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly "click" the control that immediately follows in the Tab-Order after the Line control, simply by pressing and

holding the ALT key while pressing the specified hot-key. For example, "&Test Suite " makes ALT+T the hot-key.

A line control will only send messages to a callback if the %SS_NOTIFY style is used. The following notifications are sent to the Callback Function:

%STN_CLICKED	Sent when the user clicks a line control (unless the control has been disabled).
%STN_DBLCLK	Sent when the user double-clicks a line control (unless the control has been disabled).
%STN_DISABLE	Sent when a line control has been disabled.
%STN_ENABLE	Sent when a line control has been enabled.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification message.

See also [Dynamic Dialog Tools](#), [CONTROL HANDLE](#), [CONTROL SEND](#)

CONTROL ADD LISTBOX statement

[Top](#) [Previous](#) [Next](#)

Purpose	Add a list box control to a dialog. A list box contains a set of predefined entries that permit a user to select one or more items. A list box may contain strings, images, or both. To put numbers in a list box, convert them to strings with the FORMAT\$, USING\$, or STR\$ functions.
Syntax	<code>CONTROL ADD LISTBOX, <i>hDlg</i>, <i>id&</i>, [<i>items\$()</i>], <i>x</i>, <i>y</i>, <i>xx</i>, <i>yy</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]] [[,] CALL <i>callback</i>]</code>
<i>hDlg</i>	Handle of the dialog in which the list box will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>items\$()</i>	Optional dynamic (variable length) string array containing the initial items to be displayed in the list box. Items are copied from the array to the list box, starting at the lowest subscript of the array (LBOUND), continuing on toward the end of the array until an empty string is encountered, or the highest subscript is reached. If an array with an LBOUND of zero (the default) is specified, be sure that the 1st element (0) contains data. Also see Restrictions below. To create a list box that is initially empty, either omit this parameter, or specify an array whose first element contains an empty string. If the list box uses the %LBS_SORT style, the items are sorted alphanumerically as they are added to the list box.
<i>x</i> , <i>y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 100 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the

height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.

style&

Primary [style](#) of the list box control. The default list box style comprises %LBS_SORT, %LBS_NOTIFY, %WS_TABSTOP, and %WS_VSCROLL (along with the %WS_EX_CLIENTEDGE extended style). The default list box style is used if both the primary and extended style parameters are omitted from the statement. For example:

```
CONTROL ADD LISTBOX, hDlg, id&, items$(), 100, 100, 150, 200,
' ' _
CALL ListboxCallback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary list box style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%LBS_DISABLENOSCROLL	Show a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items. Used in conjunction with the %WS_VSCROLL style.
%LBS_EXTENDEDSEL	Allow selection of multiple items in the list box by using the SHIFT key with mouse and/or keyboard actions.
%LBS_MULTICOLUMN	List box has multiple columns, and can be scrolled horizontally. To set the width, send the %LB_SETCOLUMNWIDTH message to the list box control.
%LBS_MULTIPLESEL	Allow selection of multiple items in the list box (without needing to use the SHIFT key) with mouse and/or keyboard actions.
%LBS_NOINTEGRALHEIGHT	Force the size of the list box to be exactly the size specified when the control is created. Otherwise,

	Windows may resize the list box to ensure that items are not partially displayed (clipped).
%LBS_NOSEL	The list box can contain items that can be viewed but not selected.
%LBS_NOTIFY	Send the callback a message whenever the user clicks or double-clicks a string in the list box.
%LBS_SORT	Automatically sort strings added to the list box in alphanumeric order.
%LBS_STANDARD	Equivalent to the combination of %LBS_SORT, %LBS_NOTIFY, %WS_VSCROLL and %WS_BORDER styles.
%LBS_USETABSTOPS	Expand tab (\$TAB , CHR\$(9)) characters. The default tab positions are for every 32 dialog units. To change the tab stop positions, send the %LB_SETTABSTOPS message to the list box control.
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user.
%WS_HSCROLL	Allow the control to display a horizontal scroll bar. By default this is disabled unless the controls horizontal scroll width has been configured by sending a %LB_SETHORIZONTALEXTENT message to the control. Use in conjunction with %LBS_DISABLENOSCROLL to make the scroll bar(s) visible at all times.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group

%WS_TABSTOP

and the start of a new group.

Allow the control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

%WS_VSCROLL

Allow the control to display a vertical scroll bar if the list is longer than the height of the list box. Use in conjunction with %LBS_DISABLENOSCROLL to make the scroll bar(s) visible at all times.

Do not intermix list box styles with similarly named [combo box](#) styles as the numeric values of similar styles can produce unexpected results. For example, %LBS_SORT = &H2 and %CBS_SORT = &H100. List box styles are prefixed with %LBS.

exstyle&

Extended style of the list box control. The default extended list box style comprises %WS_EX_CLIENTEDGE, and %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD LISTBOX statement, in the same manner as *style&* above.

The extended list box style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_CLIENTEDGE

Apply a sunken edge border to the control.

%WS_EX_LEFT

The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT

The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

	<p>%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).</p> <p>%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.</p> <p>%WS_EX_WINDOWEDGE Apply a raised edge border to the control.</p>
<i>callback</i>	<p>Optional name of a Callback Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the #MESSAGES metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.</p> <p>If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.</p>
Remarks	<p>The following notifications are sent to the Callback Function:</p> <p>%LBN_DBLCLK Sent when the user double-clicks a string in the list portion of a list box.</p> <p>%LBN_ERRSPACE Sent when a list box cannot allocate enough memory to meet a specific request.</p> <p>%LBN_KILLFOCUS Sent when a list box loses the keyboard focus.</p> <p>%LBN_SELCANCEL Sent when the user selects an item, but then selects another control or closes the dialog box. It indicates the user's initial selection is to be ignored.</p>

	<p>%LBN_SELCHANGE Sent when the selection in the list box is about to be changed as a result of the user either clicking in the list box or changing the selection by using the arrow keys.</p> <p>%LBN_SETFOCUS Sent when a list box receives the keyboard focus.</p> <p>When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification message.</p>
Restrictions	Under Windows 95/98/ME, a list box is limited to 32,736 items. In all versions of Windows, the actual string data contained by the list box is limited only by available memory.
See also	Dynamic Dialog Tools , CONTROL SET COLOR , CONTROL SET FONT , LISTBOX

CONTROL ADD LISTVIEW statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose	Add a ListView control to a dialog . A ListView displays a set of predefined string data items in one or more columns. The user may then view the items, selecting one or more of them for use in the program at a later time.
Syntax	<pre>CONTROL ADD LISTVIEW, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the ListView will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the ListView control. A ListView control does not display this text, so it is common to set this value to a null, empty string literal ("").
<i>x,y</i>	Integer expressions, variables, or numeric literal values specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx,yy</i>	Integer expressions, variable, or numeric literal values, specifying the width and height of the control. xx is the width and yy is the height, given in the same terms (pixels or dialog units) as the parent dialog.
<i>style&</i>	Primary style of the ListView control. The default ListView style comprises %WS_TABSTOP, %LVS_REPORT, and %LVS_SHOWSELALWAYS. This default ListView style is used if the style parameters are omitted from the statement, as in the following example: <pre>CONTROL ADD LISTVIEW, hDlg, id&, "", 100, 100, 150, 200, , , CALL LVCallback()</pre> If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters. The primary ListView style value can be a combination of any values below, combined together with the OR operator to form a bitmask: %LVS_ALIGNLEFT Items are left-aligned in icon and small icon

	view.
%LVS_ALIGNTOP	Items are aligned with the top of the control in icon and small icon view.
%LVS_AUTOARRANGE	Icons are automatically kept arranged.
%LVS_EDITLABELS	Item text can be edited by the user. The parent window must process notification messages.
%LVS_ICON	This style specifies icon view.
%LVS_LIST	This style specifies list view.
%LVS_NOCOLUMNHEADER	In report view, there are no headers on the columns.
%LVS_NOLABELWRAP	Item text is displayed on a single line in icon view.
%LVS_NOSCROLL	No scroll bars are provided. Incompatible with list view and report view.
%LVS_NOSORTHEADER	Report view column headers are flat, not like buttons. User can not click on the header to generate a column click notification.
%LVS_OWNERDATA	This style specifies a virtual ListView control.
%LVS_OWNERDRAWFIXED	The owner window can paint items in report view.
%LVS_REPORT	This style specifies report view. The first column is always left-aligned and columns have headers.
%LVS_SHAREIMAGELISTS	The image list will not be deleted when the control is destroyed.
%LVS_SHOWSELALWAYS	Selections are always shown, even without the focus.
%LVS_SINGLESEL	Only one item at a time can be selected. By default, multiple items may be selected.
%LVS_SMALLICON	This style specifies small icon view.
%LVS_SORTASCENDING	Item indexes are sorted as added in ascending order.

%LVS_SORTDESCENDING	Item indexes are sorted as added in descending order.
%WS_DISABLED	Create a control that is initially disabled . A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group.
%WS_TABSTOP	Allow the control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP.

exstyle&

Extended style of the ListView control. The default extended style is %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD LISTVIEW statement, in the same manner as *style&* above.

The extended ListView style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE	Apply a sunken edge border to the control.
%WS_EX_LEFT	The control has generic "left-aligned" properties. (default)
%WS_EX_RIGHT	The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.
%WS_EX_STATICEDGE	Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).
%WS_EX_TRANSPARENT	Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already

been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return [TRUE](#) (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification messages.

See also

[Dynamic Dialog Tools](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#), [LISTVIEW](#)

Purpose	Add an option button to a dialog. An option button is just like a conventional "radio button" control.
Syntax	<code>CONTROL ADD OPTION, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>hDlg</i>	Handle of the dialog in which the option button will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%DefCon5</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to be displayed next to the option button. An ampersand (&) may be included in <i>txt\$</i> to specify a hot-key. See the Remarks section below.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 40 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 14 dialog units.
<i>style&</i>	<p>Primary style of the option button control. The default option button styles are <code>%WS_TABSTOP</code>, <code>%BS_LEFT</code>, and <code>%BS_VCENTER</code>. The default styles are used if both the primary and extended style parameters are omitted from the statement. For example:</p> <pre>CONTROL ADD OPTION, hDlg, id&, txt\$, 100, 100, 150, 200, , , _ CALL OptionButtonCallback() ' Use default styles</pre> <p>Custom style values replace the default values. That is, they are not in addition to the default style values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary option button style value can be a combination of any values below, combined together with the OR operator to form a bitmask:</p>

%BS_BOTTOM	Place the text at the bottom of the control.
%BS_CENTER	Center the text horizontally in the control.
%BS_LEFT	Place the text on the left side of the control. Also see %BS_LEFTTEXT. (default)
%BS_LEFTTEXT	Place the option button to the right of the text portion of the control. Combine with %BS_RIGHT to right-align text against the left side of the option button.
%BS_MULTILINE	Wrap the caption text across multiple lines, if the text string is too long to fit on a single line. To force a wrap, insert a \$CR (or \$CRLF) into the caption text at the desired wrap position.
%BS_NOTIFY	Enable the %BN_KILLFOCUS and %BN_SETFOCUS notification messages for the option button.
%BS_PUSHLIKE	Button state alternates (toggles) between normal (raised) and depressed (sunken) modes.
%BS_RIGHT	Place the text on the right side of the control. Also see %BS_LEFTTEXT.
%BS_TOP	Place the text at the top of the control.
%BS_VCENTER	Center the text vertically in the control. (default)
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group. Groups configured this way permit the arrow keys to shift focus between the controls within the group, and focus can jump from group to group with the usual TAB and SHIFT+TAB keys. Both tab stops and groups are permitted to wrap from the end of the tab order back to the start.
%WS_TABSTOP	Allow the option control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP. (default)

exstyle&

Extended style of the option button control. The default extended option button style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended style parameters are omitted from the CONTROL ADD OPTION statement completely, in the same manner as *style&* above.

The extended option button style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

Option buttons are used for presenting a list of choices, only one of which may be selected. So, there is no point in having just a single option button. If what you want is to allow turning a single item on or off, use a [Checkbox](#)

instead.

When a group of option buttons are created, you should explicitly set the "selected" and "unselected" state of all option buttons, using the [CONTROL SET OPTION](#) statement to set the Check State of all the buttons in the group.

In addition, the first OPTION control in a group should have the style %WS_GROUP (to mark the beginning of a group of buttons) and %WS_TABSTOP. The remainder of the OPTION controls in the group should not have %WS_GROUP or %WS_TABSTOP styles. However, the very next non-OPTION control to appear in the tab order after the group should be given the %WS_GROUP and %WS_TABSTOP styles (the latter may depend on the type of control it is). If there are no other controls after the group, add %WS_GROUP to the first control in the dialog. This ensures that keyboard navigation with the arrow keys will operate within the group of OPTION controls, and that the TAB and SHIFT+TAB keys will switch focus between whole groups of controls (instead of individual controls as is common when each group member has the %WS_TABSTOP style).

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will be displayed underscored. This adds a control accelerator (hot-key) to enable the user to directly select the Option control, simply by pressing and holding the ALT key while pressing the specified hot-key. For example, "Level &3" makes ALT+3 the hot-key.

When the user clicks an option button, a message is sent to the Callback Function designated for the control. If there is no Callback Function designated then the message is sent to the callback for the dialog.

The following notifications are sent to the Callback Function:

%BN_CLICKED	Sent when the user clicks a mouse button, or activates the button with the hot-key (unless the button has been disabled).
%BN_KILLFOCUS	Sent then the option button loses keyboard focus, provided the button has the %BS_NOTIFY style.
%BN_SETFOCUS	Sent when the option button receives keyboard focus, provided the option button has the %BS_NOTIFY style.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification message.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD CHECK3STATE](#), [CONTROL ADD CHECKBOX](#), [CONTROL GET CHECK](#), [CONTROL SET COLOR](#),

[CONTROL SET FONT](#), [CONTROL SET OPTION](#)

Example

Refer to the example in the [CONTROL SET OPTION](#) topic.

CONTROL ADD PROGRESSBAR statement New!

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a ProgressBar control to a dialog . A ProgressBar is a rectangle that is gradually filled, left to right, as some work progresses.
Syntax	<code>CONTROL ADD PROGRESSBAR, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</code>
<i>hDlg</i>	Handle of the dialog in which the ProgressBar will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the ProgressBar control. A ProgressBar control does not display this text, so it is common to set this value to a null, empty string literal ("").
<i>x,y</i>	Integer expressions, variables, or numeric literal values specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx,yy</i>	Integer expressions, variable, or numeric literal values, specifying the width and height of the control. xx is the width and yy is the height, given in the same terms (pixels or dialog units) as the parent dialog.
<i>style&</i>	Primary style of the ProgressBar control. The default ProgressBar style is %WS_BORDER. This default style is used if both the primary and extended style parameters are omitted from the statement, as in the following example: <pre>CONTROL ADD PROGRESSBAR, hDlg, id&, "",90,90,90,20, , , CALL PBCallback()</pre> If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters. The primary style value can be a combination of the standard window values, and the values specific to a ProgressBar (below), which are combined together with the OR operator to form a bitmask:

%PBS_SMOOTH The bar is smooth rather than segmented.

%PBS_VERTICAL The control is advanced vertically.

exstyle&

Extended style of the control. The value can be a combination of the values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return [TRUE](#) (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification messages.

See also

[Dynamic Dialog Tools](#), [CONTROL SET COLOR](#), [PROGRESSBAR](#)

CONTROL ADD SCROLLBAR

statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a scroll bar control to a dialog . A scroll bar allows the user to scroll information left and right, or up and down. Your program, in response to notification messages from the scroll bar control, must do the actual scrolling itself.
Syntax	<pre>CONTROL ADD SCROLLBAR, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the scroll bar will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <i>%ReportScrollUpDown</i> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the scroll bar. A scroll bar control does not display text, so it is possible to use this string for your own purposes; however, an ampersand (&) may be included in txt\$ to specify a (hidden) hot-key. See the Remarks section below. Typically, this parameter is specified as an empty string ("") or a \$NUL string equate.
<i>x, y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 10 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 11 dialog units.
<i>style&</i>	Primary style of the scroll bar control. The default scroll bar style is %SBS_HORZ; however, if the width is less than the height, the control is automatically switched to %SBS_VERT, regardless of whether %SBS_HORZ is specified or not. If %SBS_VERT is specified, the control

will always be created as a vertical scroll bar regardless of the dimensions of the control. The default style is used if both the primary and extended style parameters are omitted from the statement. For example:

```
CONTROL ADD SCROLLBAR, hDlg, id&, txt$, 100, 100, 150, 14, , , _  
CALL Scrollbar1Callback() ' Use default styles
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.

The primary scroll bar style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%SBS_BOTTOMALIGN	Align the bottom edge of the scroll bar with the bottom edge of the dialog, and use the default height of system scroll bars. Used with %SBS_HORZ.
%SBS_HORZ	Make the control a horizontal scroll bar (default - see style& above).
%SBS_LEFTALIGN	Align the left edge of the scroll bar with the left edge of the dialog, and use the default width of system scroll bars. Used with %SBS_VERT.
%SBS_RIGHTALIGN	Align the right edge of the scroll bar with the right edge of the dialog, and use the default width of system scroll bars. Used with %SBS_VERT.
%SBS_TOPALIGN	Align the top edge of the scroll bar with the top edge of the window, and use the default height of system scroll bars. Used with %SBS_HORZ.
%SBS_VERT	Make the control a vertical scroll bar (see style& above).
%WS_DISABLED	Create a control that is initially disabled. A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group.
%WS_TABSTOP	Allow the scrollbar control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the

%WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP.

exstyle&

Extended style of the scroll bar control. The default extended scroll bar style comprises %WS_EX_LEFT. The default extended style is used if both the primary and extended style parameters are omitted from the CONTROL ADD SCROLLBAR statement, in the same manner as *style&* above.

The extended scroll bar style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE non-zero if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages automatically.

Remarks

If the ampersand (&) character appears in the *txt\$* parameter, the letter that follows will become a control accelerator (hot-key) to enable the user to directly select the scroll bar control, simply by pressing and holding the ALT key while pressing the specified hot-key. For example, "&9" makes ALT+9 the hot-key. The actual text in *txt\$* is not displayed in a scroll bar control.

When the user clicks on a scroll bar, drags the thumb (also called the scroll box), or initiates a scroll event with the keyboard, a message is sent to the

Callback Function designated for the control. If there is no Callback Function designated, the message is sent to the callback for the dialog.

The following notifications are sent to the Callback Function:

%WM_HSCROLL Sent when the user adjusts a horizontal scroll bar.

%WM_VSCROLL Sent when the user adjusts a vertical scroll bar.

When a Callback Function receives a %WM_HSCROLL or %WM_VSCROLL message, it should retrieve and set the scroll bar control settings through the GetScrollInfo API and [SCROLLBAR SET POS](#) function calls. Be sure to use the %SB_CTL flag with these API functions, rather than the %SB_HORZ or %SB_VERT flags.

See also

[Dynamic Dialog Tools](#), [CONTROL HANDLE](#), [CONTROL SEND](#), [CONTROL SET COLOR](#), [SCROLLBAR](#)

CONTROL ADD STATUSBAR statement New!

[Top](#) [Previous](#)
[Next](#)

Purpose	Add a StatusBar control to a dialog . A StatusBar is a horizontal window, typically at the bottom of a dialog <u>client area</u> , which displays various kinds of status information. It can be divided into parts to display multiple items.								
Syntax	<pre>CONTROL ADD STATUSBAR, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>								
<i>hDlg</i>	Handle of the dialog in which the StatusBar will be created. The dialog will become the parent of the control.								
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.								
<i>txt\$</i>	Text to initially display in the StatusBar control.								
<i>x,y</i>	Integer expressions to specify control location. In the case of a StatusBar, location parameters are ignored since the control is placed on the top or the bottom of the dialog, based upon the chosen style. These location parameters are usually defined as 0, 0.								
<i>xx,yy</i>	Integer expressions to specify control size. In the case of a ToolBar, size parameters are ignored since the control is created with a default size. These size parameters are usually defined as 0, 0.								
<i>style&</i>	Primary style of the StatusBar control. The default StatusBar style is %CCS_BOTTOM. This default style is used if both the primary and extended style parameters are omitted from the statement, as in the following example: <pre>CONTROL ADD STATUSBAR, hDlg, id&, "", 5, 5, 5, 5, , , CALL SBCallback()</pre> <p>If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters.</p> <table><tr><td>%CCS_TOP</td><td>The StatusBar is placed at the top of the dialog.</td></tr><tr><td>%CCS_BOTTOM</td><td>The StatusBar is placed at the bottom of the dialog.</td></tr><tr><td>%SBARS_SIZEGRIP</td><td>A sizegrip is added to the StatusBar.</td></tr><tr><td>%SBARS_TOOLTIPS</td><td>Use this style to enable tooltips.</td></tr></table>	%CCS_TOP	The StatusBar is placed at the top of the dialog.	%CCS_BOTTOM	The StatusBar is placed at the bottom of the dialog.	%SBARS_SIZEGRIP	A sizegrip is added to the StatusBar.	%SBARS_TOOLTIPS	Use this style to enable tooltips.
%CCS_TOP	The StatusBar is placed at the top of the dialog.								
%CCS_BOTTOM	The StatusBar is placed at the bottom of the dialog.								
%SBARS_SIZEGRIP	A sizegrip is added to the StatusBar.								
%SBARS_TOOLTIPS	Use this style to enable tooltips.								

<i>exstyle&</i>	<p>Extended style of the StatusBar control. The extended StatusBar style value can be a combination of the values below, combined together with the OR operator to form a bitmask:</p> <p>%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.</p> <p>%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).</p> <p>%WS_EX_WINDOWEDGE Apply a raised edge border to the control.</p>
<i>callback</i>	<p>Optional name of a Callback Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the #MESSAGES metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.</p> <p>If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.</p>
Remarks	<p>When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification messages.</p>
See also	<p>Dynamic Dialog Tools, CONTROL SET FONT, STATUSBAR</p>

Purpose	Add a Tab Control to a dialog . A Tab Control is analogous to the dividers in a notebook. It displays one particular page, selecting it from multiple pages, when the user chooses the corresponding tab.				
Syntax	<code>CONTROL ADD TAB, <i>hDlg</i>, <i>id&</i>, <i>txt\$</i>, <i>x</i>, <i>y</i>, <i>xx</i>, <i>yy</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]]] [[,] CALL <i>callback</i>]</code>				
<i>hDlg</i>	Handle of the dialog in which the Tab Control will be created. The dialog will become the parent of the control.				
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.				
<i>txt\$</i>	Text to associate with the Tab Control. A Tab Control does not display this text, so it is common to set this value to a null, empty string literal ("").				
<i>x,y</i>	Integer expressions, variables, or numeric literal values specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.				
<i>xx,yy</i>	Integer expressions, variable, or numeric literal values, specifying the width and height of the control. <i>xx</i> is the width and <i>yy</i> is the height, given in the same terms (pixels or dialog units) as the parent dialog.				
<i>style&</i>	Primary style of the Tab control. The default Tab style is %WS_CHILD and %WS_TABSTOP. This default style is used if both the primary and extended style parameters are omitted from the statement, as in the following example: <pre>CONTROL ADD TAB, hDlg, id&, "",90,90,200,90, , , CALL PBCallback()</pre> <p>If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary style value can be a combination of the standard window values, and the values specific to a Tab Control (below), which are combined together with the OR operator to form a bitmask:</p> <table><tr><td>%TCS_FORCEICONLEFT</td><td>Icons are forced to the left</td></tr><tr><td>%TCS_FORCELABELLEFT</td><td>Labels are forced to the left</td></tr></table>	%TCS_FORCEICONLEFT	Icons are forced to the left	%TCS_FORCELABELLEFT	Labels are forced to the left
%TCS_FORCEICONLEFT	Icons are forced to the left				
%TCS_FORCELABELLEFT	Labels are forced to the left				

%TCS_FIXEDWIDTH	All tabs are the same size
%TCS_RAGGEDRIGHT	Tabs are not stretched
%TCS_FOCUSONBUTTONDOWN	Tabs receive the focus when clicked
%TCS_OWNERDRAWFIXED	Parent window is responsible for drawing tabs
%TCS_TOOLTIPS	A Tooltip control is associated with the control
%TCS_FOCUSNEVER	Tab never receives the focus

exstyle&

Extended style of the control. The value can be a combination of the values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE	Apply a sunken edge border to the control.
%WS_EX_STATICEDGE	Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).
%WS_EX_WINDOWEDGE	Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return [TRUE](#) (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification messages.

See also

[Dynamic Dialog Tools](#), [CONTROL SET FONT](#), [TAB](#)

CONTROL ADD TEXTBOX statement [Top](#) [Previous](#) [Next](#)

Purpose	Add a text box control to a dialog. A text box is very similar to a conventional edit control, and it is used to enter text into an application. Text boxes support single-line and multiple-line input.
Syntax	<code>CONTROL ADD TEXTBOX, <i>hDlg</i>, <i>id&</i>, <i>txt\$</i>, <i>x</i>, <i>y</i>, <i>xx</i>, <i>yy</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]]] [[,] CALL <i>callback</i>]</code>
<i>hDlg</i>	Handle of the dialog in which the text box will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate <code>%CustomerName</code> is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Default text to be displayed in text box. <i>txt\$</i> may be a string, string equate , or string expression . <i>txt\$</i> can be empty if there is no default text.
<i>x</i> , <i>y</i>	Integer expressions, variables , or numeric literal values, specifying the location of the control inside the dialog client area. <i>x</i> is the horizontal position, and <i>y</i> is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx</i>	Integer expression, variable, or numeric literal value, specifying the width of the control. The width is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 100 dialog units.
<i>yy</i>	Integer expression, variable, or numeric literal value, specifying the height of the control. The height is given in the same terms (pixels or dialog units) as the parent dialog. The most common value used in the Microsoft Dialog Editor and Visual Studio is 12 dialog units.
<i>style&</i>	<p>Primary style of the text box control. The default text box style comprises <code>%WS_TABSTOP</code>, <code>%WS_BORDER</code>, <code>%ES_LEFT</code>, and <code>%ES_AUTOHSCROLL</code>. The default style is used if both the primary and extended style parameters are omitted from the statement. For example:</p> <pre>CONTROL ADD TEXTBOX, hDlg, id&, txt\$, 100, 100, 150, 200, , , _ CALL EditControlCallback() ' Use default styles</pre> <p>Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary text box style value can be a combination of any values below, combined together with the OR operator to form a bitmask:</p>

%ES_AUTOHSCROLL	Automatically scroll text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, scroll all text back to position zero. Also see %WS_AUTOHSCROLL.
%ES_AUTOVSCROLL	Automatically scroll text up one page when the user presses the ENTER key on the last line. This must be combined with the %ES_WANTRETURN and %ES_MULTILINE styles. Also see %WS_VSCROLL.
%ES_CENTER	Center text in a multi-line edit control.
%ES_LEFT	Left-aligns text. (default)
%ES_LOWERCASE	Convert all characters to lowercase as they are typed into the edit control.
%ES_MULTILINE	<p>Allow the control to accept multiple lines of input. By default, the ENTER key activates the default button on the dialog. To use the ENTER key as a carriage return in the text box control, include the %ES_WANTRETURN style.</p> <p>If the %ES_AUTOHSCROLL style is included, the control automatically scrolls horizontally when the caret goes past the right edge of the control. Otherwise, the control automatically wraps words to the beginning of the next line when necessary. The control size determines the position of the word wrap.</p>
%ES_NOHIDESEL	Negate the default behavior for a text box. The default behavior hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. If you specify %ES_NOHIDESEL, the selected text is inverted, even if the control does not have the focus.
%ES_NUMBER	Allow only digits ("0123456789") instead of characters. Although Windows does not consider the negation symbol (-) or period symbol (.) to be digits, subclassing a TextBox that does not use %ES_NUMBER and rejecting "unwanted" keystrokes is common practice among advanced programmers.
%ES_OEMCONVERT	Text is converted from the windows character set

	to OEM, then back to Windows, as it is entered.
%ES_PASSWORD	Display an asterisk (*) for each character typed into the control in order to obscure the password.
%ES_READONLY	Prevent the user from typing or editing text in the control. Text can still be selected and copied from the control to the clipboard with the mouse.
%ES_RIGHT	Right-align text in a multi-line text box.
%ES_UPPERCASE	Convert all characters to uppercase as they are typed into the control.
%ES_WANTRETURN	Allow the ENTER key to insert a carriage return in a multi-line text box. Otherwise, the ENTER key works as the dialog box's default push button. This style has no effect on a single-line text box.
%WS_BORDER	Add a thin line border around the text box control.
%WS_HSCROLL	Add a horizontal scroll bar to the edit control, when used in conjunction to the %ES_AUTOHSCROLL style.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group.
%WS_TABSTOP	Allow the textbox control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP . (default)
%WS_VSCROLL	Add a vertical scroll bar to the edit control. This style should be used in conjunction to the %ES_MULTILINE and %ES_AUTOVSCROLL styles.

exstyle&

Extended style of the text box control. The default extended text box style comprises **%WS_EX_CLIENTEDGE** with **%WS_EX_LEFT**. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD TEXTBOX statement, in the same manner as *style&* above.

The extended text box style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_CLIENTEDGE	Apply a sunken edge border to the control.
%WS_EX_LEFT	The control has generic "left-aligned" properties. (default)
%WS_EX_RIGHT	The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.
%WS_EX_STATICEDGE	Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).
%WS_EX_TRANSPARENT	Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see MSDN for more information.
%WS_EX_WINDOWEDGE	Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

Remarks

If you specify the %ES_AUTOHSCROLL style, the control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key.

If you do not specify %ES_AUTOHSCROLL, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The control size determines the position of the word wrap.

The following notifications are sent to the Callback Function:

%EN_CHANGE	Sent when the user has taken an action that may have altered text in the text box. Unlike the
-------------------	---

	%EN_UPDATE notification, this message is sent after Windows updates the screen. Programmatically changing the text in a control also triggers this message.
%EN_ERRSPACE	Sent when the text box cannot allocate enough memory to meet a specific request.
%EN_HSCROLL	Sent when the user clicks a multi-line text box's horizontal scroll bar. The callback is notified before the screen is updated.
%EN_KILLFOCUS	Sent when an edit control loses the keyboard focus.
%EN_MAXTEXT	Sent when the current text insertion has exceeded the specified number of characters for the text box. The text insertion is truncated.
%EN_SETFOCUS	Sent when an edit control receives the keyboard focus.
%EN_UPDATE	Sent when a text box is about to display altered text. This notification message is sent after the control has formatted the text, but before it displays the text. Also see %EN_CHANGE.
%EN_VSCROLL	Sent when the user clicks a text box's vertical scroll bar. The callback is notified before the screen is updated.

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the notification message.

Use [CONTROL GET TEXT](#) to retrieve the text from a text box control, and use [CONTROL SET TEXT](#) to change the text in a text box control.

Changing the text in a text box control (in response to a %EN_CHANGE or %EN_UPDATE message) will trigger a second set of %EN_CHANGE and %EN_UPDATE messages. Unless this is compensated for, these notifications can unwittingly cause an endless loop.

For example, the following is potentially fatal:

```
CALLBACK FUNCTION EditControlCallback()
    IF CB.CTL = %ID_EDITBOX1 AND CB.CTLMSG = %EN_CHANGE THEN
        CONTROL SET TEXT CB.HNDL, CB.CTL, "New Text"
    EXIT FUNCTION
END IF
[statements]
```

As CONTROL SET TEXT is a "blocking" statement (that is, the statement does not complete until the text has been changed), it is a simple matter to block the endless loop effect:

```
CALLBACK FUNCTION EditControlCallback()
```

```
STATIC EditBusy&
IF CB.CTL = %ID_EDITBOX1 AND CB.CTLMSG = %EN_CHANGE THEN
    IF EditBusy& THEN EXIT FUNCTION
    EditBusy& = -1
    CONTROL SET TEXT CB.HNDL, CB.CTL, "New Text"
    RESET EditBusy&
    EXIT FUNCTION
END IF
[statements]
```

See also

[Dynamic Dialog Tools](#), [CONTROL GET TEXT](#), [CONTROL SET COLOR](#),
[CONTROL SET FONT](#), [CONTROL SET TEXT](#)

CONTROL ADD TOOLBAR statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose	Add a ToolBar control to a dialog . A ToolBar overlays part of a dialog's <u>client area</u> , typically at the top.
Syntax	<pre>CONTROL ADD TOOLBAR, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the ToolBar will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the ToolBar control. A ToolBar control does not display this text, so it is common to set this value to a null, empty string literal ("" or \$NUL).
<i>x,y</i>	Integer expressions to specify control location. In the case of a ToolBar, location parameters are ignored since the control is placed on the top or the bottom of the dialog, based upon the chosen style. These location parameters are usually defined as 0, 0.
<i>xx,yy</i>	Integer expressions to specify control size. In the case of a ToolBar, size parameters are ignored since the control is created with a default size. These size parameters are usually defined as 0, 0.
<i>style&</i>	<p>Primary style of the ToolBar control. The default ToolBar style is %WS_CHILD or %WS_VISIBLE or %WS_BORDER or %CCS_TOP or %TBSTYLE_FLAT. This default style is used if both the primary and extended style parameters are omitted from the statement, as in the following example:</p> <pre>CONTROL ADD TOOLBAR, hDlg, id&, "", 1, 1, 1, 1, , , CALL TBcallback()</pre> <p>If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary ToolBar style value can be a combination of the values below, combined together with the OR operator to form a bitmask:</p>

%CCS_TOP The ToolBar is placed at the top of the dialog.

%CCS_BOTTOM The ToolBar is placed at the bottom of the dialog.

<i>exstyle</i> &	Extended style of the ToolBar control. The extended ToolBar style value can be a combination of the values below, combined together with the OR operator to form a bitmask:
	<p><code>%WS_EX_CLIENTEDGE</code> Apply a sunken edge border to the control.</p> <p><code>%WS_EX_STATICEDGE</code> Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).</p> <p><code>%WS_EX_WINDOWEDGE</code> Apply a raised edge border to the control.</p>
<i>callback</i>	<p>Optional name of a Callback Function that receives all <code>%WM_COMMAND</code> and <code>%WM_NOTIFY</code> messages for the control. See the #MESSAGES metastatement to choose which messages will be received. Generally speaking, ToolBar command messages result from clicking a ToolBar Button, so the message is sent to the callback specified in TOOLBAR ADD BUTTON or the dialog callback specified in Dialog Show. Message routing by button allows you to easily determine which button generated the event, and eliminates virtually all <code>%WM_COMMAND</code> messages here. This callback is primarily used to process <code>%WM_NOTIFY</code> messages.</p> <p>If the Callback Function processes a message, it should return TRUE (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the DDT engine processes unhandled messages.</p>
Remarks	When a Callback Function receives a <code>%WM_COMMAND</code> message, it should explicitly test the value of CB.CTL and CB.CTLMSG to guarantee it is responding appropriately to the notification messages.
See also	DIALOG SHOW MODAL , DIALOG SHOW MODELESS , Dynamic Dialog Tools , CONTROL SET FONT , TOOLBAR

CONTROL ADD TREEVIEW statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose	Add a TreeView control to a dialog . A TreeView displays a set of string data items with a parent-child relationship between the items. This creates a hierarchical list of data which can have any number of levels. The user may view the items, selecting them for use in the program at a later time.
Syntax	<pre>CONTROL ADD TREEVIEW, hDlg, id&, txt\$, x, y, xx, yy [, [style&] [, [exstyle&]]] [[,] CALL callback]</pre>
<i>hDlg</i>	Handle of the dialog in which the TreeView will be created. The dialog will become the parent of the control.
<i>id&</i>	Unique identifier for the control in the range 1 to 65535, frequently specified with numeric equates for clarity of the code. For example, the equate %PickList is more informative than a literal value such as 497. Best practice suggests identifiers should start at 100 to avoid conflict with any of the standard predefined identifiers.
<i>txt\$</i>	Text to associate with the TreeView control. A TreeView control does not display this text, so it is common to set this value to a null, empty string literal ("").
<i>x,y</i>	Integer expressions, variables, or numeric literal values specifying the location of the control inside the dialog client area. x is the horizontal position, and y is the vertical position. 0,0 refers to the upper left corner of the dialog box client area. Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.
<i>xx,yy</i>	Integer expressions, variable, or numeric literal values, specifying the width and height of the control. xx is the width and yy is the height, given in the same terms (pixels or dialog units) as the parent dialog.
<i>style&</i>	<p>Primary style of the TreeView control. The default TreeView style comprises %WS_TABSTOP, %TVS_HASBUTTONS, %TVS_LINESATROOT, %TVS_HASLINES, and %TVS_SHOWSELALWAYS. This default TreeView style is used if the style parameters are omitted from the statement, as in the following example:</p> <pre>CONTROL ADD TREEVIEW, hDlg, id&, "", 100, 100, 150, 200, , , CALL TVCallback()</pre> <p>If you include explicit style values, they replace the default values. That is, they are not added to the default styles values - your code must specify all necessary primary and extended style parameters.</p> <p>The primary TreeView style value can be a combination of the values below, combined together with the OR operator to form a bitmask:</p>

%TVS_HASBUTTONS	Displays +/- signs next to parent items so the user can expand or collapse a list of child items.
%TVS_HASLINES	Uses lines to show the hierarchy of data items.
%TVS_LINESATROOT	Uses lines to link items at the root level.
%TVS_EDITLABELS	Allows the user to edit the labels of the data items.
%TVS_DISABLEDRAHDROP	Prevents drag and drop
%TVS_SHOWSELALWAYS	A selected item remains selected when the control loses focus.
%TVS_NOTOOLTIPS	Disables ToolTips.
%TVS_CHECKBOXES	Enables check boxes for items with an image.
%TVS_TRACKSELECT	Enables hot tracking.
%TVS_SINGLEEXPAND	Only one item can be expanded at a time.
%TVS_INFOTIP	Obtains ToolTip information.
%TVS_FULLROWSELECT	The entire row of a selected item is highlighted.
%TVS_NOSCROLL	Disables horizontal and vertical scrolling.
%TVS_NONEVENHEIGHT	Sets the height of items to an odd height.
%TVS_NOHSCROLL	Disables horizontal scrolling.
%WS_DISABLED	Create a control that is initially disabled . A disabled control cannot receive input from the user.
%WS_GROUP	Define the start of a group of controls. The first control in each group should also use %WS_TABSTOP style. The next %WS_GROUP control in the tab order defines the end of this group and the start of a new group.
%WS_TABSTOP	Allow the control to receive keyboard focus when the user presses the TAB and SHIFT+TAB keys. The TAB key shifts

keyboard focus to the next control with the %WS_TABSTOP style, and SHIFT+TAB shifts focus to the previous control with %WS_TABSTOP.

exstyle&

Extended style of the TreeView control. The default extended style is %WS_EX_LEFT. The default extended style is used if both the primary and extended parameters are omitted from the CONTROL ADD TREEVIEW statement, in the same manner as style& above.

The extended TreeView style value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%WS_EX_CLIENTEDGE Apply a sunken edge border to the control.

%WS_EX_LEFT The control has generic "left-aligned" properties. (default)

%WS_EX_RIGHT The control has generic "right-aligned" properties. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored.

%WS_EX_STATICEDGE Apply a three-dimensional border style to the control (intended to be used for items that do not accept user input).

%WS_EX_TRANSPARENT Controls/windows beneath the control are drawn before the control is drawn. The control is deemed transparent because elements behind the control have already been painted - the control itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

%WS_EX_WINDOWEDGE Apply a raised edge border to the control.

callback

Optional name of a [Callback](#) Function that receives all %WM_COMMAND and %WM_NOTIFY messages for the control. See the [#MESSAGES](#) metastatement to choose which messages will be received. If a callback for the control is not designated, you must create a dialog Callback Function to process messages from your control.

If the Callback Function processes a message, it should return [TRUE](#) (non-zero) to prevent the message being passed unnecessarily to the dialog callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise,

the [DDT](#) engine processes unhandled messages.

Remarks

When a Callback Function receives a %WM_COMMAND message, it should explicitly test the value of [CB.CTL](#) and [CB.CTLMSG](#) to guarantee it is responding appropriately to the messages.

See also

[Dynamic Dialog Tools](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#), [TREEVIEW](#)

Purpose	Disable a control so that it no longer receives any messages or accepts user interaction.
Syntax	<code>CONTROL DISABLE <i>hDlg</i>, <i>id&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control. <i>id&</i> is the unique control identifier as assigned to the control with a CONTROL ADD statement.</p> <p>A disabled control will not receive any messages when clicked with the mouse or selected with the keyboard. Most, but not all, controls will redraw themselves as "gray" when disabled.</p>
See also	Dynamic Dialog Tools , CONTROL ENABLE , CONTROL KILL

CONTROL ENABLE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Enable a control so that it can receive messages when the user interacts with it via the mouse or keyboard.
Syntax	<code>CONTROL ENABLE <i>hDlg</i>, <i>id&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control. <i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p>An enabled control will receive messages when clicked with the mouse or selected with the keyboard.</p>
See also	Dynamic Dialog Tools , CONTROL DISABLE , CONTROL KILL

Purpose Get the Check State of a [CHECK3STATE](#), [CHECKBOX](#), or [OPTION](#) button.

Syntax `CONTROL GET CHECK hDlg, id& TO lResult&`

Remarks *hDlg* refers to the dialog that owns the control.
id& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.
lResult& receives the Check State of the control as follows:

<i>lResult</i> &	Check State of control
0 (Zero)	Button is unchecked (unset, or cleared)
1 (One)	Button is checked (set)
2 (Two)	Button is indeterminate (grayed) (CHECK3STATE only)

Note that a grayed (indeterminate) CHECK3STATE control does not mean the control is disabled. Rather, the Check State of the control is both checked and unchecked.

See also [Dynamic Dialog Tools](#), [CONTROL ADD CHECK3STATE](#), [CONTROL ADD CHECKBOX](#), [CONTROL ADD OPTION](#), [CONTROL SET CHECK](#), [CONTROL SET OPTION](#), [TREEVIEW GET CHECK](#), [TREEVIEW SET CHECK](#)

CONTROL GET CLIENT statement

[Top](#) [Previous](#) [Next](#)

Purpose Return the client size of the specified child control.

Syntax `CONTROL GET CLIENT hDlg, id& TO wide&, high&`

Remarks *hDlg* refers to the dialog that owns the control.

id& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.

The size of the control client area is placed in the *wide&* (width) and *high&* (height) [variables](#). The size is specified in the same terms (pixels or dialog units) as the parent dialog.

See also [Dynamic Dialog Tools](#), [CONTROL GET LOC](#), [CONTROL GET SIZE](#), [CONTROL SET CLIENT](#), [CONTROL SET CLIENT](#), [CONTROL SET LOC](#), [CONTROL SET SIZE](#), [DIALOG UNITS](#), [DIALOG PIXELS](#)

CONTROL GET LOC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Get the location of the specified control in a dialog.
Syntax	<code>CONTROL GET LOC <i>hDlg</i>, <i>id&</i> TO <i>x&</i>, <i>y&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p>The location of the top left corner of the control is placed in the <i>x&</i> (horizontal location) and <i>y&</i> (vertical location) variables. The location is relative to the upper-left corner of the client area in the parent dialog. The coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.</p>
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , CONTROL GET SIZE , CONTROL SET LOC , CONTROL SET SIZE

CONTROL GET SIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Get the size of a control in the specified dialog.
Syntax	<code>CONTROL GET SIZE <i>hDlg</i>, <i>id&</i> TO <i>wide&</i>, <i>high&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p>The location of the top left corner of the control is placed in the <i>x&</i> (horizontal location) and <i>y&</i> (vertical location) variables. The location is relative to the upper-left corner of the client area in the parent dialog. The coordinates are specified in the same terms (pixels or dialog units) as the parent dialog.</p>
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , CONTROL GET LOC , CONTROL SET LOC , CONTROL SET SIZE

Purpose	Get the text from a control.
Syntax	<code>CONTROL GET TEXT <i>hDlg</i>, <i>id&</i> TO <i>txt\$</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement. Any text in the control is placed into the <i>txt\$</i> variable.</p> <p>With combo boxes, CONTROL GET TEXT returns the text entered in the edit portion of the control. To retrieve the selected text from the list portion of a combo box or a list box control, use the COMBOBOX GET TEXT statement or LISTBOX GET TEXT statement respectively.</p>
See also	Dynamic Dialog Tools , COMBOBOX GET TEXT , CONTROL SET TEXT , LISTBOX GET TEXT , LISTVIEW GET TEXT , TREEVIEW GET TEXT

CONTROL GET USER statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve a value from the user data area of a DDT control.
Syntax	<code>CONTROL GET USER <i>hDlg</i>, <i>id</i>&, <i>index</i>& TO <i>retvar</i>&</code>
Remarks	<p>Each DDT control has a user data area consisting of eight Long-integer values which may be used at the programmer's discretion to save relevant data. CONTROL GET USER allows one of the values to be retrieved, based upon the index parameter value (1 through 8).</p> <p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id</i>& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p><i>index</i>& is the index number of the user data value to retrieve, in the range 1 to 8 inclusive.</p> <p><i>retvar</i>& receives the Long-integer data value store in the nominated user data index.</p>
Restrictions	Data in the user data area is lost when the control is destroyed. The data area is completely separate from the %GWL_USERDATA area maintained by Windows.
See also	Dynamic Dialog Tools , COMBOBOX GET USER , COMBOBOX SET USER , CONTROL SET USER , DIALOG GET USER , DIALOG SET USER , LISTBOX GET USER , LISTBOX SET USER , LISTVIEW GET USER , LISTVIEW SET USER , TREEVIEW GET USER , TREEVIEW SET USER

CONTROL HANDLE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Return a window handle for the specified control ID.
Syntax	<code>CONTROL HANDLE <i>hDlg</i>, <i>id&</i> TO <i>hCtl&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control. <i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p>The returned value is a window handle for the control, assigned by Windows when the control was initially created, uniquely identifying the control from all other controls. Some API functions require a window handle value rather than a control ID value.</p>
See also	Dynamic Dialog Tools , CONTROL SEND , WINDOW GET ID , WINDOW GET PARENT

CONTROL KILL statement

[Top](#) [Previous](#) [Next](#)

Purpose Remove a control from a dialog.

Syntax `CONTROL KILL hDlg, id&`

Remarks *hDlg* refers to the dialog that owns the control. *id&* is the unique control identifier as assigned to the control with a CONTROL SHOW statement. The control is destroyed and removed from the dialog. The [Callback Function](#) for the control or dialog will no longer receive messages for the control.

Restrictions A control should not be destroyed while processing a notification message from the same control, but it is permissible to kill a different control in the notification handler. If it is absolutely necessary to kill a control while processing one of its notification messages, use the PostMessage API function (or the [DIALOG POST](#) or [CONTROL POST](#) statements) to post a user-defined message to the dialog callback, and kill the control when processing the user-defined message.

See also [Dynamic Dialog Tools](#), [CONTROL DISABLE](#), [CONTROL ENABLE](#)

Example

```
' How to avoid "suicide" conditions
CALLBACK FUNCTION DlgCallBack() AS LONG
    SELECT CASE CB.MSG
        CASE %WM_COMMAND
            IF CB.CTLMSG = %BN_GOTFOCUS AND CB.CTL = %MyBtn THEN
                DIALOG POST CB.HNDL, %WM_USER + 999&, 0, 0
            END IF
        CASE %WM_USER + 999&
            CONTROL KILL CB.HNDL, %MyBtn
            FUNCTION = 1
    END SELECT
END FUNCTION
```

Purpose	Place a message in the message queue to be processed at the leisure of the target control.
Syntax	<code>CONTROL POST <i>hDlg</i>, <i>id</i>&, <i>Msg</i>&, <i>wParam</i>&, <i>lParam</i>&</code>
Remarks	<p>CONTROL POST places the message in the message queue and returns immediately. The message is processed by the control at a later time, when it reads the message from the queue.</p> <p>This behavior is quite different to the CONTROL SEND statement, which forces the control to process the message immediately before returning. Since CONTROL POST is an asynchronous operation, it is not possible to retrieve a return code from the message.</p> <p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id</i>& is the unique control identifier as assigned to the control with a CONTROL ADD statement.</p> <p><i>Msg</i>& is the message you want to post to the control.</p> <p><i>wParam</i>& is the first message parameter. <i>lParam</i>& is the second message parameter. The values of <i>wParam</i>& and <i>lParam</i>& are message-dependent. By Default, Classic PowerBASIC passes these parameters BYVAL. If the target control is expected to alter the values held by variables passed in the <i>wParam</i>& and <i>lParam</i>& parameters, pass them using VARPTR() or the changes will likely be discarded.</p> <p>Note that the address of the data must remain valid until after the control has processed the message and accessed the data. In this case, using STATIC or GLOBAL variables can be very important or a General Protection Fault (GPF) may occur (that is, if the variables have gone out of scope by the time the message is processed).</p> <p>An example of posting the addresses of variables to a control:</p> <pre>' Retrieve an Edit controls Current Selection ' Sel1& and Sel2& must be STATIC or GLOBAL CONTROL POST CB.HNDL, %ID_EDIT6, %EM_GETSEL, VARPTR(Sel1&), VARPTR(Sel2&)</pre> <p>CONTROL POST returns immediately after the placing the message in the queue.</p>
Restrictions	To post a custom message to a control, use a message value in the range of (%WM_USER + 500) to (%WM_USER + &H07FFF), or use the RegisterWindowMessage API to obtain a unique message value from the operating system. Using messages with a numeric value of less than %WM_USER + 500 may conflict with Windows Common Control messages.

See also

[Dynamic Dialog Tools](#), [CONTROL HANDLE](#), [CONTROL SEND](#), [DIALOG POST](#), [DIALOG SEND](#)

Example

```
' Programmatically post a click message to a button:  
CONTROL POST hDlg, %ID_BTN1, %BM_CLICK, 0, 0
```

CONTROL REDRAW statement

[Top](#) [Previous](#) [Next](#)

Purpose	Schedule a designated control to be redrawn.
Syntax	<code>CONTROL REDRAW <i>hDlg</i>, <i>id&</i></code>
Remarks	<p>CONTROL REDRAW invalidates the target control and schedules a redraw event to occur.</p> <p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p>
Restrictions	<p>Redrawing of individual controls is considered a low priority event, and a control redraw may not happen instantly if there are pending messages in the message queue. That is, pending messages in the message queue may need to be processed before the scheduled redraw event occurs.</p> <p>It is not advisable to use CONTROL REDRAW or DIALOG REDRAW within the %WM_PAINT and associated message handling code, or an infinite redraw loop could occur.</p>
See also	Dynamic Dialog Tools , CONTROL SET COLOR , DIALOG SET COLOR , DIALOG REDRAW
Example	<code>CONTROL REDRAW hDlg, %ID_LABEL1</code>

Purpose	Send a message to a control.
Syntax	<code>CONTROL SEND <i>hDlg</i>, <i>id</i>&, <i>Msg</i>&, <i>wParam</i>&, <i>lParam</i>& [TO <i>lResult</i>&]</code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id</i>& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p><i>Msg</i>& is the message you want to send the control.</p> <p><i>wParam</i>& is the first message parameter. <i>lParam</i>& is the second message parameter. The values of <i>wParam</i>& and <i>lParam</i>& are message-dependent. By default, Classic PowerBASIC passes these parameters BYVAL. If the target control is expected to return or alter the values passed in the <i>wParam</i>& and <i>lParam</i>& parameters, pass them using VARPTR or the return values will be discarded. For example:</p> <pre>' Retrieve an Edit control's Current Selection CONTROL SEND CB.HNDL, %ID_EDIT1, %EM_GETSEL, VARPTR(Sel1&), VARPTR(Sel2&)</pre> <p>CONTROL SEND does not return from execution until the control's callback has processed the message. This synchronous behavior is quite different to the behavior of CONTROL POST, which simply places the message in the control's message queue (for processing at a later time) and immediately returns. On this basis, CONTROL SEND can receive a return value from the message, but CONTROL POST cannot.</p>
TO	<p>The return value from the message can optionally be assigned to <i>lResult</i>&.</p> <p>If CONTROL SEND sends a message that arrives back in the same callback as the message originated, care should be exercised to ensure that critical STATIC and GLOBAL variables are not unexpectedly altered by the second message processing code in the callback. This is known as re-entrant code design.</p>
Restrictions	<p>To send a custom message to a dialog, use a message value in the range of (%WM_USER + 500) to (%WM_USER + &H07FFF), or use the RegisterWindowMessage API to obtain a unique message value from the operating system. Using messages with a numeric value of less than %WM_USER + 500 may conflict with Windows Common Control messages.</p>
See also	Dynamic Dialog Tools , CONTROL HANDLE , CONTROL POST , DIALOG POST , DIALOG SEND
Example	<pre>' Programmatically click a button: CONTROL SEND hDlg, %ID_BTN1, %BM_CLICK, 0, 0</pre>

Purpose	Set the Check State for a CHECK3STATE or CHECKBOX control.
Syntax	<code>CONTROL SET CHECK <i>hDlg</i>, <i>id&</i>, <i>checkstate&</i></code>
Remarks	<p>With a checkbox control, the Check State is set (checked) when an 'X' symbol is shown in the check box. The Check State is deemed unset (unchecked or cleared) when the check box is empty.</p> <p>A CHECK3STATE control supports a third state, known as indeterminate. In this state, the check box is grayed.</p> <p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the button control with a CONTROL SHOW statement.</p> <p>For CHECKBOX controls, set <i>checkstate&</i> to zero (0) to uncheck (unset or clear) the Check State of the control, or one (1) to check (set) the Check State of the control.</p> <p>For CHECK3STATE controls, set <i>checkstate&</i> to zero (0) to uncheck (unset or clear) the Check State of the control, one (1) to check (set) the Check State (display an 'X' symbol in the box), or two (2) to set the indeterminate state (display a grayed check box).</p> <div>To set the Check State of OPTION controls, use the CONTROL SET OPTION statement.</div>
See also	Dynamic Dialog Tools , CONTROL ADD CHECK3STATE , CONTROL ADD CHECKBOX , CONTROL ADD OPTION , CONTROL GET CHECK , CONTROL SET OPTION , TREEVIEW GET CHECK , TREEVIEW SET CHECK

CONTROL SET CLIENT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the size of a control to a specific client area size.
Syntax	<code>CONTROL SET CLIENT <i>hDlg</i>, <i>id&</i>, <i>wide&</i>, <i>high&</i></code>
Remarks	Client size may be smaller than total size, depending on the type of borders used. The client area is the part inside the borders of a control, which varies depending upon the style and <code>exstyle</code> at creation. In a control without borders, the client size and total size is the same. As an alternate example, a control with the <code>%WS_BORDER</code> style will typically have a client area a few pixels smaller than the total size.
<i>hDlg</i>	Handle of the dialog that owns the control.
<i>id&</i>	The unique control identifier, assigned to the control with the <code>CONTROL SHOW</code> statement.
<i>wide&</i> , <i>high&</i>	Integer expressions, variables , or numeric literal values, specifying the desired size of the client area. Width and height are specified in pixels or dialog units, depending upon the system used when the parent dialog was created.
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , CONTROL GET LOC , CONTROL GET SIZE , CONTROL SET LOC , CONTROL SET SIZE

Purpose	Set the color of a control to a specific RGB foreground and background color.
Syntax	<code>CONTROL SET COLOR <i>hDlg</i>, <i>id</i>&, <i>foreclr</i>&, <i>backclr</i>&</code>
Remarks	<p><i>hDlg</i> identifies the controls parent dialog, and <i>id</i>& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p>Color values of <i>foreclr</i>& and <i>backclr</i>& must be in the range of &H0 to &H00FFFFFF. RGB can be a useful function to derive a 32-bit color value from discrete Red, Green and Blue values:</p>
<i>foreclr</i> &	The foreground color parameter <i>foreclr</i> & is used to set the color of the text displayed in the control. If <i>foreclr</i> & = -1&, the default foreground text color is used.
<i>backclr</i> &	<p>The <i>backclr</i>& parameter specifies the color of the background behind the text in the control. If <i>backclr</i>& = -1&, the default background text color is used. If <i>backclr</i>& = -2&, the text background is not painted, allowing the background to show "through" the text. The non-visible background style may produce undesirable side effects with some controls. For example, on a FRAME control, the caption text will appear superimposed over an unbroken frame.</p> <p>In 16-bit or greater color-depth mode, the specified RGB color is used when the background of the control is drawn. However, in 8-bit (256-color) mode, the color system works quite differently. Behind the scenes in Windows, the base system palette usually contains 20 solid colors that are not dithered when drawn on the controls background. These solid-colors are ideal for control background colors with DDT dialogs in 256-color mode.</p> <p>Conversely, when using a non-solid RGB color value, Windows will dither (approximate) the color to draw the control, using combinations of two or more colors. This usually produces an undesirable pattern effect.</p> <p>To avoid these problems when in 256-color mode, controls should be colored with one of the 20 standard (solid) system colors, or the default color should be used instead. Classic PowerBASIC includes the following 10 built-in equates for help with the selection of a standard solid color:</p> <pre>%RGB_BLACK %RGB_BLUE %RGB_GREEN %RGB_CYAN %RGB_RED %RGB_MAGENTA %RGB_YELLOW %RGB_WHITE %RGB_GRAY %RGB_LIGHTGRAY</pre> <p>Many non standard colors are also built into the compiler, see the Built In RGB Color Equates topic for a complete list.</p> <p>If you prefer to disable color in 256-color mode, the number of colors can</p>

be easily tested with the following code:

```
' Determine number of colors
LOCAL hDC AS DWORD, iColors AS LONG

hDC = GetWindowDC(GetDesktopWindow())
iColors = 2 & ^ (GetDeviceCaps(hDC, %BITSPIXEL) * GetDeviceCaps(hDC,
%PLANES))
ReleaseDC GetDesktopWindow(), hDC
IF iColors > 256 THEN
    CONTROL SET COLOR hDlg, idctl&, -1, RGB(0,255,100)
```

In 256-color mode on most computers, the values of the standard 20 system colors can be found by requesting the first and last 10 (0 to 9, and 246 to 255 inclusive) entries from the `GetSystemPaletteEntries` API function, as follows:

```
' Fill array with solid colors
DIM hDC AS DWORD, Cols AS LONG, x AS LONG

hDC = GetWindowDC(GetDesktopWindow)
Cols = GetDeviceCaps(hDC, %NUMRESERVED)
REDIM lp(1 TO Cols) AS LONG
x& = GetSystemPaletteEntries(hDC, 0, Cols \ 2, BYVAL VARPTR(lp(1)))
x& = GetSystemPaletteEntries(hDC, 256 - x&, Cols - x&, BYVAL
VARPTR(lp(x& + 1)))
ReleaseDC GetDesktopWindow, hDC
' Array lp() now contains the solid color table
```

For more information on working with palettes in 256-color mode, please consult WIN32.HLP or visit <http://msdn.microsoft.com>.

Restrictions Windows does not permit the color of standard push button controls, line controls, image controls, image buttons, and most common controls to be altered by the standard `CONTROL SET COLOR` techniques.

To create a colored push button or colored region on a dialog, the preferred solution is to use an [IMGBUTTON/IMAGEBUTTONX](#) or [IMAGE/IMAGEX](#) control, with a suitably colored bitmap. Some common controls offer specific ways to set their color. For example, the background color of a List View control can be set with the `%LVM_SETBKCOLOR` message.

When dynamically changing colors of a control from within a [callback](#) (i.e., after the `DIALOG SHOW` statement), a `CONTROL SET COLOR` statement should be immediately followed by an explicit [CONTROL REDRAW](#) statement.

Without this forced control redraw, the control background color change may not become evident to the user until the control is eventually repainted in the normal course of user interaction. For example, a normal repaint may only occur if the control becomes obscured and then uncovered by another window. Ensuring a timely repaint of the control will guarantee the control maintains an up-to-date appearance at all times.

When updating the colors of multiple controls at the same time, a [DIALOG](#)

[REDRAW](#) is usually more efficient than multiple CONTROL REDRAW statements.

See also

[Built In RGB Color Equates](#), [Dynamic Dialog Tools](#), [CONTROL GET CLIENT](#), [CONTROL GET SIZE](#), [CONTROL REDRAW](#), [CONTROL SET FONT](#), [DIALOG REDRAW](#), [DIALOG SET COLOR](#)

Example

```
' Set the color with discrete RGB values
CONTROL SET COLOR hDlg, idBtn&, RGB(255,255,255), RGB(0,0,255)
```

```
' Or we could use the built-in equates:
CONTROL SET COLOR hDlg, idBtn&, %RGB_WHITE, %RGB_BLUE
```

Purpose Set the keyboard focus to the specified control.

Syntax `CONTROL SET FOCUS hDlg, id&`

Remarks *hDlg* refers to the dialog that owns the control.

id& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.

The control that owns the focus will receive all keyboard messages. Many controls change their appearance when they receive (and lose) keyboard focus, usually by the display of a "focus rectangle" around or on the control that has keyboard focus. Only one control can have keyboard focus at any moment, and situations can arise where no controls have focus.

Controls that include a "notify" [style](#) (i.e., %BS_NOTIFY) will receive a notification message when focus is gained or lost. That is, when one such control loses focus, it receives a message to that effect and the control gaining focus may also receive an appropriate focus notification message.

When a control gains focus the parent dialog will also be set as the foreground window.

Windows does not guarantee the order in which focus notification messages are dispatched to the control losing focus and the control gaining focus. Applications should not rely on the order in which these types of messages are received.

Restrictions CONTROL SET FOCUS cannot be used to set the focus of a control in a separate thread.

See also [Dynamic Dialog Tools](#)

CONTROL SET FONT statement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose	Select a font to be used for a particular Windows Control.
Syntax	<code>CONTROL SET FONT <i>hDlg</i>, <i>id</i>&, <i>fonthndl</i>&</code>
<i>hDlg</i>	Handle of the dialog in which owns the control.
<i>id</i> &	Unique identifier for the control which was assigned with a CONTROL SHOW statement.
<i>fonthndl</i> &	The numeric handle returned by the FONT NEW statement when the font was created.
Remarks	<p>The font specified by <i>fonthndl</i>& is selected to be used by this particular control, until or unless it is changed with another CONTROL SET FONT statement.</p> <p>You can predefine virtually any number of fonts and attributes by executing FONT NEW statements for each of them. That makes them ready for immediate use when selected by CONTROL SET FONT, GRAPHIC SET FONT, and XPRINT SET FONT.</p>
See also	DIALOG FONT , FONT END , FONT NEW , GRAPHIC SET FONT , XPRINT SET FONT

CONTROL SET IMAGE statement

[Top](#) [Previous](#) [Next](#)

Purpose Change the icon or bitmap displayed in an [IMAGE](#) control. The new image is not re-sized to fit the size of the control.

Syntax `CONTROL SET IMAGE hDlg, id&, newimage$`

Remarks *hDlg* refers to the dialog that owns the control.
id& is the unique control identifier as assigned to the control with the [CONTROL ADD IMAGE](#) statement.
newimage\$ specifies the name of the bitmap or icon in the [resource file](#) (.PBR). If the image resource uses an identifier, *newimage\$* should begin with a Number symbol (#), followed by the integer identifier in ASCII format. For example, "#998". Otherwise, the text identifier name should be used.

Restrictions Images can only be exchanged with images of the same type. That is, if the control is displaying a bitmap then the replacement image must also be a bitmap. If the control is displaying an icon, the replacement image must also be an icon. For best results, icons should be 32x32 pixels.

When an image is changed, CONTROL SET IMAGE automatically releases the old image from memory. Previous versions of Classic PowerBASIC placed the onus on the programmer to release the old image handle.

See also [Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGE](#), [CONTROL ADD IMAGEX](#), [CONTROL ADD IMGBUTTON](#), [CONTROL ADD IMGBUTTONX](#), [CONTROL GET CLIENT](#), [CONTROL GET SIZE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTON](#), [CONTROL SET IMGBUTTONX](#)

CONTROL SET IMAGEX statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the icon or bitmap displayed in an IMAGEX control. The new image is re-sized to fit the size of the control.
Syntax	<code>CONTROL SET IMAGEX <i>hDlg</i>, <i>id&</i>, <i>newimage\$</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with the CONTROL ADD IMAGEX statement.</p> <p><i>newimage\$</i> specifies the name of the bitmap or icon in the resource file (.PBR). If the image resource uses an identifier, <i>newimage\$</i> should begin with a Number symbol (#), followed by the integer identifier in ASCII format. For example, "#998". Otherwise, the text identifier name should be used.</p>
Restrictions	<p>Images can only be exchanged with images of the same type. That is, if the control is displaying a bitmap then the replacement image must also be a bitmap. If the control is displaying an icon, the replacement image must also be an icon. For best results, icons should be 32x32 pixels.</p> <div><p>When an image is changed, CONTROL SET IMAGEX automatically releases the old image from memory. Previous versions of Classic PowerBASIC placed the onus on the programmer to release the old image handle.</p></div>
See also	Dynamic Dialog Tools , CONTROL ADD GRAPHIC , CONTROL ADD IMAGE , CONTROL ADD IMAGEX , CONTROL ADD IMGBUTTON , CONTROL ADD IMGBUTTONX , CONTROL SET IMAGE , CONTROL SET IMGBUTTON , CONTROL SET IMGBUTTONX

CONTROL SET IMGBUTTON statement

[Top](#) [Previous](#)
[Next](#)

Purpose Change the icon or bitmap displayed in an [IMAGE](#) control. The new image is not re-sized to fit the size of the control.

Syntax `CONTROL SET IMGBUTTON hDlg, id&, newimage$`

Remarks *hDlg* refers to the dialog that owns the control.
id& is the unique control identifier as assigned to the control with the [CONTROL ADD IMGBUTTON](#) statement.
newimage\$ specifies the name of the bitmap or icon in the [resource file](#) (.PBR). If the image resource uses an identifier, *newimage\$* should begin with a Number symbol (#), followed by the integer identifier in ASCII format. For example, "#998". Otherwise, the text identifier name should be used.

Restrictions Images can only be exchanged with images of the same type. That is, if the control is displaying a bitmap then the replacement image must also be a bitmap. If the control is displaying an icon, the replacement image must also be an icon. For best results, icons should be 32x32 pixels.

When an image is changed, CONTROL SET IMGBUTTON automatically releases the old image from memory. Previous versions of Classic PowerBASIC placed the onus on the programmer to release the old image handle.

See also [Dynamic Dialog Tools](#), [CONTROL ADD GRAPHIC](#), [CONTROL ADD IMAGE](#), [CONTROL ADD IMAGEX](#), [CONTROL ADD IMGBUTTON](#), [CONTROL ADD IMGBUTTONX](#), [CONTROL SET IMAGE](#), [CONTROL SET IMAGEX](#), [CONTROL SET IMGBUTTONX](#)

CONTROL SET IMGBUTTONX statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Change the icon or bitmap displayed in an IMAGEX control. The new image is re-sized to fit the size of the control.
Syntax	<code>CONTROL SET IMGBUTTONX <i>hDlg</i>, <i>id&</i>, <i>newimage\$</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with the CONTROL ADD IMGBUTTONX statements.</p> <p><i>newimage\$</i> specifies the name of the bitmap or icon in the resource file (.PBR). If the image resource uses an identifier, <i>newimage\$</i> should begin with a Number symbol (#), followed by the integer identifier in ASCII format. For example, "#998". Otherwise, the text identifier name should be used.</p>
Restrictions	<p>Images can only be exchanged with images of the same type. That is, if the control is displaying a bitmap then the replacement image must also be a bitmap. If the control is displaying an icon, the replacement image must also be an icon. For best results, icons should be 32x32 pixels.</p> <div><p>When an image is changed, CONTROL SET IMGBUTTONX automatically releases the old image from memory. Previous versions of Classic PowerBASIC placed the onus on the programmer to release the old image handle.</p></div>
See also	Dynamic Dialog Tools , CONTROL ADD GRAPHIC , CONTROL ADD IMAGE , CONTROL ADD IMAGEX , CONTROL ADD IMGBUTTON , CONTROL ADD IMGBUTTONX , CONTROL SET IMAGE , CONTROL SET IMAGEX , CONTROL SET IMGBUTTON

CONTROL SET LOC statement

[Top](#) [Previous](#) [Next](#)

Purpose Move the control to a new location in the dialog.

Syntax `CONTROL SET LOC hDlg, id&, x&, y&`

Remarks *hDlg* refers to the dialog that owns the control.
id& is the unique control identifier as assigned to the control with a CONTROL SHOW statement.
x& and *y&* specify the new location for the upper left corner of the control. These coordinates are relative to the upper left corner of the client area of the parent dialog (0,0), and are specified in the same terms (pixels or dialog units) as the parent dialog.
The location coordinates may be negative or larger than the width of the parent dialog's client area, causing the control to be clipped (partially displayed) or completely hidden.
This technique is often employed to capture the ENTER key, by creating a default button (%BS_DEFAULT) and positioning the control outside of the client area of the dialog - even though the control is not visible, it is still active and can respond to control accelerator keystrokes, etc.

See also [Dynamic Dialog Tools](#), [CONTROL GET CLIENT](#), [CONTROL GET LOC](#), [CONTROL GET SIZE](#), [CONTROL SET SIZE](#)

CONTROL SET OPTION statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the Check State for an OPTION (radio) control, and unset the Check State for other OPTION buttons in a group.
Syntax	<code>CONTROL SET OPTION <i>hDlg</i>, <i>id&</i>, <i>minid&</i>, <i>maxid&</i></code>
Remarks	<p>The Check State is deemed set (checked) when the check box is selected, and unset (unchecked or clear) if the check box is empty. Only one OPTION control in a group of OPTION controls should ever have its Check State set at any given time. OPTION controls in a group should be assigned unique sequential identifier numbers.</p> <p><i>hDlg</i> refers to the dialog that owns the OPTION controls.</p> <p><i>id&</i> is the unique control identifier as assigned to the button control with a CONTROL ADD OPTION statement. CONTROL SET OPTION sets the Check State for this control, and unsets the Check State for all of the remaining OPTION controls whose identifiers are included in the range <i>minid&</i> through <i>maxid&</i>, inclusive.</p> <p>The first OPTION control in a group should have the style %WS_GROUP to mark the beginning of a group of buttons, and the first non-OPTION control after the group should also have this style set. If there are no other controls after the group, add %WS_GROUP to the first control in the dialog. This ensures keyboard navigation with the arrow buttons will operate within the group of OPTION controls.</p>

See also [Dynamic Dialog Tools](#), [CONTROL ADD OPTION](#), [CONTROL GET CHECK](#)

Example

```
#INCLUDE "DDT.INC"
%OPT1 = 101
%OPT2 = 102
%OPT3 = 103
%OPT4 = 104
%OPT5 = 105

FUNCTION PBMAIN
    DIM hDlg AS DWORD
    DIALOG NEW 0, "OPTION control test", , ,100, 100, _
        %WS_SYSMENU OR %WS_CAPTION TO hDlg
    CONTROL ADD OPTION, hDlg, %OPT1, "Option 1", 10, 6, 50, 14, _
        %WS_GROUP OR %WS_TABSTOP
    CONTROL ADD OPTION, hDlg, %OPT2, "Option 2", 10, 20, 50, 14
    CONTROL ADD OPTION, hDlg, %OPT3, "Option 3", 10, 34, 50, 14
    CONTROL ADD OPTION, hDlg, %OPT4, "Option 4", 10, 48, 50, 14
    CONTROL ADD OPTION, hDlg, %OPT5, "Option 5", 10, 62, 50, 14
    CONTROL ADD BUTTON, hDlg, %IDOK, "OK", 25, 80, 50, 14, _
        %WS_GROUP OR %WS_TABSTOP

    ' Set the initial state to OPTION button 3
    CONTROL SET OPTION hDlg, %OPT3, %OPT1, %OPT5

    DIALOG SHOW MODAL hDlg
```

END FUNCTION

Purpose	Change the size of a control.
Syntax	<code>CONTROL SET SIZE <i>hDlg</i>, <i>id&</i>, <i>wide&</i>, <i>high&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with the CONTROL SHOW statement.</p> <p><i>wide&</i> and <i>high&</i> specify the new dimensions, in the same terms (pixels or dialog units) as the parent dialog. A control may extend past the edge of the client area of the parent dialog, but will appear clipped (partially displayed).</p>
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , CONTROL GET LOC , CONTROL GET SIZE , CONTROL SET LOC

CONTROL SET TEXT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the text in a control.
Syntax	<code>CONTROL SET TEXT <i>hDlg</i>, <i>id&</i>, <i>txt\$</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p><i>txt\$</i> is the new text for the control. Any existing text in the control is replaced with the new text.</p>
See also	Dynamic Dialog Tools , CONTROL GET TEXT

Purpose	Set a value in the user data area of a DDT control.
Syntax	<code>CONTROL SET USER <i>hDlg</i>, <i>id&</i>, <i>index&</i>, <i>usrval&</i></code>
Remarks	<p>Each DDT control has a user data area consisting of eight Long-integer values which may be used at the programmer's discretion to save relevant data. CONTROL SET USER allows one of the values to be set, based upon the index parameter value (1 through 8).</p> <p><i>hDlg</i> refers to the dialog that owns the control.</p> <p><i>id&</i> is the unique control identifier as assigned to the control with a CONTROL SHOW statement.</p> <p><i>index&</i> is the index number of the user data value to set, in the range 1 to 8 inclusive.</p> <p><i>usrval&</i> is the Long-integer data value to store in the user data area.</p>
Restrictions	Data in the user data area is lost when the control is destroyed. The data area is completely separate from the %GWL_USERDATA area maintained by Windows.
See also	Dynamic Dialog Tools , COMBOBOX GET USER , COMBOBOX SET USER , CONTROL GET USER , DIALOG GET USER , DIALOG SET USER , LISTBOX GET USER , LISTBOX SET USER , LISTVIEW GET USER , LISTVIEW SET USER , TREEVIEW GET USER , TREEVIEW SET USER

Purpose Change the visible state of a control.

Syntax `CONTROL SHOW STATE hDlg, id&, showstate& [TO lResult&]`

Remarks CONTROL SHOW STATE is used to alter the state and/or appearance of the specified control, identified by the parent dialog handle *hDlg*, and control *id&* unique identifier combination.

showstate& can be one of the following (with a value in the range from 0 to 10) as defined in the [WIN32API.INC](#) file):

<code>%SW_HIDE</code>	Hide the control.
<code>%SW_MAXIMIZE</code>	Maximize the specified control.
<code>%SW_MINIMIZE</code>	Minimize the specified control.
<code>%SW_RESTORE</code>	Activate and display the control. If the control is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized control.
<code>%SW_SHOW</code>	Activate the control and display it in its current size and position.
<code>%SW_SHOWMAXIMIZED</code>	Synonym of <code>%SW_MAXIMIZE</code> .
<code>%SW_SHOWMINIMIZED</code>	Activate the control and minimize it.
<code>%SW_SHOWNA</code>	Display the control in its current state without activating it. The currently active window/control remains active.
<code>%SW_SHOWNOACTIVATE</code>	Display the control in its most recent size and position without activating it. The currently active window/control remains active.
<code>%SW_SHOWNORMAL</code>	Activate and display the control. If the control is minimized or maximized, it is restored it to its original size and position.

If the optional TO clause is included, the *lResult&* [variable](#) is assigned the value zero if the control was previously not visible, or non-zero if it was previously visible.

See also [Dynamic Dialog Tools](#), [CONTROL DISABLE](#), [CONTROL ENABLE](#), [DIALOG SHOW STATE](#)

Purpose	Return the cosine of an angle.
Syntax	<code>y = COS (numeric_expression)</code>
Remarks	<p><i>numeric_expression</i> is an angle specified in radians. To convert radians to degrees, multiply by 57.29577951308232###. To convert degrees to radians, multiply by 0.0174532925199433###. For more information on radians, see the ATN function.</p> <p>COS returns an Extended-precision value that always ranges between -1 and +1 inclusive.</p> <p>The Inverse Cosine (ARCCOS) of a value can be calculated as follows:</p> <pre>pi## = 3.141592653589793## ArcCos = pi## / 2 - ATN(Value / SQR(1 - Value * Value))</pre> <p>The Hyperbolic Cosine (COSH) of a value can also be calculated:</p> <pre>CosH = (EXP(Value) + EXP(-Value)) / 2</pre> <p>The Inverse Hyperbolic Cosine (ARCCOSH) of a value can also be calculated:</p> <pre>ArcCosH = LOG(Value + SQR(Value * Value - 1))</pre> <pre>' Useful Macro functions MACRO Pi = 3.141592653589793## MACRO DegreesToRadians(dpDegrees) = (dpDegrees*0.0174532925199433##) MACRO RadiansToDegrees(dpRadians) = (dpRadians*57.29577951308232##)</pre>
See also	ATN , SIN , TAN
Example	<pre>pi## = 3.141592653589793## ' we could also use pi## = ATN(1) * 4 FOR I& = 0 TO 360 STEP 45 x\$ = "Cosine of " + FORMAT\$(I&, "* 0") + _ " degrees = " + FORMAT\$(COS(pi## / 180 * _ I&), "* 0.00") NEXT I&</pre>
Result	<pre>Cosine of 0 degrees = 1.00 Cosine of 45 degrees = 0.71 Cosine of 90 degrees = 0.00 Cosine of 135 degrees = -0.71 Cosine of 180 degrees = -1.00 Cosine of 225 degrees = -0.71 Cosine of 270 degrees = 0.00 Cosine of 315 degrees = 0.71 Cosine of 360 degrees = 1.00</pre>

Purpose	Center a string within the space of another string or User-Defined Type .
Syntax	<code>CSET [ABS] result_var = string_expression [USING string_expression]</code>
Remarks	CSET centers a string into the space of another string or variable of a User-Defined Type.
ABS	If ABS is specified, or <i>ustring_expression</i> is null (empty), CSET leaves the padding positions unchanged from their original content, rather than replacing them with spaces.
USING	<p>If <i>string_expression</i> is shorter than <i>result_var</i>, CSET centers <i>string_expression</i> within <i>result_var</i>, padding both sides with the first character in <i>ustring_expression</i>, or spaces if not specified.</p> <p>If <i>string_expression</i> is longer than <i>result_var</i>, CSET truncates <i>string_expression</i> from the right until it fits in <i>result_var</i>.</p> <p>CSET can be used to assign the content of a User-Defined Type to a User-Defined Type variable of a different class, or assign a dynamic string to a User-Defined Type. For example:</p> <pre>CSET MyType = MyType2 CSET MyType = a\$</pre> <p>LSET and RSET work similarly, but performs left and right-justification respectively.</p>
See also	CSET\$, GET , LET , LET (With Types) , LSET , LSET\$, PUT , RESET , RSET , RSET\$, STRINSERT\$, TYPE SET
Example	<pre>a\$ = RTRIM\$(REPEAT\$(5,"COOL ")) CSET ABS a\$ = "..Classic PowerBASIC.." ' result: "COOL ..Classic PowerBASIC.. COOL" CSET a\$ = "Classic PowerBASIC" USING "*" ' result: "*****Classic PowerBASIC*****"</pre>

Purpose	Return a string containing a centered (padded) string.
Syntax	<code>a\$ = CSET\$(string_expression, strlen& [USING ustring_expression])</code>
Remarks	CSET\$ centers the string string_expression into a string of <i>strlen&</i> characters.
USING	<p>If <i>ustring_expression</i> is null (empty) or is not specified, CSET\$ pads <i>string_expression</i> with space characters. Otherwise, CSET\$ pads the string with the first character of <i>ustring_expression</i>.</p> <p>If <i>string_expression</i> is shorter than <i>strlen&</i>, CSET\$ centers <i>string_expression</i> within <i>result_var</i>, padding both sides as described above; otherwise, CSET\$ returns the left-most <i>strlen&</i> bytes of <i>string_expression</i>.</p>
See also	CSET , GET , LET , LSET , LSET\$, PUT , RESET , RSET , RSET\$, STRINSERT\$, TYPE SET
Example	<pre>a\$ = CSET\$("Classic PowerBASIC", 20) ' result: " Classic PowerBASIC "</pre> <pre>a\$ = CSET\$("Classic PowerBASIC",20 USING "*") ' result: "*****Classic PowerBASIC*****"</pre>

CURDIR\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the current directory path for the specified drive.
Syntax	<code>s\$ = CURDIR\$[(drive\$)]</code>
Remarks	<i>drive\$</i> is an optional string expression , containing the drive letter of the target disk drive. If <i>drive\$</i> is not specified or is an empty string, the current directory path is returned for the default drive.
See also	CHDRIVE , CHDIR
Example	<pre>FUNCTION PBMAIN LOCAL FullCurrentPath\$ LOCAL CurrentDrive\$ FullCurrentPath\$ = CURDIR\$ IF MID\$(CURDIR\$,2,1) = ":" THEN CurrentDrive\$ = LEFT\$(CURDIR\$,2) END IF END FUNCTION</pre>

CVBYT, CVCUR, CVCUX, CVD, CVDWD, CVE, CVI, CVL, CVQ, CVS and CVWRD functions

Purpose Convert string data to numeric form.

Syntax

<i>bytevar?</i>	=	CVBYT(<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>curvar@</i>	=	CVCUR(<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>cuxvar@@</i>	=	CVCUX(<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>doublevar#</i>	=	CVD (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>doublewordvar???</i>	=	CVDWD(<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>extendedvar##</i>	=	CVE (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>integervar%</i>	=	CVI (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>longintvar&</i>	=	CVL (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>quadintvar&&</i>	=	CVQ (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>singlevar!</i>	=	CVS (<i>stringexpr</i> [<i>,</i> <i>offset</i>])
<i>wordvar??</i>	=	CVWRD(<i>stringexpr</i> [<i>,</i> <i>offset</i>])

Remarks The CVx functions return a number corresponding to a binary pattern stored in a string value. The [MKx](#) functions are complementary to the CVx functions. Do not confuse these functions with the [VAL](#) function, which converts a number stored as a printable text string (such as "123.45") into a numeric expression.

The CVx functions allow retrieving values beyond the first byte of the input string variable, where offset indicates at what position in the string the function should retrieve value. variable may be either a [User-Defined Type](#) or a string.

For example: "Value& = CVL(x\$, 3)" would extract the 3rd through 6th bytes of x\$ and convert these 4 bytes to the corresponding Long-integer value. In this example, x\$ must be at least 6 bytes long.

Function	Variable	Converts to
CVBYT	1-byte string	Byte
CVCUR	8-byte string	Currency
CVCUX	8-byte string	Extended-currency
CVD	8-byte string	Double-precision float
CVDWD	4-byte string	Double-word
CVE	10-byte string	Extended-precision float
CVI	2-byte string	Integer
CVL	4-byte string	Long-integer

CVQ	8-byte string	Quad-integer
CVS	4-byte string	Single-precision float
CVWRD	2-byte string	Word

Restrictions Expressions involving [Numeric Equates](#) and conditional compilation ([#IF](#)) may also include the CVQ function. This allows you to easily assign numeric values to an equate, based upon a meaningful mnemonic. In this context, the CVQ expression is limited to a length of eight bytes. For example:

```
%Mode  = CVQ("DEBUG")
%Style = CVQ("Cool")
```

CVS limits string display to seven significant digits.

See also [MKI\\$ and associated functions](#)

Purpose	Declare string constants within the source code to be read by READ\$ function.
Syntax	<code>DATA ["]item["] [[, ["]item["]] ...]</code>
Remarks	<p>DATA statements may only appear inside of Subs, Functions, Method, or Properties, and are visible only to the code in the procedure in which they appear. Each procedure may therefore have its own private data.</p> <p>Data may consist of virtually any text characters. Data items may be enclosed in quotes to preserve leading/trailing spaces, which are otherwise stripped during compilation.</p>
Restrictions	<p>There is a limit of 64 Kilobytes and 16384 separate data items per procedure. Previous versions of Classic PowerBASIC ignored plain text located immediately after a quoted literal up to the next comma or end-of-line; however, this is no longer acceptable and generates an Error 477 ("Syntax error").</p> <p>DATA statements cannot extend across more than one physical source code line using line continuation characters. Special care should be used when formatting DATA statements, especially if the data is to contain underscore and/or colon characters. The following examples highlight data items in blue:</p> <p>If an underscore appears <u>after</u> a comma, it is treated as the start of a quoted data string, rather than a line continuation character:</p> <pre>' Three data items exist in this line: DATA "Tom", "Dick", _Harry</pre> <p>The colon (statement separator) character, when used <u>within unquoted</u> string data, performs as a regular statement separator:</p> <pre>' Two data items and a separate statement DATA "Tom","Dick" : Harry& = 1</pre> <p>However, if a colon character appears <u>within</u> a <u>quoted</u> data string, it is treated as part of the data string:</p> <pre>' 3 data items DATA "Tom",Dick,":Harry& = 1"</pre>
See also	DATACOUNT , READ\$, VAL
Example	<pre>DATA "Abc", Bob, "Sally", 123 DATA 456.78, " leading space" DATA embedded "quotes within data"</pre>

DATACOUNT function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the total count of the number of local DATA items that can be read with the READ\$ function.
Syntax	<i>Count%</i> = DATACOUNT
Remarks	<p>DATACOUNT only returns the number of DATA items in the Sub, Function, Method, or Property in which it appears (i.e., local DATA statements).</p> <p>While it is not possible to directly read data from outside of the scope of current procedure, global data can be emulated easily by placing it inside a procedure returns data to the calling code. There is a limit of 64 Kilobytes and 16384 separate data items per procedure.</p>
See also	DATA , READ\$
Example	<pre>FUNCTION GetCategories(Category() AS STRING) AS LONG LOCAL x AS INTEGER REDIM Category(1 TO DATACOUNT) AS STRING FOR x = 1 TO DATACOUNT Category(x) = READ\$(x) NEXT x FUNCTION = DATACOUNT DATA Animal, Mineral, Vegetable, Alien END FUNCTION</pre>

DATE\$ system variable

[Top](#) [Previous](#) [Next](#)

Purpose Set or retrieve the system date.

Syntax

```
DATE$ = s$      ' sets system date according to s$
s$ = DATE$      ' s$ now contains system date
```

Remarks Assigning a properly formatted string value to DATE\$ sets the system date. You can also assign DATE\$ to a string variable, which stores 10 characters in the form "mm-dd-yyyy", where mm represents the month, dd the day, and yyyy the year.

To change the date, your date string must be formatted in one of the following ways:

```
mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy
```

For example, DATE\$ = "11-09-84" sets the system date to November 9, 1984.

Restrictions The year assigned to the DATE\$ system variable must be within the range 1980 to 2099. DATE\$ never returns locale-specific date formats. When assigning a two digit year, any value less than 81 will be assumed to be in the 2000's and any value greater than 80 will be assumed to be in the 1900's.

See also [TIMES\\$](#)

Purpose	Explicitly declare a Sub , Function or Callback Function.
Syntax	<pre>DECLARE {SUB FUNCTION} <i>ProcName</i> [BDECL CDECL SDECL] [LIB "<i>LibName</i>"] [ALIAS "<i>AliasName</i>"] [(<i>arguments</i>)] [AS <i>type</i>] DECLARE CALLBACK FUNCTION <i>ProcName</i> [(<i>()</i>) AS LONG] DECLARE THREAD FUNCTION <i>ProcName</i> (BYVAL <i>var</i> AS (LONG DWORD)) AS {LONG DWORD}</pre>
Remarks	The DECLARE statement has the following parts:
CALLBACK	<p>When declaring a Callback Function, the CALLBACK keyword <i>follows</i> the DECLARE keyword, and no parameters should be specified. Parentheses and the AS LONG return type may be added for clarity.</p> <p>Callback Functions are reserved for use with Dynamic Dialog Tools (DDT) functions.</p>
THREAD	When declaring a Thread function, the THREAD keyword follows the DECLARE keyword. It must take exactly one Long-integer or Double-word (DWORD) parameter by value (BYVAL), and must return a Long-integer or Double-word value.
<i>ProcName</i>	<p>The name of the Sub or Function to be declared. For Functions, a type-specifier may be appended (just like an ordinary variable name) to specify the data type of the Function's return value, in place of the [AS type] clause.</p> <p>Future versions of Classic PowerBASIC may not support type-specifier symbols for the Function return type. Specify the return data type with an explicit AS type clause in all DECLARE and FUNCTION definitions to ensure future compatibility.</p>
BDECL	<p>Specifies that the declared procedure uses the BASIC/Pascal calling convention. When a procedure calls a BDECL procedure, it passes its parameters on the stack from left to right.</p> <p>It is the responsibility of the called procedure to clean up the stack before returning to the calling procedure. Therefore, all Classic PowerBASIC Subs and Functions that specify the BDECL convention automatically clean up the stack before execution returns to the calling code.</p>
CDECL	<p>Specifies that the declared procedure uses the C calling convention. When a procedure calls a CDECL procedure, it passes its parameters on the stack from right to left.</p> <p>In addition, the calling procedure must remove any passed parameters from the stack as part of the return process. When Classic PowerBASIC code calls Subs and Functions using the CDECL convention, the stack is</p>

cleaned automatically after execution returns from the called code.

In the event the called procedure is imported or exported, Classic PowerBASIC will automatically create a lowercase ALIAS, prefixed with an underscore. Any periods in the name are replaced with underscores at the same time.

CDECL may be used for declaring external procedures written in the C language, or another language that follows C calling and parameter-passing conventions. The following two declarations are equivalent, indicating how the default ALIAS name would be created by Classic PowerBASIC:

```
DECLARE SUB C_Function CDECL ()  
DECLARE SUB C_Function CDECL ALIAS "_c_function" ()
```

SDECL

This is the default if neither BDECL nor CDECL are specified. SDECL (and its synonym STDCALL), specifies that the declared procedure uses the "Standard Calling Convention" as defined by Microsoft. When calling an SDECL procedure, parameters are passed on the stack from right to left. Classic PowerBASIC Subs and Functions that use the SDECL/STDCALL convention automatically clean up the stack before execution returns to the calling code.

LIB

A [string literal](#) or [equate](#) that specifies the name of the module in which a procedure or function is located. This allows you to call Subs or Functions that reside in [DLLs](#). Unlike 16-bit Windows, you must include the .DLL extension in the name of the DLL you wish to access.

```
DECLARE SUB MySub LIB "ZUSER32.DLL" ()
```

ALIAS

A string literal that identifies the name and capitalization of the procedure in the external DLL. This lets you call a procedure or function by a name *other* than what it was originally named. This is useful if you want to abbreviate a long name, or if the original name of a function contains characters that are illegal in Classic PowerBASIC. The *AliasName* is the routine's actual name, and *Name* is the title that you can use in Classic PowerBASIC. For example:

```
DECLARE SUB ShortName ALIAS "VeryLongProcName"()  
DECLARE FUNCTION LegalName ALIAS "Illegal$Name"()
```

Although a *ProcName* must be unique, you may use the same *AliasName* in multiple declarations. This is particularly useful for avoiding AS ANY in cases where a procedure is designed to receive several different types of parameters.

```
DECLARE FUNCTION AddAtom LIB "KERNEL32.DLL" ALIAS "AddAtomA"  
    (lpString AS ASCIIZ) AS WORD  
DECLARE FUNCTION AddIntAtom LIB "KERNEL32.DLL" ALIAS "AddAtomA"  
    (BYVAL lpString AS DWORD) AS WORD
```

The ALIAS clause is very important when importing or exporting

Subs and Functions from DLLs. Omitting the **ALIAS** clause or incorrectly capitalizing the alias name are common causes of DLL load failure problems. Please refer to the [SUB](#) and [FUNCTION](#) sections for more information.

Passing parameters

Arguments

Contains the name(s) or the type of each parameter, in the order they are passed, for up to 32 parameters. If you wish to call a SUB or FUNCTION in a DLL, you must describe the target SUB or FUNCTION with an explicit DECLARE statement. The DECLARE must physically precede any reference to the target procedure.

Previous versions of Classic PowerBASIC required that you create an explicit DECLARE statement if you wished to execute a SUB or function which did not physically precede the reference to it. This extra work is no longer required, as Classic PowerBASIC resolves all forward references to internal procedures automatically.

DECLARE statements for a Sub/Function imported from a DLL must still precede any reference to the procedure.

The complete *arguments* list must be specified for each routine. Each parameter may be defined in one of three ways:

- List only its type name ([INTEGER](#), [DOUBLE](#), etc.)
- List a variable name with a type-specifier appended (count%, txt\$)
- Use the AS clause to specify the type (count AS INTEGER, text AS [STRING * 100](#), etc.).

Legal type names for *arguments* include ANY, [ASCIIZ](#), [BYTE](#), [CUR](#), [CUX](#), [DOUBLE](#), [DWORD](#), [EXT](#), [INTEGER](#), [LONG](#), [PTR](#), [QUAD](#), [SINGLE](#), [STRING](#), [WORD](#) and [ARRAY](#). The ARRAY keyword is used in conjunction with one of the other types to specify an entire array of that type. For example:

```
DECLARE SUB KerPlunk(INTEGER ARRAY, DOUBLE)
```

declares a procedure called *KerPlunk*, which takes an entire Integer array and a Double-precision variable as parameters. You can also name the parameters using the AS keyword:

```
DECLARE SUB KerPlunk(iArray() AS INTEGER, dVar AS DOUBLE)
```

The following four declare statements are equivalent:

```
DECLARE SUB KerPlunk(x) ' if DEFINT A-Z is in effect
DECLARE SUB KerPlunk(x%)
DECLARE SUB Plunk(x AS INTEGER)
DECLARE SUB KerPlunk(INTEGER)
```

When parameters are passed by reference (BYREF), the full 32-bit

address of the variable passed to the routine is placed on the stack.

Using ANY disables type checking for a given parameter, and passes the full 32-bit address of the variable passed on the stack. Since the internal format of variables differ greatly by type, you must use caution to be absolutely certain your code knows the data type in each invocation. Normally, a second parameter is used to specify the actual type of the ANY parameter.

You can use the BYVAL or BYREF keywords to specify that a given parameter should always be passed by value or by reference, respectively. When a Sub/Function definition specifies either a BYREF parameter or a [pointer](#) variable parameter, the calling code may freely pass a BYVAL DWORD or a pointer instead. While the use of the explicit BYVAL override in the calling code is optional, it is recommended for clarity. It is necessary to explicitly declare all pointer parameters as BYVAL (BYVAL x AS BYTE PTR). Failure to do so will generate compile-time [Error 549](#) ("BYVAL required with pointers").

Additional information on BYVAL/BYREF/BYCOPY parameter passing can be found in the [CALL](#) statement topic.

Optional parameters

Classic PowerBASIC now supports two syntax formats for optional parameters: the classic optional parameter syntax using brackets "[.]", and the new syntax using the OPTIONAL (or OPT) keyword. We'll discuss each one in turn.

Using OPTIONAL/OPT

DECLARE statements may specify one or more parameters as optional by preceding the parameter with either the keyword OPTIONAL (or the abbreviation OPT). Optional parameters are only allowed with CDECL or SDECL calling conventions, not BDECL.

When a parameter is declared optional, all subsequent parameters in the declaration are optional as well, whether or not they specify an explicit OPTIONAL or OPT directive. The following two lines are equivalent, with both second and third parameters being optional:

```
DECLARE SUB sABC(a&, OPTIONAL BYVAL b&, OPTIONAL BYVAL c&) AS LONG
DECLARE SUB sABC(a&, OPT BYVAL b&, BYVAL c&) AS LONG
```

[VARIANT](#) variables are particularly well suited for use as an optional parameter. If the calling code omits an optional VARIANT parameter, (BYVAL or BYREF), Classic PowerBASIC (and most other compilers) substitute a variant of type %VT_ERROR which contains an error value of %DISP_E_PARAMNOTFOUND (&H80020004). In this case, you can check for this value directly, or use the [ISMISSING\(\)](#) function to determine

whether the parameter was physically passed or not.

When optional parameters (other than VARIANT) are omitted in the calling code, the stack area normally reserved for those parameters is zero-filled.

If the parameter is defined as a BYVAL parameter, it will have the value zero. For [TYPE](#) or [UNION](#) variables passed BYVAL, the compiler will pass a string of binary zeroes of length [SIZEOF](#)(*Type_or_union_var*).

If the parameter is defined as a BYREF parameter, [VARPTR](#) (*Varname*) will equal zero; when this is true, any attempt to use *Varname* in your code will result in a General Protection Fault or memory corruption. You should use the ISMISSING() function first to determine whether it is safe to access the parameter.

Using classic optional parameters

When declaring a CDECL SUB or FUNCTION, you can specify trailing parameters as optional, using a set of brackets [...]:

```
DECLARE SUB KerPlunk CDECL (x%, y% [, z%])
```

Note that the comma separating the y% parameter from the optional z% parameter is inside the brackets. The following calling sequences would then be valid:

```
CALL KerPlunk (x%, y%)  
CALL KerPlunk (x%, y%, z%)
```

Optional parameters must be the last parameters designated in the list. The following is invalid:

```
DECLARE SUB KerPlunk CDECL ([x%,] y%, z%)
```

Because the SUB or FUNCTION being called does not know how many parameters are being passed at the time it is called, you should pass the number of parameters as one of the required parameters in the list.

Classic PowerBASIC continues to support the use of classic optional parameter syntax using brackets ([...]) but this will not be the case in future versions of Classic PowerBASIC. Existing code should be changed to the new OPTIONAL syntax as soon as possible to ensure compatibility with future versions of Classic PowerBASIC.

AS type

You may specify the type of data returned by a Function to the calling code. If you do not specify a type, Classic PowerBASIC assumes that the Function returns the data type specified by a [DEFtype](#) statement.

However, if no DEFtype or AS type has been specified, a [compile-time error](#) is generated.

Therefore, there are two ways to specify the return type of a Function:

- Include a type-specifier character at the end of *ProcName*
- Include the AS type clause as the last part of the DECLARE

statement (this is the recommended syntax to ensure compatibility with future versions of Classic PowerBASIC).

For example, the following statements are equivalent:

```
DECLARE FUNCTION aFunction?()  
DECLARE FUNCTION aFunction() AS BYTE
```

While most FUNCTION calling conventions are fairly well defined throughout the industry, there are a few exceptions. In the case of functions which return a Quad Integer value, some programming languages (including Classic PowerBASIC) return the quad value in the FPU, while others return it in EDX:EAX. Classic PowerBASIC automatically detects the method used by imported functions and adjusts accordingly for you, but that's not a feature found in other compilers. Therefore, we recommend that you do not EXPORT QUAD FUNCTIONS unless they will only be accessed by Classic PowerBASIC programs. A simple equivalent functionality would be to return the quad-integer value to the caller in a BYREF QUAD parameter.

Restrictions A Sub/Function may be imported and exported within the same module. That is, a function in the module may be stated as EXPORT, while a DECLARE in the same module specifies it as an imported function by the option LIB "filename.dll", as long as FILENAME.DLL is the name of the module. This may be particularly valuable when you wish to build an [#INCLUDE](#) file with all of the DECLARE statements for a project.

See also [CALL](#), [CALL DWORD](#), [FUNCTION/END FUNCTION](#), [ISMISSING](#), [SUB/END SUB](#)

Example

```
' Main program  
DECLARE SUB Calculate LIB "A.DLL" (EXT, CUR, QUAD, INTEGER)  
  
CALL Calculate(w###, x@, y&&, z%)
```

Purpose	Decrement a variable by 1; Decrement a pointer by the size of its target; or decrement the target of a numeric pointer by 1.
Syntax	<code>DECR <i>variable</i></code>
Remarks	<p><i>variable</i> can be a numeric variable or a pointer variable. When DECR is used with a numeric variable, 1 is subtracted from the numeric variable. If DECR is used on the target of a numeric pointer variable (i.e., DECR @IntPtr), the target numeric variable is decremented by one. However, when using DECR on a pointer variable, the value of the pointer variable is decremented by the size of its target.</p> <p>For example, given a pointer to element 1000 of an Integer array, DECR of the pointer variable itself would result in a decrement of 2, which should point to the previous element in the array (element 999). This is because an Integer is two bytes wide, so the pointer value is reduced by 2 bytes.</p>
See also	INCR , LET
Example	<pre>DIM x&, LongPtr AS LONG POINTER DECR x& DECR LongPtr DECR @LongPtr</pre>

DEFBYT, DEFCUR, DEFCUX, DEFDBL, DEFDWD, DEFEXT, DEFINT, DEFLNG, DEFQUD, DEFSNG, DEFSTR and DEFWRD statements

Purpose	Declare the default type for variable identifiers that begin with specified letters.
Syntax	<code>DEFtype letter_range [, letter_range] [, ...]</code>
Remarks	<i>type</i> represents one of Classic PowerBASIC's variable types: INT (Integer), LNG (Long-integer), QUD (Quad-integer), SNG (Single-precision floating-point), DBL (Double-precision floating-point), EXT (Extended-precision floating-point), CUR (Currency), CUX (Extended-currency), STR (String), BYT (Byte), WRD (Word), and DWD (Double-word).
<i>letter_range</i>	is either a single alphabetic character (A through Z, case insignificant), or a range of letters (two letters separated by a hyphen, for example, A-M).
DEFtype	<p>Tells the compiler that variables and user-defined functions, whose names begin with the specified letter or range of letters, are of the specified type. Normally, when the compiler finds a variable name without a type specifier, a compile-time error is generated. If however, there was a preceding DEFtype statement such as DEFINT A-Z, the variable would default to that type (in this case, an Integer variable).</p> <p>You may use multiple DEFtype statements. If there is overlap between two DEFtype statements, no error is generated; but the definition of the latter DEFtype statement overrides the former where the two overlap.</p> <p>The DEFtype statement may not be supported in future editions of Classic PowerBASIC, so we recommend explicit variable declarations, using DIM, INSTANCE, LOCAL, STATIC, THREADED, or GLOBAL.</p>
Restrictions	Deftype only applies to implicitly-defined variables. It has no effect on variables that are defined explicitly. If a #DIM ALL statement exists in the application then Deftype statements will have no effect, #DIM ALL requires all variables to be defined explicitly

Example	<pre>DEFINT A-E, G, Q, Y-Z DEFCUX B, F, H-P, R-X FUNCTION PBMAIN A = 1 ' A is Integer B = 2 ' B is Extended-currency.</pre>
----------------	---

```
[statements]  
END FUNCTION
```

See Also

[#DIM ALL](#), [DIM](#), [INSTANCE](#), [LOCAL](#), [STATIC](#), [THREADED](#), [GLOBAL](#)

DESKTOP GET CLIENT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the size of the client area of the desktop, in pixels.
Syntax	<code>DESKTOP GET CLIENT TO <i>ncWidth</i>&, <i>ncHeight</i>&</code>
Remarks	<p>The desktop client size is the part of the screen that is not obscured by the system tray.</p> <p>This can be used in combination with DESKTOP GET LOC or DESKTOP GET SIZE for exact positioning of windows on the desktop.</p>
See also	Dynamic Dialog Tools , DESKTOP GET LOC , DESKTOP GET SIZE

Purpose	Retrieve the location of the top, left corner of the client area of the desktop, in pixels.
Syntax	<code>DESKTOP GET LOC TO x&, y&</code>
Remarks	<p>The desktop client area is the part of the screen that is not obscured by the system tray. The system tray's position on the screen determines the upper, left position of the client area. If the tray is located at the bottom of the screen (default), left and top coordinates are 0,0. If the tray is located on the right side of the screen, left and top coordinates are 0,0. If the tray is located on the left side of the screen, left and top coordinates are TrayWidth,0. If the tray is located at the top of the screen, left and top coordinates are 0,TrayHeight.</p> <p>This can be used in combination with DESKTOP GET CLIENT or DESKTOP GET SIZE for exact positioning of windows on the desktop.</p>
See also	DESKTOP GET CLIENT , DESKTOP GET SIZE

DESKTOP GET SIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the size of the entire desktop, in pixels.
Syntax	<code>DESKTOP GET SIZE TO <i>ncWidth</i>&, <i>ncHeight</i>&</code>
Remarks	The desktop size includes the space taken up by the system tray and is same as the screen size. This can be used in combination with DESKTOP GET CLIENT or DESKTOP GET LOC for exact positioning of windows on the desktop.
See also	DESKTOP GET CLIENT , DESKTOP GET LOC

Purpose	Disable a dialog so that it no longer receives any mouse or keyboard messages.
Syntax	<code>DIALOG DISABLE hDlg</code>
Remarks	<p><i>hDlg</i> refers to the dialog you want to disable. A disabled dialog will not receive any messages when it is clicked with the mouse or selected with the keyboard. Disabling a dialog that is already disabled has no effect.</p> <p>If the dialog has a Callback Function, a %WM_DISABLE message is sent to the Callback Function before DIALOG DISABLE finishes.</p>
See also	Dynamic Dialog Tools , DIALOG ENABLE , DIALOG SHOW MODAL , DIALOG SHOW MODELESS , DIALOG SHOW STATE

Purpose	Process pending window or dialog messages for MODELESS dialogs. If there are no pending messages, DIALOG DOEVENTS pauses execution of the current thread for a length of time specified by the programmer.
Syntax	<code>DIALOG DOEVENTS [<i>sleep</i>&] [TO <i>count</i>&]</code>
Remarks	<p>DIALOG DOEVENTS is usually used to create a "message pump" for modeless dialog boxes.</p> <p>If a window message is pending, it is processed appropriately. If no messages are pending, execution of the current thread is paused for the time specified by the <i>sleep</i>& parameter. If <i>sleep</i>& is zero (0), the remainder of the current time slice is relinquished to other threads or processes. If <i>sleep</i>& is greater than zero, the current thread is paused for that number of milliseconds to allow other threads or processes to continue. If <i>sleep</i>& is not specified, it defaults to a value of one (1). During the sleep period, all time-slices for the current thread are given to other threads and processes. If there are no other threads of equal priority, execution continues immediately. The time-slice duration (also known as the Quantum) can vary from version to version of Windows, ranging from 20 mSec to 120 mSec. If the optional TO clause is included, the number of active dialogs is returned in the <i>count</i>& variable, once all of the pending messages have been processed.</p>
Restrictions	The DIALOG DOEVENTS loop must run for the duration of the modeless dialog(s), or they will not respond or be redrawn correctly.
See also	Dynamic Dialog Tools , DIALOG NEW , DIALOG SHOW MODELESS , SLEEP
Example	<pre>' Single modeless dialog message pump example. ' (Assume dialog already created with DIALOG NEW) DIALOG SHOW MODELESS hDlg CALL DlgCallback DO DIALOG DOEVENTS 0 TO Count& LOOP WHILE Count& ' Application code continues here... ' Multiple modeless dialog message pump example. ' In some applications, the number of modeless dialogs can vary at any given moment, ' we want to break the message loop when the 'main' dialog is closed. ' (Assume dialogs already created with DIALOG NEW) DIALOG SHOW MODELESS hMainDlg& CALL DlgCallback DIALOG SHOW MODELESS hChildDlg1& DIALOG SHOW MODELESS hChildDlg2& [statements] DO DIALOG DOEVENTS</pre>

```
DIALOG GET SIZE hMainDlg TO x&, x&  
LOOP WHILE x& ' When x& = 0, dialog has ended  
' Application code continues here...
```

DIALOG ENABLE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Enable a dialog so that it can receive messages when the user interacts with it via the mouse or keyboard.
Syntax	<code>DIALOG ENABLE <i>hDlg</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog you want to enable. An enabled dialog will receive messages when it is clicked with the mouse or selected with the keyboard. Enabling a dialog has no effect if the dialog is already enabled.</p> <p>If the dialog has a Callback Function, a %WM_ENABLE message is sent to the Callback Function before DIALOG ENABLE finishes.</p>
See also	Dynamic Dialog Tools , DIALOG DISABLE , DIALOG SHOW MODAL , DIALOG SHOW MODELESS , DIALOG SHOW STATE

Purpose	Close and destroy the specified dialog.
Syntax	<code>DIALOG END <i>hDlg</i> [, <i>lResult</i>&]</code>
Remarks	The dialog specified by the <i>hDlg</i> variable is destroyed. <i>lResult</i> & optionally specifies a value to return to the DIALOG SHOW MODAL or DIALOG SHOW MODELESS statement that activated the dialog initially.
Restrictions	DIALOG END cannot close or destroy a dialog in a separate thread. In this case, send or post a message to the dialog to signal it to close, and respond to the message in the callback for the specified dialog. For example:

```
' Trigger a DIALOG END in a separate thread
DIALOG SEND hDlg, %WM_SYSCOMMAND, %SC_CLOSE, 0
```

DIALOG END cannot be used during processing of the %WM_INITDIALOG message. If this effect is necessary, the solution is to post a user-defined message to the dialog and use DIALOG END at that point. For example:

```
CALLBACK FUNCTION MyDialogCallback
SELECT CASE CB.MSG
CASE %WM_INITDIALOG
    IF gMustEnd& THEN _ ' We have to stop!
        DIALOG POST CB.HNDL, %WM_USER+999&, 0, 0

CASE %WM_USER + 999&
    DIALOG END CB.HNDL
    FUNCTION = 1
END SELECT
END FUNCTION
```

See also [Dynamic Dialog Tools](#), [DIALOG NEW](#), [DIALOG SHOW MODELESS](#)

Purpose	Specify the default DDT font information.
Syntax	DIALOG FONT [DEFAULT] <i>fontname\$</i> [, <i>points&</i> , <i>style&</i> , <i>charset&</i>]
<i>fontname\$</i>	Name of the font.
<i>points&</i>	Size of the font, in points.
<i>style&</i>	Font style attribute. 0 Normal 2 Italic
<i>charset&</i>	CharSet identifier. 0 ANSI CharSet 1 Default CharSet 2 Symbol CharSet 77 Mac CharSet 128 Shiftjis CharSet 129 Hangeul CharSet 130 Johab CharSet 136 Chinese CharSet 161 Greek CharSet 162 Turkish CharSet 177 Hebrew CharSet 178 Arabic CharSet 186 Baltic CharSet 204 Russian CharSet 222 Thai CharSet 238 East Europe CharSet 255 OEM CharSet
Remarks	<p>The DIALOG FONT statement specifies the font which will be used for all subsequent dialogs created with DIALOG NEW, until another DIALOG FONT statement is executed. When a DIALOG NEW statement is executed, the selected default font is associated with it, and its child controls, for the lifetime of the dialog.</p> <p>We recommend that the optional descriptor word DEFAULT be included, to aid in documenting your source code. The word DEFAULT may become mandatory in future versions of Classic PowerBASIC.</p> <p>If the requested font is not available on the computer, Windows will search for a substitute font, which is as similar as possible.</p>

You may use the value zero (0) for any of the numeric parameters to designate that the compiler should use the default for that item. If parameter(s) are missing, the compiler substitutes the default value for all remaining parameters.

If no DIALOG FONT DEFAULT statement is executed, the default font is MS Sans Serif, 8 point, with no style attributes.

When specifying a font, care should be exercised to use a standard font that is available in all versions of Windows, such as "Arial", "Courier", "Times New Roman", "MS Sans Serif", etc. Specifying a font name that is not available forces Windows to substitute a font that may not be visually appealing, and may also alter the relative size of the dialog.

DIALOG FONT is module-specific. That is, a DIALOG FONT statement only affects subsequent dialogs created by code in the same EXE or [DLL](#) as the DIALOG FONT statement appears. For example, a DIALOG FONT statement in a DLL, will not affect dialogs created in the calling EXE or other DLLs loaded by the EXE. The DIALOG FONT statement is thread-safe.

See also

[CONTROL SET FONT](#), [Dynamic Dialog Tools](#), [DIALOG NEW](#), [DIALOG SET COLOR](#), [FONT END](#), [FONT NEW](#), [GRAPHIC SET FONT](#), [XPRINT SET FONT](#)

DIALOG GET CLIENT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Return the client size of the specified dialog .
Syntax	<code>DIALOG GET CLIENT <i>hDlg</i> TO <i>wide&</i>, <i>high&</i></code>
Remarks	<i>hDlg</i> refers to the dialog to examine. The size of the dialog client area is placed in the <i>wide&</i> (width) and <i>high&</i> (height) variables . The size is specified in the same terms (pixels or dialog units) as the parent dialog.
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG PIXELS , DIALOG SET CLIENT , DIALOG SET LOC , DIALOG SET SIZE , DIALOG UNITS

DIALOG GET LOC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Return the location of the specified dialog .
Syntax	<code>DIALOG GET LOC <i>hDlg</i> TO <i>x&</i>, <i>y&</i></code>
Remarks	<p><i>hDlg</i> refers to the dialog to examine. The location of the dialog is placed in the <i>x&</i> (horizontal position) and <i>y&</i> (vertical position) variables as a relative location. If the dialog was created with the PIXELS option in the DIALOG NEW statement, the values are returned in pixel units. If the UNITS option was used (or no scaling option was specified), the values are returned in dialog units.</p> <p>If the parent of the dialog is zero (or %HWND_DESKTOP), the location is relative to the upper-left corner of the display. Otherwise, it is relative to the upper-left corner of client area of the parent window.</p>
See also	Dynamic Dialog Tools , DIALOG GET CLIENT , DIALOG GET SIZE , DIALOG PIXELS , DIALOG SET LOC , DIALOG SET SIZE , DIALOG UNITS

DIALOG GET SIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Return the size of the specified dialog .
Syntax	<code>DIALOG GET SIZE <i>hDlg</i> TO <i>x&</i>, <i>y&</i></code>
Remarks	<i>hDlg</i> refers to the dialog to examine. The total size of the dialog is placed in the <i>x&</i> (width) and <i>y&</i> (height) variables . If the dialog was created with the PIXELS option in the DIALOG NEW statement, the values are returned in pixel units. If the UNITS option was used (or no scaling option was specified), the values are returned in dialog units.
See also	Dynamic Dialog Tools , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG PIXELS , DIALOG SET LOC , DIALOG SET SIZE , DIALOG UNITS

DIALOG GET TEXT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the text in a dialog or window caption.
Syntax	<code>DIALOG GET TEXT <i>hDlg</i> TO <i>titletext\$</i></code>
Remarks	The text of the dialog or window caption specified by <i>hDlg</i> . For DDT dialogs, <i>hDlg</i> is the dialog handle returned by the DIALOG NEW statement. In a dialog Callback Function, the CB.HNDL function will return the parent dialog handle and this can also be used with DIALOG GET TEXT.
titletext\$	The text is returned and placed into the string variable <code>titletext\$</code> . If the window or dialog is invalid, <code>titletext\$</code> will be set to an empty string.
Restrictions	<i>hDlg</i> is a dialog or window handle, so DIALOG GET TEXT works with both DDT dialogs and conventional windows and dialogs.
See also	CB Callback functions , CONTROL GET TEXT , CONTROL SET TEXT , DIALOG NEW , DIALOG SET TEXT
Example	<code>DIALOG GET TEXT hDlg1& TO a\$</code>
Result	Variable <i>a\$</i> contains the caption text of the dialog referenced by <i>hDlg</i>

Purpose	Retrieve a value from the user data area of a DDT dialog.
Syntax	<code>DIALOG GET USER <i>hDlg</i>, <i>index</i>& TO <i>retvar</i>&</code>
Remarks	<p>Each DDT dialog has a user data area consisting of eight Long-integer values which may be used at the programmer's discretion to save relevant data. DIALOG GET USER allows one of the values to be retrieved, based upon the index parameter value (1 through 8).</p> <p><i>hDlg</i> refers to the dialog that contains the user data.</p> <p><i>index</i>& is the index number of the user data value to retrieve, in the range 1 to 8 inclusive.</p> <p><i>retvar</i>& receives the Long-integer data value stored in the nominated user data index.</p>
Restrictions	Data in the user data area is lost when the dialog is destroyed. The data area is completely separate from the %GWL_USERDATA area maintained by Windows.
See also	Dynamic Dialog Tools , COMBOBOX SET USER , CONTROL GET USER , CONTROL SET USER , DIALOG SET USER , LISTBOX GET USER , LISTBOX SET USER , LISTVIEW GET USER , LISTVIEW SET USER , TREEVIEW GET USER , TREEVIEW SET USER

Purpose	Create a new dialog in memory, ready for display.
Syntax	<code>DIALOG NEW [PIXELS, UNITS,] <i>hParent</i>, <i>title\$</i>, [<i>x&</i>], [<i>y&</i>], <i>xx&</i>, <i>yy&</i> [, [<i>style&</i>] [, [<i>exstyle&</i>]] [,] TO <i>hDlg</i></code>
Remarks	<p>A new empty dialog is created, but not yet displayed. Once the dialog has been created and all of the desired controls have been added with CONTROL ADD statements, the dialog can be displayed with the DIALOG SHOW MODELESS, or DIALOG SHOW MODAL statements.</p> <p>If a modeless dialog is created, the application must create a DIALOG DOEVENTS message pump for the duration of the dialog. Failure to provide a message pump can result in disruptions to the display of the dialog, or the inability of the dialog to respond to messages such as button clicks, etc. Modal dialogs do not require a message pump.</p> <p>To change the displayed state of a dialog (i.e., hidden, minimized, etc) after the dialog has been created, use the DIALOG SHOW STATE statement.</p> <p>If a dialog does not have either %WS_CHILD or %WS_POPUP styles, Windows may enforce a minimum dialog width of some 60-70 dialog units.</p>
PIXELS	If the PIXELS keyword is specified, all size and position parameters are specified in pixels. In this case, related statements such as DIALOG GET LOC will also return values in Pixels.
UNITS	<p>If UNITS is specified (or no scaling option is specified), all size and position parameters are specified in Dialog Units. (default)</p> <p>DIALOG NEW takes the following parameters.</p>
<i>hParent</i>	Handle of the parent window or dialog. If there is no parent, use zero (0) or %HWND_DESKTOP. If the dialog is MODAL, the parent window/dialog will be disabled while this "child" dialog is running.
<i>title\$</i>	The text displayed in the title or caption bar of the dialog.
<i>x&</i> , <i>y&</i>	<p>Optional location of the top-left corner for the dialog, in dialog units. If neither <i>x&</i> and <i>y&</i> are specified, the dialog is centered on the screen.</p> <p>If %CW_USEDEFAULT (&H08000000) is specified, the default Windows position is used (cascading from the upper-left corner).</p>
<i>xx&</i> , <i>yy&</i>	<p>The width and height of the dialog, specified in dialog units.</p> <p>If the default dialog style (or any other dialog style that includes the %WS_CAPTION style) is used, the width and height parameters specify the client size only, and this does not include any caption and border</p>

dimensions.

If the style does **not** include %WS_CAPTION, the width and height specify the overall dialog size, including the caption and border, if any.

Note that %WS_CAPTION is a combination of the %WS_BORDER and %WS_DLGFRAME styles. The default dialog style includes %WS_BORDER and %WS_DLGFRAME styles, so it implicitly includes the %WS_CAPTION style.

style&

An optional bitmask describing how the dialog should be displayed. default style of &H084C000D4& is made up %DS_3DLOOK, %DS_SETFONT, %DS_MODALFRAME, %DS_NOFAILCREATE, %WS_BORDER, %WS_CLIPSIBLINGS, %WS_DLGFRAME and %WS_POPUP. used if parameter omitted from statement completely. For example:

```
DIALOG NEW 0, "Dialog Title",,, 100, 200,, TO hDlg
```

Custom style values replace the default values. That is, they are not additional to the default style values - your code must specify all necessary style parameters (with the exception of %DS_NOFAILCREATE, %DS_SETFONT and %DS_3DLOOK, which are automatically added into the *style&* parameter by Classic PowerBASIC).

This also applies to the extended styles parameter - if your code specifies a custom primary style, the default extended style will no longer be in effect either. In this case, an explicit extended style may also need to be added to the DIALOG NEW statement if an explicit primary style is specified.

The primary *style&* value can be a combination of any values below, combined together with the [OR](#) operator to form a bitmask:

%DS_3DLOOK Give the dialog box a non-bold font, and draw three-dimensional borders around controls in the dialog box. The %DS_3DLOOK style is not required by applications marked with [#OPTION VERSION4](#) or [#OPTION VERSION5](#); as Windows automatically applies the 3D appearance. DDT dialogs are always created with this style. (default)

%DS_3DLOOK

Give the dialog box a non-bold [font](#), and draw three-dimensional borders around controls in the dialog box. The %DS_3DLOOK style is not required by applications marked with [#OPTION VERSION4](#) or [#OPTION VERSION5](#); as Windows automatically applies the 3D appearance. [DDT](#) dialogs are always created with this style. (default)

%DS_ABSALIGN

Indicate that the coordinates of the dialog box are screen coordinates; otherwise,

Windows assumes they are client coordinates.

%DS_CENTER

Center the dialog box in the working area (the area not obscured by the task bar and system tray). This is the default if `x&` and `y&` are not specified.

%DS_CENTERMOUSE

Center the mouse cursor in the dialog box when the dialog is initially created.

%DS_CONTEXTHELP

Include a question mark in the title bar of the dialog box. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a control in the dialog box, the dialog callback receives a `%WM_HELP` message. This style cannot be used with the `%WS_MAXIMIZEBOX` and `%WS_MINIMIZEBOX` styles. Also see `%WS_EX_CONTEXTHELP`.

%DS_CONTROL

Create a dialog that works as a child control of another dialog, smoothing the keyboard focus interface across the two dialogs when the TAB key or control accelerators are used. Typically used for dialogs that form the "pages" for tab controls and property-sheets.

%DS_MODALFRAME

Create a dialog box with a modal dialog-box frame that can be combined with a title bar and System menu by specifying the `%WS_CAPTION` and `%WS_SYSMENU` styles. (default)

%DS_NOFAILCREATE

The dialog is created regardless of any errors that may occur during the creation phase. DDT dialogs are always created with this style. (default)

%DS_SETFONT

The font to be used by a DDT dialog and its controls can be predetermined with the [DIALOG FONT](#) statement. If the DIALOG FONT statement is not used, the default font (MS Sans Serif, 8 point) is used. The size of the dialog font proportionately

	affects the conversion of dialog units values into pixels, so an increase in default font size will automatically create a larger dialog, even through the dialog dimensions have remained constant. As child controls are added to a %DS_SETFONT dialog, they will be sent a %WM_SETFONT message to ensure they also make use of the specified dialog font. DDT dialogs are always created with this style. (default)
%DS_SETFOREGROUND	Bring the dialog box to the foreground. Internally, Windows calls the SetForegroundWindow API function for the dialog box.
%DS_SYSMODAL	Create a system-modal dialog box. This style causes the dialog box to have the %WS_EX_TOPMOST style, but otherwise has no effect on the dialog box or the behavior of other applications and windows when the dialog box is displayed.
%WS_BORDER	Create a dialog that has a thin-line border.
%WS_CAPTION	Create a dialog that has a title bar. Includes the %WS_BORDER and %WS_DLGFRAME styles. When this style is used, the <i>width&</i> and <i>height&</i> parameters specify the size of the client area of the dialog; otherwise, they specify the outer dimensions of the dialog. (default)
%WS_CHILD	Create a child dialog. Cannot be used with the %WS_POPUP style. Typically used with %DS_CONTROL for tab control and property sheet "pages".
%WS_CLIPCHILDREN	Exclude the area occupied by child controls when drawing occurs on the dialog background. FRAME and LINE controls on a dialog with this style usually use the extended style %WS_EX_TRANSPARENT so the background of those controls is drawn by

	the dialog before the controls are drawn. %WS_CLIPCHILDREN is commonly used to reduce redraw flicker when a %WS_THICKFRAME style dialog is being resized.
%WS_CLIPSIBLINGS	Child controls are clipped (not overdrawn) by one another when the dialog window is repainted. (default)
%WS_DISABLED	Create a dialog that is initially disabled. A disabled dialog cannot receive input from the user.
%WS_DLGFRAME	Create a window that has a border of the style typically used with dialog boxes. (default)
%WS_HSCROLL	Dialog contains a horizontal scroll bar.
%WS_ICONIC	Create a dialog that is initially minimized, the same as the %WS_MINIMIZE style.
%WS_MAXIMIZE	Create a dialog that is initially maximized.
%WS_MAXIMIZEBOX	Create a dialog that has a Maximize button. Use with the %WS_SYSMENU style.
%WS_MINIMIZE	Create a dialog that is initially minimized, the same as the %WS_ICONIC style.
%WS_MINIMIZEBOX	Create a dialog that has a Minimize button. Use with the %WS_SYSMENU style.
%WS_OVERLAPPED	Create an overlapped window. An overlapped window has a title bar (caption) and a border. Synonym of the obsolete style %WS_TILED.
%WS_OVERLAPPEDWINDOW	Combination style producing an overlapping dialog. Comprises %WS_CAPTION, %WS_SYSMENU, %WS_THICKFRAME, %WS_MINIMIZEBOX, %WS_MAXIMIZEBOX, and %WS_OVERLAPPED styles.
%WS_POPUP	Create a popup dialog. When used by itself, a flat dialog is created with no caption or borders. Combine with

	%DS_MODALFRAME to create a 3D border. A popup dialog can overlap another window or dialog. (default)
%WS_POPUPWINDOW	Create a popup dialog but with a border and system menu. Comprises %WS_BORDER, %WS_POPUP and %WS_SYSMENU. Combine %WS_POPUPWINDOW with %WS_CAPTION to make the Window menu visible.
%WS_SYSMENU	Create a dialog that has a System-menu box in its title bar. Must be used with the %WS_CAPTION style.
%WS_THICKFRAME	Create a dialog that has a sizing border. That is, the dialog will be resizable.
%WS_VSCROLL	Dialog contains a vertical scroll bar. Also see %WS_EX_LEFTSCROLLBAR.

exstyle&

An optional extended style bitmask describing how the dialog should be displayed. The default extended dialog style comprises %WS_EX_LEFT with %WS_EX_LTRREADING and %WS_EX_RIGHTSCROLLBAR. The default extended style is used only if there are no explicit primary or extended styles parameters in the DIALOG NEW statement.

An explicit extended style value can be a combination of any values below, combined together with the OR operator to form a bitmask:

%WS_EX_ACCEPTFILES	The dialog accepts drag+drop files. The dialog Callback Function receives a %WM_DROPFILES message when files have been dropped onto the dialog.
%WS_EX_APPWINDOW	Force a top-level dialog onto the application taskbar when the window is minimized.
%WS_EX_CLIENTEDGE	Dialog has a border with a sunken edge.
%WS_EX_CONTEXTHELP	Include a question mark in the title bar of the dialog. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the

	user then clicks a child window, the child receives a %WM_HELP message. Also see %DS_CONTEXTHELP.
%WS_EX_CONTROLPARENT	The user may navigate among the child dialogs of the window by using the TAB key. See %DS_CONTROL.
%WS_EX_LEFT	Dialog has generic "left-aligned" properties. (default)
%WS_EX_LEFTSCROLLBAR	If present, the vertical scroll bar is positioned to the left of the client area. Also see %WS_VSCROLL.
%WS_EX_LTRREADING	Display the dialog text using Left to Right reading-order properties. (default)
%WS_EX_NOPARENTNOTIFY	Suppress %WM_PARENTNOTIFY messages when dialog is created or destroyed.
%WS_EX_OVERLAPPEDWINDOW	Comprised of the %WS_EX_CLIENTEDGE and %WS_EX_WINDOWEDGE styles.
%WS_EX_PALETTEWINDOW	Comprised of the %WS_EX_WINDOWEDGE, %WS_EX_TOOLWINDOW and %WS_EX_TOPMOST styles.
%WS_EX_RIGHT	Dialog has generic "right-aligned" properties that depend on the window class. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment. Otherwise, the style is ignored.
%WS_EX_RIGHTSCROLLBAR	If present, the vertical scroll bar is positioned to the right of the client area. See %WS_VSCROLL. (default)
%WS_EX_RTLREADING	If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the dialog text is displayed using Right to Left reading-order properties. For

`%WS_EX_STATICEDGE`

other languages, the style is ignored.
Dialog has a 3D border. Primarily used for dialogs that do not require user-input.

`%WS_EX_TOOLWINDOW`

Create a tool window (a window intended to be used as a floating toolbar). A tool window has a shorter than normal caption area and the dialog caption is drawn using a smaller font. A tool window does not appear in the task bar, or in the window that appears when the user presses ALT+TAB. The hybrid versions of Windows (95/98/ME) may require this extended style to be added after creation, using the `SetWindowLong` API function.

`%WS_EX_TOPMOST`

Place dialog above all non-topmost windows and keep it above them, even while the dialog is deactivated.

`%WS_EX_TRANSPARENT`

Controls/windows beneath the dialog are drawn before the dialog is drawn. The dialog is deemed transparent because elements behind the dialog have already been painted - the dialog itself is not drawn differently. True transparency is achieved by using Regions - see [MSDN](#) for more information.

`%WS_EX_WINDOWEDGE`

Dialog has a border with a raised edge.

hDlg

[Long-integer](#) Variable where the Windows window handle of the dialog is stored after it has been created and assigned by Windows. This handle should be used with subsequent `DIALOG` and `CONTROL` statements, and may be directly used with Windows API calls.

If the dialog could not be created (i.e., due to low Windows resources), zero is returned.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD](#), [DIALOG DOEVENTS](#), [DIALOG END](#), [DIALOG SET COLOR](#), [DIALOG SHOW MODAL](#), [DIALOG SHOW MODELESS](#), [DIALOG SHOW STATE](#)

DIALOG PIXELS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Convert pixels (device units) into dialog units.
Syntax	<code>DIALOG PIXELS <i>hDlg</i>, <i>x</i>&, <i>y</i>& TO UNITS <i>xx</i>&, <i>yy</i>&</code>
Remarks	The pixel values specified in the <i>x</i> & and <i>y</i> & variables are converted into dialog units, based on the default font of the dialog specified by <i>hDlg</i> . The resulting value in dialog units is stored in the <i>xx</i> & and <i>yy</i> & variables.
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG SET LOC , DIALOG SET SIZE , DIALOG UNITS

Purpose	Place a message in the message queue to be processed at the leisure of the target dialog.
Syntax	<code>DIALOG POST <i>hDlg</i>, <i>Msg</i>&, <i>wParam</i>&, <i>lParam</i>&</code>
Remarks	<p>DIALOG POST places the message in the message queue and returns immediately. The message is processed by the dialog at a later time, when it reads the message from the queue.</p> <p>This behavior is quite different to the DIALOG SEND statement, which forces the control to process the message immediately before returning. Since DIALOG POST is an asynchronous operation, it is not possible to retrieve a return code from the message.</p> <p><i>hDlg</i> refers to the target dialog.</p> <p><i>Msg</i>& is the message you want to post to the dialog.</p> <p><i>wParam</i>& is the first message parameter. <i>lParam</i>& is the second message parameter. The values of <i>wParam</i>& and <i>lParam</i>& are message-dependent. By Default, Classic PowerBASIC passes these parameters BYVAL. If the target dialog is expected to alter the values held by variables passed in the <i>wParam</i>& and <i>lParam</i>& parameters, pass them using VARPTR() or the changes will likely be discarded.</p> <p>Note that the address of the data must remain valid until after the dialog has processed the message and accessed the data. In this case, using STATIC or GLOBAL variables can be very important or a General Protection Fault (GPF) may occur (that is, if the variables have gone out of scope by the time the message is processed).</p> <p>An example of posting the addresses of variables to a dialog:</p> <pre>' Sel1& and Sel2& must be STATIC or GLOBAL DIALOG POST CB.HNDL, %WM_USER + 999&, VARPTR(Sel1&), VARPTR(Sel2&)</pre> <p>DIALOG POST returns immediately after the placing the message in the queue.</p> <p>To post a custom message to a dialog, use a message value in the range of (%WM_USER + 500) to (%WM_USER + &H07FFF), or use the RegisterWindowMessage API to obtain a unique message value from the operating system. Using messages with a numeric value of less then %WM_USER + 500 may conflict with Windows Common Control messages.</p>
See also	Dynamic Dialog Tools , CB Callback functions , CONTROL POST , CONTROL SEND , DIALOG SEND

Example

```
' Programmatically post a message to a dialog:  
DIALOG POST hDlg, %WM_CLOSE, 0, 0
```

DIALOG REDRAW statement

[Top](#) [Previous](#) [Next](#)

Purpose	Signal a designated dialog and all child controls to be redrawn immediately.
Syntax	<code>DIALOG REDRAW <i>hDlg</i></code>
Remarks	<p>DIALOG REDRAW invalidates the target dialog area, and signals a redraw/repaint to occur immediately, even if there are pending messages in the message queue.</p> <p><i>hDlg</i> refers to the dialog that is to be redrawn.</p>
Restrictions	It is not advisable to use DIALOG REDRAW or CONTROL REDRAW statements within the %WM_PAINT and associated message handling code, or an infinite redraw loop could occur.
See also	Dynamic Dialog Tools , CONTROL REDRAW , CONTROL SET COLOR , DIALOG SET COLOR
Example	<code>DIALOG REDRAW hDlg</code>

DIALOG SEND statement

[Top](#) [Previous](#) [Next](#)

Purpose	Send a message to a dialog, then wait until the message has been processed before continuing.
Syntax	<code>DIALOG SEND <i>hDlg</i>, <i>msg</i>&, <i>wParam</i>&, <i>lParam</i>& [TO <i>lResult</i>&]</code>
Remarks	<p><i>hDlg</i> identifies the dialog which should receive the message specified by <i>msg</i>&. <i>wParam</i>& is the first message parameter, and <i>lParam</i>& is the second message parameter.</p> <p>By default, Classic PowerBASIC passes these parameters BYVAL. If the target dialog is expected to return or alter the values passed in the <i>wParam</i>& and <i>lParam</i>& parameters, pass them using VARPTR() or the return values will be discarded. For example:</p> <pre>DIALOG SEND CB.HNDL, %WM_USER, VARPTR(Param1&), VARPTR(Param2&)</pre>
TO	The return value may be returned and stored in the variable <i>lResult</i> & after the message was processed by the dialog.
Restrictions	<p>If the target dialog was not created by the same thread, the DIALOG SEND statement becomes blocked until the thread processes the message. The <code>InSendMessage</code> API function will return TRUE (non-zero) if the callback code is currently processing a message from a separate thread.</p> <p>To send a custom message to a dialog, use a message value in the range of (<code>%WM_USER + 500</code>) to (<code>%WM_USER + &H07FFF</code>), or use the <code>RegisterWindowMessage</code> API to obtain a unique message value from the operating system.</p> <p>A dialog callback can send a message to its own dialog, but care should be taken not to create an infinite loop. Also, if DIALOG SEND sends a message that arrives back in the same callback as the message originated, care should be exercised to ensure that critical STATIC and GLOBAL variables are not unexpectedly altered by the second message processing code in the callback. This is known as re-entrant code design.</p>
See also	Dynamic Dialog Tools , CONTROL SEND

DIALOG SET CLIENT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the size of a dialog to a specific client area size.
Syntax	<code>DIALOG SET CLIENT <i>hDlg</i>, <i>x&</i>, <i>y&</i></code>
Remarks	<p><i>hDlg</i> refers to the handle of the dialog to change. <i>x&</i> and <i>y&</i> specify the new width and height of the dialog client area. <i>x&</i> and <i>y&</i> are specified in dialog units or pixels, depending upon the system used when the dialog was created.</p> <p>The dialog client size may be smaller than total size, depending on the type of borders used. The client area is the part below the dialog caption, and an eventual menu, where controls can be placed.</p>
See also	Dynamic Dialog Tools , DIALOG NEW , DIALOG PIXELS , DIALOG UNITS , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG SET LOC , DIALOG SET SIZE
Example	<pre>LOCAL hDlg, hMnu, hSubMenu AS DWORD, h, w AS LONG DIALOG NEW 0, "My Dialog",,, 400, 300, %WS_CAPTION OR %WS_SYSMENU, 0 TO hDlg ' Retrieve dialog client area DIALOG GET CLIENT hDlg TO w, h MENU NEW BAR TO hMnu MENU NEW POPUP TO hSubMenu MENU ADD POPUP, hMnu, "&File", hSubMenu, %MF_ENABLED MENU ADD STRING, hSubMenu, "E&xit", %IDCANCEL, %MF_ENABLED MENU ATTACH hMnu, hDlg ' Restore client area to desired size DIALOG SET CLIENT hDlg, w, h</pre>

Purpose	Set the background color of a dialog to a specific RGB color.
Syntax	<code>DIALOG SET COLOR <i>hDlg</i>, <i>foreclr&</i>, <i>backclr&</i></code>
Remarks	<p><i>hDlg</i> identifies the dialog to colorize.</p> <p>Color values <i>foreclr&</i> and <i>backclr&</i> must be in the range of &H0 to &H00FFFFFF, while the value -1& is used to specify the system default color. RGB can be a useful function to derive a 32-bit color value from discrete Red, Green and Blue values.</p> <p><i>foreclr&</i></p> <p>In the current implementation of Classic PowerBASIC, the dialog foreground color parameter <i>foreclr&</i> is not used, but the syntax is retained for future implementation. It is recommended that the foreground color parameter be set to -1&.</p> <p><i>backclr&</i></p> <p>In 16-bit or greater color-depth mode, the RGB color specified is used when the background of the dialog is drawn. If <i>backclr&</i> = -1&, the default dialog background color is used. If <i>backclr&</i> = -2&, the dialog background is not painted, allowing the content behind the dialog to become visible through the dialog.</p> <p>In 16-bit or greater color-depth mode, the specified RGB color is used when the background of the dialog is drawn. However, in 8-bit (256-color) mode, the color system works quite differently. Behind the scenes in Windows, the base system palette usually contains 20 solid colors that are not dithered when drawn on a dialog background. These solid-colors are ideal for background colors with DDT dialogs in 256-color mode.</p> <p>Conversely, when using a non-solid RGB color value, Windows will dither (approximate) the color to draw the dialog, using combinations of two or more colors. This usually produces an undesirable pattern effect.</p> <p>To avoid these problems when in 256-color mode, dialogs should either be colored with one of the 20 standard (solid) system colors, or the default color should be used instead. Classic PowerBASIC includes the following 10 built-in equates for help with the selection of a standard solid color:</p> <pre>%RGB_BLACK %RGB_BLUE %RGB_GREEN %RGB_CYAN %RGB_RED %RGB_MAGENTA %RGB_YELLOW %RGB_WHITE %RGB_GRAY %RGB_LIGHTGRAY</pre> <p>Many non standard colors are also built into the compiler, see the Built In RGB Color Equates topic for a complete list.</p> <p>If you prefer to disable color in 256-color mode, the number of colors can be easily tested with the following code:</p> <pre>' Determine number of colors LOCAL hDC AS DWORD, iColors AS LONG</pre>

```

hDC = GetWindowDC(GetDesktopWindow())
iColors = 2 & ^ (GetDeviceCaps(hDC, %BITSPIXEL) * GetDeviceCaps(hDC,
%PLANES))
ReleaseDC GetDesktopWindow(), hDC
IF iColors > 256 THEN
DIALOG SET COLOR hDlg, -1, RGB(0,100,192)

```

In 256-color mode on most computers, the values of the standard 20 system colors can be found by requesting the first and last 10 (0 to 9, and 246 to 255 inclusive) entries from the `GetSystemPaletteEntries` API function, as follows:

```

' Fill array with solid colors
DIM hDC AS DWORD, Cols AS LONG, x AS LONG

hDC = GetWindowDC(GetDesktopWindow)
Cols = GetDeviceCaps(hDC, %NUMRESERVED)
REDIM lp(1 TO Cols) AS LONG
x& = GetSystemPaletteEntries(hDC, 0, Cols \ 2, BYVAL VARPTR(lp(1)))
x& = GetSystemPaletteEntries(hDC, 256 - x&, Cols - x&, BYVAL
VARPTR(lp(x& + 1)))
ReleaseDC GetDesktopWindow, hDC
' Array lp() now contains the solid color table

```

For more information on working with palettes in 256-color mode, please consult WIN32.HLP or visit <http://msdn.microsoft.com>.

When dynamically changing colors of a dialog from within a callback (i.e., after the `DIALOG SHOW` statement), a `DIALOG SET COLOR` statement should be immediately followed by an explicit [DIALOG REDRAW](#) statement.

Without a forced dialog redraw, the dialog background color change may not become evident to the user until the dialog is **eventually** repainted in the normal course of user interaction. For example, a normal repaint may only occur if the dialog becomes obscured and then uncovered by another window. Ensuring a timely repaint of the dialog will guarantee the dialog maintains an up-to-date appearance at all times.

See also

[Built In RGB Color Equates](#), [Dynamic Dialog Tools](#), [CONTROL REDRAW](#), [DIALOG REDRAW](#), [CONTROL SET COLOR](#), [DIALOG SET ICON](#)

Example

```

DIALOG NEW 0, "Dialog",,, 160, 120, TO hDlg

' Set the color with an RGB value
DIALOG SET COLOR hDlg, -1, RGB(0,0,255)

' Or we could use the built-in %BLUE equate:
DIALOG SET COLOR hDlg, -1, %BLUE

```

Purpose	Change both the dialog icon in the caption, and the icon shown in the ALT+TAB task list.
Syntax	<code>DIALOG SET ICON <i>hDlg</i>, <i>newicon</i>\$</code>
Remarks	DIALOG SET ICON changes both the small icon (as used in the dialog caption bar), and the large icon (visible in the icon task list presented during ALT+TAB task switching).
<i>hDlg</i>	Handle of the dialog that is to have its icon changed.
<i>newicon</i> \$	A string expression which specifies the name of the icon in the resource file (.PBR). If the icon resource uses an integer identifier, <i>newicon</i> \$ should begin with a Number symbol (#), followed by the integer identifier in ASCII format. For example, "#998". Otherwise, the text identifier name should be used.
Restrictions	DIALOG SET ICON cannot use bitmap files. 32x32 pixel icons produce the most visually pleasing results.
See also	Dynamic Dialog Tools , CONTROL ADD IMAGE , CONTROL ADD IMAGEX , CONTROL ADD IMGBUTTON , CONTROL ADD IMGBUTTONX , CONTROL SET IMAGEX , CONTROL SET IMGBUTTON , CONTROL SET IMGBUTTONX , DIALOG SET TEXT

Purpose	Change the position of a dialog .
Syntax	<code>DIALOG SET LOC <i>hDlg</i>, <i>x&</i>, <i>y&</i></code>
Remarks	<p><i>hDlg</i> identifies the dialog to reposition. <i>x&</i> and <i>y&</i> specify the new coordinates of the upper-left corner of the target dialog. <i>x&</i> and <i>y&</i> are the horizontal and vertical coordinates respectively. If the dialog was created with the PIXELS option in the DIALOG NEW statement, the values are returned in pixel units. If the UNITS option was used (or no scaling option was specified), the values are returned in dialog units.</p> <p>If the dialog has a parent, the coordinates are relative to the upper-left corner of the parent dialog client area. Otherwise, the coordinates are relative to the upper-left corner of the desktop workspace.</p>
See also	Dynamic Dialog Tools , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG PIXELS , DIALOG SET SIZE , DIALOG UNITS

DIALOG SET SIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the size of a dialog .
Syntax	<code>DIALOG SET SIZE <i>hDlg</i>, <i>wide&</i>, <i>high&</i></code>
Remarks	<i>hDlg</i> identifies the dialog to resize. <i>wide&</i> and <i>high&</i> specify the new width and height, in dialog units, for the dialog. If the dialog was created with the PIXELS option in the DIALOG NEW statement, the values are set in pixel units. If the UNITS option was used (or no scaling option was specified), the values are set in dialog units.
See also	Dynamic Dialog Tools , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG PIXELS , DIALOG SET CLIENT , DIALOG SET LOC , DIALOG UNITS

Purpose	Set the text in a dialog or window caption.
Syntax	<code>DIALOG SET TEXT <i>hDlg</i>, <i>titletext\$</i></code>
Remarks	The caption of the dialog or window specified by <i>hDlg</i> is set with the DIALOG SET TEXT statement. For DDT dialogs, <i>hDlg</i> is the dialog handle returned by the DIALOG NEW statement. In a dialog Callback Function, the CB.HNDL function will return the parent dialog handle and this can also be used with DIALOG SET TEXT.
<i>titletext\$</i>	The caption text is specified in <i>titletext\$</i> . If the window or dialog is invalid, the operation is ignored.
Restrictions	<i>hDlg</i> is a dialog or window handle, so DIALOG SET TEXT works with both DDT dialogs and conventional windows and dialogs.
See also	Dynamic Dialog Tools , CB Callback functions , CONTROL GET TEXT , CONTROL SET TEXT , DIALOG GET TEXT , DIALOG NEW , DIALOG SET ICON
Example	<code>DIALOG SET TEXT hDlgMine, "This is my dialog!"</code>

Purpose	Set a value in the user data area of a DDT dialog.
Syntax	<code>DIALOG SET USER <i>hDlg</i>, <i>index</i>&, <i>usrval</i>&</code>
Remarks	<p>Each DDT dialog has a user data area consisting of eight Long-integer values which may be used at the programmer's discretion to save relevant data. DIALOG SET USER allows one of the values to be set, based upon the index parameter value (1 through 8).</p> <p><i>hDlg</i> refers to the dialog that owns the user data area.</p> <p><i>index</i>& is the index number of the user data value to set, in the range 1 to 8 inclusive.</p> <p><i>usrval</i>& is the Long-integer data value to store in the user data area.</p>
Restrictions	Data in the user data area is lost when the dialog is destroyed. The data area is completely separate from the %GWL_USERDATA area maintained by Windows.
See also	Dynamic Dialog Tools , COMBOBOX SET USER , CONTROL GET USER , CONTROL SET USER , DIALOG GET USER , LISTBOX GET USER , LISTBOX SET USER , LISTVIEW GET USER , LISTVIEW SET USER , TREEVIEW GET USER , TREEVIEW SET USER

DIALOG SHOW MODAL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Display and activate a dialog , allowing it to receive user input and messages. The DIALOG SHOW MODAL statement blocks (halts) until the dialog is destroyed with DIALOG END .
Syntax	<code>DIALOG SHOW MODAL <i>hDlg</i> [[,] CALL <i>callback</i>] [TO <i>lResult&</i>]</code>
Remarks	<p><i>hDlg</i> identifies a dialog created using DIALOG NEW. You can specify a Callback Function for all dialog messages using the CALL keyword, followed by the name of the Callback Function.</p> <p>When a modal dialog is displayed, the DIALOG SHOW MODAL statement is blocked until the dialog is destroyed with DIALOG END. During the duration of the dialog, the Callback Function code is executed in response to messages for the dialog.</p> <p>If a parent was specified in the DIALOG NEW statement, the parent window is disabled until the modal dialog is destroyed.</p>
<i>callback</i>	<p>If specified, dialog messages are routed to the nominated Callback Function.</p> <p>Just before a dialog is initially displayed, the dialog Callback Function is sent a %WM_INITDIALOG message. By processing this message within the dialog callback, an application can take the opportunity to load controls with data before the controls become visible to the user. For example, a list box control could be loaded with a list of items so that the control appears populated with data when initially displayed.</p> <p>The nominated callback function name must be a CALLBACK FUNCTION or a compile-time Error 547 ("Callback function required") will occur.</p>
<i>lResult&</i>	<p>When the modal dialog is destroyed using the DIALOG END statement, the resulting value is assigned to the <i>lResult&</i> variable, if specified.</p> <p><i>lResult&</i> is excluded from becoming a Register variable by the compiler, since this value can be assigned from outside of the function containing the DIALOG SHOW MODAL statement, and this may only be performed with a memory variable. However, if the target variable is explicitly declared as a register variable, Classic PowerBASIC raises a compile-time Error 491 ("Invalid register variable").</p>
See also	Dynamic Dialog Tools , DIALOG END , DIALOG NEW , DIALOG SHOW MODELESS

DIALOG SHOW MODELESS statement

[Top](#) [Previous](#)
[Next](#)

Purpose	Display and activate a dialog , allowing it to receive user input and messages. Execution of the code continues at the same time as the dialog is displayed. Modeless dialogs require a message pump to be running for the duration of the dialog.
Syntax	<code>DIALOG SHOW MODELESS <i>hDlg</i> [[,] CALL <i>callback</i>] [TO <i>lResult&</i>]</code>
Remarks	<p><i>hDlg</i> identifies a dialog created using DIALOG NEW. You can specify a Callback Function for all dialog messages, using the CALL keyword followed by the name of the Callback Function.</p> <p>Once a modeless dialog is displayed, the DIALOG SHOW MODELESS statement completes, and execution of the code continues. At the same time, the dialog can receive messages and process them via the Callback Function. A DIALOG SHOW MODELESS statement is usually followed by a message pump loop. For more information, please refer to the examples under DIALOG DOEVENTS.</p>
callback	<p>If specified, dialog messages are routed to the nominated Callback Function.</p> <p>The nominated callback function name must be a CALLBACK FUNCTION or a compile-time Error 547 ("Callback function required") will occur.</p>
lResult&	<p>When the modeless dialog is destroyed using the DIALOG END statement, the resulting value is assigned to the <i>lResult&</i> variable, if specified.</p> <p><i>lResult&</i> is excluded from becoming a Register variable by the compiler, since this value can be assigned from outside of the function containing the DIALOG SHOW MODELESS statement, and this may only be performed with a memory variable. However, if the target variable is explicitly declared as a register variable, Classic PowerBASIC raises a compile-time Error 491 ("Invalid register variable").</p>
See also	Dynamic Dialog Tools , DIALOG DOEVENTS , DIALOG END , DIALOG NEW , DIALOG SHOW MODAL

Purpose	Change the visible state of a dialog .																				
Syntax	<code>DIALOG SHOW STATE <i>hDlg</i>, <i>showstate&</i> [TO <i>lResult&</i>]</code>																				
Remarks	<p>DIALOG SHOW STATE changes the visible state of the dialog identified by <i>hDlg</i>.</p> <p><i>showstate&</i> can be one of the following values:</p> <table><tr><td>%SW_HIDE</td><td>Hide the dialog.</td></tr><tr><td>%SW_MAXIMIZE</td><td>Maximize the specified dialog.</td></tr><tr><td>%SW_MINIMIZE</td><td>Minimize the specified dialog.</td></tr><tr><td>%SW_RESTORE</td><td>Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized dialog.</td></tr><tr><td>%SW_SHOW</td><td>Activate the dialog and displays it in its current size and position.</td></tr><tr><td>%SW_SHOWMAXIMIZED</td><td>Synonym of %SW_MAXIMIZE.</td></tr><tr><td>%SW_SHOWMINIMIZED</td><td>Activate the dialog and minimize it.</td></tr><tr><td>%SW_SHOWNA</td><td>Display the dialog in its current state without activating it. The currently active window remains active.</td></tr><tr><td>%SW_SHOWNOACTIVATE</td><td>Display the dialog in its most recent size and position without activating it. The currently active window remains active.</td></tr><tr><td>%SW_SHOWNORMAL</td><td>Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position.</td></tr></table> <p>If the optional <i>lResult&</i> parameter is used, it will contain the previous visibility state. If <i>lResult&</i> is set to TRUE (non-zero), the dialog was visible. If the dialog was previously hidden, <i>lResult&</i> is set to FALSE (zero).</p>	%SW_HIDE	Hide the dialog.	%SW_MAXIMIZE	Maximize the specified dialog.	%SW_MINIMIZE	Minimize the specified dialog.	%SW_RESTORE	Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized dialog.	%SW_SHOW	Activate the dialog and displays it in its current size and position.	%SW_SHOWMAXIMIZED	Synonym of %SW_MAXIMIZE.	%SW_SHOWMINIMIZED	Activate the dialog and minimize it.	%SW_SHOWNA	Display the dialog in its current state without activating it. The currently active window remains active.	%SW_SHOWNOACTIVATE	Display the dialog in its most recent size and position without activating it. The currently active window remains active.	%SW_SHOWNORMAL	Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position.
%SW_HIDE	Hide the dialog.																				
%SW_MAXIMIZE	Maximize the specified dialog.																				
%SW_MINIMIZE	Minimize the specified dialog.																				
%SW_RESTORE	Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position. An application should specify this flag when restoring a minimized dialog.																				
%SW_SHOW	Activate the dialog and displays it in its current size and position.																				
%SW_SHOWMAXIMIZED	Synonym of %SW_MAXIMIZE.																				
%SW_SHOWMINIMIZED	Activate the dialog and minimize it.																				
%SW_SHOWNA	Display the dialog in its current state without activating it. The currently active window remains active.																				
%SW_SHOWNOACTIVATE	Display the dialog in its most recent size and position without activating it. The currently active window remains active.																				
%SW_SHOWNORMAL	Activate and display the dialog. If the dialog is minimized or maximized, Windows restores it to its original size and position.																				
Restrictions	In previous versions of Classic PowerBASIC, the DIALOG SHOW STATE was not permitted to be executed before a DIALOG SHOW MODAL or DIALOG SHOW MODELESS statement had been executed for that specific dialog. Starting with this version of Classic PowerBASIC, DIALOG SHOW STATE may be executed before or after the dialog is activated with DIALOG SHOW MODAL or DIALOG SHOW MODELESS statement.																				

When utilized prior to dialog activation, the attributes %SW_HIDE, %SW_MAXIMIZE, and %SW_MINIMIZE are remembered for use when activated. All other possible attributes are translated to the standard %SW_SHOW. Generally speaking, it is unwise to use %SW_HIDE with a modal dialog.

DIALOG SHOW STATE can be used to show a dialog before the message pump for a modeless dialog begins operating (i.e., after the DIALOG SHOW MODELESS statement, etc). However, until the message pump begins its operation, the dialog may not be drawn or displayed completely. For more information on message pumps, see [DIALOG DOEVENTS](#) and [DIALOG SHOW MODELESS](#).

See also

[Dynamic Dialog Tools](#), [CONTROL SHOW STATE](#), [DIALOG DOEVENTS](#), [DIALOG SHOW MODAL](#), [DIALOG SHOW MODELESS](#)

Purpose	Convert dialog units into pixels.
Syntax	<code>DIALOG UNITS <i>hDlg</i>, <i>x</i>&, <i>y</i>& TO PIXELS <i>xx</i>&, <i>yy</i>&</code>
Remarks	The dialog units specified in the <i>x</i> & and <i>y</i> & variables are converted into pixels, based on the default font of the dialog specified by <i>hDlg</i> . The resultant pixel values are stored in the <i>xx</i> & and <i>yy</i> & variables.
See also	Dynamic Dialog Tools , CONTROL GET CLIENT , DIALOG GET CLIENT , DIALOG GET LOC , DIALOG GET SIZE , DIALOG PIXELS , DIALOG SET LOC , DIALOG SET SIZE

Purpose	Declare and dimension arrays , scalar variables , and pointers .
Syntax	<p>Arrays:</p> <pre>DIM var[(subscripts)] [AS [GLOBAL INSTANCE LOCAL STATIC THREADED]type] [PTR POINTER] [AT address] [, ...] DIM var[(subscripts)] ' var may include a type-specifier</pre> <p>Scalar variables:</p> <pre>DIM var AS [GLOBAL INSTANCE LOCAL STATIC THREADED] type [PTR POINTER] [, ...] DIM var ' var must include a type-specifier</pre>
Remarks	<p>DIM declares var to be a variable or array whose type is specified by appending a type-specifier to the name or by using the AS type keyword. If the AS clause is used, the variable name cannot end with a type-specifier character.</p> <p>DIM can only be used inside a SUB, FUNCTION, METHOD, or PROPERTY. Outside of Subs, Functions, Methods, or Properties, use GLOBAL or INSTANCE to declare variables and arrays.</p> <p>DIM can also be used to dimension an "absolute array" - one that occupies a specific location in memory. This can be useful to dynamically "superimpose" one type of array directly over the top of an existing block of memory (which could be another type of array, or data structure). This would form a Union-like overlay structure. See below.</p> <p>In addition, it is possible to create an array of pointers with the DIM statement, and it is also possible to do so at a specific location in memory. This is termed an "<i>absolute pointer array</i>".</p>

Dimensioning arrays

[subscripts](#) may take one of the following forms for each array dimensioned:

(a) A comma-delimited list of one or more [Long-integer](#) expressions, each defining a dimension of the array. This form is used to declare arrays whose [subscript](#) (index) range starts at 0. For example, the following lines are equivalent ways of dimensioning the same array:

```
DIM lArray(20) AS LONG ' With an AS type clause
□ DIM lArray&(20)      ' With a type-specifier
```

Both lines above define a one dimension Long-integer array that has 21 elements, from lArray(0) to lArray(20) inclusive. The second line uses a type-specifier symbol to specify the data type, and this uses a simplified syntax (trailing clauses/keywords are not permitted). The simplified syntax is only valid for data types that have a type-specifier symbol (\$, !,

@, @@, #, ##, %, &, &&, ?, ??, ???), or the specifier can be omitted if there is a [DEFtype](#) statement in effect. The specifier must be omitted if [#DIM ALL](#) is in effect.

Declarations of [multiple-dimension arrays](#) take the following forms:

```
DIM sArray(20,40,2) AS STRING
```

□

```
DIM sArray$(20,40,2)
```

These two lines of code define a [dynamic string](#) array with three dimensions, 21 elements by 41 elements by 3 elements, totaling 2583 string elements. As before, the second line uses the simplified syntax form.

(b) A comma-delimited list where both the upper and lower subscript bounds are explicitly declared for each dimension of the array. For each dimension, the lower bound is listed first, followed by a colon (:) character or the TO keyword (preferred), followed by the upper bound. For example:

```
DIM MyArray(1:20) AS LONG      ' Classic syntax
```

□

```
DIM MyArray(1 TO 20) AS LONG  ' Preferred syntax
```

both define an array of one dimension that has 20 elements, from MyArray(1) to MyArray(20). The lower bound does not have to be zero or one; for example:

```
DIM SalesByYear(1980 TO 2000) AS INTEGER
```

□

```
DIM SalesByYear%(1980 TO 2000)
```

Use the TO keyword instead of the colon (:) syntax, as the colon syntax may not be supported in future versions of Classic PowerBASIC.

Each array can have up to 4,294,967,295 elements in the range of -2,147,483,648 to 2,147,483,647 (Long-integer range). It is recommended that an explicit [variable scope](#) clause ([GLOBAL/LOCAL/STATIC](#)) be added to each DIM statement that uses an explicit *type* clause. See Restrictions below.

Array Initialization and Absolute Arrays

Classic PowerBASIC generates an error message when it encounters an array that hasn't been dimensioned. If the array has already been dimensioned, the DIM statement is ignored. A new array is not created and a [run-time error](#) is not generated.

When a program is first executed, Classic PowerBASIC sets each element of a numeric array to zero, and sets each element of regular string arrays to a null string (length zero). However, when an absolute array is Dimensioned (at a specific location in memory using the AT *address* syntax), Classic PowerBASIC does not initialize the memory

occupied by the array. Further, when an absolute array is erased, the memory is not released either. This provides a powerful mechanism to create [Union](#)-like overlay structures in memory.

The most common use of an absolute array is when manipulating Visual Basic arrays directly from a DLL. This involves obtaining a pointer to the array, the element size, and the number of elements. With this information, an absolute array can be dimensioned in Classic PowerBASIC and the array memory manipulated directly. Another common use involves using a large dynamic or [fixed-length string](#) memory block, overlaid with an absolute numeric array.

Care must be exercised when using absolute arrays, since the contents of an absolute array can only be valid for the scope of the memory the array references. If an absolute array references memory that is LOCAL to the procedure, the array contents become invalidated if the target memory block is released. For example, by either explicitly deallocating the memory block, or exiting the procedure itself. Attempting to access absolute array memory that has been deallocated will likely trigger a General Protection Fault (GPF). On this basis, absolute arrays should be LOCAL to the procedure in which they are to be used.

While Classic PowerBASIC supports [LBOUND](#) values that are non-zero, Classic PowerBASIC generates the most efficient code if the LBOUND parameter is omitted (i.e., the array uses the default LBOUND of zero). You should also avoid specifying an explicit LBOUND of zero, since this imposes a small efficiency penalty with no meaningful benefits

Declaring scalar (non-array) variables

If you have specified #DIM ALL or [OPTION EXPLICIT](#), you have to declare all variables used in your programs. Classic PowerBASIC provides a variation of the DIM statement for this job, because of the reduced level of syntax required for scalar variables. The following is a simplified syntax for DIM that just applies to scalar variables:

```
DIM var AS [GLOBAL | INSTANCE | LOCAL | STATIC | THREADED] type [PTR  
| POINTER] [, ...]  
DIM var ' var must include a type-specifier
```

Here are some sample variable declarations:

```
DIM a AS LOCAL INTEGER  
DIM b AS STATIC WORD  
DIM c AS GLOBAL DOUBLE POINTER  
DIM d AS ASCIIZ * 255  
DIM e AS THREADED STRING  
DIM f AS INSTANCE SINGLE
```

AS type	Type
BYTE	Byte

WORD	Word
INTEGER	Integer
DWORD	Double-word
LONG	Long-integer
QUAD	Quad-integer
SINGLE	Single-precision floating-point
DOUBLE	Double-precision floating-point
EXT	Extended-precision floating-point
EXTENDED	Extended-precision floating-point
CUR	Currency
CURRENCY	Currency
CUX	Extended-currency
CURRENCYX	Extended-currency
STRING	Dynamic (variable-length) string
STRING * x	Fixed-length string
ASCIIZ * x	Nul-terminated string
ASCIZ * x	Nul-terminated string
Pointer	Pointer
Ptr	Pointer
VARIANT	Variant
IAUTOMATION	Automation Interface
IDISPATCH	Dispatch Interface
IUNKNOWN	Direct Interface
GUID	16-byte GUID string
FIELD	Field string

Restrictions

LOCAL ASCIIZ, LOCAL fixed-length strings, and LOCAL [UDTs](#) are created on the [stack](#) frame of the Sub/Function/Method/Property in which they are declared. You must therefore use caution so that the combined local variable size does not exceed the allocated stack size. Unless you declare otherwise, Classic PowerBASIC sets a default stack size of

1MB. If more stack space is required, you can allocate it with the [#STACK](#) metastatement. There are no such limitations with GLOBAL, [INSTANCE](#), [THREADED](#), or STATIC variables.

When a DIM statement is used (*without an explicit scope clause*), to declare a variable in a procedure, *and* an identical variable has already been declared as GLOBAL, the variable in the procedure will be given GLOBAL scope. For example:

```
GLOBAL xyz AS LONG
...
SUB MySub
  DIM xyz AS LONG
  ' Here, xyz is a GLOBAL variable
END SUB
```

To ensure that the variable scope is LOCAL to the Sub/Function/Method/Property, use a LOCAL statement rather than a DIM statement. Alternatively, add an explicit scope clause to the DIM statement. For example:

```
GLOBAL xyz AS LONG
[statements]
SUB MySub
  DIM xyz AS LOCAL LONG
  ' Here, xyz is a LOCAL variable
END SUB
```

Declaring pointer variables

A pointer *must* be declared before it can be used. You use the DIM statement to declare pointers, and describe the type of data to which they point. When a pointer is declared, it is automatically initialized to a value of zero. This is known as a null-pointer. You *must* remember to initialize it to a valid address, or you will get a General Protection Fault (GPF). The syntax for declaring pointer variables is similar to that of regular variables:

```
DIM var[(subscripts)] AS [GLOBAL | INSTANCE | LOCAL | STATIC |  
THREADED] type [PTR | POINTER] [, ...]
```

Here are some examples of pointer variable declarations:

```
DIM a          AS BYTE PTR
DIM b          AS INTEGER POINTER
DIM c          AS STRING PTR * 25
DIM d          AS MyType POINTER
DIM e(500)     AS INTEGER PTR
```

Pointers themselves are stored as DWORD values.

Options

The scope of a variable or array is set using the GLOBAL, INSTANCE, LOCAL, STATIC, or THREADED keywords.

Restrictions

When returning a pointer to a calling Sub, Function, Method, or Property, make sure the pointer target remains valid when the current routine terminates. For example, returning a pointer to a LOCAL variable is

certain to trigger a GPF, since local storage is released when the routine ends. In this case, the pointers target should be STATIC, GLOBAL, or INSTANCE, or be valid within the scope of the calling code.

IAUTOMATION, IDISPATCH, IUNKNOWN, VARIANT and GUID variables have special uses with [COM](#).

See also

[#DIM](#), [ARRAYATTR](#), [ERASE](#), [GLOBAL](#), [INSTANCE](#), [Just what is COM?](#), [LOCAL](#), [REDIM](#), [RESET](#), [STATIC](#), [THREADED](#), [Variables](#), [Variable Scope](#), [What is an object, anyway?](#)

Purpose

Return a filename and/or directory entry that matches a file name mask and an optional attribute.

Syntax

```
file$ = DIR$(mask$ [, [ONLY] attribute&, TO DirDataVar))  
file$ = DIR$([NEXT] [TO DirDataVar])
```

Remarks

There are two forms to the DIR\$() function. The first form, which includes a mask string and optional attribute, is used to find the first filename which matches. The second form, without those parameters, returns subsequent matching filenames. When the returned string is null (zero-length), there are no further matching filenames.

The second form may optionally specify the key-word NEXT to aid in self-documentation of the source code.

mask\$ specifies a filename or path which can include a drive name and system wildcard characters (* and ?). If the numeric attribute parameter is zero (or not specified), DIR\$ returns only "Normal" files. If *mask\$* is a null (zero-length) string, the function call is equivalent to the second form of the function to find subsequent matching filenames. In that case, an optional attribute is ignored.

If an attribute& is specified, it must use a standard operating system numeric attribute code. This causes DIR\$ to include filenames with specific attributes in the search, in addition to normal files. "Normal" files are those which are not hidden or system files, nor are they a directory or a volume label.

Attribute	Description	Equate
0	Normal	%NORMAL
2	Hidden	%HIDDEN
4	System	%SYSTEM
8	Volume Label	%VLABEL
16	Directory	%SUBDIR

You can search for filenames with multiple attributes set by adding the attribute codes together. For example, to search for hidden and system files, you'd add those codes together (2 and 4) to get 6. All other attribute codes (except for volume label) are normally inclusive. For example, specifying both hidden and system results in DIR\$ returning all hidden files, system files, normal files, and files that are both hidden and system.

If the ONLY option is included, normal files are excluded from the file search. For example: DIR\$(mask\$, ONLY 16) just the directory entries which match *mask\$* are returned.

An attribute of 8 will return the volume label, if one exists. In this case, *mask\$* must reference the drive letter of the target drive, and additional path information is ignored. Additionally, you may specify a UNC name for a shared drive (subject to operating system restrictions), and retrieve the volume label, if one exists, and you have suitable access rights. You can also obtain the volume label for a 'hidden' share with NT/2000/XP by appending a trailing dollar symbol to the share name.

```
' Retrieve volume for share \\server\drive0
A$ = DIR$("\\server\drive0", 8)
```

```
' Retrieve volume for hidden share D: (\d$)
A$ = DIR$("\\server\d$", %VLABEL)
```

The DIR\$ function may optionally assign the complete directory entry to an appropriate UDT variable if you include the TO clause as a parameter. The complete directory entry contains 318 bytes of data, corresponding to the following TYPE definition. This definition (DIRDATA) is built into Classic PowerBASIC, and need not necessarily be included in your source code.

```
TYPE DirData
  FileAttributes      AS DWORD
  CreationTime        AS QUAD
  LastAccessTime      AS QUAD
  LastWriteTime       AS QUAD
  FileSizeHigh        AS DWORD
  FileSizeLow         AS DWORD
  Reserved0           AS DWORD
  Reserved1           AS DWORD
  FileName            AS ASCIIZ * 260
  ShortName           AS ASCIIZ * 14
END TYPE
```

You can declare a variable as DIRDATA for this purpose, or use any other user-defined type of at least 318 data bytes. The additional data may be used for any other purpose in your program.

Restrictions Classic PowerBASIC performs file matching with both the long (LFN) and short (SFN) filename versions of filenames. This means that DIR\$ will also return filenames that start with the specified extension (as per standard Windows operating system behavior).

For example, A\$ = DIR\$("*.htm") will match filenames such as "Index.htm", "Default.html", "Homepages.htm", "cgilib.htmlpages", etc. Similarly, A\$ = DIR\$("*.h??") and DIR\$("*.ht*") will match the same filenames.

DIR\$ is thread-safe, so DIR\$ operations in one thread do not interfere with DIR\$ operations in another thread.

However, you should be aware that changing the mask during a DIR\$ search loop (within the same thread) will affect the search loop operation.

That is, if a typical DIR\$ search loop block such as shown in the *Example code* below, changes the search mask mid-way through the loop, subsequent calls to DIR\$ will begin to return files matching the new mask. This is rarely a problem if the mask change takes place within the search loop code block, since the DIR\$("mask") change will be plainly visible in the loop block code. However, the mask can also be changed from within a [Sub/Function/Method/Property](#) that is called from within the search loop block (provided that the procedure is located within the same compiled module).

While such techniques are perfectly valid, the cause of the mask change may not be immediately obvious when viewing just the search loop code block. Classic PowerBASIC allows the mask to be changed in this manner, since it may be an intended behavior on the part of the programmer, but in doing so, it opens up the potential for subtle bugs to be introduced into your code if this technique is not anticipated.

See also

[CURDIR\\$](#), [DIR\\$ CLOSE](#), [DISPLAY BROWSE](#), [DISPLAY OPENFILE](#), [FILEATTR](#), [GETATTR](#), [ISFILE](#), [PATHNAME\\$](#), [PATHSCAN\\$](#), [SETATTR](#)

Example

The following code shows a typical method of retrieving filenames from a directory:

```
DIM Listing(1000) AS DirData
DIM x%, temp$
temp$ = DIR$("*.*", TO Listing(x%))
WHILE LEN(temp$) AND x% < 1000 ' max = 1000
    INCR x%
    temp$ = DIR$(NEXT, TO Listing(x%))
WEND
```

DIR\$ CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Force the release of the operating system FindNext handle.
Syntax	DIR\$ CLOSE
Remarks	<p>DIR\$ CLOSE will cause the operating system FindNext handle to be closed. Each time a new DIR\$() sequence is initiated within a thread, or DIR\$() returns an empty string, Classic PowerBASIC automatically closes the FindNext handle to avoid overuse of system resources.</p> <p>However, in unusual circumstances (such as a recursive directory scan with delete or rename), it may be necessary to close the FindNext handle sooner, through the use of this explicit statement, so that a directory can be removed or renamed.</p>
Restrictions	It is never necessary to execute DIR\$ CLOSE to simply avoid a "System Handle Leak".
See also	DIR\$

DISKFREE function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the amount of available space on a disk, in bytes.
Syntax	<code>bytes&& = DISKFREE(<i>drive</i>\$)</code>
Remarks	<i>drive</i> \$ specifies the drive letter or UNC share name (subject to operating system restrictions) of the disk to examine. If <i>drive</i> \$ is an empty string, information on the default drive is returned.
Restrictions	With Windows 95 versions before OSR2, and Windows NT versions before 4.0, DISKFREE may return a negative or inaccurate value for drives larger than 2 GB.
See also	DISKSIZE
Example	<code>DisplayText "Free bytes on C: " + FORMAT\$(DISKFREE("C"), "#,###")</code>

DISKSIZE function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the total amount of space on a disk, in bytes .
Syntax	<code>bytes&& = DISKSIZE(<i>drive</i>\$)</code>
Remarks	<i>drive</i> \$ specifies the drive letter or UNC share name (subject to operating system restrictions) of the disk to examine. If <i>drive</i> \$ is an empty string, information on the default drive is returned.
Restrictions	With Windows 95 versions before OSR2, and Windows NT versions before 4.0, DISKSIZE may return a negative or inaccurate value for drives larger than 2 GB.
See also	DISKFREE
Example	<code>DisplayText "Total bytes on C: " + FORMAT\$(DISKSIZE("C"), "#,###")</code>

Purpose	Display a folder selection dialog to return the user's choice.										
Syntax	<code>DISPLAY BROWSE [hParent], [xpos&], [ypos&], title\$, start\$, flags& TO folder\$</code>										
<i>hParent</i>	Handle of the parent window or dialog . If there is no parent, use zero (0) or %HWND_DESKTOP.										
<i>xpos&</i>	Horizontal position, in pixels, relative to the parent window. If omitted, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).										
<i>ypos&</i>	Vertical position, in pixels, relative to the parent window. If missing, PowerBASIC selects the position (offset from the parent, or centered if no parent).										
<i>title\$</i>	The caption to be displayed below the caption bar of the dialog box. If this parameter is a null string, the title "Open" is displayed.										
<i>start\$</i>	A string which specifies the starting path to be used as the initial default folder. This may be disabled by passing a nul, zero-length string ("").										
<i>flags&</i>	<p>The style attributes of the BROWSE Dialog. The following values may be used alone or combined, and are predefined in the Classic PowerBASIC compiler:</p> <table><tr><td>%BIF_BROWSEINCLUDEFILES (4.71)</td><td>The dialog box will display both files and folders.</td></tr><tr><td>%BIF_BROWSEINCLUDEURLS</td><td>The dialog box can display URL's if %BIF_USENEWUI and %BIF_BROWSEINCLUDEFILES are also set.</td></tr><tr><td>%BIF_DONTGOBELOWDOMAIN</td><td>Does not include network folders below the domain level in the treeview control.</td></tr><tr><td>%BIF_EDITBOX</td><td>Includes an edit control in the dialog box that allows the user to type the name of an item.</td></tr><tr><td>%BIF_NEWDIALOGSTYLE (5.0)</td><td>Provides the new user interface, a larger dialog box that can be resized. It also offers drag-and-drop capability within the dialog box, reordering, shortcut menus, new folders, delete, and other shortcut menu commands. This is the default style implemented by</td></tr></table>	%BIF_BROWSEINCLUDEFILES (4.71)	The dialog box will display both files and folders.	%BIF_BROWSEINCLUDEURLS	The dialog box can display URL's if %BIF_USENEWUI and %BIF_BROWSEINCLUDEFILES are also set.	%BIF_DONTGOBELOWDOMAIN	Does not include network folders below the domain level in the treeview control.	%BIF_EDITBOX	Includes an edit control in the dialog box that allows the user to type the name of an item.	%BIF_NEWDIALOGSTYLE (5.0)	Provides the new user interface, a larger dialog box that can be resized. It also offers drag-and-drop capability within the dialog box, reordering, shortcut menus, new folders, delete, and other shortcut menu commands. This is the default style implemented by
%BIF_BROWSEINCLUDEFILES (4.71)	The dialog box will display both files and folders.										
%BIF_BROWSEINCLUDEURLS	The dialog box can display URL's if %BIF_USENEWUI and %BIF_BROWSEINCLUDEFILES are also set.										
%BIF_DONTGOBELOWDOMAIN	Does not include network folders below the domain level in the treeview control.										
%BIF_EDITBOX	Includes an edit control in the dialog box that allows the user to type the name of an item.										
%BIF_NEWDIALOGSTYLE (5.0)	Provides the new user interface, a larger dialog box that can be resized. It also offers drag-and-drop capability within the dialog box, reordering, shortcut menus, new folders, delete, and other shortcut menu commands. This is the default style implemented by										

Classic PowerBASIC.

%BIF_NONEWFOLDERBUTTON (6.0)	Do not include the "New Folder" button in the dialog box.
%BIF_NOTRANSLATETARGETS (6.0)	When the selected item is a shortcut, return the PIDL of the shortcut itself rather than its target.
%BIF_RETURNFSANCESTORS	Only returns file system ancestors. With any other selection, the OK button is grayed.
%BIF_RETURNONLYFSDIRS	Only returns file system directories. With any other selection, the OK button is grayed.
%BIF_SHAREABLE (5.0)	The dialog box can display shareable resources on remote systems. It is intended for applications that want to expose remote shares on a local system. The %BIF_NEWDIALOGSTYLE flag must also be set.
%BIF_UAHINT (6.0)	When this flag is combined with %BIF_NEWDIALOGSTYLE, adds a usage hint to the dialog box in place of the edit box.
%BIF_USENEWUI (5.0)	Use the new user interface, plus an edit box.

folder\$

Contains the drive letter and path to the folder the user selected. If an error occurs or the user clicks the cancel button, this variable is set to a nul, zero-length string.

See also

[DISPLAY COLOR](#), [DISPLAY FONT](#), [DISPLAY OPENFILE](#), [DISPLAY SAVEFILE](#)

Purpose	Display a color selection dialog to return the user's choice.						
Syntax	<code>DISPLAY COLOR [hParent], [xpos&], [ypos&], firstcolor&, custcolors, flags& TO colorval&</code>						
hParent	Handle of the parent window or dialog. If there is no parent, use zero (0) or %HWND_DESKTOP.						
xpos&	Horizontal position, in pixels, relative to the parent window. If omitted, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).						
ypos&	Vertical position, in pixels, relative to the parent window. If missing, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).						
firstcolor&	Specifies the RGB color which is initially selected when the dialog box is created.						
custcolors	User-Defined Type variable which is used to initialize and return 16 custom colors on the dialog. The UDT must have 16 members, each of which is a long integer or dword . They may be scalar members, or a member array .						
flags&	<div>The style attributes of the COLOR Dialog. The following values may be used alone or combined, and are predefined in the Classic PowerBASIC compiler:</div> <table><tr><td>%CC_FULLOPEN</td><td>Causes the entire dialog box to appear when created, including the section which allows the user to create custom colors.</td></tr><tr><td>%CC_PREVENTFULLOPEN</td><td>Disables the "Define Custom Colors" button, preventing the creation of custom colors.</td></tr><tr><td>%CC_SHOWHELP</td><td>Causes the Help Button to be displayed. The <i>hParent</i> parameter must not be zero or %HWND_DESKTOP.</td></tr></table>	%CC_FULLOPEN	Causes the entire dialog box to appear when created, including the section which allows the user to create custom colors.	%CC_PREVENTFULLOPEN	Disables the "Define Custom Colors" button, preventing the creation of custom colors.	%CC_SHOWHELP	Causes the Help Button to be displayed. The <i>hParent</i> parameter must not be zero or %HWND_DESKTOP.
%CC_FULLOPEN	Causes the entire dialog box to appear when created, including the section which allows the user to create custom colors.						
%CC_PREVENTFULLOPEN	Disables the "Define Custom Colors" button, preventing the creation of custom colors.						
%CC_SHOWHELP	Causes the Help Button to be displayed. The <i>hParent</i> parameter must not be zero or %HWND_DESKTOP.						
colorval&	The RGB value of the selected color. If the user fails to make a color selection, or chooses CANCEL, the value -1 is assigned to the <i>colorval&</i> variable.						
Remarks	If you offer the user the ability to create custom colors, it is suggested you retain the <i>custcolors&</i> UDT variable without change. It may then be used again on a later invocation of DISPLAY COLOR with the user's custom colors intact.						
See also	Built In RGB Color Equates , DISPLAY BROWSE , DISPLAY FONT ,						

[DISPLAY OPENFILE](#), [DISPLAY SAVEFILE](#), [RGB](#)

Purpose	Display a font selection dialog to return user choices.	
Syntax	<code>DISPLAY FONT [hParent], [xpos&], [ypos&], defname\$, defpoints&, defstyle&, flags& _ TO fontname\$, points&, style& [,colorval&, charset&]</code>	
hParent	Handle of the parent window or dialog. If there is no parent, use zero (0) or %HWND_DESKTOP.	
xpos	Horizontal position, in pixels, relative to the parent window. If omitted, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).	
ypos	Vertical position, in pixels, relative to the parent window. If missing, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).	
defname\$	The name of the default, pre-selected font which will be initially highlighted when the font dialog is displayed. A default font may be disabled by passing a nul, zero-length string ("").	
defpoints&	The point size of the default, pre-selected font.	
defstyle&	The style attribute of the default, pre-selected font. See the specific definition of <i>style&</i> below.	
flags&	The style attributes of the FONT Dialog. The following values may be used alone or combined, and are predefined in the Classic PowerBASIC compiler:	
	%CF_BOTH	Causes the dialog box to list both screen and printer fonts.
	%CF_TTONLY	Specifies that Font selection dialog should only enumerate and allow the selection of TrueType fonts.
	%CF_EFFECTS	Specifies that Font selection dialog should enable strikeout, underline, and color effect choices.
	%CF_FIXEDPITCHONLY	Specifies that Font selection dialog should select only fixed-pitch fonts.
	%CF_FORCEFONTEXIST	Specifies that Font selection dialog should indicate an error condition if the user attempts to select a font or style that does not exist.
	%CF_NOSTYLESEL	Specifies that Font selection dialog should not make an initial style selection.

<code>%CF_NOSIZESEL</code>	Specifies that Font selection dialog should not make an initial size selection.
<code>%CF_NOSIMULATIONS</code>	Specifies that Font selection dialog should not allow graphics device interface (GDI) font simulations.
<code>%CF_NOVECTORFONTS</code>	Specifies that Font selection dialog should not allow vector font selections.
<code>%CF_PRINTERFONTS</code>	Causes the Font selection dialog box to list only the fonts supported by the printer.
<code>%CF_SCALABLEONLY</code>	Specifies that Font selection dialog should allow only the selection of scalable fonts. (Scalable fonts include vector fonts, scalable printer fonts, TrueType fonts, and fonts scaled by other technologies.)
<code>%CF_SCREENFONTS</code>	Causes the Font selection dialog box to list only the screen fonts supported by the system.
<code>%CF_WYSIWYG</code>	Specifies that the Font selection dialog should allow only the selection of fonts available on both the printer and the display. If this flag is specified, the <code>%CF_BOTH</code> and <code>%CF_SCALABLEONLY</code> flags should also be specified.

<i>fontname</i> \$	The name of the font selected by the user
<i>points</i> &	The point size of the font selected by the user.
<i>style</i> &	<p>The style attribute of the selected font. Any of the following values can be combined or used alone:</p> <ul style="list-style-type: none"> 0 Normal 1 Bold 2 Italic 4 Underline 8 Strikeout <p>For example, if a <i>style</i>& value of 3 is returned, it specifies that a combination of both bold and italic attributes was selected by the user.</p>
<i>colorval</i> &	The RGB value of the selected color.
<i>charset</i> &	The chosen character set - 0 if a standard U.S. charset.

See also

[CONTROL SET FONT](#), [DIALOG FONT](#), [DISPLAY BROWSE](#), [DISPLAY COLOR](#), [DISPLAY OPENFILE](#), [DISPLAY SAVEFILE](#), [FONT END](#), [FONT NEW](#), [GRAPHIC SET FONT](#), [XPRINT SET FONT](#)

Purpose	Display an OpenFile selection dialog to return user choices.
Syntax	<pre>DISPLAY OPENFILE [hParent], [xpos&], [ypos&], title\$, folder\$, filter\$, _start\$, defextn\$, flags& TO filevar\$ [,countvar&]</pre>
<i>hParent</i>	Handle of the parent window or dialog . If there is no parent, use zero (0) or %HWND_DESKTOP.
<i>xpos&</i>	Horizontal position, in pixels, relative to the parent window. If omitted, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).
<i>ypos&</i>	Vertical position, in pixels, relative to the parent window. If missing, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).
<i>title\$</i>	The title to be displayed in the title bar of the dialog box. If this parameter is a null string, the title "Open" is displayed.
<i>folder\$</i>	The name of the initial file directory to be displayed. If this parameter is a null string, the current directory is used. Future invocations remember and use the ending directory, rather than honoring a null string for the current directory.
<i>filter\$</i>	<p>A string expression containing pairs of null-terminated filter strings. The first string in each pair describes the filter, and the second the filter pattern. For example, if you wish to display BASIC source files, you might use an expression like:</p> <pre>"BASIC" + CHR\$(0) + "*.BAS" + CHR\$(0)</pre> <p>A simpler method using the unique characteristics of the CHR\$() function in Classic PowerBASIC to achieve the same result:</p> <pre>CHR\$("BASIC", 0, "*.BAS", 0)</pre> <p>Multiple filters can be designated for a single item by separating filter pattern strings with a semicolon:</p> <pre>CHR\$("BASIC", 0, "*.BAS;*.INC;*.BAK", 0)</pre>
<i>start\$</i>	A string which specifies the starting file name to be used as the initial file selection. This may be disabled by passing a null, zero-length string ("").
<i>defextn\$</i>	A default extension to be appended to the selected file name if the user does not enter it. This may be disabled by passing a null, zero-length string ("").
<i>flags&</i>	The style attributes of the OPENFILE Dialog. The following values may be used alone or combined, and are predefined in the Classic PowerBASIC

compiler:

%OFN_ALLOWMULTISELECT	Multiple selections are allowed. If the user chooses multiple items, the return value consists of multiple file names which are null-terminated.
%OFN_CREATEPROMPT	The user may specify a file which does not exist.
%OFN_ENABLESIZING	The dialog may be resized by the user, but future invocations remember and use the ending size and screen location, rather than honoring <i>xpos</i> and <i>ypos</i> parameter values. The position parameters are ignored.
%OFN_EXPLORER	The dialog uses the Explorer style interface. This is the default condition, even if the flag is not set.
%OFN_FILEMUSTEXIST	The user may not specify a file which does not exist.
%OFN_NODEREFERENCELINKS	The dialog returns the name of the selected shortcut (.LNK) file. If this value is not given, the name of the file referenced by the shortcut is returned.
%OFN_NONETWORKBUTTON	Hides and disables the network button.
%OFN_NOTESTFILECREATE	The file is not created before the dialog is closed.
%OFN_NOVALIDATE	The file name is not validated for invalid characters.
%OFN_PATHMUSTEXIST	The user may type only valid paths and filenames.
%OFN_SHAREAWARE	If the dialog fails because of a network sharing violation, the error is ignored and the selected filename is returned.
%OFN_SHOWHELP	The help button is displayed.

Return Values

filevar\$ If the user selects one file, this variable receives the drive, path, and name of that file. If the user selects no files, an error occurs, or cancel/close is chosen, this variable is set to a null, zero-length string.

If the user selects multiple files, and specified the flag %OFN_ALLOWMULTISELECT, the returned string consists of the path

name (which applies all selected files), followed by each of the file names of the selected files. Each of these text items are delimited in the returned string by a nul - [CHR\\$\(0\)](#). You can extract each of the multiple names with the [PARSE\\$\(\)](#) function or the [PARSE](#) statement.

Windows imposes a text limit of 32K (32,768 bytes) for the returned string value. If it is exceeded, a nul, zero-length string is returned.

countvar&

If this optional [long integer](#) variable is included, it receives a count of the number of file names which were selected by the user.

Remarks

The current default directory is never altered by this statement, even if the user changes the directory while searching for files.

See also

[DISPLAY BROWSE](#), [DISPLAY COLOR](#), [DISPLAY FONT](#), [DISPLAY SAVEFILE](#)

Purpose	Display a SaveFile selection dialog to return user choices.
Syntax	<pre>DISPLAY SAVEFILE [hParent], [xpos&], [ypos&], title\$, folder\$, filter\$, _start\$, defext\$, flags& TO filevar\$ [,countvar&]</pre>
<i>hParent</i>	Handle of the parent window or dialog . If there is no parent, use zero (0) or %HWND_DESKTOP.
<i>xpos&</i>	Horizontal position, in pixels, relative to the parent window. If omitted, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).
<i>ypos&</i>	Vertical position, in pixels, relative to the parent window. If missing, Classic PowerBASIC selects the position (offset from the parent, or centered if no parent).
<i>title\$</i>	The title to be displayed in the title bar of the dialog box. If this parameter is a null string, the title "Save As" is displayed.
<i>folder\$</i>	The name of the initial file directory to be displayed. If this parameter is a null string, the current directory is used.
<i>filter\$</i>	<p>A string expression containing pairs of null-terminated filter strings. The first string in each pair describes the filter, and the second the filter pattern. For example, if you wish to display BASIC source files, you might use an expression like:</p> <pre>"BASIC" + CHR\$(0) + "*.BAS" + CHR\$(0)</pre> <p>A simpler method using the unique characteristics of the CHR\$() function in Classic PowerBASIC to achieve the same result:</p> <pre>CHR\$("BASIC", 0, "*.BAS", 0)</pre> <p>Multiple filters can be designated for a single item by separating filter pattern strings with a semicolon:</p> <pre>CHR\$("BASIC", 0, "*.BAS;*.INC;*.BAK", 0)</pre>
<i>start\$</i>	A string which specifies the starting file name to be used as the initial file selection. This may be disabled by passing a nul, zero-length string ("").
<i>defext\$</i>	A default extension to be appended to the selected file name if the user does not enter it. This may be disabled by passing a nul, zero-length string ("").
<i>flags&</i>	<p>The style attributes of the SAVEFILE Dialog. The following values may be used alone or combined, and are predefined in the Classic PowerBASIC compiler:</p> <p>%OFN_ALLOWMULTISELECT Multiple selections are allowed. If the</p>

	user chooses multiple items, the return value consists of multiple file names which are null-terminated.
%OFN_CREATEPROMPT	The user may specify a file which does not exist.
%OFN_ENABLESIZING	The dialog may be resized by the user, but future invocations remember and use the ending size and screen location, rather than honoring <i>xpos</i> and <i>ypos</i> parameter values. The position parameters are ignored.
%OFN_EXPLORER	The dialog uses the Explorer style interface. This is the default condition, even if the flag is not set.
%OFN_FILEMUSTEXIST	The user may not specify a file which does not exist.
%OFN_NODEREFERENCELINKS	The dialog returns the name of the selected shortcut (.LNK) file. If this value is not given, the name of the file referenced by the shortcut is returned.
%OFN_NONETWORKBUTTON	Hides and disables the network button.
%OFN_NOTESTFILECREATE	The file is not created before the dialog is closed.
%OFN_NOVALIDATE	The file name is not validated for invalid characters.
%OFN_PATHMUSTEXIST	The user may type only valid paths and filenames.
%OFN_OVERWRITEPROMPT	The user may select a filename that already exists.
%OFN_SHAREAWARE	If the dialog fails because of a network sharing violation, the error is ignored and the selected filename is returned.
%OFN_SHOWHELP	The help button is displayed.

Return Values

filevar\$

If the user selects one file, this variable receives the drive, path, and name of that file. If the user selects no files, an error occurs, or cancel/close is chosen, this variable is set to a nul, zero-length string.

If the user selects multiple files, and specified the flag %OFN_ALLOWMULTISELECT, the returned string consists of the path name (which applies all selected files), followed by each of the file names

of the selected files. Each of these text items are delimited in the returned string by a nul - [CHR\\$\(0\)](#). You can extract each of the multiple names with the [PARSE\\$\(\)](#) function or the [PARSE](#) statement.

Windows imposes a text limit of 32K (32,768 bytes) for the returned string value. If it is exceeded, a nul, zero-length string is returned.

countvar&

If this optional [long integer](#) variable is included, it receives a count of the number of file names which were selected by the user.

Remarks

The current default directory is never altered by this statement, even if the user changes the directory while searching for files.

See also

[DISPLAY BROWSE](#), [DISPLAY COLOR](#), [DISPLAY FONT](#), [DISPLAY
OPENFILE](#)

Purpose LIBMAIN (or its synonym DLLMAIN) is an optional user-defined function called by Windows each time a [DLL](#) is loaded into, and unloaded from, memory. The [PBLIBMAIN](#) function performs a similar task to LIBMAIN, but takes no parameters.

Syntax

```
FUNCTION { LIBMAIN | DLLMAIN } ( _  
    BYVAL hInstance AS DWORD, _  
    BYVAL lReason AS LONG, _  
    BYVAL lReserved AS LONG ) AS LONG
```

In 32-bit Windows, LIBMAIN is called by Windows each time a DLL is loaded or unloaded by an application or process, and (usually) when a thread is started and stopped. Your code should never call LIBMAIN.

Remarks The LIBMAIN / DLLMAIN function provides the following parameters:

hInstance The unique instance handle of the DLL. This handle is used by the calling application to identify the DLL. The instance handle value is commonly used to load [resources](#) embedded within the DLL, and to obtain the actual file name of the DLL (via the GetModuleFilename API function). In these cases, it is common to copy the *hInstance* value to a [global](#) variable, allowing the instance handle value to be utilized elsewhere in the DLL.

lReason This flag indicates why the DLL entry-point is being called. It can be one of the following values (as defined in [WIN32API.INC](#)):

%DLL_PROCESS_ATTACH Indicates that the DLL is being loaded by a process (another DLL or EXE is loading the DLL). DLLs can use this opportunity to initialize any instance or global data, such as [arrays](#). *lReserved* is zero if the DLL is being loaded explicitly (run-time linking) using LoadLibrary(), or non-zero if the DLL is being loaded implicitly (load-time linking) during process initialization.

%DLL_PROCESS_DETACH Indicates that the DLL is being cleanly unloaded or detached from the calling application. DLLs can take this opportunity to clean up all resources for all threads attached and known to the DLL. This is functionally equivalent to the WEP function in 16-bit DLLs. *lReserved* is zero if LIBMAIN was executed via the FreeLibrary API and

	the DLLs reference count reached zero (no further instances of the DLL are loaded), or non-zero if LIBMAIN is executed during process termination. A %DLL_PROCESS_DETACH does not generate %DLL_THREAD_DETACH for active threads.
%DLL_THREAD_ATTACH	Indicates that the DLL is being loaded by a new thread in the calling application. DLLs can use this opportunity to initialize any Thread Local Storage (TLS). This execution occurs in the context of the new thread.
%DLL_THREAD_DETACH	Indicates that the thread is exiting cleanly. If the DLL has allocated any thread-specific storage (Thread Local Storage or TLS), it should be released. This may occur even if there was no matching %DLL_THREAD_ATTACH call. A %DLL_PROCESS_DETACH does not generate %DLL_THREAD_DETACH for active threads.
<i>Reserved</i>	The <i>Reserved</i> parameter specifies further aspects of the DLL initialization and cleanup. If <i>Reason</i> is %DLL_PROCESS_ATTACH, <i>Reserved</i> is zero (0) for explicit (dynamic) loads and non-zero for implicit loads. If <i>Reason</i> is %DLL_PROCESS_DETACH, <i>Reserved</i> is zero if LIBMAIN has been called by using the FreeLibrary API call, and non-zero if LIBMAIN has been called during process termination.
Return value	If LIBMAIN is called with %DLL_PROCESS_ATTACH, your LIBMAIN function should return a zero (0) if any part of your initialization process fails, or a one (1) if no errors were encountered. If a zero is returned, Windows will abort and unload the DLL from memory. When LIBMAIN is called with any other value than %DLL_PROCESS_ATTACH, the return value is ignored.
Restrictions	Note that Windows does not guarantee that LIBMAIN will be called in a "balanced" manner. For example, a %DLL_PROCESS_ATTACH is not followed by a %DLL_THREAD_ATTACH for the primary thread. In some conditions, %DLL_THREAD_DETACH may not occur at all. Further discussion on these Windows traits are beyond the scope of this documentation; however, an excellent source of information can be found in "Win32 Programming", Rector/Newcomer, ISBN 0-201-63492-9. At the point where a DLL is loaded into memory during process startup,

Windows only guarantees that the KERNEL32.DLL system library will be loaded in memory. On this basis, API calls made from within LIBMAIN must be restricted to the range of API functions present in KERNEL32.DLL, with the exception of the LoadLibrary, LoadLibraryEx, and FreeLibrary API functions.

In addition, code within LIBMAIN must not call API functions in any other DLL (for example, USER32.DLL, SHELL32.DLL, ADVAPU32.DLL, GDI32.DLL, etc), because some API functions in those DLLs may attempt to load other libraries via LoadLibrary, etc. For example, never call the MessageBox API function from within LIBMAIN, nor use the related [MSGBOX](#) function or [MSGBOX](#) statement.

Failure to observe these restrictions will result in Access Violation or General Protection Faults (GPFs), typically caused by the execution of code in DLLs that has yet to be initialized.

See also

[DLLMAIN](#), [PBLIBMAIN](#), [PBMAIN](#), [THREAD CREATE](#), [WINMAIN](#)

Example

```
#DIM ALL
#COMPILE DLL "LIBTEST.DLL"
#include "WIN32API.INC"

GLOBAL gNumOfTimes AS DWORD

FUNCTION LIBMAIN(BYVAL hInstance AS DWORD, _
    BYVAL lReason AS LONG, _
    BYVAL lReserved AS LONG) AS LONG

    INCR gNumOfTimes

    SELECT CASE AS LONG lReason

        CASE %DLL_PROCESS_ATTACH
            ' This DLL has been mapped into the memory context of
            ' the calling program, and can be initialized as required.
            ' Here we return a non-zero LIBMAIN result to indicate success.
            LIBMAIN = 1
            EXIT FUNCTION

        CASE %DLL_PROCESS_DETACH
            ' This DLL is about to be unloaded
            EXIT FUNCTION

        CASE %DLL_THREAD_ATTACH
            ' A [New] thread is starting (see THREADID)
            EXIT FUNCTION

        CASE %DLL_THREAD_DETACH
            ' This thread is closing (see THREADID)
            EXIT FUNCTION

    END SELECT

    ' Theoretically execution should never get to this point.
    ' However, if the DLL is being implicitly linked then return
    ' Zero (0) and the process (program) will fail to start
```

```
' running. For Explicit linking, returning Zero (0) will
' simply cause the LoadLibrary/LoadLibraryEx API call to fail.
LIBMAIN = 0 ' Indicate failure to initialize the DLL!
END FUNCTION

SUB TestIt ALIAS "TestIt" () EXPORT
    MSGBOX "TestIt" + $CRLF + _"gNumOfTimes =" + STR$(gNumOfTimes)
END SUB
```

Purpose	Define a group of program statements that are executed repetitively as long as a certain condition is met.
Syntax	<pre>DO [{WHILE UNTIL} <i>expression</i>] [<i>statements</i>] [EXIT LOOP] [<i>statements</i>] [ITERATE LOOP] [<i>statements</i>] LOOP [{WHILE UNTIL} <i>expression</i>]</pre>
Remarks	<p><i>expression</i> is a numeric expression, in which non-zero values represent logical TRUE, and zero values represent logical FALSE. If a string expression is used (i.e., A\$ <> ""), Classic PowerBASIC returns TRUE if the length of result of the string expression is greater than zero.</p> <p>DO/LOOP statements are extremely flexible. They can be used to create loops for almost any imaginable programming situation. They allow you to create loops with the test for the terminating condition at the top of the loop, the bottom of the loop, both places, or none of the above.</p> <p>A DO statement must always be paired with a matching LOOP statement at the bottom of the loop. Failure to match each DO with a LOOP results in either a compile-time Error 448 ("DO loop expected") or an Error 456 ("LOOP/WEND expected").</p> <p>The WHILE and UNTIL keywords are used to add tests to a DO/LOOP. Use the WHILE if the loop should be repeated if <i>expression</i> is TRUE, and terminated if <i>expression</i> is FALSE. UNTIL has the opposite effect; that is, the loop will be terminated if <i>expression</i> is TRUE, and repeated if FALSE.</p> <p>For example:</p> <pre>DO WHILE a = 13 [<i>statements</i>] LOOP</pre> <p>executes the statements between DO and LOOP as long as <i>a</i> is 13. If <i>a</i> is not 13 initially, the statements in the loop are never executed. Conversely:</p> <pre>DO UNTIL a = 13 [<i>statements</i>] LOOP</pre> <p>executes the statements between DO and LOOP as long as <i>a</i> is not 13. If <i>a</i> equals 13 initially, the loop is never executed.</p> <p>At any point in a DO/LOOP, you can include an EXIT LOOP or ITERATE LOOP statement. EXIT LOOP causes the loop to terminate, so that execution continues <i>after</i> the terminating loop statement. ITERATE LOOP causes the loop to continue <i>at</i> the terminating loop statement.</p> <p>The WHILE/WEND statements can be used in many cases to perform the</p>

same functions as DO/LOOP. For example, this DO/LOOP:

```
DO WHILE a < b
    [statements]
LOOP
```

has the same effect as this WHILE/WEND loop:

```
WHILE a < b
    [statements]
WEND
```

When using nested loops, be careful that inner loops do not modify variables that are used by the outer loop's terminating condition test. For example, the following code was intended to get all 20 elements of a 10x2 [array](#) (dimensioned `array(9,1)`):

```
Count1 = 0
DO WHILE Count1 < 10
    FOR Count2 = 0 TO 1
        x = array(Count1,Count2)
        Count1 = Count1 + 1
    NEXT Count2
LOOP
```

Because *Count1* is incremented within the inner loop, which executes twice for each pass through the outer loop, this code would not get all the array values, but would only get the values for `array(0,0)`, `array(1,1)`, `array(2,0)`, `array(3,1)` and so on. By moving the `Count1 = Count1 + 1` statement to just below the [NEXT](#) Count2 statement, the code functions as intended.

If an EXIT LOOP statement is used within nested loops, it exits only the current loop, not the entire nest. Similarly, an ITERATE within nested loops iterates the current loop. For advice on exiting nested block structures, please refer to the EXIT statement. The Classic PowerBASIC logical operators can be used to construct multiple test conditions for loop control. For example:

```
DO WHILE x < 10 AND y < 10
    [statements]
LOOP
```

is executed only as long as both x and y are less than 10. Similarly, the loop:

```
DO UNTIL x > 10 OR y > 10
    [statements]
LOOP
```

is executed until either x or y (or both) is (are) greater than 10. See the Data Types and [Arithmetic Operators](#) topics for more information about using logical operators.

Although the compiler doesn't care about such things, it is a good idea when writing your source code to indent the statements between DO and LOOP. The same is true of [FOR/NEXT](#) loops, WHILE/WEND loops, and multi-line [IF](#) statements. Such indenting makes the appearance of your

source code reflect the logical structure of your program, resulting in greater readability. Indenting is particularly valuable when nesting multiple loops of the same type, since it makes it easier to see which LOOP goes with which DO.

Also see the discussion on the IF statement for notes on Classic PowerBASIC's Short-circuit evaluation and its possible side effects.

See also

[#OPTIMIZE](#), [EXIT](#), [FOR/NEXT](#), [ITERATE](#), [WHILE/WEND](#)

Purpose	Modify the current program's environment table.
Syntax	<code>ENVIRON <i>envstring</i></code>
Remarks	Modify the environment table for the current program and any subsequent child programs that are launched. A single string expression parameter sets both the name of the environment variable and its value, delimited by an equal ("=") sign. If a value is not specified, the variable is removed from the environment table.
See also	ENVIRON\$
Example	<pre>ENVIRON "SETMODE=YES" ' SETMODE = "YES" ENVIRON "SETMODE=" ' Removes SETMODE</pre>

Purpose	Retrieve information from the current program's environment table.
Syntax	<code>s\$ = ENVIRON\$({<i>parameter_string</i> <i>n</i>})</code>
Remarks	<p><i>parameter_string</i> is a string expression denoting which environment parameter is to be retrieved. <i>n</i> is an integer class expression, starting at 1.</p> <p>If a string argument is used, ENVIRON\$ returns the text that follows <i>parameter_string</i> (after the equal sign) in the environment table. If <i>parameter_string</i> is not found, or no text follows the equal sign in the environment string table, an empty string is returned.</p> <p>If the numeric argument is used, it acts as an index into the environment table. ENVIRON\$ returns a string containing the <i>n</i>th parameter from the start of the table. If there is no <i>n</i>th parameter, an empty string is returned. If the index is negative, private Windows variables are returned.</p> <p>When launching a program from within the IDE, Classic PowerBASIC sets the "PBIDE" environment variable with the IDE name and version number. For example, "CCEDIT 5.00" or "PBEDIT 9.00". Similarly, when running in the debugger, the "PBDEBUG" environment string will return the IDE name and version.</p> <p>Programs can use these environment strings to detect their "mode" of operation, for example, to signal a program to save internal data to a disk file, and when to display helpful debugging information. DLLs created with PB/Win can also examine these environment strings and adapt behavior accordingly. This will be of particular interest to 3rd-party DLL programmers who create libraries and add-ons for other Classic PowerBASIC programmers.</p>
Restrictions	When a program (process) starts, it is given its own local environment table, which is typically a copy of the parent program's environment table. ENVIRON\$ works with this local table, not the parent's table.
See also	ENVIRON
Example	<pre>' Retrieve the PATH environment variable Path\$ = ENVIRON\$("PATH") IF LEN(ENVIRON\$("PBDEBUG")) THEN _ CALL DisplayMyDebugData() ' Enumerate all Environment strings RESET x& DO INCR x& a\$ = ENVIRON\$(x&) ' process a\$ here LOOP WHILE LEN(a\$)</pre>

Purpose	Return the end-of-file status of an opened file or TCP/UDP transmission.
Syntax	<code>y = EOF([#] <i>filenum</i>&)</code>
Remarks	<p>Use EOF to determine when the end of a file has been reached while reading its data. <i>filenum</i>& is the file number specified when the file was Opened. EOF returns -1 (TRUE) if the end of the specified file has been reached, or if an error occurs trying to check for the end of the file. Otherwise, EOF returns 0 (FALSE).</p> <p>If <i>filenum</i>& is not a valid, open file, a run-time Error 53 will occur ("File not found"). If <i>filenum</i>& is for a binary file, EOF returns TRUE only if the most recent file operation was a read operation, and that operation could not read the requested number of bytes.</p> <p>The EOF function may also be used with the COMM LINE and TCP LINE statements to detect that an incomplete line was received. Normally, these statements read data until a \$CRLF character pair is found, and in that case, EOF will return 0 (FALSE). However, even if no \$CRLF has been found, the statements will end when no additional data is available. In that case, they will return whatever data has already been accumulated, and set EOF to -1 (TRUE).</p> <p>In many cases, it would be prudent to test EOF after every COMM LINE and TCP LINE to verify that a full line has been received. In some cases, you may wish to execute the statement one or more additional times, combining the data, in order to obtain a full line of text.</p>
See also	COMM LINE , LOC , LOF , OPEN , TCP LINE
Example	<pre>' Open an ASCII text file and read it hFile = FREEFILE OPEN "TEXTFILE.TXT" FOR INPUT AS hFile WHILE ISFALSE EOF(hFile) LINE INPUT# hFile, x\$ WEND CLOSE hFile</pre>

EQV operator

- Purpose

The EQV operator works as both a logical and a bitwise [arithmetic operator](#).
- Syntax

`p EQV q`
- Remarks

Using EQV as a logical operator

EQV returns TRUE (non-zero) if *at least* one bit in one operand contains the same value as the identical bit position in the other operand. Further, EQV will return zero if and only if there are no matching bit values between the two operands. This can occur when one operand is equal to the bitwise [NOT](#) value of the other operand. For example:

```
IF x EQV y = 0 THEN statement
```

is equivalent to:

```
IF x = NOT y THEN statement
```

The EQV operator can be used for comparing signed and unsigned values of the same bit size, such as [Long-integer](#) and [Double-word](#). This use of EQV is similar to using the [BITS functions](#); however, care must be exercised to test the return value of EQV correctly, since EQV will return an unsigned value with all bits set *only* if the bit patterns of the two operands are an *exact match*.

The EQV truth table looks like this:

Truth table		
x	y	x EQV y
T	T	T
T	F	F
F	T	F
F	F	T

See also

[Arithmetic Operators](#), [AND](#), [IMP](#), [ISFALSE](#), [ISTRUE](#), [LET](#), [NOT](#), [OR](#), [XOR](#)

Example

```
IF (Var1& EQV Var2???) = BITS???(~1&) THEN ...
IF (Val1% EQV Var2???) = &H0FFFF?? THEN ...
IF ~1& EQV BITS???(~1&) = &H0FFFFFFFF THEN ...
IF ~1% EQV BITS??(~1%) = &H0FFFF THEN ...
```

Purpose Deallocate [array](#) memory and release it from memory.

Syntax `ERASE array[()] [, array[()]] ...`

Remarks Any memory assigned to the individual elements (if they are [dynamic strings](#), [Objects](#), [Variants](#), etc.) is also released. ERASE deallocates all the memory for [LOCAL](#), [STATIC](#), and [GLOBAL](#) arrays. After an array is erased, attempting to access the array may produce a General Protection Fault (GPF). Local arrays are implicitly erased upon exit from the [Sub/Function/Method/Property](#) that created them.

array The name of the array to deallocate. Parentheses are optional, but are recommended for clarity of the source code.

One method to check if an array has been dimensioned without triggering a GPF is to use the [LBOUND](#) and [UBOUND](#) functions, as follows:

```
IF UBOUND(array) - LBOUND(array) = -1 THEN
    ' array() is not allocated
END IF
```

ERASE can deallocate an array that was passed as a parameter to a procedure, but only if the array was passed by reference (BYREF). To clear the contents of an array back to its initialized state, use [REDIM](#) or [RESET](#).

Restrictions Absolute arrays (those created by [DIM...AT](#)) are handled differently by ERASE. An explicit ERASE will release the individual elements of an absolute array, if needed, but the full data block is left as-is because no assumptions can be made as to its origin. It is the programmer's responsibility to ensure that the memory block overlaid by the absolute array is handled correctly. In an implied ERASE (Sub/Function/Method/Property exit) of a LOCAL absolute array, the internal array descriptor is deactivated, but no changes of any kind are made to the individual data elements or the full block. RESET may be used to set arrays to zeroes or empty strings without releasing the data block.

See also [ARRAYATTR](#), [DIM](#), [REDIM](#), [RESET](#)

Example `ERASE Array1$(), MyArray%()`

ERL system variable

[Top](#) [Previous](#) [Next](#)

Purpose	Return the last line number encountered before the most recent error .
Syntax	<code>nline = ERL</code>
Remarks	Return the last (most recent) line number that was encountered before the most recent run-time error , within the current Sub , Function , Method , or Property . With ERL, line numbers are of the traditional-basic line numbering variety, not the physical source code line.
See also	ERL\$, ERR , ERRCLEAR , ERROR , ERROR\$, Error Overview , Error Trapping , ON ERROR
Example	<pre>10 ERRCLEAR 20 NAME "a nonexistent filename.txt" AS "abc.txt" 30 IF ERR THEN lErrLine = ERL ' lErrLine = 20</pre>

Purpose	Return the last label , line number , or procedure name executed prior to the most recent error .
Syntax	<i>position\$</i> = ERL\$
Remarks	Return a string containing the name of the last (most recent) label, line number, or procedure that was executed prior to the most recent run-time error , within the current Sub , Function , Method , or Property . In order to maintain high efficiency levels, the returned name is limited to the first 8 characters of the actual name.
See also	ERL , ERR , ERRCLEAR , ERROR , ERROR\$, Error Overview , Error Trapping , ON ERROR
Example	<pre>MyLabel: ERRCLEAR NAME "a nonexisting filename.txt" AS "abc.txt" IF ERR THEN Position\$ = ERL\$</pre>

ERR and ERRCLEAR system variables

[Top](#) [Previous](#)
[Next](#)

Purpose	Return the error code of the most recent Classic PowerBASIC run-time error.
Syntax	<pre>y = ERR ERR = ErrNum y = ERRCLEAR ERRCLEAR</pre>
Remarks	<p>ERR and ERRCLEAR return the error code of the most recent run-time error in the current Sub, Function, Method, or Property. This number can be tested after any critical operation, so that appropriate error-handling code can be executed.</p> <p>You can also assign a value to ERR. This is similar to executing an ERROR statement, except no error is generated. Instead, subsequent tests of ERR and ERRCLEAR reflect <i>ErrNum</i>. To explicitly raise an exception in a TRY/END TRY block, use the ERROR statement.</p> <p>ERRCLEAR returns the error code of the most recent run-time error. In addition, it resets Classic PowerBASIC's internal error code variable ERR to zero after you reference it. This ensures that the next time you test ERR or ERRCLEAR, you are guaranteed to get a zero, unless there has actually been an error since the last ERRCLEAR. ERR, on the other hand, does not reset the internal error code. It retains its value until it is changed, either by a run-time error or explicitly by your code (using the ERR= or ERROR =statements).</p> <p>ERRCLEAR can also be used as a statement to reset ERR to zero. ERRCLEAR, when used as a statement, is equivalent to ERR = 0.</p> <p>IMPORTANT: Be sure to study the Errors and Error Trapping.</p>
Restrictions	Valid run-time error values are in the range 1 through 255. Attempting to set an error value (with the ERROR statement) outside of that range will convert the value to a run-time Error 5 ("Illegal function call").
See also	ERROR , ERROR\$, Error Overview , Error Trapping , ON ERROR
Example	<pre>y = ERR ' sets y = ERR ERR = 6 ' sets ERR to 6 y = ERRCLEAR ' sets y = ERR and ERR = 0 ERRCLEAR ' sets ERR = 0</pre>

Purpose	Cause a run-time error to be generated and sets ERR to the specified error number.
Syntax	<code>ERROR <i>ErrNum</i></code>
Remarks	<p>ERROR <i>ErrNum</i> causes a run-time error to be generated and sets ERR to <i>ErrNum</i>. Run-time errors are only caught if you have an active ON ERROR GOTO in your code. ERROR is useful to explicitly raise an exception in a TRY/END TRY block.</p> <p>Valid errors are in the range 1 through 255. Attempting to set an error value outside of that range will convert the value to a run-time Error 5 ("Illegal function call").</p> <p>The compiler reserves codes 0 through 150, and 241 through 255 for run-time errors. You may freely use error codes 151 through 240 for your own purposes.</p>
See also	#DEBUG DISPLAY , ERL , ERR , ERRCLEAR , ERROR\$, Error Overview , Error Trapping , ON ERROR
Example	<pre>ERROR 5 ' generates error 5 "Illegal Function Call"</pre>

ERROR\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string containing the descriptive name of a specified Classic PowerBASIC run-time error code.
Syntax	<code>msg\$ = ERROR\$ [(<i>ErrNum</i>)]</code>
Remarks	<p>ERROR\$ returns the verbose text title of a Classic PowerBASIC run-time error identified by <i>ErrNum</i>.</p> <p><i>ErrNum</i> must be in the range 1 to 255 inclusive. Values outside of this range return "No error". If <i>ErrNum</i> is not specified, ERROR\$ returns the description of the current value of ERR.</p>
See also	#DEBUG DISPLAY , ERL , ERR , ERRCLEAR , ERROR , Error Overview , Error Trapping , ON ERROR
Example	<code>a\$ = ERROR\$(5) ' Returns "Illegal function call"</code>

Purpose Declare an [event](#) interface within a [Class](#) definition

Syntax `EVENT SOURCE InterfaceName`

Remarks With [objects](#), normally a client module calls a server module to perform specific operations as they are needed. However, in many situations, it's convenient and efficient for a server to notify its client of a condition or event immediately, without forcing the client to inquire about the status. At the appropriate time, the server calls back to a client method, passing information via the [method](#) parameters. This is the exact opposite of normal communication, because the server module is now calling the client module. In effect, the client is acting as a server for the purpose of handling these events. In the world of objects, a server which can call such "Event Methods" is said to offer a "[Connection Point](#)". A Connection Point can be used with [COM objects](#) or internal objects. Further, it may use either a [direct interface](#) or the [DISPATCH interface](#). Event methods may take parameters, but may not return a result.

Each server class created by Classic PowerBASIC may offer up to four event interfaces. A client module may subscribe to any or all of these event interfaces. When it's time for the server object to notify the client of an event, the [RAISEEVENT](#) statement is used. For the Dispatch interface, [OBJECT RAISEEVENT](#) is used instead. RAISEEVENT may only appear within a class which declares the Event Source interface.

The client must initiate a connection to the server with [EVENTS FROM](#) statement, and disconnect when done with [EVENTS END](#) statement.

A Connection Point may be attached to one Event Method, multiple Event Methods, or no Event Method at all. Whenever a RAISEEVENT statement or OBJECT RAISEEVENT statement is executed, all Event Methods attached to the source object are called, one after another. There is no guarantee of the sequence of the calls, and you must consider the possibility that RAISEEVENT with a ByRef parameter could change the value of a parameter variable before any particular Event Method is executed.

InterfaceName Specifies the "Event Source" Interface name. If *InterfaceName* is DISPATCH, you can reference it with the OBJECT RAISEEVENT statement -- otherwise, regular Method references are used.

See also [EVENTS](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [Just what is COM?](#), [METHOD](#), [OBJECT RAISEEVENT](#), [RAISEEVENT](#), [What are Connection Points?](#)

Example

```
#COMPILE EXE
#DIM ALL
CLASS EvClass AS EVENT
    INTERFACE IStatus AS EVENT
        INHERIT IUNKNOWN
        METHOD Done
            ? "Done!"
        END METHOD
    END INTERFACE
END CLASS

CLASS MyClass
    INTERFACE IMath
        INHERIT IUNKNOWN
        METHOD DoMath
            ? "Calculating..." ' Do some math calculations here
            RAISEEVENT IStatus.Done()
        END METHOD
    END INTERFACE
    EVENT SOURCE IStatus
END CLASS

FUNCTION PBMAIN()
    LOCAL oMath AS IMath
    LOCAL oStatus AS IStatus
    oMath = CLASS "MyClass"
    oStatus = CLASS "EvClass"
    EVENTS FROM oMath CALL oStatus
    oMath.DoMath
    EVENTS END oStatus
END FUNCTION

' Dispatch Interface Example
#COMPILE EXE
#DIM ALL
CLASS EvClass AS EVENT
    INTERFACE IStatus AS EVENT
        INHERIT IDISPATCH
        METHOD Done
            ? "Done!"
        END METHOD
    END INTERFACE
END CLASS

CLASS MyClass
    INTERFACE IMath
        INHERIT IDISPATCH
        METHOD DoMath
            ? "Calculating..." ' Do some math calculations here
            OBJECT RAISEEVENT IStatus.Done()
        END METHOD
    END INTERFACE
    EVENT SOURCE DISPATCH
END CLASS

FUNCTION PBMAIN()
    LOCAL oMath AS IMath
    LOCAL oStatus AS DISPATCH

    oMath = CLASS "MyClass"
    oStatus = CLASS "EvClass"
```

EVENTS FROM oMath CALL oStatus

oMath.DoMath

EVENTS END oStatus

END FUNCTION

Purpose Attach or detach an [event handler](#) to/from an event source.

Syntax

```
DIM oSource AS InterfaceName
DIM oEvent AS EventInterface
LET oSource = NEWCOM CLSID $ClassId
LET oEvent = CLASS "EventClass"
EVENTS FROM oSource CALL oEvent
[statements]
EVENTS END oEvent
```

Remarks

In the above source code sample, *oEvent* is an [object](#) variable which references an event handler object, and *oSource* is an object variable which references an [event source](#) object which generates events.

The EVENTS FROM statement attaches event handler code to an event source object variable. The object variable *oEvent* must be declared as a supported event interface, while *EventClass* specifies the class which implements the event handler code. The object variable *oSource* specifies the event source. EVENTS END detaches the event handler from the event source.

Generally speaking, a server object "sources" events, and a client object "handles" events by supplying a [METHOD](#) which is called by the server to perform a user-defined notification. This event handler is code in the client object, which is sometimes referred to as an "event sink" (analogous to the electrical engineering terms source/sink).

One or more clients may choose to "subscribe" to events from a server object by executing the EVENTS FROM statement. The subscription is terminated by execution of the EVENTS END statement. When the server executes [RAISEEVENT](#) or [OBJECT RAISEEVENT](#), all clients which have unsubscribed to these events are called. Classic PowerBASIC servers support up to 32 concurrent client subscribers per server object.

Event sources and event handlers may be used within a single module, or through [COM](#) services supplied by the Windows operating system.

See also [CLASS](#), [EVENT SOURCE](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [Just what is COM?](#), [OBJECT RAISEEVENT](#), [RAISEEVENT](#), [What is an object, anyway?](#), [What are Connection Points?](#)

Example

```
#COMPILE EXE

CLASS EvClass AS EVENT
    INTERFACE EvStatus AS EVENT
        INHERIT IUNKNOWN

        METHOD Done
            MSGBOX "Done!"
        END METHOD
```



```

    END INTERFACE
END CLASS

CLASS MyClass
    INTERFACE MyMath
        INHERIT IUNKNOWN

        METHOD DoMath
            MSGBOX "Calculating..."    ' Do some math calculations here
            RAISEEVENT EvStatus.Done()
        END METHOD
    END INTERFACE

    EVENT SOURCE EvStatus
END CLASS

FUNCTION PBMAIN()
    DIM oMath    AS MyMath
    DIM oStatus AS EvStatus

    LET oMath    = CLASS "MyClass"
    LET oStatus = CLASS "EvClass"

    EVENTS FROM oMath CALL oStatus

    oMath.DoMath

    EVENTS END oStatus
END FUNCTION

```

EXE read-only user defined type **New!**

[Top](#) [Previous](#) [Next](#)

Purpose Return the path and/or name of the executing program.

Syntax

```
f$ = EXE.Extn$  
f$ = EXE.Full$  
f$ = EXE.Name$  
f$ = EXE.Namex$  
f$ = EXE.Path$
```

Remarks You can use EXE to retrieve the complete path and file name of the executing program, or just a selected part of it. If the reference is physically located within a DLL, the name of the executable program which loaded it is returned.

EXE.Extn\$ This returns the extension (with a leading period) of the program which is currently executing.

EXE.Full\$ This returns the complete drive, path, file name, and extension of the program which is currently executing.

EXE.Name\$ This returns just the file name of the program which is currently executing.

EXE.Namex\$ This returns the file name and the extension of the program which is currently executing.

EXE.Path\$ This returns the complete drive and path of the program which is currently executing.

See also [COMMAND\\$](#), [PATHNAME\\$](#), [PATHSCAN\\$](#)

Purpose

Syntax

Remarks

Transfer program execution out of a block structure.

```
EXIT [FOR | FUNCTION | IF | DO | LOOP | MACRO | METHOD | PROPERTY |
SELECT | SUB | TRY ]
EXIT [, EXIT] [, ...] [, ITERATE]
```

The EXIT statement lets you leave a structure prematurely. Using EXIT by itself will leave the most recently executed FOR/Loop, Do/Loop, or WHILE-WEND/Loop. For example:

```
FOR ix = 1 TO 10
  DO UNTIL x > 10
    [statements]
    EXIT ' will exit from the DO LOOP
  LOOP
NEXT
```

<i>EXIT option</i>	Structure exited
FOR	FOR/NEXT loop
FUNCTION	FUNCTION/END FUNCTION function
IF	IF/END IF block (see <i>notes below</i>)
DO LOOP or	DO/LOOP or WHILE/WEND loop
MACRO	MACRO/END MACRO block
METHOD	METHOD block
PROPERTY	PROPERTY block
SELECT	SELECT statement
SUB	SUB/END SUB procedure
TRY	TRY/END TRY block

If you want to exit a structure other than the one most recently executed, you must include that type of structure as part of the EXIT statement, as follows:

```
FOR ix = 1 TO 10
  DO UNTIL x > 10
    [statements]
    EXIT FOR ' will exit DO and FOR NEXT loop
  LOOP
NEXT
```

Using EXIT is preferred over using [GOTO](#).

You can exit nested loops by using multiple EXIT statements without the loop type, and separated by commas:

```
DO
DO
    EXIT, EXIT ' exit both loops
LOOP
LOOP
```

You can also exit one loop and immediately iterate another:

```
FOR x = 1 TO 10
DO
    EXIT, ITERATE
LOOP
' never gets here
NEXT
```

If you EXIT the [PBMAIN](#) or [WINMAIN](#) function, your program will stop running.

Care should be exercised when constructing exits within a nested IF/THEN block. For example:

```
IF x& THEN
    IF y& THEN
        [statements]
        EXIT IF
        [statements]
    END IF
    [statements]
END IF
```

In the above nested block, the EXIT IF statement exits only the *inner* IF/THEN block (in which the EXIT statement is located), resulting in execution continuing at the {statements3} portion. That is, EXIT IF does not exit the overall IF/THEN structure.

Whilst EXIT can provide a convenient mechanism to escape a structure, subtle logic problems can arise when converting a nested single-line IF statement into an IF-block structure. For example:

```
IF x& = 1 THEN
    IF y& = 82 THEN EXIT IF
    y& = 12
END IF
```

is *not* logically equivalent to:

```
IF x& = 1 THEN
    IF y& = 82 THEN
        EXIT IF
    END IF
    y& = 12
END IF
```

Restrictions Before using EXIT to leave a [Function](#), [Method](#) with a return value, or a [Property Get](#), you should assign the desired return result before EXIT is executed. By default, the value returned by the function will be zero or an

empty string as determined by the Function return type. For example:

```
FUNCTION x() AS LONG
    [statements]
    IF condition THEN
        FUNCTION = -1&
        EXIT FUNCTION ' return -1
    END IF
    [statements]
END FUNCTION
```

See also

[DO/LOOP](#), [FOR/NEXT](#), [FUNCTION/END FUNCTION](#), [IF/END IF block](#), [ITERATE](#), [MACRO/END MACRO](#), [METHOD](#), [PROPERTY](#), [SELECT](#), [SUB/END SUB](#), [TRY/END TRY](#), [WHILE/WEND](#)

EXP, EXP2 and EXP10 functions

[Top](#) [Previous](#) [Next](#)

Purpose	Return a base number raised to a power. The base is <i>e</i> for EXP, 2 for EXP2, and 10 for EXP10.
Syntax	$y = \text{EXP}(n)$ $y = \text{EXP2}(n)$ $y = \text{EXP10}(n)$
Remarks	<p>EXP returns <i>e</i> to the <i>n</i>th power, where <i>n</i> is a numeric variable or expression and <i>e</i> is the base for natural logarithms, approximately 2.718282. Among other uses, this provides a simple way to obtain the value of <i>e</i> itself:</p> $e = \text{EXP}(1)$ <p>EXP2(<i>n</i>) returns 2 to the <i>n</i>th power, where <i>n</i> is a numeric variable or expression.</p> <p>EXP10(<i>n</i>) returns 10 to the <i>n</i>th power, where <i>n</i> is a numeric variable or expression.</p> <p>The EXP functions provide a convenient alternative to the ^ operator, which works with any base. The EXP functions return results in Extended-precision.</p>
See also	LOG , LOG2 , LOG10 , SQR , Arithmetic Operators

EXTRACT\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose

Return the portion of a string leading up to the first occurrence of a specified character or string.

Syntax

```
x$ = EXTRACT$([start,] MainStr, [ANY] MatchStr)
```

Remarks

start is the optional starting position to begin extracting. If *start* is not specified, it will start at position 1. *MainStr* is the [string expression](#) from which to extract. *MatchStr* is the string expression to extract up to. EXTRACT\$ is case-sensitive.

EXTRACT\$ returns a sub-string of *MainStr*, starting with its first character (or the character specified by *start*) and up to (but not including) the first occurrence of *MatchStr*. If *MatchStr* is not present in *MainStr*, all of *MainStr* is returned.

If the ANY keyword is included, *MatchStr* specifies a list of single characters to be searched for individually, a match on any one of which will cause the extract operation to be performed up to that character.

EXTRACT\$ is especially useful when parsing a string containing arguments to a program, or when manipulating nul-terminated or delimited strings received from a routine written in another language.

The complementary function to EXTRACT\$ is [REMAINS\\$](#), which returns the part of the string that EXTRACT\$ leaves behind. A similar function to EXTRACT\$ is [PARSE\\$](#), which extracts delimited substrings from a string.

See also

[INSTR](#), [JOIN\\$](#), [LEFT\\$](#), [LTRIM\\$](#), [MID\\$](#), [PARSE](#), [PARSE\\$](#), [PARSECOUNT](#), [REMAINS\\$](#), [REMOVES\\$](#), [RETAIN\\$](#), [RIGHT\\$](#), [RTRIM\\$](#), [TALLY](#), [TRIM\\$](#), [VERIFY](#)

Example

```
' x$ = first command-line argument, assuming spaces,  
' commas, periods, and tabs are valid delimiters  
x$ = EXTRACT$(COMMAND$, ANY " ,."+CHR$(9))  
  
' the following line returns "aba" (match on "cad")  
x$ = EXTRACT$("abacadabra","cad")  
  
' the following line returns nothing (match on first character "a")  
x$ = EXTRACT$("abacadabra", ANY "cad")
```

Purpose	Bind a field string variable to a random file buffer or a dynamic string variable.
Syntax	<pre>FIELD # <i>filenum</i>, <i>nSize</i> AS <i>fieldvar</i>, [FROM] <i>nStart</i> TO <i>nEnd</i> AS <i>fieldvar</i> [, ...] FIELD <i>DynamicStr</i>, <i>nSize</i> AS <i>fieldvar</i>, [FROM] <i>nStart</i> TO <i>nEnd</i> AS <i>fieldvar</i> [, ...] FIELD RESET <i>fieldvar</i> [, ...] FIELD STRING <i>fieldvar</i> [, ...]</pre>
Remarks	<p>A field variable is a special form of string variable which may be used just like a standard dynamic string variable, or it may be declared to reference a particular sub-section of a random file buffer or a dynamic string variable. Because of the added capabilities, it requires 12 bytes more storage space (16 vs 4) than a standard string variable. A field variable may not be used as a member of a User-Defined TYPE or UNION.</p> <p>By default, a field variable mimics a dynamic string variable, and may be considered a virtual replacement. Then, at any time, the FIELD statement can be used to declare that the field variable now refers to a specific portion of a random file buffer or a dynamic string. FIELD RESET is used to change it back to a nul (zero-length) dynamic string. FIELD STRING also changes it back to a dynamic string, but first assigns the current sub-section data to it. This last action is particularly useful in the case where the sub-section data might be lost when the bound random file is closed.</p> <p>In the first form, FIELD binds a field string variable to a specific sub-section of a random-access file buffer. In the second form, FIELD binds a field string variable to a specific sub-section of a dynamic string variable. If the sub-section extends beyond the actual size of the file buffer or string, that portion of the FIELD is empty. Otherwise, it represents a fixed size string, and may be referenced as any other string variable.</p> <p>When used with a file:</p> <p>A random-access file buffer is automatically created for use when GET or PUT are used without a target variable. In this case, the file data is read or written using this buffer, which is accessed with one or many field variables.</p> <p>If a field is defined by a single field (size) parameter, it represents the length of the field, with the start position implied by the preceding field within the statement. If two parameters are used, they represent the start (<i>nStart</i>) and end (<i>nEnd</i>) positions, indexed to one.</p> <p>If a string value shorter than the declared size is assigned to a field string, it is padded with blank spaces as it is placed into the file buffer. There is no requirement to use LSET for assignment.</p>

Finally, it should be noted that FIELD statements are tied to an [open file](#), i.e. they are valid only as long as the file is open. Once the [file is closed](#), any field strings that had been defined for the file will return nul (empty), not a string of the previously specified length.

```
LOCAL sFirst AS FIELD, sSecond AS FIELD
OPEN "ABC.TXT" FOR RANDOM AS #1 LEN=20
FIELD #1, 10 AS sFirst, 10 AS sSecond
sFirst = "0123456789"
sSecond = "9876543210"
Put #1 ' creates a record of: 01234567899876543210
```

When used with a dynamic string:

A field variable bound to a dynamic string works very much like a pointer, so the programmer must use care in field variable selection. For example, if you bind a [GLOBAL](#) FIELD variable to a [LOCAL](#) string variable, then attempt to reference the global string after the local is destroyed (i.e., released when the owning [Sub/Function/Method/Property](#) exits), a fatal exception error (GPF) is likely to occur. The same could happen after an [array](#) has been [erased](#), or a [REDIM](#) is used to change the memory allocation.

To avoid problems with [scope](#), it is suggested that field variables be bound only with strings within the same scope (LOCAL, GLOBAL, etc.).

```
LOCAL x$, sFirst AS FIELD, sSecond AS FIELD
FIELD x$, 3 AS sFirst, 3 AS sSecond
x$ = SPACE$(6) ' Allocate the space for the field
sFirst = "111"
sSecond = "222"
? x$           ' Displays 111222
x$ = "abcd"
? sFirst       ' Displays abc
? sSecond      ' Displays d
```

Restrictions Field string variables must be explicitly declared using DIM, [INSTANCE](#), LOCAL, [STATIC](#), GLOBAL, or [THREADED](#). Attempting to bind a variable other than a declared field variable results in a compile-time [Error 544](#) ("Field variable expected"). Field strings cannot be used in [UDT](#) or [UNION](#) structures. Attempting to do so results in a compile-time [Error 485](#) ("Dynamic/Field strings not allowed").

See also [Field Strings](#), [GET](#), [PUT](#), [TYPE/END TYPE](#), [User-Defined Types](#), [Unions](#), [UNION/END UNION](#)

Purpose Return information about an [open file](#).

Syntax `lResult& = FILEATTR([#] filenum&, fattr)`

Remarks *filenum&* is the handle of a currently open file. *fattr* is an [integer](#) between -3 and 3 that specifies the type of information required, according to the following table:

<i>fattr</i>	Definition																		
-3	The device type. Returns 1 for a file , 2 for a device. COMM , TCP and UDP are classified as devices.																		
-2	Logical first byte (base) position of a disk file. By default, Classic PowerBASIC opens files with a default first location of 1, but this can be overridden via the BASE= clause of the OPEN statement. This function can be useful when the base is not known or when performing SEEK operations.																		
-1	The minimum amount of data that can be read or written at one time. For RANDOM files , it is the record length. For INPUT files , it is the input buffer length (set with LEN= in the OPEN statement). For BINARY , OUTPUT and APPEND , there is no buffering, so it always returns 1 (1 byte).																		
0	The open state. TRUE (non-zero) if open, FALSE (zero) if closed.																		
1	The file mode (which may be a combination of the following): <table><tr><th><i>result&</i></th><th>File mode</th></tr><tr><td>1</td><td>Input</td></tr><tr><td>2</td><td>Output</td></tr><tr><td>4</td><td>Random</td></tr><tr><td>8</td><td>Append</td></tr><tr><td>16</td><td>Serial Communications (COMM)</td></tr><tr><td>32</td><td>Binary</td></tr><tr><td>64</td><td>TCP Winsock</td></tr><tr><td>128</td><td>UDP Winsock</td></tr></table> <p>(for example, an APPEND file will return 8 + 2 = 10).</p>	<i>result&</i>	File mode	1	Input	2	Output	4	Random	8	Append	16	Serial Communications (COMM)	32	Binary	64	TCP Winsock	128	UDP Winsock
<i>result&</i>	File mode																		
1	Input																		
2	Output																		
4	Random																		
8	Append																		
16	Serial Communications (COMM)																		
32	Binary																		
64	TCP Winsock																		
128	UDP Winsock																		
2	The operating system file handle for the file. This handle can be																		

	used with particular Windows API calls files to manipulate files opened with Classic PowerBASIC, and with the OPEN HANDLE statement.
3	<p>Enumerates existing file numbers. This mode enumerates existing file numbers, in the range of 1 to 32767. FILEATTR(1,3) returns the first located file number, FILEATTR(2,3) the second, and so on until -1 is returned to indicate that there are no more file numbers active. The file numbers returned are not guaranteed to be returned in any particular sequence, nor be open. You can use FILEATTR(#filenum,0) to determine whether a given file number is open or closed.</p> <p>The number symbol [#] is optional, but recommended for clarity.</p>

See also

[COMM OPEN](#), [EOF](#), [FILENAME\\$](#), [GETATTR](#), [LOF](#), [OPEN](#), [SEEK function](#), [SEEK statement](#), [SETATTR](#), [TCP OPEN](#), [UDP OPEN](#)

Example

```
OPEN "TEST.DOC" FOR OUTPUT AS #1 LEN = 28
x& = FILEATTR(#1,1)
Result  x& = 2
```

Purpose	Copy a file.
Syntax	<code>FILECOPY <i>sourcefile</i>, <i>destfile</i></code>
Remarks	<p>Copy the file <i>sourcefile</i> to the file <i>destfile</i>. Both <i>sourcefile</i> and <i>destfile</i> must be filenames, not merely drives or directories (although it's OK to include drive and directory specifications along with the filenames). Wildcards are not supported. If you attempt to copy a file that is read locked (preventing read access), a run-time Error 70 will occur ("Permission denied").</p> <p>If the destination file already exists, it will be overwritten. If it is not possible to overwrite the existing destination file (for example, it is marked as read-only or in use by another program), the result will be a run-time Error 70 ("Permission denied").</p> <p>The attributes of the source file are inherited by the destination file, with the exception of the Archive attribute, which is always set ON for the destination file. File attributes may be examined or modified with the GETATTR and SETATTR statements.</p>
See also	FILEATTR , GETATTR , SETATTR
Example	<code>FILECOPY "C:\AUTOEXEC.BAT", "C:\AUTOEXEC.BAK"</code>

FILENAME\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the file-system name of an open file .
Syntax	<code>a\$ = FILENAME\$(<i>filenum</i>&)</code>
Remarks	<code>a\$</code> receives the name of the open file identified by the file number <i>filenum</i> &. This function is not valid with a file opened using OPEN HANDLE , COMM OPEN , TCP OPEN , or UDP OPEN .
See also	CLOSE , FILEATTR , FREEFILE , GETATTR , OPEN , SETATTR
Example	<pre>OPEN "MYFILE.TXT" FOR INPUT AS #1 a\$ = FILENAME\$(1) CLOSE #1</pre>

FILESCAN statement

[Top](#) [Previous](#) [Next](#)

Purpose	Rapidly scan a file opened for INPUT or BINARY mode, in order to obtain size information about variable length string data.
Syntax	<code>FILESCAN [#] <i>fnum</i>&, RECORDS TO <i>y</i>& [, WIDTH TO <i>x</i>&]</code>
Remarks	<p>FILESCAN assigns a count of the lines/records/strings to <i>y</i>&, and if the WIDTH clause is specified, the length of the longest string to <i>x</i>&.</p> <p>In INPUT mode, it is assumed the data is standard text, with lines delimited by a CR/LF (\$CRLF) pair. FILESCAN stops reading the file if it encounters an "end of file" (EOF) marker byte (CHR\$(26) or \$EOF). Text that occurs after the last CR/LF but before the EOF is considered the last record of the file. Use the LINE INPUT# statement to read a complete text file into an array.</p> <p>In BINARY mode, it is assumed the file was written in the Classic PowerBASIC and/or VB packed string format using PUT of an entire string array. If a string is shorter than 65535 bytes, a 2-byte length WORD is followed by the string data. If a string is equal to or longer than 65535 bytes, a 2-byte value of 65535 is followed by a length DWORD value, then finally the string data.</p> <p>Use the GET statement to read a complete binary file into an array.</p>
Restrictions	If FILESCAN is applied to other file formats, the results are undefined.
See also	GET , GET\$, LINE INPUT# , PUT , PUT\$
Example	<pre>OPEN "datafile.dat" FOR INPUT AS #1 FILESCAN #1, RECORDS TO count& DIM TheData(1 TO count&) AS STRING LINE INPUT #1, TheData() TO count& CLOSE #1</pre>
Result	The entire text file comprising <i>y</i> & lines is read into the string array .

FIX function

[Top](#) [Previous](#) [Next](#)

Purpose Truncate a floating-point number to an integer.

Syntax `y = FIX(numeric_expression)`

Remarks FIX strips off the fractional part of its argument, and returns the integer part. Unlike [CINT](#) and [INT](#), FIX does not perform any form of rounding or scaling.

See also [CEIL](#), [CINT](#), [INT](#), [FRAC](#), [ROUND](#)

Example

```
x$ = "The integer part of 50.67 is" + STR$(FIX(50.67))
y$ = STR$(FIX(-1.1)) & ", " & STR$(INT(-1.1)) & ", " & STR$(CINT(-1.1))
Output  The integer part of 50.67 is 50
-1, -2, -1
```

FLUSH statement

[Top](#) [Previous](#) [Next](#)

Purpose	Flush file buffers to disk, to ensure that the disk information is up-to-date.
Syntax	<code>FLUSH [[#] <i>filenum</i>& [, [#] <i>filenum</i>&] ...]</code>
Remarks	FLUSH ensures that all data you have written to disk files has actually been written to disk. The CLOSE statement also flushes the buffers, but FLUSH has the advantage of leaving the files open.
<i>filenum</i> &	The file number of an OPEN file. If <i>filenum</i> & is specified, only the data for that file is flushed. Otherwise, data for all open files is flushed. The Number (#) symbol is optional, but recommended for the purposes of clarity.
See also	CLOSE , OPEN

Purpose Destroy a font when it is no longer needed.

Syntax `FONT END fonthdl&`

fonthdl& Handle of the font to be destroyed.

Remarks When you have no further need for a font originally created with [FONT NEW](#), you can destroy it and reclaim the memory space which was originally allocated for it.

If the specified font is still in use by a control, a [Graphic Control](#), a [Graphic Window](#), or an [XPrint](#) page, an error 5 (Illegal Function Call) will be generated.

When your program ends, any existing fonts are automatically destroyed by Classic PowerBASIC.

See also [CONTROL SET FONT](#), [FONT NEW](#), [GRAPHIC PRINT](#), [GRAPHIC SET FONT](#), [XPRINT](#), [XPRINT SET FONT](#)

Purpose	Create a new font for use with control, GRAPHIC PRINT , XPRINT , etc.
Syntax	<code>FONT NEW fontname\$ [,points!, style&, charset&, pitch&, escapement&] TO fhndl</code>
fontname\$	Name of the font.
points!	Size of the font, in points. This may be specified as a floating point value for fractional point sizes.
style&	Font style attribute. Any of the following values can be combined or used alone: 0 Normal 1 Bold 2 Italic 4 Underline 8 Strikeout
charset&	CharSet identifier. 0 ANSI CharSet 1 Default CharSet 2 Symbol CharSet 77 Mac CharSet 128 Shiftjis CharSet 129 Hangeul CharSet 130 Johab CharSet 136 Chinese CharSet 161 Greek CharSet 162 Turkish CharSet 177 Hebrew CharSet 178 Arabic CharSet 186 Baltic CharSet 204 Russian CharSet 222 Thai CharSet 238 East Europe CharSet 255 OEM CharSet
pitch&	Pitch and Font Family attribute. One of each group of values can be combined or used alone: 0 Default 1 Fixed width font

- 2 Variable width font
- 16 Roman font (Times Roman...)
- 32 Swiss font (Helvetica, Swiss...)
- 48 Modern font (Pica, Courier...)
- 64 Script font (Cursive...)
- 80 Decorative (OldEnglish...)

escapement& Specifies the angle, in tenths of degrees, between the character base line and the x axis. Allows printing of text on an angle.

fhndl Upon successful creation of a new font, a unique Classic PowerBASIC handle is assigned to this [Long Integer](#) or [DWord](#) variable. This handle is used with other statements and functions to specify the created font. If the font creation fails, the value zero (0) is assigned to *fhndl*.

Remarks This is the preferred method of creating and specifying fonts in Classic PowerBASIC. Using FONT NEW, you can create a group of fonts, in advance, and switch between them easily using [CONTROL SET FONT](#), [GRAPHIC SET FONT](#), and [XPRINT SET FONT](#).

If the requested font is not available on the computer, Windows will search for a substitute font, which is similar to the attributes specified (CharSet, Font Family, etc.).

You may use the value zero (0) for any of the numeric parameters to designate that the compiler should use the default for that item. If parameters are missing, the compiler substitutes the default value for all remaining parameters.

See also [CONTROL SET FONT](#), [FONT END](#), [GRAPHIC PRINT](#), [GRAPHIC SET FONT](#), [XPRINT](#), [XPRINT SET FONT](#)

Purpose	Define a loop of program statements whose execution is controlled by an automatically incrementing or decrementing counter.
Syntax	<pre>FOR Counter = start TO stop [STEP increment] [statements] [EXIT FOR] [statements] [ITERATE FOR] [statements] NEXT [Counter]</pre>
Remarks	<p><i>Counter</i> is a numeric variable serving as the loop counter.</p> <p><i>start</i> is a numeric expression specifying the value initially assigned to <i>Counter</i>.</p> <p><i>stop</i> is a numeric expression giving the value that <i>Counter</i> must reach for the loop to be terminated.</p> <p><i>increment</i> is an optional numeric expression defining the amount by which <i>Counter</i> is incremented with each loop execution. If not specified, <i>increment</i> defaults to 1.</p> <p>Note that <i>increment</i> must be the same data type or in the same range as <i>Counter</i>. For example:</p> <pre>FOR x?? = 50 TO 1 STEP -1</pre> <p>will fail because -1 is not within the range of an unsigned Word variable.</p> <p>When a FOR statement is encountered, <i>start</i> is assigned to <i>Counter</i>, and <i>Counter</i> is tested to see if it is greater than (or, for negative <i>increment</i>, less than) <i>stop</i>. If not, the statements within the FOR/NEXT loop are executed, <i>increment</i> is added to <i>Counter</i>, and <i>Counter</i> again tested against <i>stop</i>. The statements in the loop are executed repeatedly until the test fails, at which time control passes to the statement immediately following the NEXT.</p> <p>If <i>increment</i> is equal to the maximum value of a variable class (255 for a byte, 32767 for an Integer, 65535 for a Word, etc), the compiler will generate an error. If <i>step</i> is zero, an infinite loop can be created.</p> <p>When using floating point values with FOR/NEXT, be sure to allow for round-off errors when mixing numbers of different precision. Using constants or variables of the same type throughout will help solve this problem:</p> <pre>FOR n# = 1.0 TO 1.5 STEP 0.1 x\$ = STR\$(n#) NEXT n#</pre> <p>executes 5 times and returns:</p>

```
1.10000000149012
1.20000000298023
1.30000000447035
1.40000000596046
```

while:

```
FOR n@ = 1.0@ TO 1.5@ STEP 0.1@
  x$ = STR$(n@)
NEXT n@
```

executes 6 times and returns:

```
1
1.1
1.2
1.3
1.4
1.5
```

FOR/NEXT loops run fastest when *Counter* is a [Long-integer](#) variable, and *start* and *increment* are Long-integer constants. The value of *Counter* is available like any other variable within the loop. It is wise to avoid explicitly modifying the value of *Counter* within the loop. If you need to exit the loop prematurely, use an [EXIT FOR](#) statement. Keep range considerations in mind. For example, if *Counter* is an Integer variable, you may not use the maximum value for an Integer for *stop*, as *Counter* would be incremented outside the Integer range at the end of the loop.

The body of the loop is skipped altogether if the initial value of *Counter* is greater than *stop* (or, for a negative *increment*, if *Counter* is less than *stop*).

FOR/NEXT loops can be nested within other FOR/NEXT loops. Be sure to use unique counter variables. Note that Classic PowerBASIC allows the *Counter* in the NEXT keyword simply as a comment, which is ignored. For example, the following will compile, even though the counter variables are "crossed":

```
FOR n = 1 TO 10
  FOR m = 1 TO 20
    .
    .
    .
  NEXT n
NEXT m
```

You can omit the counter variable in the NEXT statement altogether. For example:

```
FOR n = 1 TO 10
  .
  .
  .
NEXT
```

If a NEXT is encountered without a corresponding FOR (or vice versa), a [compile-time error](#) is generated.

Previous version of Classic PowerBASIC supported a single NEXT statement used with multiple nested FOR/NEXT loops, such as NEXT c, b, a. This is no longer supported and you will need to update your code to use multiple NEXT statements.

In certain situations, previous versions of Classic PowerBASIC optimized FOR/NEXT loops to count down instead of up for improved execution speed. This optimization could cause the counter variable to contain a value which was not expected when execution of the loop was complete. This optimization has been improved so that the counter variable value is always correct upon loop completion, even if EXIT FOR was used to force an early termination.

Although the compiler does not care about such things, it is considered good programming practice to indent the statements between FOR and NEXT by two or three spaces to set off the structure of the loop.

For additional performance, use a [REGISTER](#) variable for the loop counter variable.

Restrictions A variable that has been passed by reference (BYREF) to a [Sub](#), [Function](#), [Method](#), or [Property](#) may not be used as a Counter. Use a [local](#) variable instead.

See also [#OPTIMIZE](#), [#REGISTER](#), [DO/LOOP](#), [EXIT](#), [ITERATE](#), [WHILE/WEND](#), [REGISTER](#)

FORMAT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose Format string data according to instructions contained in a format expression.

Syntax `x$ = FORMAT$(num_expression [, [digits& | fmt$])`

Remarks FORMAT\$ has the following parts:

num_expression The numeric expression, [variable](#), or [literal](#) value to be formatted. This argument is converted to full ([Extended](#)) precision before formatting commences.

digits& The maximum number of significant digits, in the range of 1 to 18. If not included, Classic PowerBASIC supplies a default value of 7 for [single precision](#) values, or 16 for more precise values. This form of the function is very similar to the [STR\\$\(\)](#) function, except that it never supplies any leading or trailing spaces. Use care that *digits&* is large enough to contain the whole part of a number, or scientific notation must be used to estimate it. For example, FORMAT\$(123.456, 2) returns the string "1.2E+2", while FORMAT\$(123.456, 5) returns the string "123.45".

fmt\$ Format characters that will determine how the numeric expression should be formatted. This expression is termed the *mask*. There may be up to 18 digit-formatting digits on either side of the decimal point. The mask may not contain literal characters unless each character is preceded with a backslash (\) escape character, or the literal characters are enclosed in quotes.

fmt\$ may contain one, two or three formatting masks, separated by semicolon (;) characters:

One mask If *fmt\$* contains just one format mask, the mask is used to format all possible values of *num_expression*. For example:

```
x$ = FORMAT$(z!, "000.00")
```

Two masks If *fmt\$* contains two format masks, the first mask is used for positive values (≥ 0), and the second mask is used for negative values (< 0). For example:

```
x$ = FORMAT$(-100, "+00000.00;-000")
```

Three masks If *fmt\$* contains three masks, the first mask is used for positive values (> 0), the second mask for negative values (< 0), and the third mask is used if *num_expression* is zero (0). For example:

```
FOR y! = -0.5! TO 0.5! STEP 0.5!
```

```

x$ = FORMAT$(y!, "+.0;-.0; .0")
NEXT y!

```

Digit placeholders in a mask do not have to be contiguous. This allows you to format a single number into multiple displayed parts. For example:

```
A$ = FORMAT$(123456, "00\:00\:00") ' 12:34:56
```

The following table shows the characters you can use to create the user-defined format strings (masks) and the definition of each formatting character:

Character Definition

Empty string [null string] No formatting takes place. The number is converted to Extended-precision and formatted similarly to STR\$, but without the leading space that STR\$ applies to non-negative numbers.

```

A$=FORMAT$(0.2) ' .200000002980232
A$=FORMAT$(0.2!, "") ' .200000002980232
A$=FORMAT$(0.2#) ' .2
A$=FORMAT$(0.2#, "") ' .2

```

0 **[zero]** Digit placeholder. Classic PowerBASIC will insert a digit or 0 in that position.

If there is a digit in *num_expression* in the position where the 0 appears in the format string, return that digit.

Otherwise, return "0". If the number being formatted has fewer digits than there are zeros (on either side of the decimal point) in the format expression, leading or trailing zeros are added. If the number has more digits to the right of the decimal point than there are zeros to the right of the decimal point in the format expression, the number is rounded to as many decimal places as there are zeros in the mask.

If the number has more digits to the left of the decimal point than there are zeros to the left of the decimal point in the format expression, the extra digits are displayed without truncation. If the numeric value is negative, the negation symbol will be treated as a decimal digit.

Therefore, care should be exercised when displaying negative values with this placeholder style. In such cases, it is recommended that multiple masks be used.

```

' Numeric padded with leading zero characters
A$ = FORMAT$(999%, "00000000") ' 00000999

```

[Number symbol] Digit placeholder. If there is a digit for this position, Classic PowerBASIC replaces this placeholder with a digit, nothing, or a user-specified character.

Unlike the 0 digit placeholder, if the numeric value has the fewer digits than there are # characters on either side of the decimal placeholder, Classic PowerBASIC will either:
a) Omit this character position from the final formatted string; or

Substitute a user-specified replacement character if one has been defined (see the asterisk (*) character for more information). To specify leading spaces, prefix the mask with "*" (asterisk and a space character).

For example:

```
' No leading spaces and trailing spaces
A$ = FORMAT$(0.75!, "####.###")      ' 0.75
' Up to 3 Leading spaces before decimal
A$ = FORMAT$(0.75!, "* ##.###")      '    0.75
' Using asterisks for padding characters
A$ = FORMAT$(0.75!, "*==##.###")      ' ==0.75=
```

FORMAT\$ may also return a string that is larger than the number of characters in the mask:

```
A$ = FORMAT$(999999.9, "#.#")      ' 999999.9
```

[period] Decimal placeholder. Determines the position of the decimal point in the resultant formatted string.

If any numeric field is specified to the left of the decimal point, at least one digit will always result, even if only a zero. The zero is not considered to be a "leading" zero if it is the only digit to the left of the decimal. Placing more than one period character in the *fmt\$* string will produce undefined results.

% [percent] Percentage placeholder. Classic PowerBASIC multiplies *num_expression* by 100, and adds a trailing percent symbol. For example:

```
x$ = FORMAT$(1 / 5!, "0.0%")      ' 20.0%
```

, [comma] Thousand separator. Used to separate thousands from hundreds within a number that has four or more digits to the left of the decimal point. In order to be recognized as a format character, the comma must be placed immediately after a digit placeholder character (also see Restrictions below).

```
A$ = FORMAT$(1234567@, "#,")      ' 1,234,567
A$ = FORMAT$(12345@, "#,.00")      ' 12,345.00
A$ = FORMAT$(12345@, "#.00,")      ' 12,345.00
A$ = FORMAT$(1212.46, "$00,000.00") ' $01,212.46
A$ = FORMAT$(1000%, ""#"#"#,")      ' #1,000
A$ = FORMAT$(1234567@, "0,")      ' 1,234,567
```

***x** [asterisk] Digit placeholder and fill-character. Instructs

Classic PowerBASIC to insert a digit or character "x" in that position. If there is a digit in *num_expression* at the position where the * appears in the format string, that digit is used; otherwise, the "x" character is used (where "x" represents your own choice of character). The *x specifier acts as two digit (#) fields.

```
A$ = FORMAT$(9999.9@, "$**####, .00") ' $**9,999.90
A$ = FORMAT$(0@, "$*#####0, .00#") ' $=====0.00=
A$ = FORMAT$(0@, "$* #####0, .00") ' $ 0.00
```

E- e- E+ [e] Scientific format. Classic PowerBASIC will use scientific notation in the formatted output. Use E- or e- to place a minus sign in front of negative exponents. Use E+ or e+ to place a minus sign in front of negative exponents and a plus sign in front of non-negative exponents.

In order to be recognized as a format sequence, the E-, e-, E+, or e+ must be placed between two digit placeholder characters. For example:

```
A$ = FORMAT$( 99.999, "0.0E-##") ' 1.0E2
A$ = FORMAT$(-99.999, "0.0E-##") ' -1.0E2
A$ = FORMAT$( 99.999, "0.0E+##") ' 1.0E+2
A$ = FORMAT$(-99.999, "0.0E+##") ' -1.0E+2
A$ = FORMAT$(0.1!, "0.0e+##") ' 1.0e-1
```

" [double-quote] Quoted string. Classic PowerBASIC treats all characters up to the next quotation mark as-is, without interpreting them as digit placeholders or format characters. Also see backslash. For example:

```
A$ = FORMAT$(12, $DQ+"##"+$DQ+"##") ' ##12
A$ = FORMAT$(5.55, ""XYZ=""#.##\#"') ' XYZ=5.55#
A$ = FORMAT$(25, ""x=""#"') ' x=25
A$ = FORMAT$(999, ""Total ""#"') ' Total 999
A$ = FORMAT$(5, $DQ+"x="+$DQ+"#"') ' x=5
```

\x [backslash] Escaped character prefix. Classic PowerBASIC treats the character "x" immediately following the backslash (\) as a literal character rather than a digit placeholder or a formatting character. Many characters in a mask have a special meaning and cannot be used as literal characters unless they are preceded by a backslash.

The backslash itself is not copied. To display a backslash, use two backslashes (\\). To display a literal double-quote, use two double-quote characters.

To simplify the mask string for common numeric formats, FORMAT\$ permits the dollar symbol, the left and right parenthesis symbols, the plus and minus symbols, and the

space character ("\$()+- ") to pass through from the mask string into the formatted output string, without requiring an escape (\) prefix character.

```
A$ = FORMAT$(23, "( * 0\%)" ) ' ( 23%)
A$ = FORMAT$(99999, "#\#") ' 99999#
A$ = FORMAT$(5, "\"+$DQ+$DQ+"x="+$DQ+ _
           "#\"+$DQ) ' "x=5"
A$ = FORMAT$(5, "\"\"\"\"x=\"\"#\#\"") ' "x=5"
A$ = FORMAT$(1000%, "\"\"#####,\"\"") ' "#####,"
```

Restrictions

You cannot pass a [string expression](#) or string variable in *num_expression*. Do not place more than one decimal point in the mask.

FORMAT\$ can return the maximum possible number of digits (up to 4932 for [Extended-precision](#)); however, the resulting digits will be meaningless beyond the actual precision of *num_expression*.

Consequently, the value of *num_expression* may produce formatted strings that are wider than the length of *fmt\$*, for example:

```
A$ = FORMAT$(3e30!, "#,###") ' returns 41 characters
```

Rounding, if necessary, is implemented by the "banker's rounding" principle: if the fractional digit being rounded off is exactly five, with no trailing digits, the number is rounded to the nearest even number. This provides better results, on average, and follows the IEEE standard. For example:

```
A$ = FORMAT$(0.5##, "0") ' 0
A$ = FORMAT$(1.5##, "0") ' 2
A$ = FORMAT$(2.5##, "0") ' 2
A$ = FORMAT$(2.51##, "0") ' 3
```

Semicolon characters, being mask delimiters, should not be used for other purposes in mask strings unless prefixed with an escaped character symbol (\).

FORMAT\$, when used with some formatting characters such as the thousands separator (comma), may not produce a "right-justified" formatted string. The simple solution is to apply separate justification with the [RSET](#) statement or the [RSET\\$](#) function. For example:

```
A$ = SPACE$(12)
RSET A$ = FORMAT$(1, "#,###.00") ' 1.00
RSET A$ = FORMAT$(1000, "#,.00") ' 1,000.00
RSET A$ = FORMAT$(1000000, "#,###.00") ' 1,000,000.00
B$ = RSET$(FORMAT$(1e6, "#,") ,10) ' " 1,000,000"
```

One further enhancement would be to combine this into a [MACRO](#) function, for example:

```
MACRO mMoney1(d,1) = "$"+RSET$(FORMAT$(d,"#",") ,1-1)
MACRO mMoney2(d,1) = RSET$(FORMAT$(d,"$#",") ,1)
' code here
A$ = mMoney1(1000,10) ' "$ 1,000"
B$ = mMoney2(1000,10) ' " $1,000"
```

See also

[BIN\\$](#), [GRAPHIC PRINT](#), [GUID\\$](#), [HEX\\$](#), [OCT\\$](#), [REPEAT\\$](#), [SPACE\\$](#), [STR\\$](#), [STRING\\$](#), [USING\\$](#), [VAL](#), [XPRINT](#)

FRAC function

[Top](#) [Previous](#) [Next](#)

Purpose Return the fractional part of a floating-point number.

Syntax `h = FRAC(float_expression)`

Remarks FRAC returns the number *after* the decimal point of a floating-point number or expression. FRAC rounds the result of fit the precision of the target *h*, as per IEEE specifications.

See also [CEIL](#), [CINT](#), [FIX](#), [INT](#), [ROUND](#)

Example `h# = FRAC(10.25#) ' = 0.25# (Double-precision)`

Purpose	Return the next available Classic PowerBASIC file number.
Syntax	<code>x% = FREEFILE</code>
Remarks	<p>FREEFILE returns an Integer value in the range 1 to 32767, which dictates the next available file number that may be used to OPEN a file or device. Using FREEFILE, your program can open files and devices without the need to keep track of which file numbers are already in use.</p> <p>FREEFILE is thread-safe, returning a new file number with <u>each</u> invocation. This means that two or more consecutive calls to FREEFILE will return different file numbers regardless of whether they were used to open a file or not. This behavior differs from previous versions of Classic PowerBASIC, where FREEFILE returned the same file number consistently until the file number was actually used to open a file or device.</p> <p>FREEFILE is vastly superior to using hard-coded file numbers because it eliminates the possibility of a file number being used more than once in a module at any given moment.</p> <p>A file number returned by FREEFILE can be used with the COMM OPEN, TCP OPEN and UDP OPEN statements, as well as standard file I/O OPEN and OPEN HANDLE statements.</p> <p>Use FILENAMES\$ to return the name of the open file that corresponds to a given file number.</p>
Restrictions	<p>FREEFILE returns file numbers within a predictable and convenient range of values. Classic PowerBASIC file numbers are also <u>specific</u> (private) to the Classic PowerBASIC module in which they are used to OPEN a file or device. This means that a file number in use in one module will have no definition or meaning if passed to another module or API function.</p> <p>However, it can sometimes be necessary for a different module or even an API function to access a file that is already open. In this case, it is necessary to use the FILEATTR function to obtain the operating system file handle, and this value can be passed to other modules or API functions. These modules would use the OPEN HANDLE statement to gain access into the already-open file.</p>
See also	COMM OPEN , FILEATTR , FILENAMES\$, OPEN , TCP OPEN , UDP OPEN
Example	<pre>x%= FREEFILE OPEN MyFileName\$ FOR OUTPUT AS #x%</pre>

FUNCNAME\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the name of the current Sub/Function/Method/Property .
Syntax	<i>f\$</i> = FUNCNAME\$
Remarks	<p>FUNCNAME\$ returns the name of the procedure in which it is located. If an ALIAS is specified, FUNCNAME\$ returns the ALIAS name; otherwise, it returns the primary name capitalized. Returning the ALIAS name provides a mechanism to disguise sensitive internal procedure names even when reporting error conditions to a user.</p> <p>FUNCNAME\$ can be useful as a debugging tool, or in situations where an error handler in a procedure passes error information on to a "central" procedure for logging and handling. FUNCNAME\$ does <u>not</u> require #TOOLS ON.</p>
See also	#TOOLS , CALLSTK , CALLSTK\$, CALLSTKCOUNT , FILENAME\$, FUNCTION , METHOD , PROFILE , PROPERTY , SUB , TRACE
Example	<pre>SUB SecretEncryptionSub ALIAS "MySub" (sData\$) x\$ = FUNCNAME\$ ' Returns "MySub" END SUB [statements] SUB SecretDecryptionSub (sData\$) x\$ = FUNCNAME\$ ' Returns "SECRETDECRYPTIONSUB" END SUB</pre>

FUNCTION/END FUNCTION statements

[Top](#) [Previous](#)
[Next](#)

Purpose Define a Function block.

Syntax

```
[CALLBACK | THREAD] FUNCTION FuncName [BDECL | CDECL | SDECL] [ALIAS  
"AliasName"] [[arguments]] [EXPORT | PRIVATE] [AS type]  
[DIM | LOCAL | STATIC variable list]  
[statements]  
[{FuncName | FUNCTION} = ReturnValue]  
[statements]  
[EXIT FUNCTION]  
[statements]  
END FUNCTION
```

Remarks All executable code must reside in a [Sub](#), Function, [Method](#), or [Property](#) block. Functions may not be nested. That is, you cannot define a code block (Sub, Function, Method, Property) inside another code block.

Previous versions of Classic PowerBASIC required that you create an explicit [DECLARE](#) statement if you wished to execute a SUB or FUNCTION which did not physically precede the reference to it. This extra work is no longer required, as Classic PowerBASIC resolves all forward references to internal procedures automatically.

DECLARE statements for a Sub/Function imported from a DLL must still precede any reference to the procedure.

CALLBACK Specifies that this is a [callback](#) function, which is used only to receive messages from the operating system. It may never be called directly from your code. Details about the message sent to the callback are retrieved using the [CB](#) group of Classic PowerBASIC functions. Callback functions may not include parameters, and always return a [long integer](#) result. For example:

```
CALLBACK FUNCTION DlgProc AS LONG  
    ' Callback code goes here  
END FUNCTION
```

Callback functions have the unique ability to optionally return two distinct values when necessary for certain Windows messages. This allows them to return the value zero (0) as a function result, while still specifying that the message has been processed. See the section CALLBACK RETURN VALUE (below) and the [CALLBACKS](#) page for more details.

THREAD Specifies that this is a thread function, which is the point where execution of a new thread begins. It may never be called directly from your code. Thread functions must take exactly one long-integer or [double-word](#) parameter by value (BYVAL), and must return either a long-integer or

double-word result. For example:

```
THREAD FUNCTION MyThreadFunction(BYVAL x AS LONG) AS LONG
' Thread code goes here
END FUNCTION
```

The [THREAD CREATE](#) statement creates and begins execution of a new thread Function.

FuncName

The name of the Function. A [type-specifier](#) may be appended (just like an ordinary variable name) to specify the data type of the Function's return value, in place of the [AS type] clause. *FuncName* must be unique: no other variable, Function, Sub, Method, Property, or [label](#) can share it. Also see ALIAS below.

Future versions of Classic PowerBASIC may not support type-specifier symbols for the Function return type, so specify the return data type with an explicit AS type clause in all [DECLARE](#) and FUNCTION definitions, to ensure future compatibility.

BDECL

Specifies that the declared procedure uses the BASIC/Pascal calling convention. When a procedure calls a BDECL procedure, it passes its parameters on the [stack](#) from left to right.

It is the responsibility of the called procedure to clean up the stack before returning to the calling procedure. Therefore, all Classic PowerBASIC procedures that specify the BDECL convention automatically clean up the stack before execution returns to the calling code.

CDECL

Specifies that the declared Function uses the C calling convention. When a CDECL Function is called, it passes its parameters on the stack from right to left.

The calling procedure removes any passed parameters from the stack as part of the return process. When Classic PowerBASIC code calls procedures using the CDECL convention, the stack is cleaned automatically after execution returns from the called code.

In the event the called procedure is imported or exported, Classic PowerBASIC will automatically create a C style ALIAS for the function name. This alias will be prefixed with an underscore followed by the original function name converted to lowercase.

The following two declarations are equivalent, indicating how the default ALIAS name would be created by Classic PowerBASIC:

```
FUNCTION C_Function CDECL () EXPORT AS LONG
FUNCTION C_Function CDECL ALIAS "_c_function" () EXPORT AS LONG
```

SDECL

This is the default if neither BDECL nor CDECL are specified. SDECL (and its synonym STDCALL) specifies that the declared Function uses the "Standard Calling Convention" as defined by Microsoft. When calling an SDECL Function, parameters are passed on the stack from right to left.

Classic PowerBASIC procedures that use the SDECL/STDCALL convention automatically clean up the stack before execution returns to the calling code.

ALIAS [String literal](#) that identifies an case-sensitive alternative name for the function. This lets you export a Function by a different unique name. This can be useful if you want to abbreviate a long name, provide a more descriptive name, or if the exported name needs to contain characters that are illegal in Classic PowerBASIC. *AliasName* is the routine's actual name as it appears in the export table, and *FuncName* is the title that you can use in Classic PowerBASIC. For example:

```
FUNCTION ShortName ALIAS "LongFuncName"() EXPORT STATIC AS LONG
```

The ALIAS clause is very important when exporting procedures. Omitting the ALIAS clause or incorrectly capitalizing the alias name are common causes of "Missing Export" errors. Please refer to the [DECLARE](#) topic for more information.

Passing parameters

arguments

An optional, comma-delimited sequence of formal parameters. The parameters used in the *arguments* list serve only to define the Function; they have no relationship to other variables in the calling code with the same name.

Normally, Classic PowerBASIC passes parameters to a Function either by reference (BYREF) or by copy (BYCOPY). Either way, the address of the variable is passed and the procedure has to look at that address to get the value of the parameter. If you do not need to modify the parameters (true in many cases), you can speed up your calls by passing the parameters by value using the BYVAL keyword.

You can also clarify that a parameter is passed by reference by using the optional BYREF keyword.

The type of the parameter is specified either by appending a type-specifier character to the name or by using an AS clause. For example:

```
FUNCTION Test&(A AS INTEGER) 'integer passed by ref
FUNCTION Test&(A%)          'integer passed by ref
FUNCTION Test&(BYREF A%)    'integer passed by ref
FUNCTION Test&(BYVAL A%)    'integer passed by val
```

Parameter restrictions

Classic PowerBASIC compilers have a limit of 32 parameters per FUNCTION. To pass more than 32 parameters to a FUNCTION, construct a [User-Defined Type](#) (UDT) and pass the UDT by reference (BYREF) instead.

[Fixed-length strings](#), [ASCIIZ strings](#), and [User-Defined Types/Unions](#) may

also be passed as BYVAL or OPTIONAL parameters. Try to avoid passing large items BYVAL, as its terribly inefficient, and there is a maximum size limit of 64 Kb for a given parameter list.

Classic PowerBASIC Functions cannot return an [array](#) or [Variant](#) variable as a Function return value. Pass these variable types as BYREF parameters instead. For example:

```
lResult& = ProcessData(TheArray&(), iSize%)
[statements]
FUNCTION ProcessData(lArr() AS LONG, iSize%) AS LONG
    REDIM lArr(iSize%) AS LONG
    lArr(iSize%) = 1&
    FUNCTION = -1&
END FUNCTION
```

Pointer parameters

When a Function definition specifies either a BYREF parameter or a [pointer](#) variable parameter, the calling code may freely pass a BYVAL DWORD or a Pointer instead. Pointer variable parameters must always be declared as BYVAL parameters.

```
' Integer Pointer (passed by value)
FUNCTION Test(BYVAL A AS INTEGER PTR) AS LONG
    @A = 56
END FUNCTION
```

Additional information on BYVAL/BYREF/BYCOPY parameter passing can be found in the [CALL](#) statement topic.

Optional parameters

Classic PowerBASIC supports two syntax formats for optional parameters: the classic optional parameter syntax using brackets "[..]", and the new syntax using the OPTIONAL (or OPT) keyword. We'll discuss each one in turn.

Using OPTIONAL/OPT

FUNCTION statements may specify one or more parameters as optional by preceding the parameter with either the keyword OPTIONAL or OPT. Optional parameters are only allowed with CDECL or SDECL calling conventions, not BDECL.

When a parameter is declared optional, all subsequent parameters in the declaration are optional as well, whether or not they specify an explicit OPTIONAL or OPT directive. The following two lines are equivalent, with both second and third parameters being optional:

```
FUNCTION sABC(a&, OPTIONAL BYVAL b&, OPTIONAL BYVAL c&) AS LONG
FUNCTION sABC(a&, OPT BYVAL b&, BYVAL c&) AS LONG
```

[VARIANT](#) variables are particularly well suited for use as an optional parameter. If the calling code omits an optional VARIANT parameter,

(BYVAL or BYREF), Classic PowerBASIC (and most other compilers) substitute a variant of type VT_ERROR which contains an error value of %DISP_E_PARAMNOTFOUND (&H80020004). In this case, you can check for this value directly, or use the [ISMISSING\(\)](#) function to determine whether the parameter was physically passed or not.

When optional parameters (other than a VARIANT) are omitted in the calling code, the stack area normally reserved for those parameters is zero-filled. This allows you to test if an optional parameter was passed or not:

If the parameter is defined as a BYVAL parameter, it will have the value zero. For TYPE or UNION variables passed BYVAL, the compiler will pass a string of binary zeroes of length [SIZEOF](#)(Type_or_union_var).

If the parameter is defined as a BYREF parameter, [VARPTR](#) (varname) will equal zero; when this is true, any attempt to use *Var_name* in your code will result in [Error #9](#) (null pointer); failure to detect this error using [error-trapping](#) may result in a General Protection Fault or memory corruption. You should use the ISMISSING() function first to determine whether it is safe to access the parameter.

The OPTIONAL directive provides the same functionality as the older syntax using square brackets "[.]". See below.

Using classic optional parameters

When declaring a CDECL procedure, you can specify trailing parameters as optional, using a set of brackets "[.]":

```
FUNCTION KerPlunk CDECL (x%, y% [, z%]) AS LONG
```

Note that the comma separating the y% parameter from the optional z% parameter is inside the brackets. The following calling sequences would then be valid:

```
x% = KerPlunk(x%, y%)
x% = KerPlunk(x%, y%, z%)
```

Optional parameters must be the last parameters designated in the list. The following is **invalid**:

```
FUNCTION KerPlunk CDECL ([x%,] y%, z%) AS LONG
```

Because the FUNCTION (SUB, METHOD, or PROPERTY) being called does not know how many parameters are being passed at the time it is called, you should pass the number of parameters as one of the required parameters in the list.

Classic PowerBASIC continues to support the use of classic optional parameter syntax using brackets ([.]) but this will not be the case in future versions of Classic PowerBASIC. Existing code should be changed to the new OPTIONAL syntax as soon as

possible to ensure compatibility with future versions of Classic PowerBASIC.

EXPORT	Functions are private by default. The EXPORT keyword can be used to make a Function accessible from another module.
PRIVATE	Functions are private by default. The PRIVATE keyword is not required, but may be used for clarity.
AS type	<p>Function blocks are constructed very much like Subs (see SUB/END SUB statement). However, Functions differ from Subs in that they always return a result, so they can be used in assignments and expressions. Therefore, there are two ways to specify the return type of a Function:</p> <p>You may specify the type of data returned by a Function to the calling code. If you do not specify a type, Classic PowerBASIC assumes that the Function returns the data type specified by a DEFtype statement. However, if no DEFtype or AS type has been specified, a compile-time error is generated.</p>

Therefore, there are two ways to specify the return type of a Function:

- Include a type-specifier character at the end of *FuncName*
- Include the AS type clause as the last part of the FUNCTION statement (this is the recommended syntax to ensure future compatibility).

For example, the following statements are equivalent:

```
FUNCTION aFunction?()  
FUNCTION aFunction() AS BYTE
```

While most FUNCTION calling conventions are fairly well defined throughout the industry, there are a few exceptions. In the case of functions which return a [Quad Integer](#) value, some programming languages (including Classic PowerBASIC) return the quad value in the FPU, while others return it in EDX:EAX. Classic PowerBASIC automatically detects the method used by imported functions and adjusts accordingly for you, but that's not a feature found in other compilers. Therefore, we recommend that you do not EXPORT QUAD FUNCTIONS unless they will only be accessed by Classic PowerBASIC programs. A simple equivalent functionality would be to return the quad-integer value to the caller in a BYREF QUAD parameter.

Assigning a return value

You can specify the return value of the Function by explicitly setting the value, either by assigning a value to the FUNCTION keyword, or by assigning a value to the function name. For example, the two lines within the following Function block are equivalent:

```

FUNCTION AddData() AS LONG
    [statements]
    AddData = 123& ' Assign value to function name
    FUNCTION = 123& ' Assign value to the function
END FUNCTION

```

Default return value

If the code within the Function does not explicitly set a return value, the default return value will be zero if the function returns a numeric data type, or an empty string if the function returns a string. For example:

```

FUNCTION AddData() AS LONG
    [statements]
    IF condition THEN
        EXIT FUNCTION ' No assignment, will return 0&
    ELSE
        FUNCTION = -1& ' An explicit return value
    END IF
END FUNCTION

```

Classic PowerBASIC Functions cannot return an array as a Function return value. Pass the array as a parameter instead. For example:

```

lResult& = CheckTheData(InTheArray&())
[statements]
FUNCTION CheckTheData(lArr() AS LONG) AS LONG
    [statements]
END FUNCTION

```

CALLBACK Return Value

Callback functions always return a long integer result. The primary purpose of this return value is to tell the Classic PowerBASIC [DDT](#) engine and the Windows operating system whether your Callback Function has processed this particular message. If you return the value [TRUE](#) (any non-zero value), you are asserting that the message was processed and no further handling is needed. If you return the value [FALSE](#) (zero), the Classic PowerBASIC DDT engine will manage the message for you, using the default message procedures in Windows. If you do not specify a return value in the function, Classic PowerBASIC chooses the value FALSE (zero) for you.

The term "process a message" may have many meanings. If it's a simple notification of a change in focus or style, which has no impact on your program, you may decide to consider it processed, yet do nothing. In other cases, your reaction could be quite complex and involved. As the programmer, that's your decision to make. But, regardless of your reaction, you should consider a message "processed" (returning a true value) whenever no further handling of the message (by DDT or Windows) is needed.

In some cases, especially when dealing with Common Controls and custom controls, you may be required to return a second result value through a special Windows data area named DWL_MSGRESULT. When you

complete a Callback Function, Classic PowerBASIC automatically copies any non-zero return value to DWL_MSGRESULT, if you haven't done so already. Therefore, it's generally safe to ignore this requirement in your code.

In most cases, when you process a message, you'll return a generic value for TRUE, such as: `FUNCTION = 1`. However, some messages require that you return a special value for TRUE, such as a graphical brush handle. As long as the value is non-zero, you can return it in the normal manner (with `FUNCTION = n`), since any non-zero value automatically implies that the message was processed.

That said, there are a few unique messages which may require special handling. Luckily, they're rare, but some just "break all the rules" listed above. For example, you might find one which requires a zero result, even when you have processed the message. You may find another which requires the return value be different from DWL_MSGRESULT. For these very special cases, you can simply specify two return values:

```
FUNCTION = 1, BrushHandle&
```

In this form, the first numeric expression specifies the value to be returned from the Callback Function. The second numeric expression tells the value to be assigned to DWL_MSGRESULT. When you use this double parameter assignment, the results are absolute. Classic PowerBASIC assumes you have processed the message, regardless of the values given. Classic PowerBASIC makes no other assumptions of any kind about these values. A double parameter function assignment is only allowed in a Callback Function.

Previous versions of Classic PowerBASIC did not offer a double parameter form of function return. This caused some difficulty with a few Windows messages which required a special return value of zero. If you return a value of zero (0) with the single parameter form, it implies the message was not processed at all by the Callback. This issue is totally circumvented by the double parameter form.

Variables within functions

[LOCAL](#) variables are created within the procedures stack frame. If a LOCAL variable exceeds the amount of stack space available, it may become necessary to use a [STATIC](#) or [GLOBAL](#) variable instead. For example, creating a LOCAL [ASCIIZ](#) or LOCAL [fixed-length](#) string that is very large (say, approaching 1 MB) can trigger a General Protection Fault (GPF) because it may overrun the stack frame.

The use of the STATIC keyword to define the default scope of variables inside the function to a static scope will be supported for a

limited time only. Existing code should be modified to use the [STATIC](#) statement as soon as possible.

See also

[DECLARE](#), [EXIT](#), [FUNCNAME\\$](#), [GLOBAL](#), [INSTANCE](#), [ISMISSING](#), [LOCAL](#), [METHOD](#), [PROPERTY](#), [STATIC](#), [SUB](#), [THREAD CREATE](#), [THREADID](#)

Example

```
FUNCTION HalfOf ALIAS "HalfOf" (X!) EXPORT AS SINGLE
  FUNCTION = X! / 2
END FUNCTION
```


Purpose Read a record from a [random-access file](#), or a [variable](#) or an [array](#) from a [binary file](#).

Syntax

Random-Access files:
GET [#] *filenum*&, [Rec], [ABS] Var
GET [#] *filenum*& [, Rec]
Binary files:
GET [#] *filenum*&, [RecPos], Var
GET [#] *filenum*&, [RecPos], Arr() [RECORDS *rcds*] [TO *count*]

Remarks The GET statement has the following parts:

filenum& The [file number](#) under which the file was opened.

Random Access files

Rec For random-access files, *Rec* is the record number to be read, from 1 to 2⁶³-1 (the maximum positive value for a [Quad-integer](#)). If *Rec* is omitted, the next record in sequence (following the one specified by the most recent GET or [PUT](#)) is read. If the file was just opened, the first record is read.

Var If *Var* is smaller than the defined record length, GET will read enough data to fill *Var*. The remainder of the record is discarded and the file pointer is placed at the next record position. If *Var* is larger than the defined record length, GET will read one record into *Var*, and the file pointer will be moved to the next record.

When GET is used to retrieve data from a random access file into a [dynamic \(variable-length\) string](#), Classic PowerBASIC looks for a 2-byte ([WORD](#)) length field at the beginning of each record which indicates the length of the string that follows. The length of the string is set by this value. The argument to the LEN clause of the [OPEN](#) statement must be at least 2 bytes greater than the actual string length. This 2-byte (WORD) length field is placed in the file by the PUT statement when used with dynamic (variable-length) strings.

(no *Var*) When the second form of GET is used (without a *Var* target string), GET reads the file data from the current file pointer into an internal buffer. This data can then be accessed using [FIELD](#) string variables.

ABS When GET is used to read a random file into a dynamic string, it normally expects the first two bytes of the record to contain the length of the valid data contained in the record. The ABS keyword specifies that no length word exists in the data, and the number of bytes to read is defined by the current length of the dynamic string variable. If the variable length is greater than the file record length, the remainder of the string variable is filled with nul's ([CHR\\$\(0\)](#) or [\\$NUL](#)). This offers greater compatibility with

the actual operation of other versions of BASIC, such as [PowerBASIC for DOS](#).

A random access file record is limited to 32768 bytes to ensure consistent behavior across all supported Win32 platforms. [GET\\$](#) and other related functions are not constrained in this manner.

Binary files

RecPos

For binary files, *RecPos* is the starting byte or seek position from where the GET should begin. The optional `BASE =` clause of the OPEN statement defines whether the first position is 0 or 1. The base position is 1 by default. If *RecPos* is greater than the number of records or bytes in the file, no error occurs but unpredictable data may be read. Use the [EOF](#) function to avoid reading past the end of the file.

Var

When used with a binary file, GET retrieves enough data from the file to fill *Var*. The *Var* parameter can be a simple (scalar) variable like an [Integer](#) or a dynamic (variable-length) string, an element in an array, or a variable of [User-Defined Type](#).

Arr()

When reading an array from the disk file, GET assigns data from the file into each element in the array, starting at the arrays [LBOUND](#) subscript. GET attempts to read the number of elements specified by *rcds* in the RECORDS option, or the number of elements in the array, whichever is smaller. The actual number of elements read is assigned to the variable *count* specified in the optional TO clause.

With a dynamic [string array](#), it is assumed the file was written in the Classic PowerBASIC and/or VB packed string format using PUT of an entire string array. If a string is shorter than 65535 bytes, a 2-byte length WORD is followed by the string data. Otherwise, a 2-byte value of 65535 is followed by a length [Double-word](#) (DWORD), then finally the string data.

With other array arrays types, the entire data area is read as a single block. In either case, it is presumed the file was created with the complementary PUT Array statement.

[EOF](#) is set just as with other GET statements.

You can use the FILESCAN statement to determine the number of records contained in the file, allowing an array of the appropriate type to be dimensioned before using the GET statement to read the file.

See also

[CSET](#), [CSET\\$](#), [EOF](#), [FIELD](#), [FILESCAN](#), [GET\\$](#), [LINE INPUT#](#), [LSET](#), [OPEN](#), [PRINT#](#), [PUT](#), [PUT\\$](#), [RSET](#), [TYPE](#), [SEEK](#), [WRITE#](#)

Example

```
' Random-access GET example
DIM uName AS STRING * 20
DIM I AS QUAD
DIM F AS LONG
```

```

F = FREEFILE
OPEN "TESTFILE.DTA" FOR RANDOM AS #F LEN = LEN(uName)
WHILE uName <> SPACE$(20)
    PUT #F,, uName
    uName = GetInput()
WEND

IF SEEK(F) > 0 THEN
    ShowText "The file contains these names:"
    FOR ix = 1 TO SEEK(F)
        GET #F, ix, uName
        ShowText uName + NL
    NEXT
ELSE
    ShowText "The file is empty"
END IF
CLOSE #F

' Binary GET Array example
OPEN "Data file to read.dat" FOR BINARY AS #1
FILESCAN #1, RECORDS TO count&
DIM TheData$(1 TO count&)
GET #1, 1, TheData$() TO y&
CLOSE #1

```

GET\$ statement

[Top](#) [Previous](#) [Next](#)

Purpose	Read a string from a file opened in binary mode.
Syntax	<code>GET\$ [#] <i>filenum</i>&, <i>Count</i>&, <i>sVar</i>\$</code>
Remarks	The GET\$ statement has the following parts:
<i>filenum</i> &	Is the file number under which the file was opened.
<i>Count</i> &	Specifies how many bytes to read.
<i>sVar</i> \$	The string variable to read the data into.
Remarks	<p>GET\$ reads <i>Count</i>& characters from file number <i>filenum</i>&, and assigns them to <i>sVar</i>\$. File <i>filenum</i>& must have been opened in binary mode. Characters are read starting at the current file pointer position, which can be set with the SEEK statement. When the file is first opened, the pointer is at the beginning of the file (unless the BASE clause is used in the corresponding OPEN statement, position 1 is used by default). After GET\$, the file pointer position will have been advanced by <i>Count</i>& bytes. GET\$, PUT\$, and SEEK provide a low-level alternative to sequential and random-access file-processing techniques, allowing you to deal with files on a byte-by-byte basis.</p>
See also	EOF , GET , INPUT# , LINE INPUT# , LOF , OPEN , PRINT# , PUT , PUT\$, SEEK , WRITE#
Example	<pre>' Open binary file, write the alphabet A-Z to it OPEN "SEEK.DTA" FOR BINARY AS #1 LEN = 1 BASE = 0 FOR I& = 65 TO 90 PUT\$ #1, CHR\$(I&) NEXT I& ' Now read five characters at a time from the file, ' starting at different pointer positions FOR I& = 0 TO 20 STEP 5 SEEK #1, I& GET\$ #1, 5, TempString\$ x\$ = "Starting at position" + STR\$(I&) + \$SPC + \$DQ + TempString\$ + \$DQ NEXT I& CLOSE #1</pre>
Result	<pre>Starting at position 0 "ABCDE" Starting at position 5 "FGHIJ" Starting at position 10 "KLMNO" Starting at position 15 "PQRST" Starting at position 20 "UVWXY"</pre>

Purpose Return the file-system attribute(s) of a disk file or directory.

Syntax `x& = GETATTR(filespec$)`

Remarks *filespec*\$ specifies a filename or directory (optionally, including a drive letter and directory path). The attribute code returned in x& is a standard operating system attribute code, or a combination of several codes ORed together:

Attribute	Description	Equate
0	Normal*	%NORMAL
1	Read-only	%READONLY
2	Hidden	%HIDDEN
4	System	%SYSTEM
8	Volume Label	%VLABEL
16	Directory	%SUBDIR
32	Archived	%ARCHIVE
128	Normal*	(synonym of %NORMAL)

*** Some operating systems may return either 0 or 128 for normal files.**

If GETATTR returns an attribute of 0 (or 128), *filespec*\$ is a regular file: not read-only, not hidden, not system, and not archived.

Additional file attributes may be supported on some file systems. See the %FILE_ATTRIBUTE equates in your Win32API.inc file for a full list.

If you want to test for a single attribute, use the bitwise [AND](#) operator to strip out any other attributes that might be set. See the example below.

GETATTR can also be used to verify the existence of a file or directory, taking advantage of the fact that [ERR](#) will be set if the file/directory does not exist. See the example below.

Restrictions If *filespec*\$ cannot be found, a run-time [Error 53](#) ("File not found") occurs. You cannot obtain the attributes of the root directory (i.e., "C:\"). Windows prevents this particular operation, triggering an Error 53.

See also [DIR\\$](#), [FILEATTR](#), [ISFILE](#), [PATHSCAN\\$](#), [SETATTR](#)

Example

```
attr& = GETATTR("C:\CONFIG.SYS")
IF (attr& AND 32&) = 32& THEN
  x$ = "CONFIG.SYS has been modified"
ELSE
  x$ = "CONFIG.SYS hasn't been modified"
END IF
```

Purpose	Declare global (shared) variables between Subs , Functions , Methods , and Properties .
Syntax	<pre>GLOBAL variable[()] [AS type] [, variable[()]] [, ...] GLOBAL variable[()] [, variable[()]] [, ...] AS type</pre>
Remarks	<p>GLOBAL declares the specified variable(s) as global to the entire program. This gives a procedure access to variable(s), without having to pass them as parameters. To declare an array as a global variable, use an empty set of parentheses in the variable list:</p> <pre>GLOBAL MyArray%() GLOBAL StringArray() AS STRING</pre> <p>You must then use the DIM or REDIM statements to dimension the array inside a procedure. A good place to do this is inside your WINMAIN or PBMAIN function.</p> <p>If an array is defined as GLOBAL outside a procedure, you should include the GLOBAL keyword in the DIM statement for clarity, and compatibility with future versions of PowerBASIC:</p> <pre>GLOBAL a() AS STRING FUNCTION PBMAIN DIM a(1 TO 500) AS GLOBAL STRING [statements] END FUNCTION</pre> <p>The GLOBAL statement may accept a list of variables, all of which are defined by the type descriptor keywords which follow them. For example:</p> <pre>GLOBAL aaa, bbb, ccc AS INTEGER GLOBAL vptr, aptr() AS LONG PTR</pre>
Restrictions	GLOBAL variables are not shared between programs and DLLs or between multiple instances of the same DLL. That is, a GLOBAL variable is only global within its own module. The simplest way to expose a variable to a DLL is to pass the variable BYREF (by reference) to the target DLL. DEFtype has no effect on variables defined by a GLOBAL statement.
See also	DIM , INSTANCE , LOCAL , STATIC , THREADED
Example	<pre>#COMPILE EXE GLOBAL Caption AS ASCIIZ * 255 FUNCTION PBMAIN() AS LONG DIM Msg AS ASCIIZ * 255 CALL SetVars IF Caption = "GLOBAL test" then Msg = "Success!" END FUNCTION SUB SetVars() Caption = "GLOBAL test" END SUB</pre>

Purpose Allocate or release a block of global memory

Syntax

```
GLOBALMEM ALLOC count TO vHndl  
GLOBALMEM FREE   mHndl TO vHndl  
GLOBALMEM LOCK    mHndl TO vPtr  
GLOBALMEM SIZE    mHndl TO vSize  
GLOBALMEM UNLOCK mHndl TO vLocked
```

Remarks GLOBALMEM allocates a block of global system memory of the requested size. This is always allocated as "moveable" memory, so it can be used with any facilities of Windows. It is the programmer's responsibility to release the allocated memory block when it's no longer needed.

There are five general forms of the GLOBALMEM statement:

GLOBALMEM ALLOC A moveable memory block of the size in [bytes](#) specified by *count* is allocated. A unique handle is assigned to this memory object (for later identification). This handle is assigned to the [LONG](#) or [DWORD](#) variable specified by *vHndl*. If the requested allocation fails for any reason, the value zero (0) is assigned to *vHndl* instead.

GLOBALMEM FREE A memory block is de-allocated and released for re-use. The *mHndl* parameter is a [variable](#) or expression which evaluates to the handle returned by GLOBALMEM ALLOC when the memory block was created. If the de-allocation operation was successful, the result variable *vHndl* is set to zero (0) to indicate that the original handle is no longer valid. If the operation fails for any reason, the value of the *mHndl* parameter is assigned to *vHndl*. It may be convenient to use the same variable for both the parameter and the result, as it will then be automatically cleared to zero when the memory block is released.

GLOBALMEM LOCK The moveable memory block referenced by *mHndl* is locked at a specific memory location. A [pointer](#) to this location is assigned to the variable specified by *vPtr*. You may only read or write the memory block while it is locked, and you use the current pointer to its location.

GLOBALMEM SIZE The size of the memory block referenced by *mHndl* is retrieved and assigned to the LONG or DWORD variable specified by *vSize*. The *mHndl* parameter is the handle originally returned by GLOBALMEM ALLOC.

GLOBALMEM The moveable memory block referenced by *mHndl* is

UNLOCK

unlocked, and the previous memory pointer is invalidated. If the memory block remains locked (perhaps because it had been locked more than once), the value [TRUE](#) (non-zero) is assigned to the result variable *vLocked*. If the memory block is now unlocked, or the parameter *mHndl* was invalid, the value [FALSE](#) (0) is assigned to *vLocked* instead.

GOSUB/GOSUB DWORD statements

[Top](#) [Previous](#) [Next](#)

Purpose	Invoke a subroutine.
Syntax	<pre>GOSUB {<i>label</i> <i>linenumber</i>} GOSUB DWORD <i>dwpointer</i></pre>
Remarks	<p>GOSUB causes execution to branch to the statement prefaced by label or linenumber, after first saving its current location on the stack. The <i>label</i> or <i>linenumber</i> must be local to the Sub, Function, Method, or Property where the GOSUB statement is located. Executing a RETURN statement returns control to the instruction immediately following the GOSUB.</p> <p>When using GOSUB, be sure that each subroutine returns to its caller gracefully through a RETURN statement. Run-away (recursive) GOSUBs that loop upon themselves will eat up large chunks of stack space, reducing available memory.</p> <p>All labels and line numbers are private. You cannot GOSUB to a label outside of the current procedure.</p> <p>For time critical or high-performance code, using a GOSUB to perform a repetitive task is almost always faster then performing a call to a procedure, since there is no overhead in setting up a stack frame for a GOSUB.</p>
DWORD	GOSUB DWORD causes execution to branch to address stored in <i>dwpointer</i> , after first saving its current location on the stack. <i>dwpointer</i> must be a Double word , Long integer , or pointer variable that contains the address of a label that is in the same procedure as the GOSUB DWORD statement. Executing a RETURN statement returns control to the instruction immediately following the GOSUB DWORD statement.
See also	#STACK , FUNCTION , METHOD , ON GOSUB , PROPERTY , SUB , RETURN
Example	<pre>FUNCTION DoCalc! (Radius!) pi# = ATN(1) * 4 ' calculate value of PI GOSUB CalcArea ' jump to subroutine Radius! EXIT FUNCTION CalcArea: FUNCTION = pi# * Radius! ^2 ' calculate area RETURN ' return from subroutine END FUNCTION</pre>

Purpose	Transfer program execution to the statement identified by a label or line number .
Syntax	<pre>GOTO {<i>label</i> <i>linenumber</i>} GOTO DWORD <i>dwpointer</i></pre>
Remarks	<p>GOTO causes program flow to jump unconditionally to the code identified by label or linenumber. The <i>label</i> or <i>linenumber</i> must be local to the Sub, Function, Method, or Property where the GOTO statement is located. GOTO differs from GOSUB and other similar control statements, in that after execution of a GOTO, the program retains no memory of where it was before it executed the jump.</p> <p>Labels and line numbers are private. You cannot GOTO a label outside of the current procedure.</p>
DWORD	GOTO DWORD causes execution to jump unconditionally to address stored in <i>dwpointer</i> . <i>dwpointer</i> must be a Double word , Long integer , or Pointer variable that contains the address of a label which is local to the procedure where the GOTO DWORD statement is located.
See also	CALL , CALL DWORD , DO/LOOP , EXIT , FOR/NEXT , FUNCTION , GOSUB , IF block , METHOD , PROPERTY , RETURN , SELECT , SUB , WHILE/WEND
Example	<pre>FUNCTION test() AS LONG RESET X Start: ' define a label INCR X ' increment X IF X < 10 THEN DoBeep EXIT FUNCTION .[<i>statements</i>] DoBeep: BEEP GOTO Start ' jump back to Start END FUNCTION</pre> <p>One method of obtaining the same results without use of GOTO is:</p> <pre>FUNCTION test() AS LONG FOR X = 1 TO 9 GOSUB DoBeep NEXT X EXIT FUNCTION [<i>statements</i>] DoBeep: BEEP RETURN END FUNCTION</pre>

Purpose	Draw an arc in the selected graphic target .
Syntax	<code>GRAPHIC ARC (x1!, y1!) - (x2!, y2!), arcStart!, arcEnd! [, rgbColor&]</code>
Remarks	<p>An arc is a section of a circle or an ellipse. To specify a particular arc, you would first define the full circle or ellipse of which it is a part, and then specify the points on the ellipse where the arc starts and stops.</p> <p>The full circle or ellipse is defined by its bounding rectangle, which is defined as the smallest rectangle which can be drawn around the circle or ellipse. For example, if the circle is centered at position (400,400), with a radius of 100 pixels, the upper left corner (x1!, y1!) of the bounding rectangle is (300,300), and the lower right corner (x2!, y2!) is (500,500). The start point and end point of the arc are specified by their angle, which must be given in radians. A complete circle or ellipse is 2*pi radians. On a 12-hour clock-face, the values 0 and 2*pi both refer to the position of 3 o'clock, while the value 1*pi refers to the position of 9 o'clock. Other positions are specified by a radian value relative to these. In Classic PowerBASIC, arcs are always drawn counter-clockwise from the starting point to the ending point.</p> <p>Prior to any graphical operations, the graphic target must first be selected with GRAPHIC ATTACH. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH. If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE. Because of the nature of an arc, GRAPHIC ARC neither uses, nor updates, GRAPHIC POS (last point referenced).</p>
<i>x1!, y1!</i>	The upper left corner of the bounding rectangle of the full circle or ellipse.
<i>x2!, y2!</i>	The lower right corner of the bounding rectangle of the full circle or ellipse.
<i>ArcStart!</i>	The starting angle of the arc, in radians, from 0 to 2*pi.
<i>ArcEnd!</i>	<p>The ending angle of the arc, in radians, from 0 to 2*pi radians. Note that arcs are always drawn counter-clockwise from <i>arcStart!</i> to <i>arcEnd!</i>.</p> <p>Compared with a 12-hour clock-face, 0 or 2*pi radians is at 3 o'clock, and 1*pi radians is at 9 o'clock.</p>
<i>rgbColor&</i>	Optional RGB color for the arc. If omitted (or -1), the current foreground color for the graphic window is used.
See also	Built In RGB Color Equates , GRAPHIC ATTACH , GRAPHIC COLOR , GRAPHIC ELLIPSE , GRAPHIC PIE , GRAPHIC STYLE , GRAPHIC WIDTH

Example

```
' Draw two arcs that combine into a circle.
' The upper half uses the default foreground color.
' The lower half is drawn in red.
LOCAL Pi AS DOUBLE
Pi = 4 * ATN(1)                                ' Calculate Pi
GRAPHIC ARC (5, 5) - (105, 105), 0, Pi         ' Upper half
GRAPHIC ARC (5, 5) - (105, 105), Pi, 0, %RED   ' Lower half
```

Purpose	Select the graphic target (window , control , or bitmap) on which future drawing operations will take place.
Syntax	<code>GRAPHIC ATTACH <i>hWin</i>, <i>id</i> [, REDRAW]</code>
Remarks	<p>This statement chooses a graphic target. All further graphic operations will be directed to this target until another GRAPHIC ATTACH or GRAPHIC DETACH statement is executed, or the graphic target is deleted. All Classic PowerBASIC graphical displays are persistent -- they will be automatically redrawn even if minimized or temporarily covered by another window.</p> <p>By default, all graphic operations are displayed immediately upon execution of a graphic statement. In many cases, this is a good choice, because the display is always up-to-date. However, as the complexity of graphic operations increases, this continuous update process does not afford the best performance. It is usually better to use the REDRAW option described below, as it will generally provide a dramatic improvement in overall performance.</p>
<i>hWin</i>	Handle of the GRAPHIC WINDOW , DIALOG , or BITMAP to be used with graphic statements.
<i>id</i>	The control id, if the target is a GRAPHIC CONTROL , or zero if the target is a GRAPHIC WINDOW or GRAPHIC BITMAP.
REDRAW	This option can provide a dramatic improvement in the execution speed of graphic statements, as it eliminates repetitive updates to the display. If this option is included, all drawing statements are buffered until a GRAPHIC REDRAW statement is executed, or the operating system chooses to update the target window. Without REDRAW, all graphical statements (Line, Box, Print, etc.) are performed immediately. However, in most cases, it's better to defer the display until a number of statements have been performed.

Example

```
' Draw a blue gradient fill.
' Each line is displayed as it's drawn.
GRAPHIC ATTACH hDlg, %IDC_GRAPHIC1
FOR y& = 0 TO 255
  GRAPHIC LINE (0, y&) - (255, y&), RGB(0, 0, y&)
NEXT

' Draw a buffered, blue gradient fill.
' Nothing is displayed before GRAPHIC REDRAW,
' this enhancing performance dramatically.
GRAPHIC ATTACH hDlg, %IDC_GRAPHIC2, REDRAW
FOR y& = 0 TO 255
  GRAPHIC LINE (0, y&) - (255, y&), RGB(0, 0, y&)
```

NEXT
GRAPHIC REDRAW

See also

[CONTROL ADD GRAPHIC](#), [GRAPHIC BITMAP LOAD](#), [GRAPHIC BITMAP NEW](#), [GRAPHIC DETACH](#), [GRAPHIC WINDOW](#)

GRAPHIC BITMAP END statement

[Top](#) [Previous](#) [Next](#)

Purpose	Close the selected graphic bitmap.
Syntax	<code>GRAPHIC BITMAP END</code>
Remarks	You must close every memory bitmap (that was created with GRAPHIC BITMAP LOAD or GRAPHIC BITMAP NEW) when you are finished using them for graphical operations. To close a bitmap, select it with the GRAPHIC ATTACH statement, then execute GRAPHIC BITMAP END .
See also	GRAPHIC ATTACH , GRAPHIC BITMAP LOAD , GRAPHIC BITMAP NEW

GRAPHIC BITMAP LOAD statement

[Top](#) [Previous](#)
[Next](#)

IMPROVED

Purpose	Create a memory bitmap and load an image into it.				
Syntax	<code>GRAPHIC BITMAP LOAD <i>BmpName\$</i>, <i>nWidth&</i>, <i>nHeight&</i> [,<i>stretch&</i>] TO <i>hBmp???</i></code>				
<i>BmpName\$</i>	The name of the bitmap image to load.				
<i>nWidth&</i>	The width of the bitmap, in pixels.				
<i>nHeight&</i>	The height of the bitmap, in pixels.				
<i>stretch&</i>	Stretch mode if the bitmap is to be resized.				
<i>hBmp???</i>	The bitmap handle.				
Remarks	<p>GRAPHIC BITMAP LOAD creates a new memory bitmap, loading a bitmap image from a resource or a disk file. This bitmap works just like a GRAPHIC WINDOW, except that it is not visible. The parameter <i>BmpName\$</i> specifies the name of the image to be loaded. If <i>BmpName\$</i> contains a period, it is presumed to be the name of a disk file. Otherwise, an attempt is made to load it from a resource -- if not found, it is then presumed to be a disk file.</p> <p>The parameters <i>nWidth&</i> and <i>nHeight&</i> specify the width and height of the bitmap, in pixels. If either of the size parameters are zero (0), the bitmap is loaded at its natural size. If either of the size parameters is different from the natural size, the bitmap is stretched or shrunk to the requested size.</p> <p>If the bitmap creation is successful, the bitmap handle is assigned to the variable <i>hbmp???</i>. If not successful, <i>hbmp???</i> is set to zero. When you are finished using this memory bitmap, you must delete it with GRAPHIC BITMAP END.</p> <p>If the <i>stretch&</i> parameter is included, it is one of the values in the following table. If not included, or it is the value zero (0), the stretch mode is unchanged. An appropriate choice of stretch mode can substantially enhance the quality of bitmaps which are changed in size. The stretch mode equates are predefined in Classic PowerBASIC.</p> <p>The 4 stretch modes are:</p> <table><tr><td>%BLACKONWHITE</td><td>This is the default Windows stretch mode, and is most appropriate for monochrome bitmaps, or those with blocks of color. Performs a boolean OR of eliminated and existing pixels. It preserves black pixels at the expense of white pixels.</td></tr><tr><td>%WHITEONBLACK</td><td>Performs a boolean OR of eliminated and existing</td></tr></table>	%BLACKONWHITE	This is the default Windows stretch mode, and is most appropriate for monochrome bitmaps, or those with blocks of color. Performs a boolean OR of eliminated and existing pixels. It preserves black pixels at the expense of white pixels.	%WHITEONBLACK	Performs a boolean OR of eliminated and existing
%BLACKONWHITE	This is the default Windows stretch mode, and is most appropriate for monochrome bitmaps, or those with blocks of color. Performs a boolean OR of eliminated and existing pixels. It preserves black pixels at the expense of white pixels.				
%WHITEONBLACK	Performs a boolean OR of eliminated and existing				

pixels. It preserves white pixels at the expense of black pixels.

%COLORONCOLOR Deletes eliminated lines of pixels without trying to preserve their information.

%HALFTONE This provides the highest quality for complex color bitmaps. The average color of the destination pixel block is kept approximately the same as the source pixel block.

The following code will retrieve the natural size of an image in a bitmap file, in pixels:

```
nFile& = FREEFILE  
OPEN "myimage.bmp" FOR BINARY AS nFile&  
GET #nFile&, 19, nWidth&  
GET #nFile&, 23, nHeight&  
CLOSE nFile&
```

See also

[CONTROL ADD GRAPHIC](#), [GRAPHIC ATTACH](#), [GRAPHIC BITMAP END](#),
[GRAPHIC BITMAP NEW](#), [GRAPHIC IMAGELIST](#), [GRAPHIC WINDOW](#)

GRAPHIC BITMAP NEW statement

[Top](#) [Previous](#) [Next](#)

Purpose Create a new memory bitmap.

Syntax `GRAPHIC BITMAP NEW nWidth&, nHeight& TO hBmp???`

Remarks GRAPHIC BITMAP NEW creates a new memory bitmap, which may be manipulated and drawn just as if it were a [GRAPHIC WINDOW](#), except that it is not visible. The parameters *nWidth*& and *nHeight*& specify the width and height of the bitmap, in pixels. If the bitmap creation is successful, the bitmap handle is assigned to the variable *hBmp*???. If not successful, *hBmp*??? is set to zero. When you are finished using this memory bitmap, you must delete it with [GRAPHIC BITMAP END](#).

See also [GRAPHIC ATTACH](#), [GRAPHIC BITMAP END](#), [GRAPHIC BITMAP LOAD](#), [GRAPHIC IMAGELIST](#)

Purpose	Draw a box with square or rounded corners in the selected graphic target .
Syntax	<code>GRAPHIC BOX (x1!, y1!) - (x2!, y2!) [, [corner&] [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]]]</code>
Remarks	<p>The graphic target must first be selected with GRAPHIC ATTACH. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH. If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE. Because of the nature of an arc, GRAPHIC ARC neither uses, nor updates, the last point referenced (POS).</p>
x1!, y1!	The upper left corner of the box.
x2!, y2!	The lower right corner of the box.
corner&	The percentage of roundness of the corners, in the range of 0 to 100. A value of zero creates square corners, while 100 creates a circle/oval. A value of 20 being most common for a pleasant, rounded appearance. If <i>corner&</i> is omitted, the default is 0, which creates a rectangle with square corners.
rgbColor&	Optional RGB color of the box edge. If omitted (or -1), the edge color defaults to the current foreground color for the selected graphic target.
fillcolor&	Optional RGB color of the box interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the box is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.
fillstyle&	<p>Optional fill style (pattern) to be used. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>fillcolor&</i>, while the background is specified by the default background color. The optional <i>fillstyle&</i> may be:</p> <ul style="list-style-type: none">0 Solid (default)1 Horizontal Lines2 Vertical Lines3 Upward Diagonal Lines4 Downward Diagonal Lines5 Crossed Lines6 Diagonal Crossed

Lines

See also

[Built In RGB Color Equates](#), [GRAPHIC ATTACH](#), [GRAPHIC COLOR](#), [GRAPHIC LINE](#), [GRAPHIC STYLE](#), [GRAPHIC WIDTH](#)

Example

```
' Draw rectangle with square corners and default colors.  
GRAPHIC BOX (10, 10) - (100, 80)
```

```
' Draw a blue rectangle with 20% rounded corners,  
' filled with a light-gray, diagonal cross pattern  
GRAPHIC BOX (15, 15) - (95, 75), 20, %BLUE, RGB(191,191,191), 6
```

Purpose	Retrieve the character size for the current font in the selected graphic target .
Syntax	<code>GRAPHIC CHR SIZE TO <i>ncWidth!</i>, <i>ncHeight!</i></code>
Remarks	<p>The graphic target must first be selected with GRAPHIC ATTACH. The character size is specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if they have been defined with a GRAPHIC SCALE statement).</p> <p>If the font is a fixed-width font, like Courier New or Lucida Console, the sizes returned are as exact as possible, given the rounding errors possible when converting from pixels to other coordinates. If the font is proportional, like Arial or Times New Roman, the width will be the average for the entire font.</p>
See also	GRAPHIC ATTACH , GRAPHIC SET FONT , GRAPHIC PRINT , GRAPHIC TEXT SIZE

Purpose	Clear the entire selected graphic target , optionally using a specified color and fill style.
Syntax	GRAPHIC CLEAR [<i>rgbColor</i> & [, <i>fillstyle</i> &]]
Remarks	The graphic target must first be selected with GRAPHIC ATTACH . The last point referenced (POS) is set to the upper left corner of the graphic window (0,0).
<i>rgbColor</i> &	Optional RGB value representing the fill color. If <i>rgbColor</i> & is omitted (or -1), the graphic target is cleared to the default background color for the selected graphic target.
<i>fillstyle</i> &	Optional fill style (pattern) to be used. If <i>fillstyle</i> & is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>rgbColor</i> &, while the background is specified by the default background color for the selected graphic window. The optional <i>fillstyle</i> & may be: <div><div>0 Solid (default)</div><div>1 Horizontal Lines</div><div>2 Vertical Lines</div><div>3 Upward Diagonal Lines</div><div>4 Downward Diagonal Lines</div><div>5 Crossed Lines</div><div>6 Diagonal Crossed Lines</div></div>
See also	Built In RGB Color Equates , GRAPHIC ATTACH

Purpose	Set the foreground color and optionally the background color for various graphic statements.
Syntax	<code>GRAPHIC COLOR <i>foreground</i>& [, <i>background</i>&]</code>
Remarks	The graphic target must first be selected with GRAPHIC ATTACH . If either parameter is -1, the default foreground/background color is used. If the background parameter is -2, the background is not painted, allowing the content behind to become visible. Otherwise, the specified RGB color is used.
See also	Built In RGB Color Equates , GRAPHIC ATTACH , GRAPHIC PRINT , GRAPHIC SET FONT
Example	<pre>' Set red foreground and blue background color. GRAPHIC COLOR %RED, RGB(0,0,191)</pre>

Purpose	Copy a bitmap to the selected graphic target .										
Syntax	<pre>GRAPHIC COPY <i>hbmpSource</i>???, <i>id</i>& [, <i>style</i>&] GRAPHIC COPY <i>hbmpSource</i>???, <i>id</i>& TO (<i>x</i>!, <i>y</i>!) [, <i>style</i>&] GRAPHIC COPY <i>hbmpSource</i>???, <i>id</i>&, (<i>x1</i>!, <i>y1</i>!)-(<i>x2</i>!, <i>y2</i>!) TO (<i>x</i>!, <i>y</i>!) [, <i>style</i>%]</pre>										
Remarks	<p>You can copy a complete bitmap, or a portion of it, to the selected graphic target. The expression <i>hbmpSource</i>??? specifies the handle of the source bitmap, GRAPHIC WINDOW, or dialog containing a GRAPHIC CONTROL. The expression <i>id</i>& is the unique control identifier in the range 1 to 65535, as assigned with the CONTROL ADD GRAPHIC statement. <i>id</i>& must be 0 for a GRAPHIC WINDOW or a GRAPHIC BITMAP. The destination of the copy operation is the window selected by GRAPHIC ATTACH. You must take care that your parameters are valid for the specified bitmap, or the results of the operation are undefined.</p> <p>The first form of the GRAPHIC COPY statement copies the complete bitmap, positioning it at (0,0), which is the upper left corner of the destination.</p> <p>The second form of GRAPHIC COPY also copies the complete bitmap, but positions it at the point specified by the parameter (<i>x</i>!, <i>y</i>!).</p> <p>The third form copies a portion of the bitmap, specified by <i>x1</i>,<i>y1</i> as the upper left corner and <i>x2</i>,<i>y2</i> as the lower right corner. It is positioned at the point specified by the parameter (<i>x</i>,<i>y</i>). You must use care that your parameters are valid for the specified bitmaps, or results of the operation are undefined.</p> <p>If the style parameter is included, it is one of the values in the following table. If not included, a default of %mix_CopySrc is presumed. There are 8 mix modes available to use for mixing drawing colors with the colors which already exist at the at the drawing location The mix mode equates are predefined in Classic PowerBASIC.</p> <table><tr><td>%mix_NotMergeSrc</td><td>Pixel is the inverse of the MergeSrc color.</td></tr><tr><td>%mix_NotCopySrc</td><td>Pixel is the inverse of the source color.</td></tr><tr><td>%mix_MaskSrcNot</td><td>Pixel is a combination of the colors common to both the source and the inverse of the pixel.</td></tr><tr><td>%mix_XorSrc</td><td>Pixel is a combination of the colors in the source and the pixel, but not in both.</td></tr><tr><td>%mix_MaskSrc</td><td>Pixel is a combination of the colors common to both the source and the pixel.</td></tr></table>	%mix_NotMergeSrc	Pixel is the inverse of the MergeSrc color.	%mix_NotCopySrc	Pixel is the inverse of the source color.	%mix_MaskSrcNot	Pixel is a combination of the colors common to both the source and the inverse of the pixel.	%mix_XorSrc	Pixel is a combination of the colors in the source and the pixel, but not in both.	%mix_MaskSrc	Pixel is a combination of the colors common to both the source and the pixel.
%mix_NotMergeSrc	Pixel is the inverse of the MergeSrc color.										
%mix_NotCopySrc	Pixel is the inverse of the source color.										
%mix_MaskSrcNot	Pixel is a combination of the colors common to both the source and the inverse of the pixel.										
%mix_XorSrc	Pixel is a combination of the colors in the source and the pixel, but not in both.										
%mix_MaskSrc	Pixel is a combination of the colors common to both the source and the pixel.										

<code>%mix_MergeNotSrc</code>	Pixel is a combination of the pixel color and the and the inverse of the source color.
<code>%mix_CopySrc</code>	Pixel is the source color (default).
<code>%mix_MergeSrc</code>	Pixel is a combination of the source color and the pixel color.

See also [GRAPHIC ATTACH](#), [GRAPHIC RENDER](#), [GRAPHIC STRETCH](#)

Purpose	Detaches a graphic target (Window , Control , or bitmap) which may be currently attached to the process.
Syntax	GRAPHIC DETACH
Remarks	Though detached from the graphic command stream, the graphic target (Window or Bitmap) is not deleted, nor is it altered in any way. Until another graphic target is attached, any graphic statements executed are ignored. If no graphic is attached, this statement performs no operation.
See also	GRAPHIC ATTACH , GRAPHIC BITMAP LOAD , GRAPHIC BITMAP NEW , GRAPHIC WINDOW

Purpose	Draw an ellipse or a circle in the selected graphic target .
Syntax	<pre>GRAPHIC ELLIPSE (x1!, y1!) - (x2!, y2!) [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]</pre>
Remarks	<p>The graphic target must first be selected with GRAPHIC ATTACH. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH. If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE. Because of the nature of an ellipse, which has no obvious beginning or end, GRAPHIC ELLIPSE neither uses, nor updates, the last point referenced (POS).</p>
<i>x1!, y1!</i>	The upper left corner of the bounding rectangle.
<i>x2!, y2!</i>	The lower right corner of the bounding rectangle.
<i>rgbColor&</i>	Optional RGB color of the ellipse edge. If omitted (or -1), the edge color defaults to the current foreground color for the selected graphic window.
<i>fillcolor&</i>	Optional RGB color of the ellipse interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the ellipse is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.
<i>fillstyle&</i>	Optional fill style (pattern) to be used. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>fillcolor&</i> , while the background is specified by the default background color for the selected graphic window. The optional <i>fillstyle&</i> may be:
	<ul style="list-style-type: none">0 Solid (default)1 Horizontal Lines2 Vertical Lines3 Upward Diagonal Lines4 Downward Diagonal Lines5 Crossed Lines6 Diagonal Crossed Lines
See also	Built In RGB Color Equates , GRAPHIC ARC , GRAPHIC ATTACH , GRAPHIC COLOR , GRAPHIC LINE , GRAPHIC PIE , GRAPHIC STYLE , GRAPHIC WIDTH

Example

```
' Draw a circle, using default colors.  
GRAPHIC ELLIPSE (10, 10) - (100, 100)
```

```
' Draw a blue ellipse filled with a light-gray,  
' diagonal cross pattern.  
GRAPHIC ELLIPSE (15, 25) - (95, 50), %BLUE, RGB(191,191,191), 6
```

Purpose	<p>Select a font for the GRAPHIC PRINT, GRAPHIC INPUT, and GRAPHIC LINE INPUT statements.</p> <p>GRAPHIC FONT has been superceded by the GRAPHIC SET FONT statement, although GRAPHIC FONT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>										
Syntax	<p><code>GRAPHIC FONT <i>fontname</i>\$ [,<i>points</i>& , <i>style</i>&]</code></p> <p><i>fontname</i>\$ The name of the font.</p> <p><i>points</i>& The size of the font, in points.</p> <p><i>style</i>& Font style attribute. Any of the following values can be combined or used alone:</p> <table><tbody><tr><td>0</td><td>normal</td></tr><tr><td>1</td><td>bold</td></tr><tr><td>2</td><td>italic</td></tr><tr><td>4</td><td>underline</td></tr><tr><td>8</td><td>strikeout</td></tr></tbody></table> <p>For example, a <i>style</i>& of 3 specifies a combination of both bold and italic attributes.</p>	0	normal	1	bold	2	italic	4	underline	8	strikeout
0	normal										
1	bold										
2	italic										
4	underline										
8	strikeout										
Remarks	<p>If the requested font is not available on the computer, Windows will search for a substitute font, which is similar to the attributes specified (CharSet, Font Family, etc.).</p> <p>You may use the value zero (0) for the numeric parameters to designate that the compiler should use the default for that item. If parameter(s) are missing, the compiler substitutes the default value for all remaining parameters.</p> <p>Prior to any graphical operations, the graphic target must first be selected with GRAPHIC ATTACH. If no GRAPHIC FONT statement is executed, the default font is MS Sans Serif, 8 point, with no style attributes.</p>										
See Also	<p>FONT NEW, GRAPHIC ATTACH, GRAPHIC CHR SIZE, GRAPHIC INPUT, GRAPHIC LINE INPUT, GRAPHIC PRINT, GRAPHIC SET FONT, GRAPHIC TEXT SIZE</p>										
Example	<pre>' Set the font for the selected graphic window to ' Times New Roman, 18 points, bold + italic + underline = (1+2+4) . GRAPHIC FONT "Times New Roman", 18, 7</pre>										

Purpose Retrieve a copy of a bitmap , storing it as a device-independent bitmap in a [dynamic string](#) variable.

Syntax GRAPHIC GET BITS TO *bitvar\$*

Remarks This statement retrieves a copy of the entire bitmap for the selected [graphic target](#), assigning it to the dynamic string [variable](#) specified by *bitvar\$*. This allows you to make many modifications to the bitmap very quickly, particularly operations which may not be directly supported by Classic PowerBASIC. For example, you might change all red pixels in a bitmap to blue. Once your operations are complete, the bitmap is replaced using [GRAPHIC SET BITS](#).

The *bitvar\$* string will contain a series of four-byte values, each of which represents a [long integer](#). You can convert the four-byte string sections to numeric values with the [CVL](#) function, and convert a numeric value to a four-byte string with [MKL\\$](#). The first four-byte value specifies the width of the bitmap, in pixels, and the second specifies the height. Following that will be one four-byte value for each pixel in the bitmap, which represents the [color](#) of that pixel. So, a 20 by 20 bitmap would have 400 pixels and require 1600 bytes (400 * 4), plus 4 bytes for the width and 4 bytes for the height, or a total of 1608 bytes.

The first four-byte pixel value in the string represents the top-left corner of the image, the second represents the second pixel of the first row, and so on. After the last pixel of the first row will be the first pixel of the second row, etc.

If execution speed is most important, it's likely that the string can be manipulated most efficiently with pointer variables.

Some Windows API functions, namely those which reference Device-Independent Bitmaps (DIB), require that colors be specified in the reverse of normal [RGB](#) sequence (Blue-Green-Red instead of Red-Green-Blue). To maximize performance, GRAPHIC GET BITS uses BGR format as well. You can use the [BGR\(\)](#) function to translate an RGB value to its BGR equivalent.

See also [Built In RGB Color Equates](#), [BGR](#), [CVL](#), [GRAPHIC SET BITS](#), [MKL\\$](#), [RGB](#)

Example

```
' Change all red pixels to blue
LOCAL PixelPtr AS LONG PTR
GRAPHIC GET BITS TO bmp$
xsize& = CVL(bmp$,1)
ysize& = CVL(bmp$,5)
PixelPtr = STRPTR(bmp$) + 8
```



```
FOR i& = 1 TO xsize& * ysize&
  IF @PixelPtr = BGR(%red) THEN @PixelPtr = BGR(%blue)
  INCR PixelPtr
NEXT
GRAPHIC SET BITS bmp$
```

Purpose	Retrieve the client size of the selected graphic target .
Syntax	<code>GRAPHIC GET CLIENT TO <i>ncWidth!</i>, <i>ncHeight!</i></code>
Remarks	GRAPHIC GET CLIENT allows you to retrieve the size of the client area (displayable area) of the selected graphic window or bitmap. The size is specified in the same terms (pixels or dialog units) as the parent dialog, or in world coordinates, if those were chosen with GRAPHIC SCALE . If no graphic target has been selected with GRAPHIC ATTACH , the values 0,0 are returned.
See also	GRAPHIC ATTACH

Purpose	Retrieve the handle of the DC (device context) for the selected graphic target .
Syntax	<code>GRAPHIC GET DC TO <i>hDC</i>???</code>
Remarks	The DC handle may be used with various Windows API functions to perform specialized graphic operations in the graphic window. The graphic window must first be selected with GRAPHIC ATTACH . If no graphic window is currently selected, zero is returned.
See also	GRAPHIC ATTACH

GRAPHIC GET LINES statement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the number of lines that can be printed on the graphic target.
Syntax	GRAPHIC GET LINES TO <i>linecount</i> &
Remarks	GRAPHIC GET LINES retrieves the number of lines of text which can be printed on the graphic window or graphic control , given the current selected font.
See also	GRAPHIC ATTACH , GRAPHIC CHR SIZE , GRAPHIC PRINT , GRAPHIC SET FONT , GRAPHIC TEXT SIZE

GRAPHIC GET LOC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the location of the selected graphic target on the screen.
Syntax	<code>GRAPHIC GET LOC TO <i>x&</i>, <i>y&</i></code>
Remarks	This statement allows you to retrieve the location of the selected graphic target. If no graphic object is selected, or it is not a graphic window , 0,0 is returned. The location is specified in specified in the same terms (pixels or dialog units) as the parent dialog, relative to the upper left corner of the screen.
See also	GRAPHIC ATTACH , GRAPHIC GET PPI , GRAPHIC SET LOC

Purpose Retrieve the color mix mode for the selected [graphic target](#).

Syntax `GRAPHIC GET MIX TO mixmode&`

Remarks The graphic target must first be selected with [GRAPHIC ATTACH](#). There are 16 mix modes available to use for mixing the drawing color with the color that already exists at the drawing location.

`%mix_Blackness` Pixel is always 0 (black).

`%mix_NotMergeSrc` Pixel is the inverse of the MergeSrc color.

`%mix_MaskNotSrc` Pixel is a combination of the colors common to both the pixel and the inverse of the source.

`%mix_NotCopySrc` Pixel is the inverse of the pen color.

`%mix_MaskSrcNot` Pixel is a combination of the colors common to both the source and the inverse of the pixel.

`%mix_Not` Pixel is the inverse of the pixel color.

`%mix_XorSrc` Pixel is a combination of the colors in the source and in the pixel, but not in both.

`%mix_NotMaskSrc` Pixel is the inverse of the MaskSrc color.

`%mix_MaskSrc` Pixel is a combination of the colors common to both the source and the pixel.

`%mix_NotXorSrc` Pixel is the inverse of the XorSrc color.

`%mix_Nop` Pixel remains unchanged.

`%mix_MergeNotSrc` Pixel is a combination of the source color and the inverse of the pixel color.

`%mix_CopySrc` Pixel is the source color (default).

`%mix_MergeSrcNot` Pixel is a combination of the source color and the inverse of the pixel color.

`%mix_MergeSrc` Pixel is a combination of the source color and the pixel color.

`%mix_Whiteness` Pixel is always 1 (white).

See also [GRAPHIC ATTACH](#), [GRAPHIC SET MIX](#)

Purpose	Retrieve the color of the pixel at the specified point in the selected graphic target .
Syntax	<code>GRAPHIC GET PIXEL (<i>x!</i>, <i>y!</i>) TO <i>rgbColor&</i></code>
Remarks	The graphic target must first be selected with GRAPHIC ATTACH . The coordinate points <i>x</i> , <i>y</i> are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE).
See also	GRAPHIC ATTACH , GRAPHIC COLOR , GRAPHIC SCALE , GRAPHIC SET PIXEL

GRAPHIC GET POS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the POS (last point referenced) by a graphic statement.
Syntax	<code>GRAPHIC GET POS TO <i>x!</i>, <i>y!</i></code>
Remarks	The graphic target must first be selected with GRAPHIC ATTACH . The coordinate points <i>x</i> , <i>y</i> are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE).
See also	GRAPHIC ATTACH , GRAPHIC SCALE , GRAPHIC SET POS

Purpose	Retrieve the resolution of the display device, in points per inch.
Syntax	<code>GRAPHIC GET PPI TO <i>ncWidth</i>&, <i>ncHeight</i>&</code>
Remarks	<p>The graphic target must first be selected with GRAPHIC ATTACH. The resolution is always specified in pixels, regardless of any GRAPHIC SCALE option. This statement is particularly useful in drawing items such as rulers and graphs to a "representative physical size". There are 25.4 millimeters per inch, so just divide by 25.4 to convert from pixels per inch to pixels per millimeter.</p> <p>"Representative physical size" means that the actual image may be close to a particular physical size, but is subject to factors including Windows default PPI setting, the driver's DPI to PPI ratio and even how the monitor has been adjusted. By using the GRAPHIC GET PPI, results, you can construct a representative graphic image that can be saved and later output at the intended scale by more precise means, for example a higher resolution Windows printer.</p>
See also	GRAPHIC ATTACH , GRAPHIC SCALE

GRAPHIC GET SCALE statement New! [Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the current coordinate limits for the graphic target .
Syntax	<code>GRAPHIC GET SCALE TO <i>x1!</i>, <i>y1!</i>, <i>x2!</i>, <i>y2!</i></code>
Remarks	<p>GRAPHIC SCALE allows you to define your own world coordinate system for subsequent graphic statements. World coordinates may be values, with the only requirement that <i>x1!</i> not equal <i>x2!</i>, and <i>y1!</i> not equal <i>y2!</i>.</p> <p>GRAPHIC GET SCALE retrieves the coordinate limits, which may be either custom world coordinates (if a GRAPHIC SCALE has been executed), or else default pixel coordinates. This allows you to save and restore a previous set of coordinates. This statement will automatically adjust to allow Dialog Unit scale factors to be retrieved.</p>
See also	GRAPHIC SCALE , GRAPHIC SCALE PIXELS

Purpose	Display an image from an IMAGELIST										
Syntax	<code>GRAPHIC IMAGELIST (x!,y!), hLst, index&, overlay&, style&</code>										
Remarks	<p>One of the images stored in an IMAGELIST is displayed on the selected graphic control, bitmap, or window. The parameters x!,y! define the upper left corner of the position of the image. hLst is the handle of the IMAGELIST and index& is the selector of the image to be displayed (1=first, 2=second, etc.). If overlay& is non-zero, it specifies an overlay image to be added to the displayed image from the image list. The parameter style& may be one of the following style bits:</p> <table><tr><td>%ILD_NORMAL</td><td>Draws the image using the background color of the image list. If the background color is the default value %CLR_NONE, the image is drawn transparently.</td></tr><tr><td>%ILD_TRANSPARENT</td><td>Draws the image transparently if there is a mask.</td></tr><tr><td>%ILD_MASK</td><td>Draws the mask.</td></tr><tr><td>%ILD_BLEND25</td><td>If there is a mask, the image is drawn blending 25% with the system highlight color.</td></tr><tr><td>%ILD_BLEND50</td><td>If there is a mask, the image is drawn blending 50% with the system highlight color.</td></tr></table>	%ILD_NORMAL	Draws the image using the background color of the image list. If the background color is the default value %CLR_NONE, the image is drawn transparently.	%ILD_TRANSPARENT	Draws the image transparently if there is a mask.	%ILD_MASK	Draws the mask.	%ILD_BLEND25	If there is a mask, the image is drawn blending 25% with the system highlight color.	%ILD_BLEND50	If there is a mask, the image is drawn blending 50% with the system highlight color.
%ILD_NORMAL	Draws the image using the background color of the image list. If the background color is the default value %CLR_NONE, the image is drawn transparently.										
%ILD_TRANSPARENT	Draws the image transparently if there is a mask.										
%ILD_MASK	Draws the mask.										
%ILD_BLEND25	If there is a mask, the image is drawn blending 25% with the system highlight color.										
%ILD_BLEND50	If there is a mask, the image is drawn blending 50% with the system highlight color.										
See also	GRAPHIC ATTACH , GRAPHIC COPY , GRAPHIC RENDER , GRAPHIC STRETCH , IMAGELIST										

Purpose Read a keyboard character if one is ready.

Syntax `GRAPHIC INKEY$ TO string_variable`

Remarks GRAPHIC INKEY\$ returns a string of 0, 1, or 2 characters that reflects the status of the keyboard buffer for the selected graphic target. A null string (LEN=0) means that the buffer is empty - no key pressed.

A string length of one means that an ASCII key was pressed and the string contains the ASCII character. An ASCII value between 1 and 31 indicate a control code.

A string length of two means that an extended key was pressed. In this case, the first character in the string has an ASCII value of zero, and the second is the extended keyboard code.

See also [GRAPHIC INPUT](#), [GRAPHIC INPUT FLUSH](#), [GRAPHIC INSTAT](#), [GRAPHIC LINE INPUT](#), [GRAPHIC WAITKEY\\$](#)

Purpose	Read data from the keyboard from within a graphic window or graphic control .
Syntax	<code>GRAPHIC INPUT [<i>prompt</i>,] <i>varlist</i></code>
<i>prompt</i>	An optional quoted string literal or string equate which is displayed to the user as a prompt.
<i>varlist</i>	A comma delimited sequence of one or more string or numeric variables.
Remarks	<p>GRAPHIC INPUT displays the prompt on the graphic window or graphic control, waits for the user to enter data from the keyboard, and assigns the data to the variables in <i>varlist</i>. Data entered from the keyboard must match the type of the variables -- that is, non-numeric characters are unacceptable for numeric variables.</p> <p>If a single GRAPHIC INPUT statement prompts for more than one variable, the user must enter the proper number of values on a single line, separated by commas. If not enough comma-delimited values are entered, remaining variables are set to zero or nul.</p>
See also	GRAPHIC INKEY\$, GRAPHIC INPUT FLUSH , GRAPHIC INSTAT , GRAPHIC LINE INPUT , GRAPHIC WAITKEY\$

GRAPHIC INPUT FLUSH statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose Remove all buffered keyboard data.

Syntax `GRAPHIC INPUT FLUSH`

See also [GRAPHIC INKEY\\$](#), [GRAPHIC INPUT](#), [GRAPHIC INSTAT](#), [GRAPHIC LINE INPUT](#), [GRAPHIC WAITKEY\\$](#)

Purpose	Determine whether a keyboard character is ready.
Syntax	<code>GRAPHIC INSTAT TO <i>n</i>&</code>
Remarks	<p>The long integer variable receives the keyboard buffer status for the selected graphic target. The value assigned is TRUE (non-zero) if a keyboard character is ready to be retrieved, or FALSE (zero) if not.</p> <p>GRAPHIC INSTAT does not remove the character from the buffer, so repeated execution will continue to return TRUE until the character is read with GRAPHIC INKEY\$, GRAPHIC INPUT, etc.</p>
See also	GRAPHIC INKEY\$, GRAPHIC INPUT , GRAPHIC INPUT FLUSH , GRAPHIC LINE INPUT , GRAPHIC WAITKEY\$

GRAPHIC LINE INPUT statement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose	Read an entire line from the keyboard from within a Graphic Window or a Graphic Control .
Syntax	<code>GRAPHIC LINE INPUT ["<i>prompt</i>"] <i>string_variable</i></code>
Remarks	GRAPHIC LINE INPUT displays the optional prompt on the Graphic Window or Control and waits for user input. Keystrokes are accepted until you press ENTER, at which time the entire typed string is assigned to the <i>string_variable</i> . Input is limited to 255 characters.
See also	GRAPHIC INKEY\$, GRAPHIC INPUT , GRAPHIC INPUT FLUSH , GRAPHIC INSTAT , GRAPHIC WAITKEY\$

Purpose	Draw a line on the selected graphic target
Syntax	<code>GRAPHIC LINE [STEP] [(<i>x1!</i>, <i>y1!</i>)] - [STEP] (<i>x2!</i>, <i>y2!</i>)[, <i>rgbColor&</i>]</code>
Remarks	The line is drawn from the first point, up to, but not including the second point. Coordinate points are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH . If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE .
<i>x1!</i> , <i>y1!</i>	Optional values which define the starting point of the line. If this optional first point is omitted, the line begins at the last point referenced (POS) in a preceding graphic statement. If the first STEP option is included, the <i>x1!</i> and <i>y1!</i> starting coordinates are relative to the last point referenced (POS).
<i>x2!</i> , <i>y2!</i>	The ending point of the line. If the second STEP option is included, the <i>x2!</i> and <i>y2!</i> ending coordinates are relative to the starting coordinates.
<i>rgbColor&</i>	Optional RGB color value for the line. If <i>rgbColor&</i> is omitted (or -1), the line color defaults to the current foreground color.
See also	GRAPHIC ARC , GRAPHIC ATTACH , GRAPHIC BOX , GRAPHIC COLOR , GRAPHIC ELLIPSE , GRAPHIC PIE , GRAPHIC STYLE , GRAPHIC WIDTH
Example	<pre>' Draw a triangle. Note that, since LINE draws up to, ' but not including the second point, one extra point ' must be added when STEP is used. GRAPHIC LINE (10, 10) - (10, 100) ' left side GRAPHIC LINE STEP - (101, 100) ' base line GRAPHIC LINE STEP - (10, 10) ' back to top</pre>

Purpose	Fill an area with a solid color or a hatch pattern.
Syntax	<code>GRAPHIC PAINT [BORDER REPLACE] [STEP] (x!, y!) [, [<i>rgbFill&</i>] [, [<i>rgbBorder&</i>] [, [<i>fillstyle&</i>]]]</code>
Remarks	The graphic target must first be chosen with GRAPHIC ATTACH . The coordinate points are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE).
<i>x!</i> , <i>y!</i>	The point where filling begins. If the STEP option is included, the x and y coordinates are relative to the last point referenced (POS) in the selected graphic target.
<i>rgbFill&</i>	Optional RGB color value for the fill area. If <i>rgbFill&</i> is omitted (or -1), the default foreground color is used.
<i>rgbBorder&</i>	Optional RGB base color of the fill area. If the REPLACE option is chosen, filling continues outward in all directions until a color other than <i>rgbBorder&</i> is found. If the BORDER option (or no option) is included, filling continues outward until the <i>rgbBorder&</i> color is found. If <i>rgbBorder&</i> is not specified, it is assumed to be the same as the <i>rgbFill&</i> parameter.
<i>fillstyle&</i>	Optional fill style (pattern) to use. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>rgbFill&</i> , while the background is specified by the default background color for the selected graphic target. The optional <i>fillstyle&</i> may be: <div><div>0</div><div>Solid (default)</div><div>1</div><div>Horizontal Lines</div><div>2</div><div>Vertical Lines</div><div>3</div><div>Upward Diagonal Lines</div><div>4</div><div>Downward Diagonal Lines</div><div>5</div><div>Crossed Lines</div><div>6</div><div>Diagonal Crossed Lines</div></div>
See also	Built In RGB Color Equates , GRAPHIC ARC , GRAPHIC ATTACH , GRAPHIC BOX , GRAPHIC COLOR , GRAPHIC ELLIPSE , GRAPHIC LINE , GRAPHIC PIE
Example	<pre>FUNCTION PBMAIN LOCAL hWin AS DWORD GRAPHIC WINDOW "Paint", 0, 0, 200, 200 TO hWin GRAPHIC ATTACH hWin, 0</pre>

```
' Draw a circle with blue foreground color
' and a box below it with red foreground color.
GRAPHIC ELLIPSE (10, 10) - (70, 70), %BLUE
GRAPHIC BOX (10, 80) - (70, 120), 0, %RED

' Fill the area inside the circle's blue borders
' with a green diagonal pattern.
GRAPHIC PAINT BORDER (40, 40), %GREEN, %BLUE, 6

'Retrieve the color at point 5,5 (outside the circle).
GRAPHIC GET PIXEL (5, 5) TO lRes&

' Fill the area outside the circle by replacing the color
' at point 5,5 and outwards with a solid yellow color.
GRAPHIC PAINT REPLACE (5, 5), RGB(255, 255, 223), lRes&, 0

SLEEP 10000
END FUNCTION
```

Purpose	Draw a pie section on the selected graphic target .
Syntax	<code>GRAPHIC PIE (x1!, y1!) - (x2!, y2!), arcStart!, arcEnd! [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]</code>
Remarks	<p>A pie section is an arc, with a line drawn from each end point to the center of the circle or ellipse. To specify a pie section, you would first define the full circle or ellipse of which it is a part, and then specify the points on the ellipse where the arc starts and stops.</p> <p>The full circle or ellipse is defined by its bounding rectangle, which is the smallest rectangle which can be drawn around the circle or ellipse. For example, if the circle is centered at position (400,400), with a radius of 100 pixels, the upper left corner (x1,y1) of the bounding rectangle is (300,300), and the lower right corner (x2,y2) is (500,500).</p> <p>The start point and end point of the arc are specified by their angle, which must be given in radians. A complete circle or ellipse is 2*pi radians. On a 12-hour clock-face, the values 0 and 2*pi both refer to the position of 3 o'clock, while the value 1*pi refers to the position of 9 o'clock. Other positions are specified by a radian value relative to these. In Classic PowerBASIC, arcs are always drawn counter-clockwise from the starting point to the ending point.</p> <p>Prior to any graphical operations, the graphic target must first be selected with GRAPHIC ATTACH. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH. If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE. Because of the nature of a pie section, GRAPHIC PIE neither uses, nor updates, graphic POS (last point referenced).</p>
<i>x1!, y1!</i>	The upper left corner of the bounding rectangle of the full circle or ellipse.
<i>x2!, y2!</i>	The lower right corner of the bounding rectangle of the full circle or ellipse.
<i>ArcStart!</i>	The starting angle of the arc, in radians, from 0 to 2*pi.
<i>ArcEnd!</i>	The ending angle of the arc, in radians, from 0 to 2*pi radians. Note that arcs are always drawn counter-clockwise from <i>arcStart!</i> to <i>arcEnd!</i> . Compared with a 12-hour clock-face, 0 or 2*pi radians is at 3 o'clock, and 1*pi radians is at 9 o'clock.
<i>rgbColor&</i>	Optional RGB color of the pie section edge. If omitted (or -1), the edge color defaults to the current foreground color.

fillcolor& Optional RGB color of the pie section interior. If *fillcolor*& is omitted (or -2), the interior of the pie section is not filled, allowing the background to show through. If *fillcolor*& is -1, the interior is painted with the same color as the edge. Otherwise, *fillcolor*& specifies the RGB color to be used.

fillstyle& Optional fill style (pattern) to be used. If *fillstyle*& is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the *fillcolor*&, while the background is specified by the default background color. The optional *fillstyle*& may be:

- 0 Solid (default)
- 1 Horizontal Lines
- 2 Vertical Lines
- 3 Upward Diagonal Lines
- 4 Downward Diagonal Lines
- 5 Crossed Lines
- 6 Diagonal Crossed Lines

See also [Built In RGB Color Equates](#), [GRAPHIC ARC](#), [GRAPHIC ATTACH](#), [GRAPHIC BOX](#), [GRAPHIC COLOR](#), [GRAPHIC ELLIPSE](#), [GRAPHIC LINE](#), [GRAPHIC STYLE](#), [GRAPHIC WIDTH](#)

Example

```
FUNCTION PBMAIN
    LOCAL hWin AS DWORD

    GRAPHIC WINDOW "Pie", 0, 0, 200, 200 TO hWin
    GRAPHIC ATTACH hWin, 0

    ' A full circle is 2Pi radians (100%).
    ' To show a 25% Pie, use the formula 0.25 * 2Pi.
    ' The following divides a full circle into four 25% parts, each
    ' with its own colors, each slightly separated from the others.
    ' Note: 0 is at 3 O'clock, then it builds counter-clockwise.

    LOCAL Pi2 AS DOUBLE
    Pi2 = 8 * ATN(1) ' 2 * Pi can be useful here

    GRAPHIC PIE (10, 9)-(110, 109), 0, Pi2 * 0.25, %BLUE,
    %LTGRAY, 3
    GRAPHIC PIE (9, 9)-(109, 109), Pi2 * 0.25, Pi2 * 0.50, %RED,
    %LTGRAY, 4
    GRAPHIC PIE (9, 10)-(109, 110), Pi2 * 0.5, Pi2 * 0.75, RGB(0,127,0),
    %LTGRAY, 3
    GRAPHIC PIE (10, 10)-(110, 110), Pi2 * 0.75, 0, %GRAY, %LTGRAY,
    4

    SLEEP 10000
```

END FUNCTION

Purpose	Draw a polygon in the selected graphic target .
Syntax	<code>GRAPHIC POLYGON <i>points</i> [, [<i>rgbColor&</i>] [, [<i>fillcolor&</i>] [, [<i>fillstyle&</i>] [, [<i>fillmode&</i>]]]]]</code>
Remarks	<p>The graphic target must first be selected with GRAPHIC ATTACH. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). Line width can be set using GRAPHIC WIDTH. If line width is set to 1 (the default), the line style can be set with GRAPHIC STYLE. GRAPHIC POLYGON neither uses, nor updates, the last point referenced (POS).</p>
<i>points</i>	<p>User-defined type that defines the number of vertices and the location of each. There must be at least two, and no more than 1024 vertices. The first member is a long integer point count, followed directly by the appropriate number of single precision floats to specify the actual coordinates. Floating point coordinates are required, because of the possibility of their use as world coordinates with SCALE. You can use a type with a scalar list, like this:</p> <pre>TYPE PolyPoints count as long x1 as single y1 as single x2 as single y2 as single x3 as single y3 as single END TYPE</pre> <p>Or, you can create an array using point types, like this:</p> <pre>TYPE PolyPoint x as single y as single END TYPE TYPE PolyArray count as long xy(1 TO 3) as PolyPoint END TYPE</pre>
<i>rgbColor&</i>	Optional RGB color of the polygon edge. If omitted (or -1), the edge color defaults to the current foreground color.
<i>fillcolor&</i>	Optional RGB color of the polygon interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the ellipse is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.

fillstyle&

Optional fill style (pattern) to be used. If *fillstyle*& is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the *fillcolor*&, while the background is specified by the default background color. The optional *fillstyle*& may be:

- 0 Solid (default)
- 1 Horizontal Lines
- 2 Vertical Lines
- 3 Upward Diagonal
Lines
- 4 Downward Diagonal
Lines
- 5 Crossed Lines
- 6 Diagonal Crossed
Lines

fillmode&

If *fillmode*& is missing (or zero), the winding mode is selected. This fills any region with a non-zero winding value. If *fillmode*& is non-zero, the alternate mode is selected. This fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, it fills the area between the first side and the second side, between the third side and fourth side, etc.

See also

[Built In RGB Color Equates](#), [GRAPHIC ARC](#), [GRAPHIC ATTACH](#), [GRAPHIC BOX](#), [GRAPHIC COLOR](#), [GRAPHIC ELLIPSE](#), [GRAPHIC LINE](#), [GRAPHIC POLYLINE](#), [GRAPHIC STYLE](#), [GRAPHIC WIDTH](#)

Purpose Draw a series of connected line segments.

Syntax `GRAPHIC POLYLINE points [, rgbColor&]`

Remarks The graphic target must first be selected with [GRAPHIC ATTACH](#). The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with [GRAPHIC SCALE](#)). Line width can be set using [GRAPHIC WIDTH](#). If line width is set to 1 (the default), the line style can be set with [GRAPHIC STYLE](#). [GRAPHIC POLYLINE](#) neither uses, nor updates, the last point referenced (POS).

points User-defined type that defines the number of vertices and the location of each. There must be at least two, and no more than 1024 vertices. The first member is a long integer point count, followed directly by the appropriate number of single precision floats to specify the actual coordinates. Floating point coordinates are required, because of the possibility of their use as world coordinates with SCALE. You can use a type with a scalar list, like this:

```
TYPE PolyPoints
  count as long
  x1 as single
  y1 as single
  x2 as single
  y2 as single
  x3 as single
  y3 as single
END TYPE
```

Or, you can create an array using point types, like this:

```
TYPE PolyPoint
  x as single
  y as single
END TYPE

TYPE PolyArray
  count as long
  xy(1 TO 3) as PolyPoint
END TYPE
```

rgbColor& Optional [RGB](#) color of the polyline. If omitted (or -1), the [color](#) defaults to the current foreground color.

See also [Built In RGB Color Equates](#), [GRAPHIC ARC](#), [GRAPHIC ATTACH](#), [GRAPHIC BOX](#), [GRAPHIC COLOR](#), [GRAPHIC ELLIPSE](#), [GRAPHIC LINE](#), [GRAPHIC POLYGON](#), [GRAPHIC STYLE](#), [GRAPHIC WIDTH](#)

Purpose	Output text to the selected graphic target .
Syntax	<code>GRAPHIC PRINT [<i>expr</i>] [<i>;</i> <i>expr</i>] [<i>;</i>]</code>
Remarks	<p>Prior to any graphical operations, the graphic target must first be selected with GRAPHIC ATTACH. The text color and the text background color are set with GRAPHIC COLOR. Text which extends beyond the bounds of the graphic target is clipped. The size of the text to be printed can be determined in advance with the GRAPHIC TEXT SIZE statement.</p>
<i>expr</i>	<p>A numeric or string expression to be drawn on the selected graphic target. A semicolon can be used as separator between multiple expressions in the same statement. Drawing begins at the last point referenced (POS) by another graphic statement, or the point specified by GRAPHIC SET POS. The upper left corner of the text is positioned at the POS. Upon completion, the POS is moved to the left margin of the next line. However, if a trailing semi-colon is included, movement to the next line is suppressed.</p> <div>GRAPHIC PRINT USING\$ is valid but USING\$ will be evaluated as a string expression. See the USING\$ function for more information.</div>
See also	GRAPHIC ATTACH , GRAPHIC CHR SIZE , GRAPHIC SET FONT , GRAPHIC GET POS , GRAPHIC SET POS , GRAPHIC TEXT SIZE , USING\$

Purpose	Update buffered graphical statements, drawing them to the selected graphic target .
Syntax	GRAPHIC REDRAW
Remarks	This statement is only needed when GRAPHIC ATTACH with the REDRAW option have been chosen for faster, buffered draw operations. Otherwise, it performs no operation.

All Classic PowerBASIC graphical displays are *persistent* -- they are automatically redrawn for you after resuming from being minimized or temporarily covered by other windows.

In intensive drawing operations, it is preferable to delay the display until a number of statements have been performed by using the REDRAW option with the GRAPHIC ATTACH statement and the GRAPHIC REDRAW statement. This can improve the overall performance dramatically.

See also [GRAPHIC ATTACH](#)

Example

```
FUNCTION PBMAIN
    LOCAL hWin AS DWORD

    ' Draw a buffered, blue gradient fill
    GRAPHIC WINDOW "Gradient", 0, 0, 255, 255 TO hWin
    GRAPHIC ATTACH hWin, 0, REDRAW
    FOR y& = 0 TO 255
        GRAPHIC LINE (0, y&) - (255, y&), RGB(0, 0, y&)
    NEXT
    GRAPHIC REDRAW

    SLEEP 10000
END FUNCTION
```

Purpose Render an image on the selected [graphic target](#).

Syntax `GRAPHIC RENDER BmpName$, (x1!, y1!)-(x2!, y2!)`

Remarks Renders an image, loaded from a [resource](#) or a disk file, on the selected graphic bitmap, [control](#), or [window](#). The parameter `BmpName$` contains the name of an image to be loaded. If `BmpName$` contains a period, it is presumed to be the name of a disk file. Otherwise, an attempt is made to load it from the programs resource data; if not found, it is then presumed to be a disk file. The parameters `x1!`, `y1!` define the upper left corner of the destination rectangle, while `x2!`, `y2!` define the lower right corner of that rectangle. The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with [GRAPHIC SCALE](#)). If the destination rectangle is larger or smaller than the original, the image is stretched or shrunk to the requested size.

The following code will retrieve the natural size of an image in a bitmap file, in pixels:

```
nFile& = FREEFILE
OPEN "myimage.bmp" FOR BINARY AS nFile&
GET #nFile&, 19, nWidth&
GET #nFile&, 23, nHeight&
CLOSE nFile&
```

See also [GRAPHIC ATTACH](#), [GRAPHIC COPY](#), [GRAPHIC IMAGELIST](#), [GRAPHIC STRETCH](#)

GRAPHIC SAVE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Save an image to a bitmap (.BMP) file.
Syntax	<code>GRAPHIC SAVE <i>BmpName\$</i></code>
Remarks	The selected graphic target (a graphic bitmap, control , or window , etc.) is saved to a disk file using the filename specified by BmpName\$. The bitmap is always saved in a single plane, 24-bit format, to allow for the maximum (true color) resolution.
See also	CONTROL ADD GRAPHIC , GRAPHIC ATTACH , GRAPHIC BITMAP LOAD , GRAPHIC BITMAP NEW

Purpose	Define a custom coordinate system for the graphic target .
Syntax	<pre>GRAPHIC SCALE (x1!, y1!) - (x2!, y2!) GRAPHIC SCALE PIXELS</pre>
Remarks	<p>The graphic target must first be chosen with GRAPHIC ATTACH. GRAPHIC SCALE lets you define your own world coordinate system for subsequent graphic statements. The custom coordinates remain with the graphic target until GRAPHIC SCALE is repeated, or the target is deleted. World coordinates may be values, with the only requirement that x1! not equal x2!, and y1! not equal y2!. If either is equal, the statement is ignored. If x2! is greater than x1!, coordinates grow larger as they move to the right. Otherwise, they grow larger as they move to the left. If y2! is greater than y1!, coordinates grow larger as they move downward. Otherwise, they grow larger as they move upward. GRAPHIC SCALE PIXELS sets or resets the coordinate system to pixel coordinates. This can be particularly valuable when the original coordinates are in Dialog Units, since this provides increased resolution for other graphic functions.</p>
See also	GRAPHIC ATTACH , GRAPHIC GET SCALE

Purpose	Replace a copy of a bitmap that was retrieved as a device-independent bitmap.
Syntax	GRAPHIC SET BITS <i>bitexpr</i> \$
Remarks	<p>This statement replaces a bitmap that was originally retrieved with the GRAPHIC GET BITS statement. The bitmap is assigned to the selected graphic target from the string expression specified by <i>bitexpr</i>\$. This allows you to make many modifications to the bitmap very quickly, particularly operations which may not be directly supported by Classic PowerBASIC. For example, a typical use might be to change all red pixels in a bitmap to blue.</p> <p>The <i>bitexpr</i>\$ string contains a series of four-byte values, each of which represents a long integer. You can convert the four-byte string sections to numeric values with the CVL function, and convert a numeric value to a four-byte string with MKL\$. The first four-byte value specifies the width of the bitmap, in pixels, and the second specifies the height. Following that will be one four-byte value for each pixel in the bitmap, which represents the color of that pixel. So, a 20 by 20 bitmap would have 400 pixels and require 1600 bytes (400 * 4), plus 4 bytes for the width and 4 bytes for the height, or a total of 1608 bytes.</p> <p>The first four-byte pixel value in the string represents the top-left corner of the image, the second represents the second pixel of the first row, and so on. After the last pixel of the first row will be the first pixel of the second row, etc.</p> <p>If execution speed is most important, it's likely that the string can be manipulated most efficiently with pointer variables.</p> <p>Some Windows API functions, namely those which reference Device-Independent Bitmaps (DIB), require that colors be specified in the reverse of normal RGB sequence (Blue-Green-Red instead of Red-Green-Blue). To maximize performance, GRAPHIC SET BITS uses BGR format as well. You can use the BGR function to translate an RGB value to its BGR equivalent.</p>
See also	Built In RGB Color Equates , BGR , CVL , GRAPHIC GET BITS , MKL\$, RGB
Example	<pre>' Change all red pixels to blue LOCAL PixelPtr AS LONG PTR GRAPHIC GET BITS TO bmp\$ xsize& = CVL(bmp\$,1) ysize& = CVL(bmp\$,5) PixelPtr = STRPTR(bmp\$) + 8</pre>

```
FOR i& = 1 TO xsize& * ysize&
  IF @PixelPtr = BGR(%red) THEN @PixelPtr = BGR(%blue)
  INCR PixelPtr
NEXT
GRAPHIC SET BITS bmp$
```


GRAPHIC SET FOCUS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Bring the selected graphic window to the foreground and direct focus to it.
Syntax	GRAPHIC SET FOCUS
Remarks	A graphic window must first be chosen with GRAPHIC ATTACH . The GRAPHIC SET FOCUS statement brings the graphic window to the foreground, directing focus to it. This is particularly useful when another window may overlap the graphic window.
See also	GRAPHIC ATTACH , GRAPHIC WINDOW

Purpose	Select a font for the GRAPHIC PRINT , GRAPHIC INPUT , and GRAPHIC LINE INPUT statements.
Syntax	<code>GRAPHIC SET FONT <i>fonthdl</i>&</code>
<i>fonthdl</i> &	The numeric handle returned by the FONT NEW statement.
Remarks	<p>The font specified by <i>fonthdl</i>& is selected to be used by all of the following GRAPHIC PRINT, GRAPHIC INPUT, and GRAPHIC LINE INPUT statements. This is the most efficient way to change fonts and their general appearance (size, style, etc.).</p> <p>You can predefine virtually any number of fonts and attributes by executing FONT NEW statements for each of them. That makes them ready for immediate use when selected by GRAPHIC SET FONT.</p> <p>Prior to any graphical operations, the graphic target must first be selected with GRAPHIC ATTACH. If no specific font is selected, the default font is MS Sans Serif, 8 point, with no style attributes.</p>
Restrictions	GRAPHIC SET FONT replaces GRAPHIC FONT . Note that the GRAPHIC FONT statement may not be supported in future versions of PowerBASIC, so update your code to use the new syntax.
See also	FONT NEW , GRAPHIC ATTACH , GRAPHIC CHR SIZE , GRAPHIC FONT , GRAPHIC INPUT , GRAPHIC LINE INPUT , GRAPHIC PRINT , GRAPHIC TEXT SIZE

GRAPHIC SET LOC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Change the location of the selected graphic target on the screen.
Syntax	<code>GRAPHIC SET LOC <i>x&</i>, <i>y&</i></code>
Remarks	This statement allows you to change the location of the selected graphic window. If no graphic target is selected, or it is not a graphic window, no action is taken. The location is always given in the same terms (pixels or dialog units) as the parent dialog, relative to the upper left corner of the screen.
See also	GRAPHIC ATTACH , GRAPHIC GET LOC , GRAPHIC GET PPI

Purpose Set the color mix mode for the selected [graphic target](#).

Syntax `GRAPHIC SET MIX mode&`

Remarks Prior to any graphical operations, the graphic target must first be selected with [GRAPHIC ATTACH](#). There are 16 mix modes available to use for mixing the drawing color with the color that already exists at the drawing location. The mix mode equates are predefined in Classic PowerBASIC.

%mix_Blackness	Pixel is always 0 (black).
%mix_NotMergeSrc	Pixel is the inverse of the MergeSrc color.
%mix_MaskNotSrc	Pixel is a combination of the colors common to both the pixel and the inverse of the source.
%mix_NotCopySrc	Pixel is the inverse of the pen color.
%mix_MaskSrcNot	Pixel is a combination of the colors common to both the source and the inverse of the pixel.
%mix_Not	Pixel is the inverse of the pixel color.
%mix_XorSrc	Pixel is a combination of the colors in the source and in the pixel, but not in both.
%mix_NotMaskSrc	Pixel is the inverse of the MaskSrc color.
%mix_MaskSrc	Pixel is a combination of the colors common to both the source and the pixel.
%mix_NotXorSrc	Pixel is the inverse of the XorSrc color.
%mix_Nop	Pixel remains unchanged.
%mix_MergeNotSrc	Pixel is a combination of the source color and the inverse of the pixel color.
%mix_CopySrc	Pixel is the source color (default).
%mix_MergeSrcNot	Pixel is a combination of the source color and the inverse of the pixel color.
%mix_MergeSrc	Pixel is a combination of the source color and the pixel color.
%mix_Whiteness	Pixel is always 1 (white).

See also [GRAPHIC ATTACH](#), [GRAPHIC GET MIX](#)

GRAPHIC SET PIXEL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Draw a single pixel.
Syntax	<code>GRAPHIC SET PIXEL [STEP] (<i>x!</i>, <i>y!</i>) [, <i>rgbColor</i>&]</code>
Remarks	The graphic target must first be chosen with GRAPHIC ATTACH . The coordinate point is specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). If the STEP option is included, the <i>x!</i> and <i>y!</i> coordinates are relative to the last point referenced (POS).
See also	Built In RGB Color Equates , GRAPHIC ATTACH , GRAPHIC COLOR , GRAPHIC GET PIXEL

GRAPHIC SET POS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the last point referenced (POS) for the selected graphic target .
Syntax	<code>GRAPHIC SET POS [STEP] (<i>x!</i>, <i>y!</i>)</code>
Remarks	The graphic target must first be chosen with GRAPHIC ATTACH . The coordinate point is specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). If the STEP option is included, the <i>x!</i> and <i>y!</i> coordinates are relative to the last point referenced (POS).
See also	GRAPHIC ATTACH , GRAPHIC GET POS , GRAPHIC SCALE

Purpose	Copy and resize a bitmap to the selected graphic target .																
Syntax	<code>GRAPHIC STRETCH <i>hbmSource???</i>, <i>id&</i>, (<i>x1!</i>, <i>y1!</i>)-(<i>x2!</i>, <i>y2!</i>) TO (<i>x3!</i>, <i>y3!</i>)-(<i>x4!</i>,<i>y4!</i>) [, <i>mix&</i>, <i>stretch&</i>]</code>																
Remarks	<p>You can copy a complete bitmap, or a portion of it, to the selected graphic target, while resizing it to a larger or smaller size. The expression <i>hbmSource???</i> specifies the handle of the source bitmap, control, or window. The expression <i>id&</i> is the unique control identifier in the range 1 to 65535, as assigned with the CONTROL ADD GRAPHIC statement. <i>id&</i> must be 0 for a GRAPHIC WINDOW or a bitmap.</p> <p>The Coordinates are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE). The destination of the copy operation is the target specified by GRAPHIC ATTACH. The bitmap is automatically resized to fit the destination parameters. You must use care that your parameters are valid for the specified bitmaps, or results of the operation are undefined. If the <i>mix&</i> parameter is included, it is one of the values in the following table. If not included, a default of %mix_CopySrc is presumed. There are 8 mix modes available to use for mixing drawing colors with the colors which already exist at the at the drawing location. The mix mode equates are predefined in Classic PowerBASIC.</p> <p>The 8 mix modes are:</p> <table><tr><td>%MIX_NOTMERGESRC</td><td>Pixel is the inverse of the MergeSrc color.</td></tr><tr><td>%MIX_NOTCOPYSRC</td><td>Pixel is the inverse of the source color.</td></tr><tr><td>%MIX_MASKSRCNOT</td><td>Pixel is a combination of the colors common to both the source and the inverse of the pixel.</td></tr><tr><td>%MIX_XORSRC</td><td>Pixel is a combination of the colors in the source and the pixel, but not in both.</td></tr><tr><td>%MIX_MASKSRC</td><td>Pixel is a combination of the colors common to both the source and the pixel.</td></tr><tr><td>%MIX_MERGENOTSRC</td><td>Pixel is a combination of the pixel color and the and the inverse of the source color.</td></tr><tr><td>%MIX_COPYSRC</td><td>Pixel is the source color (Default Mode).</td></tr><tr><td>%MIX_MERGESRC</td><td>Pixel is a combination of the source color and the pixel color.</td></tr></table>	%MIX_NOTMERGESRC	Pixel is the inverse of the MergeSrc color.	%MIX_NOTCOPYSRC	Pixel is the inverse of the source color.	%MIX_MASKSRCNOT	Pixel is a combination of the colors common to both the source and the inverse of the pixel.	%MIX_XORSRC	Pixel is a combination of the colors in the source and the pixel, but not in both.	%MIX_MASKSRC	Pixel is a combination of the colors common to both the source and the pixel.	%MIX_MERGENOTSRC	Pixel is a combination of the pixel color and the and the inverse of the source color.	%MIX_COPYSRC	Pixel is the source color (Default Mode).	%MIX_MERGESRC	Pixel is a combination of the source color and the pixel color.
%MIX_NOTMERGESRC	Pixel is the inverse of the MergeSrc color.																
%MIX_NOTCOPYSRC	Pixel is the inverse of the source color.																
%MIX_MASKSRCNOT	Pixel is a combination of the colors common to both the source and the inverse of the pixel.																
%MIX_XORSRC	Pixel is a combination of the colors in the source and the pixel, but not in both.																
%MIX_MASKSRC	Pixel is a combination of the colors common to both the source and the pixel.																
%MIX_MERGENOTSRC	Pixel is a combination of the pixel color and the and the inverse of the source color.																
%MIX_COPYSRC	Pixel is the source color (Default Mode).																
%MIX_MERGESRC	Pixel is a combination of the source color and the pixel color.																

If the *stretch&* parameter is included, it is one of the values in the following table. If not included, or it is the value zero (0), the stretch mode is unchanged. An appropriate choice of stretch mode can substantially

enhance the quality of bitmaps which are changed in size. The stretch mode equates are predefined in Classic PowerBASIC.

The 4 stretch modes are:

- %BLACKONWHITE** This is the default Windows stretch mode, and is most appropriate for monochrome bitmaps, or those with blocks of color. Performs a boolean OR of eliminated and existing pixels. It preserves black pixels at the expense of white pixels.
- %WHITEONBLACK** Performs a boolean OR of eliminated and existing pixels. It preserves white pixels at the expense of black pixels.
- %COLORONCOLOR** Deletes eliminated lines of pixels without trying to preserve their information.
- %HALFTONE** This provides the highest quality for complex color bitmaps. The average color of the destination pixel block is kept approximately the same as the source pixel block.

See also

[GRAPHIC ATTACH](#), [GRAPHIC COPY](#), [GRAPHIC IMAGELIST](#), [GRAPHIC RENDER](#)

Purpose Set the line style to be used by various graphic statements in the selected [graphic target](#).

Syntax `GRAPHIC STYLE linestyle&`

Remarks The graphic target must first be selected with [GRAPHIC ATTACH](#). Due to limitations in the Windows graphics device interface (GDI), styles are only applied if the line width is set to 1, the default. If the line width is greater than 1, the style is interpreted as 0, solid.

Available line styles are:

- 0 Solid
(default)
- 1 Dash
- 2 Dot
- 3 DashDot
- 4 DashDotDot

See also [GRAPHIC ARC](#), [GRAPHIC ATTACH](#), [GRAPHIC BOX](#), [GRAPHIC ELLIPSE](#), [GRAPHIC LINE](#), [GRAPHIC PIE](#), [GRAPHIC WIDTH](#)

Example

```
' Draw a square box with red, dotted lines
GRAPHIC WIDTH 1
GRAPHIC STYLE 2
GRAPHIC BOX (10, 10) - (110, 110), 0, %RED
```

Purpose	Calculate the size of text to be printed .
Syntax	<code>GRAPHIC TEXT SIZE <i>txt\$</i> TO <i>nWidth!</i>, <i>nHeight!</i></code>
Remarks	<p>This statement calculates the total size of the printed text, based upon the current font for the graphic target. The sizes returned are specified in the same terms (pixels or dialog units) as the parent dialog (or world coordinates, if those were chosen with GRAPHIC SCALE).</p> <p>This allows you to easily calculate the appropriate print position, particularly when using a proportional font.</p>
See also	FONT NEW , GRAPHIC ATTACH , GRAPHIC CHR SIZE , [****] GRAPHIC SET FONT , GRAPHIC SCALE , GRAPHIC PRINT

Purpose Read a keyboard character, waiting until one is ready.

Syntax GRAPHIC WAITKEY\$ [TO *string_variable*]

Remarks GRAPHIC WAITKEY\$ waits for a key to be pressed. It removes the character from the keyboard buffer for the selected [graphic target](#), and optionally assigns it to the *string_variable*. If the TO clause is omitted, the keyboard character is discarded.

It returns a string of 0, 1, or 2 characters that reflects the status of the keyboard buffer for the selected graphic target. A null string ([LEN](#)=0) means that there was an error, such as the case when no graphic target has been assigned with [GRAPHIC ATTACH](#).

A string length of one means that an ASCII key was pressed and the string contains the ASCII character. An ASCII value between 1 and 31 indicate a control code.

A string length of two means that an extended key was pressed. In this case, the first character in the string has an ASCII value of zero, and the second is the extended keyboard code.

See also [GRAPHIC INKEY\\$](#), [GRAPHIC INPUT](#), [GRAPHIC INPUT FLUSH](#), [GRAPHIC INSTAT](#), [GRAPHIC LINE INPUT](#)

Purpose	Set the line width to be used by various graphic statements in the selected graphic target .
Syntax	GRAPHIC WIDTH <i>linewidth</i> &
Remarks	The graphic target must first be selected with GRAPHIC ATTACH . If line width is set to a value greater than 1 (default), the line style is always interpreted to be 0 (solid).
See also	GRAPHIC ARC , GRAPHIC ATTACH , GRAPHIC BOX , GRAPHIC ELLIPSE , GRAPHIC LINE , GRAPHIC PIE , GRAPHIC STYLE
Example	<pre>' Draw a square box with red, thick lines GRAPHIC WIDTH 10 GRAPHIC BOX (10, 10) - (110, 110), 0, %RED</pre>

Purpose	Create a new graphic window.
Syntax	<code>GRAPHIC WINDOW <i>caption\$</i>, <i>x&</i>, <i>y&</i>, <i>nWidth&</i>, <i>nHeight&</i> TO <i>hWin???</i></code>
Remarks	<p>After a graphic window has been created, GRAPHIC ATTACH is used to choose it as the selected graphic target. You can then draw text, lines, circles, and other forms with various graphics statements. Due to the nature of a GRAPHIC WINDOW and its contents, it may not be resized. If a new size is needed, it should be destroyed and recreated.</p> <p>GRAPHIC WINDOW END can be used to close and destroy the selected Graphic window at any time. Otherwise, the window is automatically destroyed when the program ends.</p> <p>All Classic PowerBASIC graphical displays are <i>persistent</i> -- they are automatically redrawn for you after resuming from being minimized or temporarily covered by other windows.</p>
<i>caption\$</i>	The text to be displayed in the title or caption bar of the graphic window. If <i>caption\$</i> is empty (zero-length), the window is displayed without a title bar, so the appearance is different and the window cannot be dragged by the user.
<i>x&</i> , <i>y&</i>	The location of the window, in pixels, relative to the upper left corner of the screen.
<i>nWidth&</i>	The width of the client area of the window, not including the frame. The width is specified in pixels.
<i>nHeight&</i>	The height of the client area of the window, not including the frame. The height is specified in pixels.
<i>hWin???</i>	The handle of the newly-created window. If the window could not be created, <i>hWin???</i> will be 0.
See also	CONTROL ADD GRAPHIC , GRAPHIC ATTACH , GRAPHIC DETACH , GRAPHIC WINDOW END , GRAPHIC WINDOW CLICK
Example	<pre>FUNCTION PBMAIN () AS LONG ' Create and show a Graphic window on screen LOCAL hWin AS DWORD GRAPHIC WINDOW "Box", 300, 300, 130, 130 TO hWin GRAPHIC ATTACH hWin, 0 GRAPHIC BOX (10, 10) - (120, 120), 0, %BLUE SLEEP 5000 ' show it for 5 seconds, then end END FUNCTION</pre>

Purpose	Check whether a GRAPHIC WINDOW has been clicked with the mouse.
Syntax	<code>GRAPHIC WINDOW CLICK [<i>hwin</i>&] TO <i>click</i>&, <i>x</i>!, <i>y</i>!</code>
<i>hWin</i> &	Handle of the GRAPHIC WINDOW to check.
Remarks	<p>GRAPHIC WINDOW CLICK checks whether the specified GRAPHIC WINDOW has been clicked since the last time this statement was executed on this window. If so, the value one (1) is assigned to the <i>click</i>& variable for a single click, or two (2) for a double click. Also, the mouse position is assigned to <i>x</i>! and <i>y</i>!. If there has been no click, the value zero (0) is assigned to all three result variables.</p> <p>In case of a double click, a <i>click</i>& value of one (1) is returned immediately after the first click, and a <i>click</i>& value of two (2) is also returned after the second click.</p> <p>If the optional handle (<i>hwin</i>&) is omitted, the graphic window which is currently selected with GRAPHIC ATTACH is used.</p>
See also	GRAPHIC WINDOW

GRAPHIC WINDOW END statement

[Top](#) [Previous](#) [Next](#)

Purpose	Close and destroy the selected graphic window .
Syntax	<code>GRAPHIC WINDOW END</code>
Remarks	<p>The graphic window must first be selected with GRAPHIC ATTACH. GRAPHIC WINDOW END can be used to close and destroy the selected graphic window at any time. Otherwise, the window is automatically destroyed when the program ends.</p>
See also	GRAPHIC DETACH , GRAPHIC WINDOW

Purpose	Return a 16-byte (128-bit) Globally Unique Identifier (GUID) or Universally Unique Identifier (UUID) binary string.
Syntax	<pre>id\$ = GUID\$[()] id\$ = GUID\$(guidtext\$)</pre>
Remarks	<p>The GUID\$ function, with no parameter (or a null, zero-length string parameter) will return a new, unique 16-byte string GUID (Globally Unique Identifier). This GUID may be used as a new class identifier or an interface identifier, or for some other purpose where a unique identifier may be required, such as for a one-time encryption key.</p> <p>If <i>guidtext\$</i> is specified, GUID\$ examines a text string, and converts the first standard format, human-readable GUID it finds, and returns a 16-byte binary string. This 16-byte string contains the internal GUID representation as a 128-bit data item.</p> <p>To be valid, the GUID string in <i>guidtext\$</i> string must contain exactly 32 hexadecimal digits, optionally delimited by spaces or hyphens, but which must be enclosed overall by curly braces. For example: "{01234567-89AB-CDEF-FEDC-BA9876543210}".</p> <p>The GUID\$ function is the logical complement to the GUIDTXT\$ function.</p>
<i>id\$</i>	The return string may be assigned to a dynamic string , or a fixed-length string of at least 16 bytes, or (typically) a GUID variable. See DIM for more information on creating GUID variables.
Restrictions	<p>If any errors are encountered, GUID\$ returns a null (zero-length) string instead of the 16-byte GUID string. GUID\$ can also be used in string equate assignments provided an explicit human-readable GUIDTXT\$ argument string is assigned. For example:</p> <pre>\$AppGuid = GUID\$("{01234567-89AB-CDEF-FEDC-BA9876543210}")</pre>
See also	DIM , CLSID\$, GUIDTXT\$, How are GUID's used with objects? , INTERFACE (Direct) , INTERFACE (IDBind) , ISINTERFACE , ISNOTHING , ISOBJECT , Just what is COM? , LET (with Objects) , OBJECT , OBJACTIVE , OBJPTR , OBJRESULT , PROGID\$, What is an object, anyway?
Example	<pre>DIM oID1 AS GUID, oID2 AS GUID oID1 = GUID\$("{01234567-89AB-CDEF-FEDC-BA9876543210}") oID2 = GUID\$("The GUID we need is shown as {0123456789ABCDEFFEDCBA9876543210}")</pre>

GUIDTXT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a 38-byte human-readable Globally Unique Identifier (GUID) or Universally Unique Identifier (UUID) string from a 16-byte GUID string.
Syntax	<code>id\$ = GUIDTXT\$(guid16\$)</code>
Remarks	<p>The GUIDTXT\$ function takes a string parameter <i>guid16\$</i> that must be exactly 16-bytes long (and represents a 128-bit GUID string), and returns a 38-byte GUID text string. <i>guid16\$</i> is usually a GUID variable but may also be a dynamic or fixed-length string, etc.</p> <p>The GUIDTXT\$ function is the logical complement to the GUID\$ function.</p>
Restrictions	If any errors are encountered, GUIDTXT\$ returns a null (zero-length) string instead of the 38-byte GUID text string.
See also	DIM , CLSID\$, GUID\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISINTERFACE , ISNOTHING , ISOBJECT , LET (with Objects) , OBJECT , OBJACTIVE , OBJPTR , OBJRESULT , PROGID\$, What is an object, anyway?
Example	<pre>oID1\$ = GUID\$("{01234567-89AB-CDEF-FEDC-BA9876543210}") oID2\$ = GUIDTXT\$(oID1\$)</pre>

Purpose	Return a string that is the hexadecimal (base 16) representation of its argument.
Syntax	<code>s\$ = HEX\$(numeric_expression [, digits])</code>
Remarks	<p><i>numeric_expression</i> must be in the decimal range -2,147,483,648 to +4,294,967,295. Any fractional part of <i>numeric_expression</i> is rounded before the string is created.</p> <p>If <i>digits</i> is specified, the result string will be of length <i>digits</i>. <i>digits</i> may be in the range 1 to 8. If the value is shorter than <i>digits</i>, leading zeros are appended as needed. If the value is longer than <i>digits</i>, the result will be truncated from the left as necessary.</p> <p>Hexadecimal is a number system that uses base 16 rather than the base 10 used by the everyday decimal system. In hexadecimal, the digits 0 to 9 represent the same numbers as in decimal, and the letters A to F represent the decimal values 10 to 15. Hexadecimal notation is commonly used in programming because it is a convenient way of representing the binary bit patterns used internally by computers. A single hex digit represents 1 nibble (4 bits), two hex digits represent one byte (8 bits), and so forth.</p> <p>By convention, hex numbers are unsigned, since they represent bit patterns. If you are not familiar with two's-complement arithmetic, the result of using HEX\$ on a negative (signed) argument may surprise you. See Numeric Literals for more information on explicitly casting numeric literal values to specific data types and distinguishing between signed and unsigned values. Also see the BIN\$ function.</p>

To display a [Quad-integer](#) in hex format, use the following technique:

```
DIM q AS QUAD, r AS STRING
q = &H0ABCEF0EE1234567F&&
r = HEX$(HI(LONG, q)) + HEX$(LO(DWORD, q), 8)
```

Hexadecimal strings can be converted to numeric values with the [VAL](#) function by prefixing the hex string with "&H". If the string has a leading zero, the result is always unsigned. For example:

```
a$ = HEX$(65535)      ' a$ will contain "FFFF"
x& = VAL("&H" + a$)   ' Signed result (-1)
y& = VAL("&H0" + a$)  ' Unsigned result (65535)
```

See also [BIN\\$](#), [FORMAT\\$](#), [OCT\\$](#), [STR\\$](#), [USING\\$](#), [VAL](#)

Example	<pre>DIM a AS DWORD, b AS STRING, c AS QUAD a = &H0FFFFFFFF ' Unsigned literal b = HEX\$(a) ' "FFFFFFFF"</pre>
----------------	---

```
c = a                ' 4294967295
b = HEX$(c)          ' "FFFFFFFF"
c = VAL("&H" + b)    ' -1&& (signed conversion)
b = HEX$(c)          ' "FFFFFFFF"
c = VAL("&H0" + b)   ' 4294967295&&
b = HEX$(c)          ' "FFFFFFFF"
```

Purpose	Extract the most significant (high-order) portion of an integer class value.
Syntax	<i>result</i> = HI(<i>type</i> , <i>value</i>)
Remarks	<p>The value returned by HI is unsigned if <i>type</i> is BYTE, WORD, or DWORD, and signed if <i>type</i> is INTEGER or LONG. <i>value</i> may be up to twice the size of the data type specified by <i>type</i>. In the following example, <i>n</i> may be up to a 16-bit value (twice the size of a BYTE):</p> <pre>b = HI (BYTE, n)</pre>
Restrictions	HI replaces HIBYT , HIWRD , and HIINT . Note that those functions may not be supported in future versions of Classic PowerBASIC, so update your code to use the new syntax.
See also	LO , MAK

HIBYT function

[Top](#) [Previous](#) [Next](#)

Purpose	<p>Extract the most significant (high-order) byte from an Integer or Word value, and return it as an unsigned Byte value.</p> <p>HIBYT has been superceded by the HI function, although HIBYT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre><i>bResult?</i> = HIBYT(<i>sixteenbitvalue</i>)</pre>
Remarks	<p>The value returned by HIBYT is always unsigned, regardless of the sign of the argument. Effectively, this function provides the same result as PEEKing at the second byte of the argument. Thus, when HIBYT is used on a Long-integer or Double-word, the result is actually the high byte of the low Word of the value.</p>
See also	HI , LO , MAK

Purpose	Extract the most significant (high-order) Word from a Long-integer or Double-word (DWORD) value, and return it as a signed Integer value. HIINT has been superceded by the HI function, although HIINT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.
Syntax	<pre><i>iResult</i>% = HIINT(<i>thirtytwobitvalue</i>)</pre>
Remarks	The value return by HIINT is always signed, regardless of the sign of the argument.
See also	HI

Purpose	Extract the most significant (high-order) Word from a Long-integer or Double-word (DWORD) value and return it as an unsigned Word value. HIWRD has been superceded by the HI function, although HIWRD remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.
Syntax	<code><i>wResult??</i> = HIWRD(<i>thirtytwobitvalue</i>)</code>
Remarks	The value return by HIWRD is always unsigned, regardless of the sign of the argument.
See also	HI

Purpose	Translate a host name into a corresponding IP address.
Syntax	<pre>HOST ADDR [<i>hostname\$</i>] TO <i>ip&</i> HOST ADDR(<i>index&</i>) TO <i>ip&</i></pre>
Remarks	<p><i>hostname\$</i> is the name of a computer on the network or a domain name such as "powerbasic.com". If <i>hostname\$</i> is zero-length or not specified, the primary IP address of the current computer is returned.</p> <p><i>ip&</i> receives the IP address of the specified host name. <i>ip&</i> may be a REGISTER or memory variable.</p> <p>It is possible for a computer to have more than one IP address. For example, if you have a network card in your computer, and you are dialed into the Internet using a modem, your computer will have two IP addresses. By using the indexed form of the statement:</p> <pre>HOST ADDR(<i>index&</i>) TO <i>ip&</i></pre> <p>you can retrieve the first IP address with <i>index&</i> = 1, the second with <i>index&</i> = 2, etc. If, on return, <i>ip&</i> contains zero (0), there are no further IP addresses to retrieve on that computer.</p> <p>A numeric IP address can be easily converted to a dotted IP address string with the following code:</p> <pre>DIM p AS BYTE PTR HOST ADDR "localhost" TO <i>ip&</i> p = VARPTR(<i>ip&</i>) a\$ = USING\$("#_#_#_", @p, @p[1], @p[2], @p[3]) ' returns "127.0.0.1"</pre>
Restrictions	In order to obtain the IP address of the current computer, you must have at least one socket open, or you must first obtain the name of the computer by using the HOST NAME statement.
See also	HOST NAME , TCP and UDP Communications , TCP OPEN , UDP OPEN
Example	<pre>HOST ADDR "powerbasic.com" TO <i>ip&</i> ' Primary IP FUNCTION HowManyIPs() AS LONG DIM p AS BYTE PTR RESET <i>index&</i> DO HOST ADDR(<i>index&</i>+1) TO <i>ip&</i> IF ISTRUE <i>ip&</i> THEN INCR <i>index&</i> p = VARPTR(<i>ip&</i>) a\$ = USING\$("#_#_#_", @p, @p[1], @p[2], @p[3]) END IF LOOP UNTIL <i>ip&</i> = 0 FUNCTION = <i>index&</i> END FUNCTION</pre>

HOST NAME statement

[Top](#) [Previous](#) [Next](#)

Purpose	Translate an IP address into a corresponding host name.
Syntax	<code>HOST NAME [<i>ip&</i>] TO <i>hostname\$</i></code>
Remarks	<p><i>ip&</i> is the IP address you want to look up. If <i>ip&</i> is zero (0) or not specified, the name of the current computer is returned. <i>hostname\$</i> receives the name of the host corresponding to the IP address.</p> <p>In order to translate an IP address into an Internet domain name, your computer will need to be connected to a DNS server on the Internet or local Intranet.</p>
See also	HOST ADDR , TCP and UDP Communications , TCP OPEN , UDP OPEN
Example	<pre>HOST ADDR "powerbasic.com" TO ip& HOST NAME ip& TO hostname\$ CALL ShowResult(hostname\$, ip&)</pre>

Purpose	Sets and returns additional information about certain Dispatch Status Codes for the OBJRESULT function.																					
Syntax	<pre>info& = IDISPINFO.CODE info& = IDISPINFO.CONTEXT info\$ = IDISPINFO.DESC\$ info\$ = IDISPINFO.HELP\$ info\$ = IDISPINFO.SOURCE\$ IDISPINFO.CLEAR IDISPINFO.SET code& [, source\$, desc\$, help\$, context&]</pre>																					
Remarks	<div><div><div>GET Properties</div></div></div>																					
IDISPINFO.CODE	<div><div>When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a long integer value which represents a more specific error code. If the value is less than 65536, it is known as a WCODE, which is usually defined by the application when found in 32-bit or 64-bit Windows. Much more common are the larger values known as an SCODE. These are usually defined by Windows, although application defined values are allowed. The most common are:</div><div><table><tr><td>%E_UNEXPECTED</td><td>= &H8000FFFF&</td></tr><tr><td>%E_NOTIMPL</td><td>= &H80004001&</td></tr><tr><td>%E_NOINTERFACE</td><td>= &H80004002&</td></tr><tr><td>%E_POINTER</td><td>= &H80004003&</td></tr><tr><td>%E_ABORT</td><td>= &H80004004&</td></tr><tr><td>%E_FAIL</td><td>= &H80004005&</td></tr><tr><td>%E_ACCESSDENIED</td><td>= &H80070005&</td></tr><tr><td>%E_HANDLE</td><td>= &H80070006&</td></tr><tr><td>%E_OUTOFMEMORY</td><td>= &H8007000E&</td></tr><tr><td>%E_INVALIDARG</td><td>= &H80070057&</td></tr></table></div></div>		%E_UNEXPECTED	= &H8000FFFF&	%E_NOTIMPL	= &H80004001&	%E_NOINTERFACE	= &H80004002&	%E_POINTER	= &H80004003&	%E_ABORT	= &H80004004&	%E_FAIL	= &H80004005&	%E_ACCESSDENIED	= &H80070005&	%E_HANDLE	= &H80070006&	%E_OUTOFMEMORY	= &H8007000E&	%E_INVALIDARG	= &H80070057&
%E_UNEXPECTED	= &H8000FFFF&																					
%E_NOTIMPL	= &H80004001&																					
%E_NOINTERFACE	= &H80004002&																					
%E_POINTER	= &H80004003&																					
%E_ABORT	= &H80004004&																					
%E_FAIL	= &H80004005&																					
%E_ACCESSDENIED	= &H80070005&																					
%E_HANDLE	= &H80070006&																					
%E_OUTOFMEMORY	= &H8007000E&																					
%E_INVALIDARG	= &H80070057&																					
IDISPINFO.CONTEXT	<div><div>When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a long integer value which is the context of the topic within the help file (IDISPINFO.HELP\$). This property is only valid if IDISPINFO.HELP returns a valid string.</div></div>																					
IDISPINFO.DESC\$	<div><div>When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing a textual, human-readable description of the status. It is intended to be read by the customer. If no description is available, a null, zero-length string is returned.</div></div>																					
IDISPINFO.HELP\$	<div><div>When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing drive, path,</div></div>																					

and filename of a Help File with more information about this particular status code. If no help is available, a null, zero-length string is returned.

- IDISPINFO.PARAM When OBJRESULT is either %DISP_E_PARAMNOTFOUND or %DISP_E_TYPEREMISMATCH, this Get Property returns a long integer value which represents the parameter number of the first parameter which failed to match the requirements needed. The value is indexed to zero, which is the standard numbering convention for Dispatch parameters. The first parameter is 0, the second is 1, and so on.
- IDISPINFO.SOURCE\$ When OBJRESULT is %DISP_E_EXCEPTION, this Get Property returns a string containing a textual, human-readable description of the source of the exception. Typically, this will be the application name. If no source is available, a null, zero-length string is returned.

SET Properties

- IDISPINFO.CLEAR Clears all properties which may have been set by prior execution of IDISPINFO.SET in this thread.
- IDISPINFO.SET This statement may be executed in a [METHOD](#) or [PROPERTY](#) on a Dual Interface, so that the calling code can obtain additional information about Dispatch exception conditions. These five data items are passed back to the caller in the EXCEPINFO structure, so that they can be retrieved with IDISPINFO GET Properties, or other functions in other programming languages. This data is only available when using the Dispatch interface. It is unavailable to [Direct Methods](#). The first parameter (*code&*) is required, and must be identical to the value which you return with METHOD OBJRESULT or PROPERTY OBJRESULT. The actual OBJRESULT will then be changed to %DISP_E_EXCEPTION, so that the caller will know that this data must also be retrieved. Note that the last four parameters are optional.

Restrictions You should only execute the GET PROPERTY methods listed above when OBJRESULT returns the specified status code. In any other case,

IDISPINFO Get Properties will return zero or a null string.

See also

[OBJECT](#), [OBJRESULT](#), [OBJRESULT\\$](#), [What is an object, anyway?](#), [What is DISPATCH?](#)

Example

```
IDISPINFO.SET &H80004040, "MyApp", "Valve stem error", "C:\Help.chm",  
1773
```

Purpose	Test a condition and execute one or more program statements only if the condition is met.
Syntax	<pre>IF <i>integer_expression</i> THEN {<i>sub</i> <i>label</i> <i>statements</i>} [ELSE {<i>sub</i> <i>label</i> <i>statements</i>}]</pre>
Remarks	<p>If <i>integer_expression</i> is TRUE (evaluates to a non-zero value), the statements following THEN are executed, and the statements following the optional ELSE are not executed. If <i>integer_expression</i> is FALSE (zero), the statements following THEN are not executed, and the statements following the optional ELSE are executed. If the ELSE clause is omitted, execution continues with the next line of the program, provided <i>integer_expression</i> evaluates to FALSE.</p> <p><i>integer_expression</i> will often be a result returned by a relational operator as shown here:</p> <pre>IF Income > Expenses THEN x\$ = "OK!" ELSE x\$ = "Uh-oh"</pre> <p><i>integer_expression</i> can also be a boolean value. For example, your program could set the variable <i>BeepOn</i> to 1 (or any non-zero value) if audible beeps are requested, and to 0 if not, then use an IF statement to control output:</p> <pre>IF BeepOn THEN BEEP</pre> <p>is equivalent to:</p> <pre>IF BeepOn <> 0 THEN BEEP</pre> <p><i>integer_expression</i> can include the logical operators AND and OR, as in:</p> <pre>IF (a = b) AND (c = d) THEN x\$ = "They are equal"</pre> <p><i>label</i> If a label is specified, the label must appear within the same Sub, Function, Method, or Property as the IF statement. The GOTO keyword is implied by THEN, or can replace THEN:</p> <pre>IF EOF(1) THEN GotFile IF EOF(1) GOTO GotFile</pre> <p><i>proc</i> If <i>proc</i> is specified, it must identify a Sub, Function, Method, or Property. The IF statement and all its associated statements, including those after an ELSE, must appear on the same logical program line. The following is therefore illegal:</p> <pre>IF a < b THEN t = 15 : u = 16 : v = 17 ELSE t = 17 : u = 16 : v = 15</pre> <p>because the compiler treats the ELSE statement as a brand-new statement unrelated to the one above it. If you have more statements than you can fit on one line, you can use the line continuation character, the underscore "_", to spread a single logical line over several physical lines. For example, the following is a legal way of restating the last example:</p>

```
IF a < b THEN t = 15 : u = 16 : v = 17 _
ELSE t = 17 : u = 16 : v = 15
```

A better method of programming long and complex IF/THEN constructs is to use the [IF block](#) statement.

Also note that every statement following the ELSE will be executed if *integer_expression* is FALSE. For example, you might expect the following statement:

```
Taxable = %TRUE
Price = 1.00
Rate = .05
Total = 5.00
IF Taxable THEN Tax = Price * Rate ELSE Tax = 0: Total = Total + Tax
```

to bring *Total* to 5.05, but it won't. The *Total = Total + Tax* statement will only be executed if *Taxable* is FALSE. It's easy to get the correct results using the IF block:

```
IF Taxable THEN
    Tax = Price * Rate
ELSE
    Tax = 0
END IF
Total = Total + Tax
```

Short-Circuit Evaluation

Note that Classic PowerBASIC features short-circuit evaluation of relational expressions using [AND](#) and [OR](#). This optimization means that evaluation of a relational expression in an [IF](#), [IF/END IF](#), [DO/LOOP](#), or [WHILE/WEND](#) is terminated just as soon as it is possible to tell what the result will be. For example:

```
IF LEN(a$) AND MyFunc(a$) THEN CALL ShowText("Ok!")
```

In the above example, if [LEN\(a\\$\)](#) is zero, there is no further need to evaluate the expression, because 0 and *anything* will always be FALSE. So, if [LEN\(a\\$\)](#) is zero, *MyFunc()* is not called at all, and *ShowText()* is not executed.

To give short-circuit optimization an extra boost, AND and OR are treated as a Boolean operator rather than a bitwise operator, and this can sometimes produce unexpected results. For example, consider the following expression:

```
a& = 4
b& = 2
IF a& AND b& THEN CALL ShowText("TRUE") ELSE CALL ShowText("FALSE")
```

Applying the traditional BASIC bitwise evaluation, you would expect to see FALSE displayed because $(4 \text{ AND } 2) = 0$. Due to the short-circuit optimization though, each value is treated as a Boolean, just as if you had written:

```
IF ISTRUE a& AND ISTRUE b& THEN ...
```

If you believe this may be a problem for your particular code, you can

disable the short-circuit evaluation by surrounding the entire conditional expression in parentheses:

```
IF (a& AND b&) THEN CALL ShowText("TRUE") ELSE CALL ShowText("FALSE")
```

The parentheses force the entire expression to be evaluated, so AND reverts to being a bitwise operator.

See also

[CHOOSE](#), [CHOOSE&](#), [CHOOSE\\$](#), [IF block](#), [IIF](#), [IIF&](#), [IIF\\$](#), [MAX](#), [MAX&](#), [MAX\\$](#), [MIN](#), [MIN&](#), [MIN\\$](#), [SWITCH](#), [SWITCH&](#), [SWITCH\\$](#), [SELECT](#)

Example

```
IF x > 100 THEN y = 3 ELSE y = 4
```

Purpose Create IF/THEN/ELSE constructs with multiple lines and/or conditions.

Syntax

```
IF integer_expression THEN
    [statements]
[ELSEIF integer_expression THEN
    [statements]]
[ELSE
    [statements]]
END IF
```

Remarks In executing IF blocks, the truth of the *integer_expression* in the initial IF statement is checked first. If it evaluates to FALSE (zero), each of the following ELSEIF statements is examined in order. There can be as many ELSEIF statements as desired. As soon as one is found to be [TRUE](#) (non-zero), Classic PowerBASIC executes the statement(s) *following* the associated THEN and *before* the next ELSEIF or ELSE.

Execution then jumps to the statement just after the terminating END IF without making any further tests. If none of the test expressions evaluates to TRUE, the statement(s) in the ELSE clause (which is optional) are executed.

Note that there must be nothing following the THEN keyword in an IF block; that's how the compiler distinguishes an IF block from a conventional IF statement. There must also be nothing on the same line as the ELSE (except for remarks).

IF blocks can be nested; that is, any of the statements after any of the THENs may contain IF blocks. Although the compiler doesn't care, the clarity of source code is improved by indenting the statements controlled by each test a couple of spaces, as shown in the example.

IF blocks must be terminated with a matching END IF statement. Note that the END IF statement requires a space and the ELSEIF statement does not.

See the [IF](#) statement for notes on Classic PowerBASIC's [Short-circuit evaluation](#) and its possible side effects.

See also [CHOOSE](#), [CHOOSE&](#), [CHOOSE\\$](#), [EXIT](#), [IF](#), [IIF](#), [IIF&](#), [IIF\\$](#), [MAX](#), [MAX&](#), [MAX\\$](#), [MIN](#), [MIN&](#), [MIN\\$](#), [SELECT](#), [Short-circuit evaluation](#), [SWITCH](#), [SWITCH&](#), [SWITCH\\$](#), [SELECT](#)

Example

```
X = (RND * 500) + 1
IF X = 1 THEN
    x$ = "The number is 1"
ELSEIF X = 2 THEN
    x$ = "The number is 2"
ELSE
    IF X < 50 THEN
        x$ = "The number is less than 50"
```



```
ELSEIF X < 100 THEN
    x$ = "Greater than 49 and less than 100"
ELSE
    x$ = "The number is 100 or greater"
END IF
END IF
```

Purpose	Return one of two values based upon a TRUE/FALSE evaluation.
Syntax	<pre>var = IIF(num_expression, truepart, falsepart) var& = IIF&(num_expression, truepart&, falsepart&) var\$ = IIF\$(num_expression, truepart\$, falsepart\$)</pre>
Remarks	<p>IIF expects parts of any numeric type. IIF& expects parts optimized for long integer type. IIF\$ expects parts of string type.</p> <p>If <i>num_expression</i> evaluates to TRUE (non-zero), the <i>truepart</i> is returned, else the <i>falsepart</i> is returned. <i>num_expression</i> is evaluated as a normal Classic PowerBASIC Boolean expression, which offers short-circuit expression evaluation as needed.</p> <p>IIF(1 AND 2, 3, 4) would return the <i>truepart</i> (3) because both terms in <i>num_expression</i> are TRUE, and therefore evaluate to TRUE.</p> <p>To force a bitwise evaluation of <i>num_expression</i>, enclose it in parentheses. For example, IIF\$((1 AND 2), "True", "False") would return "False".</p> <p>IIF% is recognized as a valid synonym for IIF&.</p>
Restrictions	Contrary to the implementation in some other languages, only the selected expression (<i>truepart</i> or <i>falsepart</i>) is evaluated at run-time, not both. This ensures optimum execution speed, as well as the elimination of unanticipated side effects.
See also	CHOOSE , CHOOSE& , CHOOSE\$, SWITCH , SWITCH& , SWITCH\$
Example	<pre>iLOGFONT.lfWeight = IIF&(Weight&, 700&, 400&) Score& = Score& + IIF&(Answer = %FALSE, 0, 10)</pre>

Purpose Create and manage an IMAGELIST object to use with other functions.

Syntax

```
IMAGELIST ADD BITMAP hLst, hBmp [,hMsk] [TO data&]  
IMAGELIST ADD BITMAP hLst, Bmp$ [,Msk$] [TO data&]  
IMAGELIST ADD ICON hLst, hIcn [TO data&]  
IMAGELIST ADD ICON hLst, Icn$ [TO data&]  
IMAGELIST ADD MASKED hLst, hBmp, rgb& [TO data&]  
IMAGELIST ADD MASKED hLst, Bmp$, rgb& [TO data&]  
IMAGELIST GET COUNT hLst TO data&  
IMAGELIST KILL hLst  
IMAGELIST NEW BITMAP|ICON width&, height&, depth&, initial& TO hLst  
IMAGELIST SET OVERLAY hLst, image&, overlay&
```

hBmp Handle of a Bitmap.

hIcn Handle of an Icon.

hLst Handle of the IMAGELIST.

hMsk Handle of a Mask.

data& A [long integer](#) variable to which result data is assigned.

Remarks An IMAGELIST is a structure which contains any number of graphical images, either bitmaps or icons, but not a mixture. All of the images are automatically converted to the type, size, and color depth specified when the IMAGELIST is created. A bitmap (file type *.BMP) is a single color image, while an icon (file type *.ICO) supports transparency by including both a color bitmap and a mask bitmap. The mask bitmap is a monochrome image (one bit per pixel), where each "set" bit describes a pixel which remains transparent. The IMAGELIST structure can best be described as a set, or [array](#), of images. You can retrieve the images individually by index number, or pass the entire IMAGELIST to a control which requires it ([LISTVIEW](#), etc.).

An empty ImageList is first created with IMAGELIST NEW. Images are then added with IMAGELIST ADD, until the structure is complete. If you add an image which is wider than the size specified by *width&*, the image is separated into multiple bitmaps, each of which is added in sequence. When an IMAGELIST is attached to a control like LISTVIEW, it is usually destroyed automatically when the control is destroyed. Consult the control documentation for that information. If not, you must explicitly destroy it with IMAGELIST KILL.

IMAGELIST ADD BITMAP *hLst*, *hBmp* [,*hMsk*] [TO *data&*]

An image is added to the ImageList specified by *hLst*. With this syntax, the image is specified by a handle (*hBmp*), so it must have been loaded into

memory (e.g. with GRAPHIC BITMAP). If the ImageList is an ICON type, a second mask bitmap is also specified by a handle (hMsk). If the TO clause is included, the index position of the first added bitmap (starting with 1) is assigned to the variable designated by data&. If the operation fails, the value 0 is assigned.

IMAGELIST ADD BITMAP *hLst*, *Bmp\$* [,*Msk\$*] [TO *data&*]

An image is added to the ImageList specified by *hLst*. With this syntax, the image is specified by a name string (*Bmp\$*), which is the name of an embedded [resource](#) or a disk file. If the name string contains a period, it is presumed to be a disk file. Otherwise, an attempt is made to load it from a resource - if not found, it is then presumed to be a disk file. If the image name is a numeric resource, it should be described with a leading pound sign ("#12345"). If the ImageList is an ICON type, a second mask bitmap is also specified by a handle (hMsk). If the TO clause is included, the index position of the first added bitmap (starting with 1) is assigned to the variable designated by data&. If the operation fails, the value 0 is assigned.

IMAGELIST ADD ICON *hLst*, *hIcn* [TO *data&*]

An icon is added to the ImageList specified by *hLst*. With this syntax, the icon is specified by a handle (*hIcn*), so it must have been loaded into memory (e.g. with the WinApi LoadIcon. If the TO clause is included, the index position of the first added icon (starting with 1) is assigned to the variable designated by *data&*. If the operation fails, the value 0 is assigned.

IMAGELIST ADD ICON *hLst*, *Icn\$* [TO *data&*]

An icon is added to the ImageList specified by *hLst*. With this syntax, the icon is specified by a name string (*Icn\$*), which is the name of an embedded resource or a disk file. If the name string contains a period, it is presumed to be a disk file. Otherwise, an attempt is made to load it from a resource - if not found, it is then presumed to be a disk file. If the icon name is a numeric resource, it should be described with a leading pound sign ("#12345"). If the TO clause is included, the index position of the first added bitmap (starting with 1) is assigned to the variable designated by *data&*. If the operation fails, the value 0 is assigned.

IMAGELIST ADD MASKED *hLst*, *hBmp*, *rgb&* [TO *data&*]

A bitmap is added to the icon ImageList specified by *hLst*. With this syntax, the bitmap is specified by a handle (*hBmp*), so it must have been loaded into memory (e.g. with GRAPHIC BITMAP). The parameter *rgb&* specifies

the [RGB](#) color used in the bitmap to specify transparent pixels. Each pixel of that color is changed to the color black, and a mask bitmap is created to describe the transparent pixels. If the TO clause is included, the index position of the first added icon (starting with 1) is assigned to the variable designated by *data&*. If the operation fails, the value 0 is assigned.

IMAGELIST ADD MASKED *hLst*, *Bmp\$*, *rgb&* [TO *data&*]

A bitmap is added to the icon ImageList specified by *hLst*. With this syntax, the bitmap is specified by a name string (*Bmp\$*), which is the name of an embedded resource or a disk file. If the name string contains a period, it is presumed to be a disk file. Otherwise, an attempt is made to load it from a resource - if not found, it is then presumed to be a disk file. If the image name is a numeric resource, it should be described with a leading pound sign ("#12345"). The parameter *rgb&* specifies the RGB color used in the bitmap to specify transparent pixels. Each pixel of that color is changed to the color black, and a mask bitmap is created to describe the transparent pixels. If the TO clause is included, the index position of the first added bitmap (starting with 1) is assigned to the variable designated by *data&*. If the operation fails, the value 0 is assigned.

IMAGELIST GET COUNT *hLst* TO *data&*

The number of images in the IMAGELIST is retrieved, and assigned to the [long integer](#) variable specified by *data&*.

IMAGELIST KILL *hLst*

The IMAGELIST specified by *hLst* is destroyed. All allocated memory and resources are released.

IMAGELIST NEW BITMAP|ICON *width&*, *height&*, *depth&*, *initial&* TO *hLst*

A new ImageList structure is created. If you specify BITMAP, each image you add will be stored as a single bitmap. If you specify ICON, each image you add will be stored as two bitmaps in order to support transparent areas. The parameters *width&* and *height&* specify the size of each image in pixels. The *depth&* parameter specifies the color depth in bits per pixel (4,8,16,24,32). A depth of 4 offers 16 colors, 8 offers 256 colors, etc. The *initial&* parameter specifies the initial size of the ImageList. While it can grow beyond this number, it is most efficient to allocate space accurately at the time of creation. The variable *hLst* receives the handle of the newly created ImageList, or zero if the operation failed.

IMAGELIST SET OVERLAY *hLst*, *image&*, *overlay&*

The image specified by the index number *image*& is declared to be an overlay image. The *overlay*& parameter must be in the range of 1 to 15, and it is used later to retrieve and/or specify this particular overlay image.

See also

[GRAPHIC BITMAP LOAD](#), [GRAPHIC BITMAP NEW](#), [GRAPHIC IMAGELIST](#), [LISTVIEW](#), [TAB SET IMAGELIST](#), [TREEVIEW](#), [XPRINT IMAGELIST](#)

Purpose The IMP operator works as both a logical and a bitwise [arithmetic operator](#).

Syntax $p \text{ IMP } q$

Remarks **IMP as a logical operator**

The IMP operator returns FALSE (zero) if and only if its first operand is TRUE (non-zero), and its second operand is FALSE. In all other cases, it returns TRUE.

Truth table		
x	y	x IMP y
T	T	T
T	F	F
F	T	T
F	F	T

See also [Arithmetic Operators](#), [AND](#), [EQV](#), [ISFALSE](#), [ISTRUE](#), [NOT](#), [OR](#), [XOR](#)

Purpose	Increment a variable by 1; increment a pointer by the size of its target; or increment the target of a numeric pointer by 1.
Syntax	<code>INCR <i>variable</i></code>
Remarks	<p><i>variable</i> can be a numeric or a pointer variable. When INCR is used with a numeric variable, 1 is added to the numeric variable.</p> <p>When INCR is used with a pointer variable itself, the value of the pointer is incremented by the size of the pointer's target.</p> <p>When INCR is used on a numeric pointer's target (i.e., INCR @IntPtr) the value of the target is incremented by 1.</p> <p>For example, given a pointer to an array where the pointer targets element 1000, applying INCR to the pointer would result in the pointer aiming at element 1001. The actual address held by the pointer would have risen by two, because the target of the pointer is an Integer which is two bytes wide.</p> <p>Conversely, if INCR was used on the target of this Integer pointer, the value of the element itself would be incremented by 1.</p>
See also	DECR , LET
Example	<pre>DIM x&, LongPtr AS LONG POINTER INCR x& INCR LongPtr INCR @LongPtr</pre>

Purpose	Load variables with data from a sequential file .
Syntax	<code>INPUT #<i>filenum</i>&, <i>variable_list</i></code>
Remarks	<p><i>filenum</i>& is the file number, or variable containing a file number, given when the file was opened. <i>variable_list</i> is a comma-delimited sequence of one or more string or numeric variables. When the INPUT# statement reads an unquoted data item from a file, it removes leading and trailing spaces. If spaces are significant, place quotes around the file data, either directly or by using WRITE# to save the data to disk. Please note that data to be quoted should not contain embedded quotes.</p> <p>The data in the file must match the type(s) of the variable(s) defined in the INPUT# statement. The file data should be separated by commas with a carriage return at the end. The WRITE# statement is ideal for creating such files.</p> <p>INPUT# also supports fixed-length and ASCIIZ string variables; however, data that is longer than the string is truncated to fit into the string. Dynamic strings receive the data without truncation. UDT variables may not be used, although fixed-length and ASCIIZ UDT member variables are supported.</p>
See also	LINE INPUT# , PRINT# , WRITE#
Example	<pre>SUB MakeFile ' MakeFile opens a sequential file for output. ' Using WRITE#, it writes lines of different ' data types to 'the file. OPEN "INPUT#.DTA" FOR OUTPUT AS #1 StringVariable\$ = "I'll be back." IntegerVar% = 1000 FloatingPoint! = 30000.12 ' Write a line of text to the sequential file. WRITE #1, StringVariable\$, IntegerVar%, FloatingPoint! CLOSE #1 END SUB SUB ReadFile ' This procedure opens a sequential file for ' input. Using INPUT# it reads lines of ' different data types from the file. OPEN "INPUT#.DTA" FOR INPUT AS #1 RESET StringVariable\$ RESET IntegerVar% RESET FloatingPoint! ' Read a line of text from the sequential file. INPUT #1, StringVariable\$, IntegerVar%, FloatingPoint! CLOSE #1 END SUB</pre>

INPUTBOX\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	INPUTBOX\$ displays a dialog box containing a prompt. INPUTBOX\$ waits for the user to enter text, and accept or cancel the dialog. INPUTBOX\$ returns the contents of the text box.
Syntax	<code>sResult\$ = INPUTBOX\$(prompt\$ [, title\$], default\$) [, xpos%, ypos%]</code>
Remarks	<p><i>Prompt\$</i> is the text prompt displayed in the Inputbox dialog.</p> <p><i>Title\$</i> is the caption for the Inputbox dialog and is optional.</p> <p><i>Default\$</i> is the default result text displayed in the edit section of the Inputbox dialog, and is optional.</p> <p><i>xpos%</i> and <i>ypos%</i> specify the location on the screen to display the Inputbox, in dialog units. If these are not specified, the Inputbox dialog is centered on the screen.</p>
Restrictions	The returned string value is limited to 255 characters.
See also	MSGBOX function , MSGBOX statement
Example	<pre>sResult\$ = INPUTBOX\$("Enter your Name", , "Jane Doe")</pre>

Purpose	Declare INSTANCE variables which are unique to each object .
Syntax	<pre>INSTANCE variable[()] [AS type] [, variable[()]] INSTANCE variable[()] [, variable[()]] [, ...] AS type</pre>
Remarks	<p>INSTANCE statements are used to declare instance variables for an object. A unique set of instance variables is created for every new object, which may only be referenced from within that object. INSTANCE statements may only be placed at the beginning of a CLASS/END CLASS block, preceding all INTERFACE blocks.</p> <p>INSTANCE will optionally accept a list of variables, each of which are defined by the descriptor which follows it:</p> <pre>INSTANCE x as integer, y as long</pre> <p>INSTANCE will also accept a list of variables, all of which are defined by the single descriptor at the end of the list;</p> <pre>INSTANCE aaa, bbb, ccc AS INTEGER INSTANCE vptr, aptr() AS LONG PTR</pre> <p>To declare an array as an instance variable, use an empty set of parentheses in the variable list: You can then use the DIM statement to dimension the array.</p>
See also	GLOBAL , INTERFACE (Direct) , LOCAL , STATIC , THREADED , What is an object, anyway?

Purpose	Search a string for the existence of a second string.
Syntax	<code>y& = INSTR([Position&], Main\$, [ANY] Match\$)</code>
Remarks	<p>INSTR returns the position of <i>Match\$</i> within <i>Main\$</i>. The return value is indexed to one, while zero means "not found".</p> <p><i>Position&</i> specifies the character position to begin the search. If <i>Position&</i> is one or greater, <i>Main\$</i> is searched left to right. The value one starts at the first character, two the second, etc. If <i>Position&</i> is -1 or less, <i>Main\$</i> is searched from right to left. The value -1 starts at the last character, -2 the second to last, etc. If <i>Position&</i> is not given, the default value of +1 is assumed.</p> <pre>x& = INSTR("xyz", "y") ' returns 2 x& = INSTR("xyz", "a") ' returns 0 a\$ = "My Dog" : b\$ = " " x& = INSTR(a\$, b\$) ' returns 3</pre> <p>It is important to note that in all cases, even when <i>Position&</i> is negative, the return value of INSTR() is the absolute position of the match, from left to right, starting with the first character.</p>
ANY	<p>If the ANY keyword is included, <i>Match\$</i> specifies a list of single characters. INSTR searches for each of these characters individually. As soon as any one of these characters is found, INSTR returns the position of the match.</p> <pre>x& = INSTR(-2, "efcdef", ANY "ef") returns a result of 5</pre> <p>INSTR is case-sensitive, meaning that upper-case and lower-case letters must match exactly in <i>Match\$</i> and <i>Main\$</i>.</p>
Restrictions	<p>Special search terms are evaluated in this sequence:</p> <ol style="list-style-type: none">1. If <i>Position&</i> is zero, or beyond the length of <i>Main\$</i>, the value zero is returned.2. If <i>Main\$</i> is null, the value zero is returned.3. If <i>Match\$</i> is null, the absolute <i>Position&</i> value (default of 1) is returned.
See also	EXTRACT\$, LCASE\$, LEFT\$, LTRIM\$, MID\$, RIGHT\$, RTRIM\$, TALLY , TRIM\$, UCASE\$, VERIFY
Example	<pre>' x\$ = first command-line argument, assuming spaces, commas, ' periods, and tabs are valid delimiters. IF INSTR(COMMAND\$, ANY " ,." + CHR\$(9)) > 0 THEN x\$ = "There is more than one command-line argument" ELSE x\$ = "There is at most one command-line argument" END IF</pre>

INT function

Purpose	Convert a numeric expression to an integral value.																
Syntax	<i>y</i> = INT(<i>numeric_expression</i>)																
Remarks	INT rounds <i>numeric_expression</i> to the largest integer-class value that is less than or equal to <i>numeric_expression</i> .																
See also	CEIL , CINT , FIX , FRAC , ROUND																
Example	<pre>DIM X AS SINGLE, Y AS LONG FOR X = -1.1 TO 2.1 STEP .5 Y = INT(X) NEXT X</pre>																
Result	<table><tr><th>X</th><th>Y</th></tr><tr><td>-1.1</td><td>-2</td></tr><tr><td>-0.6</td><td>-1</td></tr><tr><td>-0.1</td><td>-1</td></tr><tr><td>0.4</td><td>0</td></tr><tr><td>0.9</td><td>0</td></tr><tr><td>1.4</td><td>1</td></tr><tr><td>1.9</td><td>1</td></tr></table>	X	Y	-1.1	-2	-0.6	-1	-0.1	-1	0.4	0	0.9	0	1.4	1	1.9	1
X	Y																
-1.1	-2																
-0.6	-1																
-0.1	-1																
0.4	0																
0.9	0																
1.4	1																
1.9	1																

INTERFACE / END INTERFACE Block (Direct) **New!**

[Top](#) [Previous](#)
[Next](#)

Purpose Declare a direct [object](#) interface and its member [Methods/Properties](#).

Syntax

```
INTERFACE interfacename [$GUID] [AS EVENT] [AS HIDDEN]
  {METHOD | PROPERTY} name [(arguments)] [AS type]
END INTERFACE
```

Remarks The first line in an Interface Block must be an [INHERIT](#) statement. INHERIT specifies the [base class](#) or the user interface upon which this new interface is built. It defines the base methods available, the optional user methods which are available, and the calling conventions which will apply. In the current version of Classic PowerBASIC, the following may be used:

INHERIT IUnknown

This defines a Custom Interface with only direct access to the interface [methods](#). [OBJRESULT](#) (an HRESULT value) is not supported. Return values are typically passed in CPU/FPU registers, just like a user defined [FUNCTION](#). This is the format most often used for internal objects, as it offers access to more data types than the other forms. You may substitute the word CUSTOM for IUNKNOWN, as they are synonyms.

INHERIT IAutomation

This defines an Automation Interface with only direct access to the interface methods. OBJRESULT (an HRESULT value) is always supported. Return values are passed as a hidden, last parameter (automatically, by Classic PowerBASIC). Parameters and return values are limited to [COM](#) data types. A [User Defined Type](#) used as a return value or parameter will be converted to a BYVAL DWORD. This is the format most often used for [COM objects](#) which do not require access to the IDispatch interface. You may substitute the word AUTOMATION for IAutomation, as they are synonyms.

INHERIT IDispatch

This defines a Dual Interface, which offers both direct access and Dispatch access to the interface methods. OBJRESULT (an HRESULT value) is always supported. This interface inherits from IAutomation, so the calling conventions are identical to IAutomation when used for direct access. You may substitute the word DUAL for IDISPATCH, as they are synonyms.

INHERIT <UserClass>, <UserInterface>

This defines an inherited user-written interface, so the new interface implements the base class IUnknown, IDispatch, etc.) and all of the Methods and Properties, as well. It's necessary to specify both the class and the interface name to be inherited, because it's possible to have multiple implementations of any particular interface.

INTERFACE / END INTERFACE statements enclose the METHOD and PROPERTY definitions which constitute a class. There are two forms of the INTERFACE / END INTERFACE block. When it appears outside of a [CLASS](#) block, it is simply a declaration of the interface, much like [DECLARE](#) statements are used for functions:

```
INTERFACE name [$GUID] [AS EVENT]
  INHERIT IUnknown
  METHOD MyMethod(xyz AS LONG)
  PROPERTY GET MyProp() AS STRING
END INTERFACE
```

The above form is used to declare an interface which is implemented in another .EXE or .DLL, but will be accessed here through COM services. It may also be used for added self-documentation of internal classes. If it appears within a CLASS block, it is the implementation of the Methods/Properties for the Class. The interface implementation must precisely match any prior interface declaration.

```
CLASS name [$GUID] [AS COM]
  INTERFACE name [$GUID] [AS HIDDEN]
    INHERIT iUnknown
    METHOD MyMethod(xyz AS LONG)
      [statements]
    END METHOD
    PROPERTY GET MyProp() AS STRING
      [statements]
    END PROPERTY
  END INTERFACE
END CLASS
```

The name and optional [\\$GUID](#) are supplied by the programmer to uniquely identify the interface. The first entry in every INTERFACE block must be the base class upon which it is built. Every interface must ultimately inherit from IUnknown, which is a requirement.

By default, a class is considered private, so that the methods are accessible only from within the EXE or DLL where it is defined. The AS COM attribute to the CLASS statement makes the class available externally, to virtually any process which is COM-aware.

The optional AS HIDDEN attribute to the INTERFACE statement prevents the interface from being documented when the type library is created. When marked as hidden, any and all uses of the interface are hidden, even if they appear in multiple classes.

With an internal class, the \$GUID on CLASS and INTERFACE statements

may be freely omitted, as Classic PowerBASIC can readily identify them by name. With a published COM class, you should insert a specific [GUID](#) of your choice. If omitted, a random GUID will be created by the compiler, but it will change every time you compile the program. This will be difficult to synchronize with other programs which wish to identify and access your object.

The following code defines a dual interface whose methods are available for both direct access and Dispatch access. This is the form you will typically use for COM objects, since it offers the best compatibility with varied client modules.

```
INTERFACE DispatchIface
    INHERIT IDispatch
    METHOD MethodDef()
        [statements]
    END METHOD
END INTERFACE
```

You should note that the IDispatch interface itself inherits from IUnknown, so that both interfaces are ultimately available. As an additional required base class, the IDispatch declaration is built into the Classic PowerBASIC Compiler.

Every method and property in a dual interface needs a positive, [long integer](#) value to identify it. That integer value is known as a DispID (Dispatch ID), and it's used internally by COM services to call the correct function on a Dispatch interface. You can specify a particular DispID by enclosing it in angle brackets immediately following the Method/Property name in an Interface definition block.

```
INTERFACE DualIface
    INHERIT IDispatch
    METHOD MethodOne <76> ()
    METHOD MethodTwo <77> ()
END INTERFACE
```

If you don't specify a DispID, Classic PowerBASIC will assign a random value for you. This is fine for internal objects, but may cause a failure for published COM objects, as the DispID could change each time you compile your program. It is particularly important that you specify a DispID for each Method/Property in a COM Event Interface.

Inherited User-Written Interfaces

Classic PowerBASIC offers Implementation Inheritance of user-written interfaces. That is, an interface can inherit all of the code in the methods and properties of a selected interface. You can then add additional methods and properties to the new interface. When you inherit a user-written interface, you must specify both the class name and the interface name, since COM allows you to have multiple implementations of any particular interface.

You can override an inherited method or property by coding a replacement which is preceded by the word **OVERRIDE**. It's possible to one or many override procedures, but they must appear in the same sequence as the ones they replace.

```
CLASS MyClass
  INTERFACE MyFace
    INHERIT IDispatch
    METHOD aaa()
      ' code...
    END METHOD
    METHOD bbb()
      ' code...
    END METHOD
    METHOD ccc()
      ' code...
    END METHOD
    METHOD ddd()
      ' code...
    END METHOD
  END INTERFACE
END CLASS

CLASS TheClass
  INTERFACE TheFace
    INHERIT MyClass, MyFace
    OVERRIDE METHOD bbb()
      ' new code...
    END METHOD
    OVERRIDE METHOD ddd()
      ' new code...
    END METHOD
    METHOD xxx()
      ' code...
    END METHOD
  END INTERFACE
END CLASS
```

Note that in the above example, the new interface "TheFace" first inherits all four methods from "MyFace" (aaa,bbb,ccc,ddd). However, because of the **OVERRIDE** statements, both bbb() and ddd() are replaced by newer versions of the methods. Because of the nature of Virtual Function Tables, the **OVERRIDE** procedures must remain in the original sequence. That is, bbb() must precede ddd(), and both must precede any added methods, such as xxx().

Because of the nature of code replacement necessary in implementation inheritance, the interface to be inherited must always physically precede the new, child interface.

See also

[INTERFACE \(IDBind\)](#), [CLASS](#), [INSTANCE](#), [ISINTERFACE](#), [LET \(with Objects\)](#), [ME](#), [METHOD](#), [MYBASE](#), [PROPERTY](#), [What does an Interface look like?](#), [What is inheritance?](#)

INTERFACE/END INTERFACE block (IDBind)

[Top](#) [Previous](#)
[Next](#)

IMPROVED

Purpose	Declare a dispatch interface and its member Methods/Properties for the purposes of IDBinding to a Dispatch COM interface.
Syntax	<pre>INTERFACE IDBIND <i>interfacename</i> MEMBER {CALL GET SET LET} <i>membername</i> <<i>dispid</i>> ([[OPTIONAL [IN OUT INOUT]] <i>paramname</i> <<i>dispid</i>> [AS <i>type</i>] [,...]]) [AS {<i>vartype</i> <i>interface</i>}] [...] END INTERFACE</pre>
Remarks	<p>In order to provide IDBinding services, Classic PowerBASIC must be able to pre-construct the references to the DISPATCH COM interface members at compile-time. Without an interface definition block, only late-binding at run-time would be possible. Late-binding is less efficient than IDBinding. You may list every Method/Property in the interface, or just the ones that are referenced in the code. They can appear in any sequence. Member names may contain (normally) reserved keywords such as INPUT or KILL, etc</p> <p>The most important aspect of an interface block is that it clearly associates a <i>dispid</i> with the each Method/Property name. Named parameters in the <i>paramname</i> list also require an appropriate <i>dispid</i> value, as does any Property which returns an object to be used in a nested object reference. All <i>dispid</i> values must be enclosed in angle brackets (< and >), and may be expressed as hexadecimal or decimal numeric literals.</p> <p>You can look up the <i>dispid</i> values of COM servers using an Object Browser, or by reading your object documentation. You can even insert additional information about the types and return value for your own reference, even though the compiler does not use them.</p> <p>Previous versions of Classic PowerBASIC compilers used an older style syntax of "INTERFACE DISPATCH <i>interfacename</i>" for this structure. It was updated to better reflect the nature of the description. While the older syntax will be recognized in this version, we suggest you update the word DISPATCH to IDBIND soon.</p>
Restrictions	<p>If the compiler cannot resolve the interface name definition specified in a DIM or LET statement, a compile-time error is generated accordingly.</p> <p><i>interfacename</i> must not be a Classic PowerBASIC keyword. If a keyword conflict arises, the addition of an arbitrary prefix is acceptable. For example, INTERFACE IDBIND Shell() could be changed to INTERFACE</p>

IDBIND MyShell() and Classic PowerBASIC will still resolve the interface correctly.

Method/Property *membername* items may freely use Classic PowerBASIC keywords without concern for conflicts with normal code syntax. For example, MEMBER CALL Open() is a valid syntax for an interface method.

See also

[DIM](#), [ID Binding](#), [INTERFACE \(Direct\)](#), [ISINTERFACE](#), [LET \(with Objects\)](#), [Late Binding](#), [LET \(with Variants\)](#), [OBJACTIVE](#), [OBJECT](#), [OBJPTR](#), [OBJRESULT](#), [PROGID\\$](#), [What is an object, anyway?](#), [What is DISPATCH?](#)

Example

```
INTERFACE IDBIND IAPPUser
  MEMBER CALL DELETE<&H1>()
  MEMBER GET Name<&H2>() AS STRING
  MEMBER LET Name<&H2>() 'Param Type As String
  MEMBER LET Password<&H3>() 'Param Type As String
  MEMBER GET ReadOnly<&H4>() AS LONG
  MEMBER LET ReadOnly<&H4>() 'Param Type As Long
  MEMBER GET ProjectRights<&H5>(OPTIONAL IN Project AS STRING<&H0>) AS
LONG
  MEMBER LET ProjectRights<&H5>(OPTIONAL IN Project AS STRING<&H0>)
  MEMBER CALL RemoveProjectRights<&H6>(IN Project AS STRING<&H0>)
END INTERFACE

INTERFACE IDBIND IAPPItems
  MEMBER GET Count<&H1>() AS LONG
  MEMBER GET Item<&H0>(IN sItem AS VARIANT<&H0>) AS IAPPItem
END INTERFACE

DIM oApp AS IAPPUser
LET oApp = NEW IAPPUser IN "com.server.0"
```

ISFALSE and ISTRUE operators

Purpose	Return the logical truth or falsity of a given expression.
Syntax	<code>ISFALSE <i>expr</i></code> <code>ISTRUE <i>expr</i></code>
Remarks	ISTRUE returns -1 (TRUE) when <i>expr</i> evaluates as non-zero; otherwise, it returns zero (FALSE). ISFALSE returns -1 when <i>expr</i> evaluates as 0 (FALSE); otherwise, it returns zero.

Truth table		
operator	<i>expr</i>	Result
ISTRUE	= 0	0
ISTRUE	<> 0	-1
ISFALSE	= 0	-1
ISFALSE	<> 0	0

Classic PowerBASIC's [NOT](#) operator serves a double duty: it returns the one's-complement of an integer class expression and "reverses" the value of a TRUE/FALSE (Boolean) expression. Usually, these two functions do not conflict, but since Classic PowerBASIC accepts any non-zero value as TRUE, the following condition can arise:

```
test1 = 0           ' test1 is FALSE (zero)
IF NOT test1 THEN   ' TRUE (-1 is non-zero)
[statements]

test2 = 1           ' test2 is TRUE (1 is non-zero)
IF NOT test2 THEN   ' still TRUE (-2 is non-zero)
[statements]
```

In this case, NOT does not reverse the TRUE/FALSE value of test2. ISFALSE ensures that the test is performed exactly as you would expect:
test2 = 1 ' test2 is TRUE (non-zero)

```
IF ISFALSE test2 THEN ' ISFALSE detects test2 is
[statements]          ' TRUE so the IF test fails
```

This problem does not exist when you're testing for logical truth. Classic PowerBASIC considers that an expression is TRUE in every case *except* when the expression is zero. However, ISTRUE converts all non-zero values to the "most true" value, -1, which provides the most consistent results with both boolean and arithmetic expressions.

Restrictions	ISTRUE and ISFALSE operators evaluate the "whole" expression following the keyword, subject to their Operator Precedence level. For example, parentheses contained within the expression are regarded as an integral part of the expression, and do not act as delimiters for the ISTRUE and
--------------	--

ISFALSE operators.

With this in mind, combining a logical test result into a further expression means that the expressions must be separated to ensure the correct evaluation.

Consider the following statement:

```
IF ISTRUE (x&) + y& THEN
```

Classic PowerBASIC evaluates the entire expression (x&) + y& and then calculates the logical truth from the overall result of that expression. That is, the parentheses around the first part of the expression do not stop ISTRUE from evaluating the whole expression. To demonstrate this, the statement can be rewritten to concisely demonstrate the scope of the logical evaluation:

```
IF ISTRUE (x& + 2) THEN
```

or it could be simplified even further:

```
IF ISTRUE x& + 2 THEN
```

If you wish to utilize the numeric result of the logical test in a further expression, parentheses must be added to separate the expressions correctly:

```
IF (ISTRUE x&) + 2 THEN
```

See also

[Arithmetic Operators](#), [NOT](#), [Short-circuit evaluation](#)

Purpose	Determine whether or not a file exists.
Syntax	<i>FileExists</i> & = ISFILE(<i>FileName</i> \$)
Remarks	<p>The file subsystem is checked to determine whether the file specified by <i>FileName</i>\$ currently exists. If it is found in any form (hidden, system, read-only, etc.), the value true (-1) is returned. Otherwise, the value false (0) is returned.</p> <p><i>Filename</i>\$ is an unambiguous file name, which may not contain an asterisk (*) or query (?). If it contains one or more of those characters, the function always returns false (0).</p>
See also	DIR\$, DISPLAY BROWSE , DISPLAY OPENFILE , DISPLAY SAVEFILE , ISFOLDER , PATHSCAN\$

Purpose	Determine whether or not a folder exists.
Syntax	<i>FolderExists</i> & = ISFOLDER(<i>FolderName</i> \$)
Remarks	<p>The file subsystem is checked to determine whether the folder specified by <i>FolderName</i>\$ currently exists. If it is found in any form (hidden, system, read-only, etc.), the value true (-1) is returned. Otherwise, the value false (0) is returned.</p> <p>The root directory (for example, "C:\") is considered to be a folder, and returns the value true (-1).</p> <p><i>FolderName</i>\$ is an unambiguous file name, which may not contain an asterisk (*) or query (?). If it contains one or more of those characters, the function always returns false (0).</p>
See also	DIR\$, DISPLAY BROWSE , DISPLAY OPENFILE , DISPLAY SAVEFILE , ISFILE , PATHSCAN\$

Purpose	Determine whether an object supports a particular interface .
Syntax	<i>IfaceValid</i> = ISINTERFACE(<i>ObjectVar</i> , <i>InterfaceName</i>)
Remarks	The object referenced by the parameter <i>ObjectVar</i> is tested to determine if the specified <i>InterfaceName</i> is supported. If so, the value true (-1) is returned. Otherwise, the value false (0) is returned.
See also	CLASS , INTERFACE (Direct) , INTERFACE (IDBind) , What is an object, anyway?

Purpose	Determine whether an optional parameter was passed by the calling code.
Syntax	<i>ParamStatus</i> = ISMISSING(<i>ParamVar</i>)
Remarks	<p>The ISMISSING function may be used to test certain optional parameters to determine whether or not the parameter was actually passed by the calling code. It may be used to test BYVAL/BYREF VARIANT parameters, or other variable types passed ByRef. An attempt to test a ByVAL parameter other than VARIANT will generate an error 579 (BYREF variable or BYVAL/BYREF variant expected) to be generated during compilation.</p> <p>A ByRef parameter is considered to be missing when the pointer has the value zero. A variant parameter is considered to be missing when it has a type of %VT_ERROR and an error value of %DISP_E_PARAMNOTFOUND.</p> <p>If the specified optional parameter is missing, the value true (-1) is returned. Otherwise, the value false (0) is returned.</p>
Restrictions	The ISMISSING function may only be used within the procedure which uses the specified optional parameter.
See also	DECLARE , FUNCTION , METHOD , PROPERTY , SUB

ISNOTHING function

[Top](#) [Previous](#) [Next](#)

Purpose	Determine the current status of a given object variable.
Syntax	<code><i>oStatus</i> = ISNOTHING(<i>objectvar</i>)</code>
Remarks	ISNOTHING is particularly useful in determining the success or failure of a LET statement. It returns TRUE (-1) if the object variable contains nothing, or FALSE (0) if it contains a valid current reference to an object interface. ISNOTHING is the complement to the ISOBJECT function.
Restrictions	<i>objectvar</i> must be an interface or IDispatch object variable.
See also	DIM , INTERFACE (Direct) , INTERFACE (IDBind) , ISOBJECT , LET (with Objects) , OBJECT , What is an object, anyway?
Example	<pre>DIM oApp AS IAPPDatabase LET oApp = NEW IAPPDatabase IN "DBApp.0" IF ISNOTHING(oApp) OR ERR THEN ' Handle error</pre>

ISOBJECT function

[Top](#) [Previous](#) [Next](#)

Purpose	Determine the current status of a given object variable.
Syntax	<code>oStatus = ISOBJECT(objectvar)</code>
Remarks	<p>ISOBJECT is particularly useful in determining the success or failure of a LET (with Objects) statement. It returns TRUE (-1) if the object variable contains a valid current reference to an object interface, or FALSE (0) if it contains nothing.</p> <p>ISOBJECT is the complement to the ISNOTHING function.</p>
Restrictions	<code>objectvar</code> must be an interface or IDispatch object variable.
See also	DIM , INTERFACE (Direct) , INTERFACE (IDBind) , ISNOTHING , LET (with Objects) , OBJECT , What is an object, anyway?
Example	<pre>DIM oApp AS IAPPDatabase LET oApp = NEWCOM "DBApp.0" IF ISOBJECT(oApp) AND ISFALSE ERR THEN 'Handle error</pre>

Purpose	Determine whether a control/ Dialog /Window currently exists
Syntax	<pre><i>DialogExists&</i> = ISWIN(<i>hDlg&</i>) <i>ControlExists&</i> = ISWIN(<i>hParentDlg&</i>, <i>Ident&</i>)</pre>
Remarks	<p>The Window subsystem is checked to determine whether the specified Dialog or Control currently exists. This function may be used for a wide range of purposes, but it's particularly valuable when you want to be sure that a CONTROL was created successfully with the CONTROL ADD statement.</p> <p>If you use a single parameter, it must specify the handle of a Window, Dialog, or Control you are checking. If you use two parameters, you would specify the handle of the parent and the identifier of the Control you are checking.</p> <p>If the target of the function currently exists, TRUE (-1) is returned. If it does not exist, the return value is FALSE (0).</p>
See also	CONTROL HANDLE , DIALOG NEW

- Purpose

Start an immediate iteration of a loop structure.
- Syntax

ITERATE [DO | LOOP | FOR]
- Remarks

ITERATE is just like using a [GOTO](#) to the line immediately before the NEXT statement (of a [FOR...NEXT](#) loop), the LOOP statement (of a [DO...LOOP](#) loop), or the WEND statement (of a [WHILE..WEND](#) loop). For example, the following code fragments are equivalent:

```
FOR ix = 1 TO 100
  [statements]
  ITERATE FOR
  [statements]
NEXT

FOR ix = 1 TO 100
  [statements]
  GOTO iterateForLoop
  [statements]
iterateForLoop:
NEXT
```

If you do not specify DO, LOOP, or FOR, ITERATE will iterate the most recently executed structure. For example:

```
FOR ix = 1 TO 10
  DO UNTIL x > 10
    [statements]
    ITERATE ' will iterate the DO LOOP
    [statements]
  LOOP
NEXT
ITERATE DO and ITERATE LOOP are interchangeable.
```

Use this statement...	To iterate this kind of loop
ITERATE FOR	FOR/NEXT
ITERATE DO, ITERATE LOOP	DO/LOOP, WHILE/WEND

See also

[DO/LOOP](#), [EXIT](#), [FOR/NEXT](#), [WHILE/WEND](#)

JOIN\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string consisting of all of the strings in an array , each separated by a delimiter.
Syntax	<code>A\$ = JOIN\$(array(), {<i>delim\$</i> BINARY})</code>
Remarks	<p>JOIN\$ requires a delimiter string <i>delim\$</i> which may be any length.</p> <p>If the delimiter expression is a null (zero-length) string, no separators are inserted between the string sections. If the delimiter expression is the 3-byte value of "," (which may be expressed in your source code as the string literal """, """), a leading and trailing double-quote is added to each string section. This ensures that the returned string contains standard, comma-delimited quoted fields that can be easily parsed.</p> <p>The array specified by <i>array()</i> may be any data type.</p>
BINARY	<p>If the array consists of fixed size elements (numeric, ASCIIZ, etc.), the returned string consists of an exact memory image of the array data in internal format. If the array contains variable length data (Dynamic string, Field string), it is stored in Classic PowerBASIC and/or Visual Basic packed string format: If a string is shorter than 65535 bytes, it starts with a 2-byte length WORD followed by the string data. Otherwise, it will start with a 2-byte value of 65535, followed by a DWORD indicating the string length, then finally the string data itself.</p> <p>The JOIN\$ function is the natural complement to the PARSE statement.</p>
See also	BUILD\$, PARSE , PARSE\$, PARSECOUNT
Example	<pre>FUNCTION PBMAIN DIM a\$(2), s1\$, s2\$ a\$(0) = "Hello" a\$(1) = "Power" a\$(2) = "BASIC" s1\$ = JOIN\$(a\$(), """, """) s2\$ = JOIN\$(a\$(), \$SPC) END FUNCTION</pre>
Result	<pre>s1\$ contains: "Hello","Power","BASIC" s2\$ contains: Hello Power BASIC</pre>

KILL statement

[Top](#) [Previous](#) [Next](#)

Purpose Delete a disk file.

Syntax `KILL filespec`

Remarks *filespec* is a [string expression](#) specifying the file or files to be deleted, and can include a path name and/or "wildcard" characters. *filespec* may be either a Short File Name (SFN) or a Long File Name (LFN). For example:

```
KILL "TEST.DOC"  
KILL "C:\MY APPLICATION DATA\INCOME.?87"  
MyFile$ = "*.BAS"  
KILL MyFile$      ' Potentially dangerous!
```

If *filespec* does not exist, [Error 53](#) ("File not found") is generated. If *filespec* is read only, [Error 70](#) ("Permission denied") occurs. You should not attempt to KILL an open file.

Files with the HIDDEN or SYSTEM attribute can not be deleted with KILL. An attempt to do so is ignored, with no error generated.

KILL is analogous to the DOS "DEL" and "ERASE" commands. KILL cannot delete a directory - use [RMDIR](#) instead, after first deleting all the files in the directory.

See also [FILEATTR](#), [FILECOPY](#), [FILENAMES](#), [GETATTR](#), [NAME](#), [RMDIR](#), [SETATTR](#), [SETEOF](#)

LBOUND function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the smallest possible subscript (boundary) for an array's specified dimension.
Syntax	<pre>y& = LBOUND(array [(dimension)]) y& = LBOUND(array, dimension)</pre>
Remarks	LBOUND can be used in combination with UBOUND to determine the size of an array. LBOUND of an undimensioned array returns zero. The LBOUND function has the following parts:
<i>array</i>	Name of the array of interest.
<i>dimension</i>	An integer indicating which dimension's lower bound is returned. If not specified, the first dimension is assumed.
Restrictions	LBOUND cannot be used on arrays within User-Defined Types .
See also	ARRAYATTR , DIM , REDIM , UBOUND
Example	<pre>' Dimension an array with lower and upper bounds DIM MyArray%(1900 TO 2000,5 TO 10) ' get the values of the array l1 = LBOUND(MyArray%) u2 = UBOUND(MyArray%) l2 = LBOUND(MyArray%(2)) u2 = UBOUND(MyArray%, 2)</pre>

LCASE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose Return a lowercase version of a string argument.

Syntax `s$ = LCASE$(string_expression [,ANSI | OEM])`

Remarks LCASE\$ returns a string equivalent to *string_expression*, except that uppercase letters in *string_expression* are converted to lowercase. The optional ANSI or OEM parameter specifies whether the conversion is made using the ANSI charset for the system, or the original IBM OEM charset. If no charset is specified, Classic PowerBASIC for Windows uses the system ANSI charset, while [PB/CC](#) uses the IBM OEM charset. Only "International" characters in the range of [CHR\\$\(128\)](#) to [CHR\\$\(255\)](#) are affected by this parameter.

The OEM charset is based upon the original IBM OEM charset to ensure compatibility with programs written for all previous versions of the Classic PowerBASIC compiler.

See also [MCASE\\$](#), [UCASE\\$](#)

Example `x$ = LCASE$("Cats aren't ALWAYS good.")`

Result `cats aren't always good.`

LEFT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the left-most <i>n</i> characters of a string.
Syntax	<i>s\$</i> = LEFT\$(<i>string_expression</i> , <i>n</i> &)
Remarks	<p><i>n</i>& is a Long-integer expression and specifies the number of characters in string_expression to be returned.</p> <p>LEFT\$ returns a string consisting of the left most <i>n</i>& characters of its string argument. If <i>n</i>& is greater than or equal to the length of <i>string_expression</i>, all of <i>string_expression</i> is returned. If <i>n</i>& is zero, LEFT\$ returns an empty string. If the length value parameter is negative, it is interpreted as LEN(<i>string_expression</i>)-ABS(<i>n</i>&). For example, LEFT\$("1234567890",-2) returns "12345678".</p>
See also	EXTRACT\$, INSTR , LTRIM\$, MID\$, RIGHT\$, RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>' Demonstrate LEFT\$ and RIGHT\$ functions DIM TestString\$, x\$, y\$, n AS LONG TestString\$ = "ABCDEFGHJKLMNOP" FOR n = 1 TO 14 STEP 2 x\$ = LEFT\$(TestString\$,n) y\$ = RIGHT\$(TestString\$,n) NEXT n</pre>

Purpose	Return the logical length of a variable , User-Defined Type , or Union .
Syntax	<code>y& = LEN(<i>target</i>)</code>
Remarks	<p>If <i>target</i> is a string variable or a string expression, LEN returns a value from 0 to the current string length, representing the number of characters in target. If target is a fixed-length string, the length of the fixed buffer is returned. If target is an ASCIIZ string, the length of the data stored in the ASCIIZ string is returned, not the maximum size of the ASCIIZ string. Use SIZEOF to determine the maximum size of an ASCIIZ string.</p> <p>When used with pointers, LEN returns a value of 4, since a pointer is always stored as a DWORD. You can use LEN with the target of a pointer to return the size of target. If the target is a dynamic string, you will receive the length of the string, not the length of the handle.</p> <p><i>target</i> can also be any other variable type, including and User-Defined Types (defined with TYPE/END TYPE). In that case, Classic PowerBASIC will return the number of bytes needed to store a variable of that type.</p> <p>When measuring the size of a padded (aligned) UDT structure with the LEN (or SIZEOF) statement, the measured length includes any padding that was added to the structure. For example, the following UDT structure:</p> <pre>TYPE LengthTestType DWORD a AS INTEGER END TYPE ' code here DIM abc AS LengthTestType x& = LEN(abc)</pre> <p>Returns a length of 4 bytes in x&, since the UDT was padded with 2 additional bytes to enforce DWORD alignment. Note that the LEN of individual UDT members returns the true size of the member without regard to padding or alignment. In the previous example, LEN(abc.a) returns 2.</p>
See also	SIZEOF
Example	<pre>DIM p AS BYTE POINTER ByteLen = LEN(p) ' size of a pointer = 4 bytes ByteLen = LEN(@p) ' size of byte (target) = 1 byte</pre>

Purpose Assign a value to a [variable](#).

Syntax

```
[LET] variable = expression
[LET] variable += expression
[LET] variable -= expression
[LET] variable *= expression
[LET] variable /= expression
[LET] variable \= expression
[LET] variable &= expression
[LET] variable AND= expression
[LET] variable OR= expression
[LET] variable EQV= expression
[LET] variable IMP= expression
[LET] variable MOD= expression
[LET] variable XOR= expression
```

Remarks **Simple assignment**

variable is a string or numeric variable, and expression is of a suitable type (that is, a string expression for string variables and numeric expression for numeric variables).

The word LET is optional in assignment statements. It is allowed to provide compatibility BASIC source files written for early versions of BASIC. In practice, the word LET is very rarely used.

To allow easy conversion, Classic PowerBASIC allows a [User-Defined Type](#) in a [string expression](#). The User-Defined Type is simply copied, byte for byte, into the expression. However, to assign a string back to a User-Defined Type, you should use the [TYPE SET](#) statement.

```
DIM abc as MyType
MyString$ = abc
```

Please refer to the following sections of the LET statement for special information regarding assignment using Object variables, Variant variables, and User-Defined Type variables.

Compound assignment

A compound assignment statement combines a binary [arithmetic operator](#), a binary bitwise operator, or a binary string operator (concatenation) as an integral part of the assignment. This offers the programmer a "shortcut" in your source code, and can even result in more efficient code generation. That's because the target variable is evaluated only once, even if an array or pointer calculation could have a side effect which changes it.

Compound assignments are available for the standard arithmetic operations of add, subtract, multiply, divide, int-divide, and modulo (+ - * / \ [MOD](#)), the bitwise operations ([AND](#), [OR](#), [XOR](#), [EQV](#), [IMP](#)), and the concatenation operators (+ &). Each are represented by one of the

following tokens:

+=	AND=
-=	OR=
/=	EQV=
\=	IMP=
&=	MOD=
*=	XOR=

Each of the following pairs of code are functionally identical:

<code>x = x + 1</code>	<code>x += 1</code>
<code>x = x / y</code>	<code>x /= y</code>
<code>x = x XOR 3</code>	<code>x XOR= 3</code>
<code>x(7) = x(7) AND 5</code>	<code>x(7) AND= 5</code>
<code>x\$ = x\$ + y\$</code>	<code>x\$ += y\$</code>

See also

[BUILD\\$](#), [JOIN\\$](#), [LET \(with Objects\)](#), [LET \(with Variants\)](#), [LET \(with Types\)](#), [TYPE SET](#)

Example

```
MyString$ = "This is a test."  
LET TempStr$ = MyString$  
LET MyVarr -= YourVar
```

LET statement (with Objects)

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose	Assign an object reference to an object variable.
Syntax	<code>[LET] <i>objvar</i> = <i>object expression</i></code>
Remarks	<p>The LET Statement, and its implied form (without using the word LET), may be used to assign an object reference to an object variable. After you declare an object variable as a particular interface, you must create an object and or assign an object reference to it before you can use the objects members (methods, properties, etc.).</p> <p>If an object creation or assignment fails for any reason, the <i>objvar</i> is set to NOTHING. If this statement fails, no errors are generated, nor is an OBJRESULT set. You should test for success of the operation with ISOBJECT(<i>objvar</i>) before trying to use the object or execute its methods.</p>

LET *objvar* = CLASS *ClassName*\$

The term *ClassName* must be specified as a quoted [string literal](#), which is the name of a class implemented within the program. Since the class is internal (the name is known at compile-time), you may not use a string variable or [expression](#). Upon execution, a new object is created, and a reference to that object is assigned to the object variable *objvar*. The interface requested is determined by the original declaration of *objvar*. If InterfaceName is DISPATCH, you can reference it with the [OBJECT](#) statement -- otherwise, regular Method and Property references are used.

LET *objvar* = NEWCOM *ProgID*\$

LET *objvar* = GETCOM *ProgID*\$

LET *objvar* = ANYCOM *ProgID*\$

This form of the LET statement is used to obtain an object reference external to the program using the [COM](#) facilities of Windows. If the requested object is in a [DLL](#) (in-process server), you will always use the NEWCOM option, as you're asking for a new object. If the request is successful, the object reference is assigned to the *objvar*.

If the requested object is in an EXE (out-of-process server), you may use any of the three options. If the director word NEWCOM is specified, a new instance of a COM application is created. With GETCOM, an interface will be opened on an existing, running application, which has been registered as the active [automation](#) object for its class. With ANYCOM, the compiler will first try to use an existing, running application if available, or a new instance if not.

The string expression *ProgID*\$ evaluates to a [ProgID](#) name on an external

COM server. If the InterfaceName is DISPATCH, you can reference it with the OBJECT statement -- otherwise, regular Method and Property references are used instead.

LET objvar = NEWCOM CLSID ClassID\$

LET objvar = GETCOM CLSID ClassID\$

LET objvar = ANYCOM CLSID ClassID\$

This form also obtains a [COM object](#), just as the examples in the above section. There is always a one-to-one relationship between a ProgID and a [CLSID](#) (Class ID). An object can be identified by either of these tokens, as long as they are both available. In some instances, you may encounter an object which has no ProgID published. You can substitute the clause "CLSID *ClassID\$*" for the *ProgID\$*. It works exactly as the usual form above, except that it describes the requested object by its 16-byte GUID which is the CLSID (Class ID) of the object.

LET objvar = NEWCOM CLSID ClassID\$ LIB DLLPath\$

Classic PowerBASIC offers the unique ability to create and reference COM objects without any reference to the registry at all. As long as you know the CLSID (Class ID) and the file path/name of the DLL to be accessed, you can do so with no registry access at all. You don't need a special type of COM server. This technique can be used with any server, whether created by Classic PowerBASIC or another compiler. By using this method of object creation, there is simply no need for the server to be registered at all. That allows you to keep local copies of the COM servers you use, with no chance they will be altered or replaced by another application. You use the above form, where the clause "CLSID *ClassID\$*" identifies the 16-byte Class ID, and the clause "LIB *DllPath\$*" identifies the file path and file name of the COM Server. Once you've obtained the COM object reference in *objvar*, it is used exactly as you would with a traditional object.

LET objvar1 = objvar2

If both object variables have been declared as the same object type (the same interface name), the source variable (*objvar2*) is copied to the destination variable (*objvar1*), and the reference count of the object is incremented. If the object variables are of different object types, a new interface (of the type implied by *objvar1*) is opened on *objvar2*, and a reference to it is assigned to *objvar1*.

LET objvar = objmethod(params)

It is assumed that the METHOD or GET PROPERTY specified by

objmethod returns an object of the type of *objvar*. The *objmethod* is evaluated, and the object reference which it returns is assigned to *objvar*.

LET *objvar* = ME

This form may only be used within a METHOD or PROPERTY. A new interface (of the type implied by *objvar*) is opened on the current object, and a reference to it is assigned to *objvar*.

LET *objvar* = NOTHING

This destroys an object variable, discontinuing its association with a specific object. This in turn releases all system and memory resources associated with the object when no more object variables refer to it.

LET *objvar* = *vrnt*

Attempts to open an interface of the specified class for *objvar* on the object of *vrnt*, and assigns a reference to *objvar*. It assumes that *vrnt* contains a reference to an object of type %VT_UNKNOWN or %VT_DISPATCH. If the desired interface can not be opened, the object variable *objvar* is set to NOTHING.

LET *vrnt* = *objvar*

This may be used to assign an object reference from an object variable to a [variant](#) variable. It attempts to open an IDispatch interface, else an IUnknown interface on the object of *objvar*, and assigns that reference to *vrnt*. Variant variables can not contain references to custom interfaces, only IDispatch or IUnknown. If the assignment is successful, [VARIANTVT](#)(*vrnt*) will return either %VT_UNKNOWN or %VT_DISPATCH. If it is unsuccessful, *vrnt* is set to %VT_EMPTY.

Previous versions of PowerBASIC Compilers supported another syntax for the creation of objects:

LET *objvar* = [NEW] Interface name ON ClassName|ProgID\$

While this syntax will still be temporarily supported for the sake of compatibility, it will be removed in the next major release of PowerBASIC. We urge you to change existing source code to the new simplified syntax as soon as possible. It's as simple as inserting a director keyword of CLASS, NEWCOM, GETCOM, or ANYCOM, and removal of InterfaceName/ON.

Previous versions of PowerBASIC Compilers used the SET statement for creation of objects. LET now includes all the functionality of the old SET statement, so you should plan to remove all SET statements as soon as possible. This involves

nothing more than changing every SET to LET, or simply deleting every SET.

See also

[LET](#), [LET \(with Variants\)](#), [LET \(with Types\)](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [ISINTERFACE](#), [ISNOTHING](#), [ISOBJECT](#), [Just what is COM?](#), [ME](#), [OBJECT](#), [What is an object, anyway?](#)

LET statement (with Types)

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose Assign data to a [user-defined type](#) variable.

Syntax `[LET] typevar = typevar`

Remarks *typevar* is a user-defined type variable. In order to perform direct assignment of data from one user-defined type variable to another, they must be dimensioned to the same type. To assign data between two different types, you should use the [TYPE SET](#) statement instead.

The word LET is optional in assignment statements. It is allowed to provide compatibility BASIC source files written for early versions of BASIC. In practice, the word LET is very rarely used.

To allow easy conversion, Classic PowerBASIC allows a User-Defined Type in a [string expression](#). The User-Defined Type is simply copied, [byte](#) for byte, into the expression. However, to assign a string back to a User-Defined Type, you should use the TYPE SET statement.

```
DIM abc as MyType
MyString$ = abc
```

See also [LET](#), [LET \(with Objects\)](#), [LET \(with Variants\)](#), [TYPE SET](#)

Example

```
MyString$ = "This is a test."
LET TempStr$ = MyString$
```

LET statement (with Variants) New!

[Top](#) [Previous](#) [Next](#)

Purpose Assign a value or an [object](#) reference to a [variant](#) variable.

Syntax `[LET] variant = variant expression`

Remarks Although notoriously lacking in efficiency, [Variant](#) variables are commonly used as [COM](#) Object parameters due to their flexibility. You can think of a Variant as a kind of container, which can hold a variable of most any data type, numeric, string, or even an entire [array](#). This simplifies the process of calling procedures in a [COM Object](#) Server, as there is little need to worry about the myriad of possible data types for each parameter.

This flexibility comes at a great price in performance, so Classic PowerBASIC limits their use to data storage and parameters only. You may assign a numeric value, a string value, or even an entire array to a Variant with the LET statement, or its implied equivalent. In the same way, you may assign one Variant value to another Variant variable, or even assign an array contained in a Variant to a compatible Classic PowerBASIC array, or the reverse.

You may extract a simple scalar value from a Variant with the [VARIANT#](#) function for numeric values or with the [VARIANT\\$](#) function for string values.

LET vrntvar = vrntvar

This form duplicates the contents of one variant variable, assigning it to a second variant variable.

LET vrntvar = expression [AS vartype]

The numeric or string expression is evaluated, and the result is assigned to the variant variable. Classic PowerBASIC automatically chooses an appropriate numeric or string data type for the internal representation of the Variant. However, you can specify a particular preferred format by adding the optional AS *vartype* clause, which could be [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), or [STRING](#). In the case of a string value, Classic PowerBASIC automatically handles [Unicode](#) conversions needed for the COM specification.

LET vrntvar = EMPTY

The variant variable is set to %VT_EMPTY, which means it contains no value of any kind.

LET vrntvar = ERROR numr

This form assigns a specific COM error number, which is usually a COM

specific error, such as %E_NOINTERFACE, etc.

LET *vrntvar* = array()

An entire Classic PowerBASIC array is assigned to a variant variable. In the case of a [string array](#), Classic PowerBASIC automatically handles [Unicode](#) conversions needed for the COM specification. Array assignment is limited to the following data types: BYTE, WORD, DWORD, INTEGER, LONG, QUAD, SINGLE, DOUBLE, CURRENCY, or STRING, as Windows does not support all Classic PowerBASIC data forms.

LET array() = *vrntvar*

An entire array is assigned from a variant variable to a Classic PowerBASIC array. In the case of a [string array](#), Classic PowerBASIC automatically handles Unicode conversions. You can not assign an array with more than eight [dimensions](#) to a Classic PowerBASIC array.

LET *vrntvar* = BYREF variable

This form is used to allow a variant to contain a typed [pointer](#) to a specific variable. Any changes to the variant will cause the variable to be changed, as it is the target of the pointer. The variable may be of any data type which is supported by variants and COM objects: Byte, Word, Dword, Integer, Long, Quad, Single, Double, Currency, Variant, and Dynamic String. If you attempt to use an unsupported variable type (like [Extended](#), [Bit](#), [ASCIIZ](#), etc.), Classic PowerBASIC will generate an [error 482](#) (Data Type Mismatch). Further, you may not use a register variable (automatic or explicit), or an [error 491](#) (Invalid Register Variable) will be generated. Note that strings used with COM objects are expected to be in Unicode format, rather than ANSI. The [ACODE\\$](#) and [UCODE\\$](#) functions may be used to convert the strings as necessary.

LET *objvar* = *vrnt*

Attempts to open an [interface](#) of the specified [class](#) for *objvar* on the object of *vrnt*, and assigns a reference to *objvar*. It assumes that *vrnt* contains a reference to an object of type %VT_UNKNOWN or %VT_DISPATCH. If the desired interface can not be opened, the object variable *objvar* is set to NOTHING. You can test for success/failure with the [ISOBJECT](#)(*objvar*) function.

LET *vrnt* = *objvar*

This may be used to assign an object reference from an object variable to a variant variable. It attempts to open an [IDispatch](#) interface, else an [IUnknown](#) interface on the object of *objvar*, and assigns that reference to

vrnt. Variant variables can not contain references to custom interfaces, only IDispatch or IUnknown. If the assignment is successful, [VARIANTVT](#)(*vrnt*) will return either %VT_UNKNOWN or %VT_DISPATCH. If it is unsuccessful, *vrnt* is set to %VT_EMPTY.

See also

[Just what is COM?](#), [LET](#), [LET \(with Objects\)](#), [LET \(with Types\)](#), [VARIANT#](#), [VARIANT\\$](#), [VARIANTVT](#)

LINE INPUT# statement

[Top](#) [Previous](#) [Next](#)

Purpose	Read line(s) from a sequential file into a string variable or string array , ignoring delimiters.
Syntax	<pre>LINE INPUT #<i>filenum</i>&, <i>string_variable</i> LINE INPUT #<i>filenum</i>&, <i>Arr</i>\$() [RECORDS <i>rcds</i>] [TO <i>count</i>]</pre>
Remarks	<p><i>filenum</i>& is the file number, or variable containing a file number, given when the file was opened. <i>string_variable</i> is the string variable to be loaded with the data read from the file.</p> <p><i>string_variable</i> may be a fixed-length, ASCIIZ, or dynamic string. For fixed-length and ASCIIZ strings, data that is longer than the string is truncated to fit into the string. Dynamic strings receive the data without truncation. <i>string_variable</i> may not be a UDT variable, although fixed-length and ASCIIZ UDT member variables are supported.</p> <p>LINE INPUT# is intended for use with text files composed of lines terminated by CR/LF (\$CRLF or CHR\$(13,10)) sequences. It reads a line from the file and returns it, minus the CR/LF delimiter. Commas, quotation marks and other characters have no special meaning for LINE INPUT#, and are treated like any other text.</p> <p>If the file consists of comma-delimited data items, INPUT# is likely to be more suitable than LINE INPUT#.</p> <p>The second syntax definition of LINE INPUT# reads a file opened for INPUT, assigning full lines of text to each element of the array.</p> <p>It is assumed the data is standard text, delimited by a CR/LF (\$CRLF) or EOF (1A hex or \$EOF). LINE INPUT# attempts to read the number of lines specified in the RECORDS <i>rcds</i> option, or the number of elements in the array, whichever is smaller.</p> <p>The actual number of lines read is assigned to the variable specified in the optional TO <i>count</i> clause. FILESCAN is useful in conjunction, to determine the dimensioned size of the string array. EOF is set just as with single Line Input.</p>
See also	EOF , FILESCAN , INPUT# , PRINT#
Example	<pre>SUB MakeFile ' Open a sequential file for output. Use PRINT# ' to write different data types to the file. OPEN "LINEINP#.DTA" FOR OUTPUT AS #1 ' Define some variables. sVar\$ = "There's trouble in River City, by George." iVar% = 1000 fpVar! = 30000.12 ' Write a line of text to the file.</pre>

```
PRINT# 1, sVar$; iVar%; fpVar!  
CLOSE #1      'close the file  
END SUB 'end procedure MakeFile
```

```
SUB ReadFile
```

```
'Open a sequential file for input, then use  
'LINE INPUT # to read lines of different  
'data types from the file.
```

```
OPEN "LINEINP#.DTA" FOR INPUT AS #1  
StringVar$ = ""
```

```
'Input an entire line regardless of length or  
'delimiters.
```

```
LINE INPUT #1, StringVar$  
CLOSE #1      'close the file  
END SUB 'end procedure ReadFile
```


Purpose Manipulate a [LISTBOX](#) control in order to set/retrieve data.

Syntax

```
LISTBOX ADD hDlg, id&, StrExpr
LISTBOX DELETE hDlg, id&, item&
LISTBOX FIND hDlg, id&, item&, StrExpr TO datav&
LISTBOX FIND EXACT hDlg, id&, item&, StrExpr TO datav&
LISTBOX GET COUNT hDlg, id& TO datav&
LISTBOX GET SELCOUNT hDlg, id& TO datav&
LISTBOX GET SELECT hDlg, id& [,item&] TO datav&
LISTBOX GET STATE hDlg, id&, item& TO datav&
LISTBOX GET TEXT hDlg, id& [,item&] TO txtv$
LISTBOX GET USER hDlg, id&, item& TO datav&
LISTBOX INSERT hDlg, id&, item&, StrExpr
LISTBOX RESET hDlg, id&
LISTBOX SELECT hDlg, id&, item&
LISTBOX SET TEXT hDlg, id&, item&, StrExpr
LISTBOX SET USER hDlg, id&, item&, NumExpr
LISTBOX UNSELECT hDlg, id& [,item&]
```

hDlg Handle of the [dialog](#) that owns the list box.

id& The control identifier assigned with [CONTROL ADD LISTBOX](#).

item& Position of data in the LISTBOX. First string=1, second=2...

NumExpr A numeric expression passed as a parameter.

StrExpr A [string expression](#) passed as a parameter.

txtv\$ A string variable to which result text is assigned.

datav& A [long integer](#) variable to which result data is assigned.

Remarks In each of the following samples and descriptions, the LISTBOX control which is the subject of the statement is identified by the handle of the dialog that owns the LISTBOX (*hDlg*), and the unique control identifier you gave it upon creation in CONTROL ADD LISTBOX.

The value *item*& refers to the position of the string data item in the LISTBOX, and is always indexed to one. The first string is position 1, the second is position 2, and so forth.

LISTBOX ADD *hDlg*, *id*&, *StrExpr*

The string value specified by *StrExpr* is added to the LISTBOX control. If the LISTBOX has the [%LBS_SORT](#) style, the new string is inserted in alphanumeric order; otherwise it is added to the end of the existing list.

LISTBOX DELETE *hDlg*, *id*&, *item*&

The string at the position specified by *item*& is deleted from the LISTBOX.

The parameter *item&* is indexed to one (1 for the first string, 2 for the second, and so on).

LISTBOX FIND *hDlg, id&, item&, StrExpr* TO *datav&*

Strings in the LISTBOX are searched to find the first string which begins with the data in *StrExpr*, regardless of any characters which follow.

Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *item&*, and ending with the last string in the LISTBOX. Searching does not wrap to the beginning of the list. The item number (*item&*) is indexed to 1 (1=first, 2=second, etc.). To search the entire LISTBOX starting with the first string, *item&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

LISTBOX FIND EXACT *hDlg, id&, item&, StrExpr* TO *datav&*

Strings in the LISTBOX are searched to find the first string which exactly matches the data in *StrExpr*. Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *item&*, and ending with the last string in the LISTBOX. Searching does not wrap to the beginning of the list. The item number (*item&*) is indexed to 1 (1=first, 2=second, etc.). To search the entire LISTBOX starting with the first string, *item&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

LISTBOX GET COUNT *hDlg, id&* TO *datav&*

The number of items in the LISTBOX is retrieved, and assigned to the [long integer](#) variable specified by *datav&*.

LISTBOX GET SELCOUNT *hDlg, id&* TO *datav&*

The number of selected items in the LISTBOX is retrieved, and assigned to the long integer variable specified by *datav&*.

LISTBOX GET SELECT *hDlg, id&* [,*item&*] TO *datav&*

The LISTBOX is searched to find the first selected item. If the *item&* parameter is included, searching starts at that position to facilitate retrieving multiple selected items. If *item&* is omitted, the search starts at the first data item. The index number of the selected item is assigned to the variable designated by *datav&*. If no item is selected, the value zero (0) is assigned to it.

LISTBOX GET STATE *hDlg, id&, item& TO datav&*

A data item is checked to see if it is currently selected. The numeric value *item&* specifies which user value is to be checked, 1 for the first item, 2 for the second item, etc. If the item is selected, -1 ([true](#)) is assigned to the variable specified by *datav&*. Otherwise, 0 ([false](#)) is assigned to it.

LISTBOX GET TEXT *hDlg, id& [,item&] TO txtv\$*

Text is retrieved from the LISTBOX and assigned to the string variable specified by *txtv\$*. If the numeric expression *item&* is included, it determines which text string is returned, 1 for the first item, 2 for the second item, etc.

The parameter *item&* may be omitted, or contain the value zero (0). In the case of a single-selection listbox, the current selected text (if any) is retrieved and assigned to *txtv\$*. With a multiple-selection listbox ([%LBS_MULTIPLESEL](#) or [%LBS_EXTENDEDSEL](#) style), the text of the first (base) selected item is assigned to *txtv\$*. To retrieve additional selected text items from a multiple-selection listbox, use LISTBOX GET SELECT to retrieve selected item numbers. Then apply the item numbers with LISTBOX GET TEXT to retrieve the string data.

LISTBOX GET USER *hDlg, id&, item& TO datav&*

Each item in a LISTBOX may have a long integer user value associated with it at the discretion of the programmer. This user value is retrieved with LISTBOX GET USER. The numeric value *item&* specifies which user value is requested, 1 for the first item, 2 for the second item, etc. The returned user value is assigned to the long integer variable specified by *datav&*. LISTBOX user values are assigned with the LISTBOX SET USER statement. In addition to these LISTBOX user values, every DDT control offers an additional eight user values which can be accessed with [CONTROL GET USER](#) and [CONTROL SET USER](#).

LISTBOX INSERT *hDlg, id&, item&, StrExpr*

The text for a new data item, specified by *StrExpr*, is inserted at the location given by *item&*. The value of *item&* = 1 for the first item, 2 for the second item, etc. The list of data items is not re-sorted, even if the LISTBOX was created with the style [%LBS_SORT](#). If you wish to sort all of the items, use LISTBOX ADD instead.

LISTBOX RESET *hDlg, id&*

Delete all contents of the specified LISTBOX.

LISTBOX SELECT *hDlg, id&, item&*

The string data item specified by *item&* is chosen as selected text for the LISTBOX control, and the selected text is scrolled into a visible position. The value of *item&* = 1 for the first item, 2 for the second item, etc. If the value of *item&* = 0 with a multiple selection listbox, then all string data items are selected. LISTBOX SELECT may be used with both single and multiple selection listboxes.

LISTBOX SET TEXT *hDlg, id&, item&, StrExpr*

The text for the data item specified by *item&* is replaced with the new text in *StrExpr*. The value of *item&* = 1 for the first item, 2 for the second item, etc. The list of data items is not re-sorted, even if the LISTBOX was created with the style %LBS_SORT. If you wish to sort the items, use LISTBOX DELETE followed by LISTBOX ADD instead.

LISTBOX SET USER *hDlg, id&, item&, NumExpr*

Each item in a LISTBOX may have a long integer user value associated with it at the discretion of the programmer. This user value is assigned with LISTBOX SET USER, and retrieved with LISTBOX GET USER. The numeric value *item&* specifies which user value is to be accessed, 1 for the first item, 2 for the second item, etc. The value specified by *NumExpr* is saved for later retrieval. In addition to these LISTBOX user values, every [DDT](#) control offers an additional eight user values which can be accessed with CONTROL GET USER and CONTROL SET USER.

LISTBOX UNSELECT *hDlg, id& [,item&]*

The string value specified by *item&* is set to an unselected state for the LISTBOX control. The value of *item&* = 1 for the first item, 2 for the second item, etc. If *item&* is missing, or has the value zero, all items are set to an unselected state. LISTBOX UNSELECT may be used with both single and multiple selection listboxes.

Restrictions Under Windows 95/98/ME, a list box is limited to 32,767 items. In all versions of Windows, the actual string data contained by the list box is limited only by available memory.

See also [Dynamic Dialog Tools](#), [CONTROL ADD LISTBOX](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

Purpose Manipulate a [LISTVIEW](#) control in order to set/retrieve data.

Syntax

```
LISTVIEW DELETE COLUMN hDlg, id&, col&
LISTVIEW DELETE ITEM hDlg, id&, row&
LISTVIEW FIND hDlg, id&, row&, StrExpr TO datav&
LISTVIEW FIND EXACT hDlg, id&, row&, StrExpr TO datav&
LISTVIEW FIT CONTENT hDlg, id&, col&
LISTVIEW FIT HEADER hDlg, id&, col&
LISTVIEW GET COLUMN hDlg, id&, col& TO datav&
LISTVIEW GET COUNT hDlg, id& TO datav&
LISTVIEW GET HEADER hDlg, id&, col& TO txtv$
LISTVIEW GET MODE hDlg, id& TO datav&
LISTVIEW GET SELCOUNT hDlg, id& TO datav&
LISTVIEW GET SELECT hDlg, id& [, row&] TO datav&
LISTVIEW GET STATE hDlg, id&, row&, col& TO datav&
LISTVIEW GET STYLEXX hDlg, id& TO datav&
LISTVIEW GET TEXT hDlg, id&, row&, col& TO txtv$
LISTVIEW GET USER hDlg, id&, row& TO datav&
LISTVIEW INSERT COLUMN hDlg, id&, col&, StrExpr, width&, format&
LISTVIEW INSERT ITEM hDlg, id&, row&, image&, StrExpr
LISTVIEW RESET hDlg, id&
LISTVIEW SELECT hDlg, id&, row& [, col&]
LISTVIEW SET COLUMN hDlg, id&, col&, NumExpr
LISTVIEW SET HEADER hDlg, id&, col&, StrExpr
LISTVIEW SET IMAGE hDlg, id&, row&, NumExpr
LISTVIEW SET IMAGE2 hDlg, id&, row&, NumExpr
LISTVIEW SET IMAGELIST hDlg, id&, hLst, NumExpr
LISTVIEW SET MODE hDlg, id&, NumExpr
LISTVIEW SET OVERLAY hDlg, id&, row&, NumExpr
LISTVIEW SET STYLEXX hDlg, id&, NumExpr
LISTVIEW SET TEXT hDlg, id&, row&, col&, StrExpr
LISTVIEW SET USER hDlg, id&, row&, NumExpr
LISTVIEW SORT hDlg, id&, col& [, options...]
LISTVIEW UNSELECT hDlg, id&, row&, [col&]
LISTVIEW VISIBLE hDlg, id&, row&
```

hDlg Handle of the [dialog](#) that owns the ListView.

hLst Handle of the [ImageList](#) to be used for graphical items.

id& The control identifier assigned with [CONTROL ADD LISTVIEW](#).

row& A horizontal row number. First=1, second=2...

col& A vertical column number. First=1, second=2...

NumExpr A numeric expression passed as a parameter.

StrExpr A [string expression](#) passed as a parameter.

txtv\$ A string variable to which result text is assigned.

datav& A [long integer](#) variable to which result data is assigned.

Remarks

There are 4 general display modes available with a LISTVIEW control. The initial display mode is established at the time the control is created, as a part of the control style parameter. It may be changed from time to time with LISTVIEW SET MODE.

- Mode 0 Icon Mode - String data items are displayed left to right, wrapped to multiple lines as necessary. If a small icon IMAGELIST is attached to the LISTVIEW control, images from that list are displayed with each data item.
- Mode 1 Report Mode - String data items are displayed as a list, top to bottom, one item per line. The control may have one or more columns, with header text to describe each of them. Additional sub-items may be displayed in each column, by specifying a column number greater than one. This is the most frequently used ListView mode, and the default mode if not specified at the time the control is created. In this mode, it's often convenient to think of the item number as a row number. If a small icon IMAGELIST is attached to the LISTVIEW control, images from that list are displayed with each data item.
- Mode 2 Small Icon Mode - String data items are displayed left to right, wrapped to multiple lines as necessary. If a small icon IMAGELIST is attached to the LISTVIEW control, images from that list are displayed with each data item.
- Mode 3 List Mode - String data items are displayed as a list, top to bottom, one item per line. This mode is very similar in appearance to a standard [LISTBOX](#) control. In this mode, it's often convenient to think of the item number as a row number. If a small icon IMAGELIST is attached to the LISTVIEW control, images from that list are displayed with each data item.

In all of the following descriptions, the LISTVIEW control which is the subject of the statement is identified by the handle of the dialog that owns the LISTVIEW (*hDlg*), and the unique control identifier (*id&*) you gave it upon creation in CONTROL ADD LISTVIEW.

Each data item (or sub-item) is referenced by a combination of its item number (*row&*) and its column number (*col&*). A primary data item always has a column number of 1, while sub-items always have a column number greater than 1. Sub-items are only displayed in Report Mode. In all other display modes, they are hidden from view.

There are some essential difference in the way Windows handles primary data items as compared to sub-items. Primary data items are always displayed in the first column. Traditionally, only primary items are selected (highlighted for retrieval). They may be selected by clicking on them, or

programmatically, by executing LISTVIEW SELECT.

A standard click automatically unselects any items which were previously selected. Sub-items are always displayed in column 2 or greater. They may be selected (highlighted for retrieval) at your discretion, but only programmatically, by executing LISTVIEW SELECT. Clicking on them has no effect. Sub-item selections are persistent -- they remain selected indefinitely, until you explicitly execute LISTVIEW UNSELECT to change the state.

It's important to note that both primary item numbers (*item&*) and sub-item column numbers (*col&*) start at 1. The first=1, the second=2, and so forth.

LISTVIEW DELETE COLUMN *hDlg, id&, col&*

The column specified by *col&*, including its associated header text (if any), is deleted from the LISTVIEW control. The column number (*col&*) is indexed to 1 (1=first, 2=second, etc.). Column one of a list-view control cannot be deleted. If you must delete column one, insert a zero length dummy column one and delete column two and above. This is a limitation of the Microsoft Windows Listview control and not a Classic PowerBASIC limitation.

LISTVIEW DELETE ITEM *hDlg, id&, row&*

The data item specified by *row&* is deleted from the LISTVIEW control. The row number (*row&*) is indexed to 1 (1=first, 2=second, etc.).

LISTVIEW FIND *hDlg, id&, row&, StrExpr TO datav&*

Strings in the first column of a LISTVIEW are searched to find the first string which begins with the data in *StrExpr*, regardless of any characters which follow. Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *row&*, and ending with the last string in the LISTVIEW. Searching does not wrap to the beginning of the list. The row number (*row&*) is indexed to 1 (1=first, 2=second, etc.). To search the entire LISTVIEW starting with the first string, *row&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

LISTVIEW FIND EXACT *hDlg, id&, row&, StrExpr TO datav&*

Strings in the first column of a LISTVIEW are searched to find the first string which exactly matches the data in *StrExpr*. Comparisons are not case-sensitive. Strings are searched beginning with the string specified by *row&*, and ending with the last string in the LISTVIEW. Searching does not wrap to the beginning of the list. The row number (*row&*) is indexed to 1

(1=first, 2=second, etc.). To search the entire LISTVIEW starting with the first string, *row&* should be set to one (1). If a matching string is found, the index value of the match is assigned to the variable specified by *datav&*. If no match is found, the value zero (0) is assigned to it.

LISTVIEW FIT CONTENT *hDlg, id&, col&*

The width of the column specified by *col&* is adjusted to fit the width of the data items displayed in that column. The column number (*col&*) is indexed to 1 (1=first, 2=second, etc.).

LISTVIEW FIT HEADER *hDlg, id&, col&*

The width of the column specified by *col&* is adjusted to fit the width of the rows displayed in that column, and the header text at the top of that column. The column number (*col&*) is indexed to 1 (1=first, 2=second, etc.). If the specified column is the last column, its width is set to fill the remaining width of the list-view control.

LISTVIEW GET COLUMN *hDlg, id&, col& TO datav&*

The width of the designated column is retrieved from the ListView and assigned to the variable specified by *datav&*. The width is specified in either pixels or dialog units, depending upon which was used at creation. The value *col&* specifies the column number (1=first, 2=second, etc.).

LISTVIEW GET COUNT

hDlg, id& TO datav&

The number of rows in the LISTVIEW is retrieved, and assigned to the long integer variable specified by *datav&*.

LISTVIEW GET HEADER *hDlg, id&, col& TO txtv\$*

Column header text is retrieved from the LISTVIEW and assigned to the string variable specified by *txtv\$*. The value *col&* specifies the column number (1=first, 2=second, etc.).

LISTVIEW GET MODE *hDlg, id& TO datav&*

The display mode of the specified LISTVIEW control is retrieved and assigned to the variable designated by *datav&*. Possible mode values are 0=icon mode, 1=report mode, 2=small icon mode, 3=list mode.

LISTVIEW GET SELCOUNT *hDlg, id& TO datav&*

The LISTVIEW is interrogated to determine the number of primary data items which are currently selected. This count is assigned to the long

integer variable specified by *datav&*. To determine the count of sub-items selections, you must execute LISTVIEW GET STATE on every active sub-item.

LISTVIEW GET SELECT *hDlg, id& [, row&] TO datav&*

The LISTVIEW is interrogated to determine the next primary data item which is currently selected. The parameter *row&* specifies the starting item number for the search, to facilitate retrieving multiple selected items. To start at the beginning, use an *row&* of one (1), or just omit that parameter. The selected item number is assigned to the long integer variable specified by *datav&*. If no selected items are found, the value zero (0) is returned. To find selected sub-items, you must execute LISTVIEW GET STATE on remaining active sub-items.

LISTVIEW GET STATE *hDlg, id&, row&, col& TO datav&*

A data item is tested to see if it is currently selected. The values of *row&/col&* specify the position of the data item (1=first, 2=second, etc.). If the item is selected, -1 ([true](#)) is assigned to the variable specified by *datav&*. Otherwise, 0 ([false](#)) is assigned to it.

LISTVIEW GET STYLEXX *hDlg, id& TO datav&*

ListView controls offer a number of optional additional style attributes which are unique and specific to a ListView. This statement retrieves the current setting of this special extended style, and assigns it to the long integer variable specified by *datav&*. A list of the available extended styles can be found under LISTVIEW SET STYLEXXX. This special extended style is named STYLEXX to distinguish it from the primary style and extended style specified in CONTROL ADD LISTVIEW.

LISTVIEW GET TEXT *hDlg, id&, row&, col& TO txtv\$*

A string data item is retrieved from the LISTVIEW control and assigned to the string variable specified by *txtv\$*. The values of *row&/col&* specify the position of the data item (1=first, 2=second, etc.).

LISTVIEW GET USER *hDlg, id&, row& TO datav&*

Each row in a LISTVIEW may have a long integer user value associated with it at the discretion of the programmer. This user value is retrieved with LISTVIEW GET USER. The numeric value *row&* specifies which user value is requested, 1 for the first row, 2 for the second row, etc. The returned user value is assigned to the long integer variable specified by *datav&*. LISTVIEW user values are assigned with the LISTVIEW SET USER statement. In addition to these LISTVIEW user values, every [DDI](#) control

offers an additional eight user values which can be accessed with [CONTROL GET USER](#) and [CONTROL SET USER](#).

LISTVIEW INSERT COLUMN *hDlg, id&, col&, StrExpr, ColWidth&, format&*

A new vertical column is defined for Report Mode of this LISTVIEW control. The value *col&* specifies the column number (1=first, 2=second, etc.). *StrExpr* describes the text name of the column header. The value *ColWidth&* specifies the width of the column in either dialog units or pixels, depending upon which was specified at creation. The value *format&* describes the format and justification of the text: 0=left, 1=right, 2=center. Column 1 is always left-justified, regardless of what is requested here. When inserting a new column 1, the contents of the original column 1 are copied to the new column 1. This only occurs when inserting a new left most column, when inserting other columns, no data is copied to the new column. This is a limitation of the Microsoft Windows Listview control and not a Classic PowerBASIC limitation.

LISTVIEW INSERT ITEM *hDlg, id&, row&, image&, StrExpr*

A new row is added to this LISTVIEW control. The value *row&* specifies the row number (1=first, 2=second, etc.), and *StrExpr* tells the text to be displayed in the first column. The remaining columns are empty, but you can fill them by executing LISTVIEW SET TEXT. If an IMAGELIST has been attached to this control, the parameter *image&* specifies which image should be displayed (1=first, 2=second, etc.). If no image is needed, the value 0 should be used.

LISTVIEW RESET *hDlg, id&*

All data items are deleted from the specified LISTVIEW control. Any columns, and their associated headers, which may have been defined for Report Display mode are retained without change.

LISTVIEW SELECT *hDlg, id&, row& [, col&]*

The string data item specified by *row&/col&* is chosen as selected text for the LISTVIEW control and the item is highlighted. The values of *row&/col&* = 1 for the first item, 2 for the second item, etc. If the optional parameter *col&* is not given, the default value of 1 is used to select the primary data item.

LISTVIEW SET COLUMN *hDlg, id&, col&, NumExpr*

The width of a LISTVIEW column is changed to that designated by the *NumExpr*. The value is specified in either dialog units or pixels, depending

upon which was used at creation. The value *col&* specifies the column number (1=first, 2=second, etc.). If *NumExpr* is -1, then the column width is adjusted to fit the data items in that column. If *NumExpr* is -2, the column width is adjusted to fit both the data items and the header text. These options are functionally identical to LISTVIEW FIT CONTENT and LISTVIEW FIT HEADER.

LISTVIEW SET HEADER *hDlg, id&, col&, StrExpr*

New column header text is displayed above the specified column on the LISTVIEW control. The string expression *StrExpr* specifies the new header text, while the value *col&* specifies the column number (1=first, 2=second, etc.).

LISTVIEW SET IMAGE *hDlg, id&, row&, NumExpr*

The image specified by *NumExpr* (1=first, 2=second, etc.) is displayed next to the item specified by *row&*. If no IMAGELIST is attached to the LISTVIEW, nothing is displayed.

LISTVIEW SET IMAGE2 *hDlg, id&, row&, NumExpr*

The image specified by *NumExpr* (1=first, 2=second, etc.) is displayed as a secondary "status" image next to the primary image. If *NumExpr* evaluates to zero, no secondary image is displayed. A secondary image is usually used to specify item status, with an image such as a check mark. Secondary images are generally not displayed in either of the icon modes. If no Status Image List is attached to the LISTVIEW (using the LISTVIEW IMAGELIST statement), nothing is displayed. A maximum of 15 status images are supported, so *NumExpr* must evaluate in the range of 1-15.

LISTVIEW SET IMAGELIST *hDlg, id&, hLst, NumExpr*

The IMAGELIST specified by *hLst* is attached to this LISTVIEW control. The value of *NumExpr* specifies the type of IMAGELIST:

%LVSIL_NORMAL	Large icons
%LVSIL_SMALL	Small icons
%LVSIL_STATE	Status images

Up to three IMAGELIST structures may be attached to each LISTVIEW to display images as needed with each data item. Depending upon the mode in effect, icons are extracted from either the large icon or small icon list for that purpose. If a status image list is also attached, the LISTVIEW SET IMAGE2 statement may be used to display a secondary image. When the LISTVIEW control is destroyed, any attached IMAGELIST is automatically destroyed unless the [%LVS_SHAREIMAGELISTS](#) style was specified at

the time the LISTVIEW was created.

LISTVIEW SET MODE *hDlg, id&, NumExpr*

The display mode of the specified LISTVIEW control is changed to that designated by the value of *NumExpr*. The possible mode values are 0=icon mode, 1=report mode, 2=small icon mode, 3=list mode.

LISTVIEW SET OVERLAY *hDlg, id&, row&, NumExpr*

The overlay image specified by *NumExpr* (1=first, 2=second, etc.) is displayed on top of the image specified by *row&*. If *NumExpr* evaluates to zero, or if no IMAGELIST is attached to the LISTVIEW, no overlay is displayed.

LISTVIEW SET STYLEXX *hDlg, id&, NumExpr*

ListView controls offer a number of optional additional style attributes which are unique and specific to a ListView. This statement allows you to alter the current setting of this special extended style. This special extended style is named STYLEXX to distinguish it from the primary style and extended style specified in CONTROL ADD LISTVIEW. *NumExpr* defines the new style from any combination of the following extended styles:

%LVS_EX_GRIDLINES	Grid lines added in report mode
%LVS_EX_SUBITEMIMAGES	Icons added to sub-items in report mode
%LVS_EX_CHECKBOXES	Enables checkboxes to items
%LVS_EX_TRACKSELECT	Enables hot track selection
%LVS_EX_HEADERDRAGDROP	Enables drag-drop reordering of columns in report mode
%LVS_EX_FULLROWSELECT	Selection highlights full row in report mode
%LVS_EX_ONECLICKACTIVATE	Notification sent on single click
%LVS_EX_TWOCLICKACTIVATE	Notification sent on double click
%LVS_EX_FLATSB	Enables flat scroll bars
%LVS_EX_REGIONAL	Sets ListView region to icons and text
%LVS_EX_INFOTIP	ListView does InfoTips for you
%LVS_EX_UNDERLINEHOT	Hot items have underlined text
%LVS_EX_UNDERLINECOLD	Non-hot items have underlined text
%LVS_EX_MULTIWORKAREAS	Will not auto-arrange until work areas defined

%LVS_EX_LABELTIP	Listview unfolds partly hidden labels
%LVS_EX_BORDERSELECT	Border selection style instead of highlight
%LVS_EX_DOUBLEBUFFER	Paints via double-buffering and reduces flicker
%LVS_EX_HIDELABELS	Hides labels in Icon and Small Icon mode
%LVS_EX_SINGLEROW	Display a single row
%LVS_EX_SNAPTOGRID	Icons automatically snap to grid
%LVS_EX_SIMPLESELECT	Changes overlay rendering to top right

LISTVIEW SET TEXT *hDlg, id&, row&, col&, StrExpr*

The text, if any, for the specified data item is replaced by the new text in *StrExpr*. You must keep in mind that this statement does not create a new item (horizontal row), but changes existing text, if any, to new text. To create a new data item (horizontal row), use LISTVIEW INSERT ITEM instead. The values of *row&/col&* specify the position of the data item (1=first, 2=second, etc.).

LISTVIEW SET USER *hDlg, id&, row&, NumExpr*

Each row in a LISTVIEW may have a long integer user value associated with it at the discretion of the programmer. This user value is assigned with LISTVIEW SET USER, and retrieved with LISTVIEW GET USER. The numeric value *row&* specifies which user value is to be accessed, 1 for the first item, 2 for the second item, etc. The value specified by *NumExpr* is saved for later retrieval. In addition to these LISTVIEW user values, every DDT control offers an additional eight user values which can be accessed with CONTROL GET USER and CONTROL SET USER.

LISTVIEW SORT *hDlg, id&, col& [, options...]*

All of the items in a LISTVIEW are sorted, based upon the value of the data in a particular column. The column number (*col&*) is specified as 1 for the first column, 2 for the second column, etc. The options are one or more comma-delimited parameters which describe the sequence and the nature of the data in the sort-key column:

ASCEND	The items are arranged in ascending sequence.
DESCEND	The items are arranged in descending sequence.
ALPHANUM	The items consist of alphanumeric data. They are sequenced based upon the ASCII value of each byte, so that case is significant. Comparison is limited to the first

255 [bytes](#) of each string.

UCASE	The items consist of alphanumeric data. The case of each alphabetic character is not significant. This is accomplished by treating all alphabetic characters as upper case letters. Comparison is limited to the first 255 bytes of each string
NUMERIC	The items start with numeric data, and evaluation is stopped at the first non-numeric character. If numeric characters are not found, the value is assumed to be zero (0). This data may be in any supported Classic PowerBASIC format: integer, floating point, scientific notation, radix format, etc.
MMDDYYYY	A date in the format mm/dd/yyyy which is exactly ten bytes in length. Leading zeros may be replaced by spaces, and delimiters may be any character.
DDMMYYYY	A date in the format dd/mm/yyyy which is exactly ten bytes in length. Leading zeros may be replaced by spaces, and delimiters may be any character.
YYYYMMDD	A date in the format yyyy/mm/dd which is exactly ten bytes in length. Leading zeros may be replaced by spaces, and delimiters may be any character.
YYYYDDMM	A date in the format yyyy/dd/mm which is exactly ten bytes in length. Leading zeros may be replaced by spaces, and delimiters may be any character.

It is important to note that Windows may overwrite USER data when sorting your ListView control. You should avoid the use of the LISTVIEW GET USER and LISTVIEW SET USER statements if you may also execute a LISTVIEW SORT on the same control.

LISTVIEW UNSELECT *hDlg, id&, row& [, col&]*

The string value specified by row&/col& is set to an unselected state for the LISTVIEW control. The values of row&/col& = 1 for the first item, 2 for the second item, etc. If the optional parameter col& is not given, the default value of 1 is used to unselect the primary data item.

LISTVIEW VISIBLE *hDlg, id&, row&*

A row is scrolled, if necessary, to ensure that the data specified by row& is visible. The value of row& = 1 for the first row, 2 for the second row, etc.

Restrictions Under Windows 95/98/ME, a ListView is limited to 32,767 items. In all versions of Windows, the actual string data contained by the ListView is

limited only by available memory.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD LISTVIEW](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#), [IMAGELIST](#)

LO function

[Top](#) [Previous](#) [Next](#)

Purpose	Extract the least significant (low-order) portion of an integral value.
Syntax	<i>result</i> = LO(<i>type</i> , <i>value</i>)
Remarks	<p>The value returned by LO is unsigned if <i>type</i> is BYTE, WORD, or DWORD, and signed if <i>type</i> is INTEGER or LONG. <i>value</i> may be up to twice the size of the data type specified by <i>type</i>. In the following example, <i>n</i> may be up to a 16-bit value (twice the size of a BYTE):</p> <pre>b = LO(BYTE, n)</pre>
Restrictions	LO replaces LOBYT , LOWRD , and LOINT . Note that those functions may not be supported in future versions of PowerBASIC, so update your code to use the new syntax.
See also	HI , MAK

LOBYT function

[Top](#) [Previous](#) [Next](#)

Purpose Extract the least significant (low-order) byte from an [Integer](#) or [Word](#) value, and return it as an unsigned [Byte](#) value.

LOBYT has been superceded by the [LO](#) function, although LOBYT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.

Syntax *bResult?* = LOBYT(*sixteenbitvalue*)

Remarks The value returned by LOBYT is always unsigned, regardless of the sign of the argument. Effectively, this function provides the same result as [PEEKing](#) at the first byte of the argument.

See also [HI](#), [LO](#), [MAK](#)

LOC function

[Top](#) [Previous](#) [Next](#)

Purpose	Determine the current seek position in an open disk file .
Syntax	<i>qResult</i> && = LOC([#] <i>filenum</i> &)
Remarks	LOC is provided for compatibility with older BASICs. It is recommended that code is modified to use the SEEK function instead. The Number symbol (#) is optional, but recommended for clarity.
See also	FILEATTR , SEEK function , SEEK statement

LOCAL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Declare local variables inside a Sub , Function , Method , or Property . Local variables retain their values only until the end of the procedure.
Syntax	<pre>LOCAL variable[()] [AS type] [, variable[()]] [...] LOCAL variable[()] [, variable[()]] [, ...] AS type</pre>
Remarks	<p>The LOCAL statement is valid only inside a Sub, Function, Method, or Property. Local variables lose their values when the procedure ends. Storage space for local variables is allocated on the stack, and each local variable is initialized to zero (or, for string variables, an empty string) each time the enclosing procedure is called.</p> <p>To declare an array as a local variable, use an empty set of parentheses in the variable list: You can then use the DIM statement to dimension the array.</p> <pre>LOCAL MyArray%() LOCAL StringArray() AS STRING</pre> <p>The LOCAL statement may, optionally, accept a list of variables, all of which are defined by the type descriptor keyword that follows them. For example:</p> <pre>LOCAL aaa, bbb, ccc AS INTEGER LOCAL vptr, aptr() AS LONG PTR</pre>
Restrictions	DEFtype has no effect on variables defined by a LOCAL statement.
See also	DIM , GLOBAL , INSTANCE , STATIC , THREADED
Example	<pre>Test% = 100 ShowText "Before: " + STR\$(Test%) CALL Locals ShowText "After: " + STR\$(Test%) SUB Locals LOCAL Test% Test% = 0 ShowText "In SUB: " + STR\$(Test%) END SUB</pre>
Result	<pre>Before: 100 In SUB: 0 After: 100</pre>

Purpose	Lock part or all of an open file , to prevent other processes from accessing it.
Syntax	<code>LOCK [#] <i>filenum</i>& [, {<i>record</i>&& <i>start</i>&& TO <i>end</i>&&}]</code>
Remarks	<p>LOCK prevents another process from accessing a record, range of records, byte, or range of bytes in a file opened as file number <i>filenum</i>&.</p> <p>If the file was opened in random-access mode, <i>record</i>&&, <i>start</i>&&, and <i>end</i>&& specify record numbers. When used with binary mode files, <i>record</i>&&, <i>start</i>&&, and <i>end</i>&& specify byte positions, starting from either zero or one (the default).</p> <p>If a record is specified, only that record (or byte) is locked. Otherwise, a range of records (or bytes) is locked, from <i>start</i>&& to <i>end</i>&&.</p> <p>If no records are specified, or if the file was opened in sequential mode, the entire file is locked.</p> <p>All records (or bytes) to be locked must be subsequently unlocked using the UNLOCK statement. Multiple locks may be placed on a file, and locks may be unlocked in any order. However, the parameters used for each UNLOCK statement must exactly match those used for the previous corresponding LOCK statement.</p> <p>All locked records (or bytes) must be unlocked using the UNLOCK statement before the file can be closed.</p> <p>If a lock attempt fails, Classic PowerBASIC sets the ERR system variable to reflect a run-time Error 70 ("Permission denied"), or Error 75 ("Path/file access error").</p>
See also	OPEN , UNLOCK
Example	<pre>OPEN "PATIENTS.DAT" FOR RANDOM AS #1 LEN = 1024 ' determine the record number to retrieve LOCK #1, recnum GET #1, recnum ' process the record here PUT #1, recnum UNLOCK #1, recnum CLOSE #1</pre>

LOF function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the length of an open disk file .
Syntax	<i>y</i> && = LOF([#] <i>filenum</i> &)
Remarks	<i>filenum</i> & is the file number with which the file was opened. LOF returns the size of the indicated file in bytes , in the Quad-integer range 0 to 2^63-1. The Number symbol (#) is optional, but recommended for clarity.
See also	FILEATTR , LOC , SEEK function , SEEK statement
Example	<pre>OPEN "RECIPES.DAT" FOR BINARY AS #1 x&& = LOF(1) CLOSE #1</pre>

LOG, LOG2 and LOG10 functions

[Top](#) [Previous](#) [Next](#)

Purpose	LOG returns the natural (base e) logarithm of its argument. LOG2 returns the base 2 logarithm. LOG10 returns the common (base 10) logarithm.
Syntax	<pre>y = LOG (numeric_expression) y = LOG2 (numeric_expression) y = LOG10 (numeric_expression)</pre>
Remarks	<p>A logarithm of a number is the power to which the base would have to be raised to yield the number. Thus:</p> $\text{logarithm (base } b) \text{ of } n = x \quad \text{if} \quad b^x = n$ <p>and:</p> $(\text{base})^{\log(n)} = n$ <p>The EXP functions complement the LOG functions. For example, if $s = \text{LOG}(t)$, then $t = \text{EXP}(s)$.</p> <p>By definition, the logarithm (any base) of 1 is 0. LOG returns the natural logarithm (base e, where $e = 2.718282\dots$) of its argument. LOG2 and LOG10 return the logarithm for base 2 and 10, respectively.</p> <p><i>numeric_expression</i> must be a value greater than zero.</p> <p>LOG, LOG2, and LOG10 return Extended-precision values.</p>
See also	EXP , EXP2 , EXP10 , SQR , Arithmetic Operators

Purpose	<p>Extract the least significant (low-order) Word from a Long-integer or Double-word (DWORD) value, and return it as a signed Integer value.</p> <p>LOINT has been superceded by the LO function, although LOINT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre><i>iResult</i>% = LOINT(<i>thirtytwobitvalue</i>)</pre>
Remarks	<p>The value returned by LOINT is always signed, regardless of the sign of the argument.</p>
See also	LO

LOWRD function

[Top](#) [Previous](#) [Next](#)

Purpose	Extract the least significant (low-order) Word from a Long-integer or Double-word (DWORD) value, and return it as an unsigned Word value. LOWRD has been superceded by the LO function, although LOWRD remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.
Syntax	<i>wResult??</i> = LOWRD(<i>thirtytwobitvalue</i>)
Remarks	The value returned by LOWRD is always unsigned, regardless of the sign of the argument.
See also	LO

Purpose	Output (device-dependent) text and data to a printer device.
Syntax	<code>LPRINT [<i>expression</i>] [SPC(<i>n</i>)] [TAB(<i>n</i>)] [,] [;]</code>
Remarks	<p>The LPRINT functionality is identical to the traditional PRINT statement, except that the data is sent directly to a line printer rather than to a display. A line printer is one which will accept standard ASCII text and associated control codes, such as \$CR, \$LF, and \$FF.</p> <p>Classic PowerBASIC inserts a carriage return and linefeed at the end of each printed line. A semi-colon between expressions is an optional delimiter which leaves the printer column position unchanged. A comma moves the printer position to the next column of 14 positions each. A trailing semi-colon suppresses the final CR/LF. If TAB(<i>n</i>) is less than the current printer position, output is placed at the requested position on the following line.</p> <p>Before you execute an LPRINT statement, you must explicitly connect to the intended line printer using the LPRINT ATTACH statement. If the connection to the device is unsuccessful, all LPRINT statements are ignored until a valid printer device has been attached. LPRINT communicates directly with the attached device, bypassing the Windows operating system and printer driver. Therefore, any settings such as "work offline" in your printer properties dialog will be ignored.</p> <p>Once all the data has been sent to the printer, detach the printer so other applications can use it., with the LPRINT CLOSE statement</p> <p>Host-based (Windows-only) printers use proprietary control protocols so, sending print data to them with LPRINT is unlikely to produce any output at all. Classic PowerBASIC supports host-based printers through XPRINT and related statements.</p>
See also	LPRINT ATTACH , LPRINT CLOSE , LPRINT FLUSH , LPRINT FORMFEED , LPRINT\$, XPRINT , XPRINT ATTACH
Example	<pre>' Typical LPRINT printing strategy ERRCLEAR LPRINT ATTACH "LPT2" ' Use LPT2 device IF ISFALSE ERR AND ISTRUE LEN(LPRINT\$) THEN LPRINT "This is your line-printer talking" LPRINT FORMFEED ' Issue a formfeed LPRINT FLUSH ' flush the buffer LPRINT CLOSE ' detach the printer END IF</pre>

LPRINT ATTACH statement

[Top](#) [Previous](#) [Next](#)

Purpose	Connect to a line-printer device for use with LPRINT .
Syntax	<code>LPRINT ATTACH <i>device</i>\$</code>
Remarks	<p>LPRINT ATTACH attempts a direct connection to the specified [line] printer device. A line printer is one that will accept standard ASCII text and any device-specific control codes, such as CR, LF, and FF.</p> <p>A line printer is named by the port to which it is attached (LPT1, etc.) because the data is sent directly to the port, not through a device driver. That is, LPRINT communicates directly with the attached line printer device, bypassing the spooler and printer driver. Therefore, any settings such as "work offline" in the Printer Properties dialog will be ignored.</p> <p>Once the printer is attached by LPRINT ATTACH, print data can be sent to it with the LPRINT statement.</p> <p>LPRINT ATTACH allows you to change the printer device used by LPRINT operation. When executed, the current connection (if any) is closed and the new connection is established. No colon is used in the device name. For example, to connect to <i>LPT2</i>:</p> <pre>LPRINT ATTACH "LPT2"</pre> <p>or to a printer on a network server:</p> <pre>LPRINT ATTACH "\\SERVER\HPLJ5"</pre> <p><i>device</i>\$ must be a valid device name and cannot exceed 32 characters in length. In some circumstances, such as with the Novell network client, LPRINT ATTACH with a UNC name may be rejected, and the LPRINT ATTACH will be unsuccessful, and a subsequent LPRINT\$ test will return an empty string.</p> <p>If LPRINT ATTACH is not successful, an Error 68 ("Device unavailable") is generated and LPRINT\$ returns a nul (empty) string. If no LPRINT ATTACH is ever executed (successful or not), Classic PowerBASIC will attempt to connect to the line printer at LPT1. Once any LPRINT ATTACH is attempted, no default to LPT1 will be presumed.</p> <p>Care must be used with line printers in Windows, since if there is no available printer attached to the port, program execution may be suspended, with no errors. So, it is wise to use LPRINT ATTACH to explicitly connect the intended printer device, and test for the successful connection by the examination of LPRINT\$ and ERR. For example:</p> <pre>ERRCLEAR LPRINT ATTACH "LPT3" IF ERR OR LPRINT\$ = "" THEN PRINT "Connection failed"</pre> <p>Once all the data has been sent to the printer, detach the printer so other</p>

applications can use it., with the [LPRINT CLOSE](#) statement

Note: The Win32 API call *EnumPrinters* can give you a list of all valid printers and print devices, or you can enumerate the list of printers with the [PRINTERCOUNT](#) and [PRINTER\\$](#) functions.

Restrictions If *device\$* is an empty string, the current connection (if any) is detached. This is equivalent to the LPRINT CLOSE statement.

See also [LPRINT](#), [LPRINT CLOSE](#), [LPRINT FLUSH](#), [LPRINT FORMFEED](#), [LPRINT\\$](#), [XPRINT](#), [XPRINT ATTACH](#)

Example

```
' Typical LPRINT printing strategy
ERRCLEAR
LPRINT ATTACH "LPT2" ' Use LPT2 device
IF (ERR<>0) OR (LEN(LPRINT$)) THEN
  LPRINT "This is your line-printer talking"
  LPRINT FORMFEED      ' Issue a formfeed
  LPRINT FLUSH         ' flush the buffer
  LPRINT CLOSE        ' detach the printer
END IF
```

LPRINT CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Disconnect the current printer device.
Syntax	LPRINT CLOSE
Remarks	<p>LPRINT CLOSE detaches the currently selected printer connection (established with the LPRINT ATTACH statement) from LPRINT operations, allowing the spooler subsystem to commence print operations. Once a connection is closed, LPRINT\$ will return an empty printer device name string until a new connection is established.</p> <p>LPRINT CLOSE is equivalent to using LPRINT ATTACH with an empty printer device name string.</p>
Restrictions	LPRINT CLOSE is an essential step in the print process. To ensure the printer device is available to other applications, printers should always be closed when not in use. Failing to close a connection may cause significant delays before printing commences. In some cases, some or all of the print data may be lost.
See also	LPRINT , LPRINT ATTACH , LPRINT FLUSH , LPRINT FORMFEED , LPRINT\$, XPRINT , XPRINT ATTACH
Example	<pre>' Typical LPRINT printing strategy ERRCLEAR LPRINT ATTACH "LPT2" ' Use LPT2 device IF ISTRUE ERR OR ISFALSE LEN(LPRINT\$) THEN LPRINT "This is your line-printer talking" LPRINT FORMFEED ' Issue a formfeed LPRINT FLUSH ' flush the buffer LPRINT CLOSE ' detach the printer END IF</pre>

LPRINT FLUSH statement

[Top](#) [Previous](#) [Next](#)

Purpose	Flush any remaining print data to a printer device and signal the start of the print process.
Syntax	<code>LPRINT FLUSH</code>
Remarks	<p>LPRINT FLUSH forces the operating system to flush any buffered data and begin printing. Use LPRINT FLUSH to ensure print data is submitted to the printer as soon as possible, rather than waiting for any timeout period to elapse first. Depending upon the printer and its drivers, printing may begin immediately, or it may be delayed until execution of an LPRINT CLOSE statement.</p> <p>Typically, an LPRINT FLUSH statement is preceded with a FORMFEED statement, so ensure that the print job is ejected normally from the printer device.</p>
See also	LPRINT , LPRINT ATTACH , LPRINT CLOSE , LPRINT FORMFEED , LPRINT\$, XPRINT , XPRINT ATTACH
Example	<pre>' Typical LPRINT printing strategy ERRCLEAR LPRINT ATTACH "LPT2" ' Use LPT2 device IF ISTRUE ERR OR ISFALSE LEN(LPRINT\$) THEN LPRINT "This is your line-printer talking" LPRINT FORMFEED ' Issue a formfeed LPRINT FLUSH ' flush the buffer LPRINT CLOSE ' detach the printer END IF</pre>

LPRINT FORMFEED statement

[Top](#) [Previous](#) [Next](#)

Purpose	Send a formfeed (page eject) character to an attached printer device.
Syntax	<code>LPRINT FORMFEED</code>
Remarks	<p>For direct connections, LPRINT FORMFEED sends a form-feed character (ASCII character 12, \$FF, or CHR\$(12)) to the attached line printer device, to ensure the current page will be ejected. For host-based connections, Classic PowerBASIC signals to the printing subsystem to perform the page eject operation.</p> <p>Typically, an LPRINT FORMFEED is performed before a LPRINT FLUSH and LPRINT CLOSE.</p>
See also	LPRINT , LPRINT ATTACH , LPRINT CLOSE , LPRINT FLUSH , LPRINT\$, XPRINT , XPRINT ATTACH
Example	<pre>' Typical LPRINT printing strategy ERRCLEAR LPRINT ATTACH "LPT2" ' Use LPT2 device IF ISTRUE ERR OR ISFALSE LEN(LPRINT\$) THEN LPRINT "This is your line-printer talking" LPRINT FORMFEED ' Issue a formfeed LPRINT FLUSH ' flush the buffer LPRINT CLOSE ' detach the printer END IF</pre>

LPRINT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the name of the printer device used for LPRINT operations.
Syntax	<code>device\$ = LPRINT\$</code>
Remarks	<p>LPRINT\$ returns the name of the currently attached printer device used by the LPRINT statement. If there is no attached device, an empty string is returned.</p> <p>LPRINT\$ is primarily used to detect if an LPRINT ATTACH operation was successful.</p>
See also	LPRINT , LPRINT ATTACH , LPRINT CLOSE , LPRINT FLUSH , LPRINT FORMFEED , XPRINT , XPRINT ATTACH
Example	<pre>ERRCLEAR LPRINT ATTACH "LPT3" IF ERR <> 0 OR LPRINT\$ = "" THEN PRINT "Printer connection failed"</pre>

Purpose	Left-align a string within the space of another string or User-Defined Type .
Syntax	<code>LSET [ABS] result_var = string_expression [USING ustring_expression]</code>
Remarks	LSET left-aligns a string into the space of another string or variable of a User-Defined Type.
ABS	If ABS is specified, or <i>ustring_expression</i> is null (empty), LSET leaves the padding positions unchanged from their original content, rather than replacing them with spaces.
USING	<p>If string_expression is shorter than <i>result_var</i>, LSET left-justifies <i>string_expression</i> within <i>result_var</i>, and pads remaining character positions on the right side using the first character in <i>ustring_expression</i> or spaces if not specified or is null (empty).</p> <p>If <i>string_expression</i> is longer than <i>result_var</i>, LSET truncates <i>string_expression</i> from the right until it fits in <i>result_var</i>.</p> <p>LSET can be used to assign the content of a User-Defined Type to a User-Defined Type variable of a different class, or assign a dynamic string to a User-Defined Type. For example:</p> <pre>LSET MyType = STRING\$(LEN(MyType) , 0) LSET MyType = a\$</pre> <p>RSET works similarly, but performs right-justification; CSET performs center-justification.</p>
See also	CSET , CSET\$, GET , LET , LET (with Types) , LSET\$, PUT , RESET , RSET , RSET\$, STRINSERT\$, TYPE SET
Example	<pre>a\$ = "SuperBASIC=SuperBASIC" LSET ABS a\$ = "PowerBASIC" ' result: "PowerBASIC=SuperBASIC" LSET a\$ = "PowerBASIC" USING "*" ' result: "PowerBASIC*****"</pre>

LSET\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string containing a left-justified (padded) string.
Syntax	<code>a\$ = LSET\$(string_expression, strlen& [USING ustring_expression])</code>
Remarks	LSET\$ left-aligns the string string_expression into a string of <i>strlen&</i> characters.
USING	<p>If <i>ustring_expression</i> is null (empty) or is not specified, LSET\$ pads <i>string_expression</i> with space characters. Otherwise, LSET\$ pads the string with the first character of <i>ustring_expression</i>.</p> <p>If <i>string_expression</i> is shorter than <i>strlen&</i>, LSET\$ left-justifies <i>string_expression</i> within <i>result_var</i>, padding the right side as described above; otherwise, LSET\$ returns the left-most <i>strlen&</i> bytes of <i>string_expression</i>.</p>
See also	CSET , CSET\$, GET , LET , LSET , PUT , RESET , RSET , RSET\$, STRINSERT\$, TYPE SET
Example	<pre>a\$ = LSET\$("PowerBASIC", 20) ' result: "PowerBASIC "</pre> <pre>a\$ = LSET\$("PowerBASIC",20 USING "*") ' result: "PowerBASIC*****"</pre>

LTRIM\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a copy of a string, with leading characters or strings removed.
Syntax	<code>x\$ = LTRIM\$(MainString [, [ANY] MatchString])</code>
Remarks	<p><i>MainString</i> is the string expression from which to remove characters, and <i>MatchString</i> is the string expression containing the characters to remove.</p> <p>If <i>MatchString</i> is not specified, LTRIM\$ removes leading spaces. LTRIM\$ returns a sub-string of <i>MainString</i>, from the first non-<i>MatchString</i> (or non-space) to the end of the string. If <i>MatchString</i> (or a space) is not present at the beginning of <i>MainString</i>, all of <i>MainString</i> is returned.</p> <p>If the ANY keyword is included, <i>MatchString</i> specifies a list of single characters to be searched for individually. A match on any one of these as a leading character will cause the character to be removed from the result. LTRIM\$ is case sensitive.</p>
See also	EXTRACT\$, INSTR , LEFT\$, MID\$, REMOVES\$, REPLACE , RIGHT\$, RTRIM\$, STRDELETES\$, STRINSERT\$, STRREVERSE\$, TALLY , TRIM\$, VERIFY
Example	<pre>A\$ = "0123ABC3210" A\$ = LTRIM\$(A\$, ANY "0123456789")</pre>
Result	ABC3210

Purpose Define a single or multi-line text substitution block.

Syntax *Single line macro:*

```
MACRO macroname [(prm1, prm2, ...)] = replacementtext
```

Multi-line macro:

```
MACRO macroname [(prm1, prm2, ...)]  
  [MACROTEMP ident1 [, ident2, ...]]  
  DIM ident1 AS type [, ident2 AS type, ...]]  
  {replacementtext}  
  [EXIT MACRO]  
  {replacementtext}  
END MACRO
```

Macro function:

```
MACRO FUNCTION macroname [(prm1, prm2, ...)]  
  [MACROTEMP ident1 [, ident2, ...]]  
  DIM ident1 AS type [, ident2 AS type, ...]]  
  {replacementtext}  
  [EXIT MACRO]  
  {replacementtext}  
END MACRO = returnexpression
```

Remarks Macro is a powerful text substitution construct that may take a single-line or multi-line format. It generates absolutely no executable code unless it is referenced, and effectively allows the programmer to design a part of the Classic PowerBASIC language to his/her own needs and requirements. For example, a simple single-line macro can allow Classic PowerBASIC to emulate the CONST syntax used in Visual Basic - see the box-out below for more information.

A macro must always be defined before it is referenced, and the parameter count must always match the definition. When a macro is referenced, the occurrence of the name is replaced by the defined replacement text, expanded with parameter substitution. The first line of a MACRO definition is termed the macro prototype, and this line may not be split into multiple logical lines with the underscore ([_](#)) [line continuation](#) character. Likewise, the END MACRO = *returnexpression* may not be split with underscores either. A Macro also cannot end with a line continuation character.

Macros may be nested, and may forward-reference other macros. However, care should be exercised to avoid circular references.

A single-line macro or a macro function may be referenced at any source code position which, when expanded, will be syntactically correct (also see the Restrictions section below). Consider the following simplistic example:

```
MACRO concatenate(prm1,prm2) = prm1 & $SPC & prm2  
  ' more code here
```

```
A$ = concatenate("Hello", "World")
```

During compilation, Classic PowerBASIC would internally expand this code to become:

```
A$ = "Hello" & $SPC & "World"
```

A multi-line macro, while more powerful in terms of coding, may be referenced only in the "statement" position, which is the first position on a line. That single reference is internally expanded into multiple lines of inline code to perform a complex task. For example:

```
MACRO Display6times(prm1)
  CALL Display(prm1) : CALL Display(prm1)
  CALL Display(prm1) : CALL Display(prm1)
  CALL Display(prm1) : CALL Display(prm1)
END MACRO
' more code here
Display6times("This is very cool...")
```

The single-line MACRO offers a cunning way to retain the CONST syntax used in MSBASIC and Visual Basic in your Classic PowerBASIC code, while maintaining the low overhead advantage of Classic PowerBASIC. For example:

```
MACRO CONST = MACRO
' more code here
CONST Version = 1&
CONST AppTitle = "My Application"
' more code here
a$ = AppTitle & " v" & FORMAT$(Version)
```

During compilation, the CONST keyword is replaced by the MACRO keyword, dynamically creating a new macro that, in turn, defines a [numeric or string literal](#). When the real macro name is referenced in the code, the literal is substituted directly.

MACROTEMP The MACROTEMP statement may be used to specify a list of one or more identifiers, each of which is automatically made unique to each expansion of a multi-line macro. This is done by internally appending the digits 0001, 0002, etc, to the identifier upon each expansion of the macro.

A text identifier may represent a [variable](#), [label](#), or any other word, which expands appropriately to avoid a duplicate name conflict in your code.

MACROTEMP just creates a symbol name. If this symbol is a variable name, the variable must still be formally declared with an appropriate [DIM](#) (or [LOCAL](#)) statement. For example:

```
MACRO CopyUntilNul(ptr1, ptr2)
  MACROTEMP LoopPoint, ByteVar
  DIM ByteVar AS BYTE
LoopPoint:
  ByteVar = @ptr1
  @ptr2 = ByteVar
  INCR ptr1
  INCR ptr2
  IF ByteVar <> 0 THEN LoopPoint
END MACRO
```

Using that MACRO definition, the code "CopyUntilNul(Source, Dest)" would expand to something like this:

```
DIM ByteVar0001 AS BYTE
LoopPoint0001:
  ByteVar0001 = @Source
  @Dest = ByteVar0001
  INCR Source
  INCR Dest
  IF ByteVar0001 <> 0 THEN LoopPoint0001
```

If the MACROTEMP statement were not used, serious naming conflicts would occur most any time that a macro was expanded more than once in a program. MACROTEMP statements may appear 0, 1, or more times in a macro definition, but they must always precede any other text in the macro.

MACROTEMP statements should be used with any label in a macro that may be expanded more than once in a program, and with any variable that should not be shared with any other expansion of the macro.

EXIT MACRO EXIT MACRO may be used to terminate execution of code in the current macro expansion. It is functionally identical to the *imaginary* concept of [GOTO](#) END-MACRO.

END MACRO A macro function block can return a value with the END MACRO = *returnexpression* statement.

Restrictions A macro definition may contain replacement text up to approximately 4000 characters. Macros may specify up to 240 parameters, which may occupy up to approximately 2000 bytes total expanded space per macro.

Macro Function substitutions are limited to an expanded total of approximately 16000 characters per line of original source code.

Macro parameters are substituted directly, so whitespace characters in the passed macro parameters may cause unexpected problems if the expanded code is syntactically incorrect with the additional whitespace. For example, this can be important when specifying [UDT](#) variables as macro parameters. Consider the following code:

```
TYPE MyType
  lCount AS LONG
  szText AS ASCIIZ * 256
END TYPE

MACRO PresetUDT(u)
  u.lCount = 1
  u.szText = SPACE$(256)
END MACRO

FUNCTION PBMAIN
  DIM x AS MyType
  PresetUDT(x)
  PresetUDT(x ) ' This line causes an Error 526
```

END FUNCTION

In the code above, the second macro expansion fails to compile because the trailing space in the passed macro parameter becomes part of the expanded code. In this situation, this additional space character breaks the syntax of the UDT variable reference within the expanded macro, triggering a compile-time [Error 526](#) ("Period not allowed"). If we examine how the two expanded macro statements would appear, the problem becomes immediately obvious:

```
x .lCount = 1
^
x .szText = SPACE$(256)
^
```

(Please note that the caret symbols (^) above have been added purely to illustrate the exact position of the problem)

When using single-line macros that contain numeric expressions, use parentheses around the macro body to guard against unexpected order of precedence problems when the macro is used within an expression. For example, consider the following macro and expansion:

```
MACRO Calculate(p1, p2, p3) = (p1 * p2) \ p3
' more code here
x = Calculate(a,b,c) ^ 3
```

When this macro is expanded, the expression would be calculated as follows:

```
x = (a * b) \ c ^ 3
```

However, if the macro body was enclosed in parentheses:

```
MACRO Calculate(p1, p2, p3) = ((p1 * p2) \ p3)
```

then the expanded expression would be calculated thus:

```
x = ((a * b) \ c) ^ 3
```

MACRO prototypes (those beginning with the MACRO keyword) and END MACRO = *returnexpression* lines must be constructed on a single line of source code. That is, they may not be split across multiple lines of source code with line continuation characters, since these interfere with the text substitution process. For example, the following prototype is invalid:

```
MACRO FUNCTION MyMacro1(sParam1, sParam2, sParam3, sParam4)
```

If a macro expands directly to a Function call, the macro can be called using the [SUB](#)-style syntax, automatically discarding the function return value. For example:

```
MACRO sm(Msg) = SendMessage(a, Msg, b, c)
```

can be called like this (if the return value is not required):

```
sm(x)
```

A macro cannot expand directly to a REMark, because [REM](#) and `'` are processed before the macro is assigned. So, MACRO hello = REM winds up as an invalid, blank macro.

Finally, it should be noted that the [Integrated Debugger](#) appears to step over macro references as if they were conventional BASIC statements. This occurs because macro expansion takes place during the compilation process and the original source code is not affected or altered by the compile-time expansion.

See also

[EXIT](#), [FUNCTION/END FUNCTION](#), [METHOD](#), [PROPERTY](#), [SUB/END SUB](#)

Example

```
' Single-line macro:
MACRO muldivide(p1, p2, p3) = ((p1 * p2) / p3)
' more code here
x = muldivide(3,3,2) + 10

' Multi-line macro and macro function example:
MACRO FUNCTION HowDidIGetHere
    MACROTEMP i, a
    DIM i AS LONG, a$
    FOR i = CALLSTKCOUNT TO 1 STEP -1
        A$ = A$ + CALLSTK$(i) + ", "
    NEXT
END MACRO = RTRIM$(A$, ANY ", ")

MACRO DisplayText(txt)
    #IF %DEF(%PB_CC32)
        PRINT txt
    #ELSE
        MSGBOX txt
    #ENDIF
END MACRO

SUB Testing2(r AS LONG,z AS ASCIIZ)
    DisplayText(HowDidIGetHere)
END SUB

SUB testing1(z AS ASCIIZ)
    DisplayText(HowDidIGetHere)
    CALL Testing2(1,z)
END SUB

FUNCTION PBMAIN
    DisplayText(HowDidIGetHere)
    CALL Testing1("This is a test")
END FUNCTION

' Useful Macro functions
MACRO Pi = 3.141592653589793##
MACRO DegreesToRadians(dpDegrees) = (dpDegrees * 0.0174532925199433##)
MACRO RadiansToDegrees(dpRadians) = (dpRadians * 57.29577951308232##)
MACRO Hex64(x) = HEX$(HI(DWORD,x),8) + HEX$(LO(DWORD,x),8)
```

MAK function

[Top](#) [Previous](#) [Next](#)

Purpose	Create an integer class value of a specified data type.
Syntax	<i>resultvar</i> = MAK(<i>datatype</i> , <i>loworderval</i> , <i>highorderval</i>)
Remarks	<p>Create an integer class value of a specified data type (WORD, DWORD, INTEGER, LONG, QUAD) from a low-order and a high-order part.</p> <p>The complements to this function are the HI and LO functions, which may be used to split a single 32-bit value into two 16-bit components.</p>
Restrictions	<p>MAK supercedes the MAKWRD, MAKDWD, and MAKPTR functions. Those functions may not be supported in future versions of Classic PowerBASIC, so update your code to use the new syntax.</p>
See also	HI , LO
Example	<pre>dwResult = MAK(DWORD, x??, y??)</pre>

MAKDWD, MAKLNG and MAKPTR functions

[Top](#) [Previous](#)
[Next](#)

Purpose	<p>Take the low-order 16-bits from each of two integer class variables and combine to produce a single 32-bit value.</p> <p>MAKDWD, MAKLNG and MAKPTR have been superceded by the MAK function, although they remain supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre>value??? = MAKDWD(<i>loword</i>, <i>hiword</i>) value& = MAKLNG(<i>loword</i>, <i>hiword</i>) value??? = MAKPTR(<i>loword</i>, <i>hiword</i>)</pre>
Remarks	<p>MAKDWD, MAKLNG and MAKPTR take the low-order 16-bits from each of <i>loword</i> and <i>hiword</i>, and combine them into a single 32-bit value. MAKDWD returns an unsigned 32-bit DWORD value, MAKLNG returns a signed 32-bit Long-integer value and MAKPTR returns an unsigned 32-bit DWORD (pointer) value.</p>
<i>loword</i>	Forms the least significant or "low-order" 16-bits of the 32-bit result value.
<i>hiword</i>	Forms the most significant or "high-order" 16-bits of an 32-bit result value.
See also	MAK

Purpose

Take the low-order byte from each of two integer class variables and combine to produce a single 16-bit value.

MAKINT and MAKWRD have been superceded by the [MAK](#) function, although they remain supported for a limited period. Existing code should be converted to the new syntax as soon as possible.

Syntax

```
value% = MAKINT(lobyte, hibyte)  
value?? = MAKWRD(lobyte, hibyte)
```

Remarks

MAKINT and MAKWRD take the low-order byte from each of *lobyte* and *hibyte*, and combine them into a single 16-bit value. MAKINT returns a signed 16-bit [integer](#) value, and MAKWRD returns an unsigned 16-bit [WORD](#) value.

lobyte Forms the least significant or "low-order" byte of the 16-bit result value.

hibyte Forms the most significant or "high-order" byte of the 16-bit result value.

See also

[MAK](#)

Purpose

To simplify Matrix Algebra calculations.

Syntax

<code>MAT a1() = CON</code>	'Set all elements of a1() to one
<code>MAT a1() = CON(expr)</code>	'Set all elements of a1() to value of expr
<code>MAT a1() = IDN</code>	'Establish a1() as an identity matrix
<code>MAT a1() = ZER</code>	'Set all elements of a1() to zero
<code>MAT a1() = a2() + a3()</code>	'Addition
<code>MAT a1() = a2()</code>	'Assignment
<code>MAT a1() = INV(a2())</code>	'Inversion
<code>MAT a1() = (expr) * a2()</code>	'Scalar Multiplication
<code>MAT a1() = a2() - a3()</code>	'Subtraction
<code>MAT a1() = a2() * a3()</code>	'Multiplication
<code>MAT a1() = TRN(a2())</code>	'Transposition

Remarks

[Array](#) names with the MAT statements may optionally include a set of empty parentheses. The following are both equally valid, but the inclusion of the parentheses improves clarity of the code:

```
MAT a1 = CON
MAT a1() = CON
```

MAT CON, IDN ZER + - = and TRN operations are valid with [Byte](#), [Word](#), [Double-word](#), [Integer](#), [Long-integer](#), [Quad-integer](#), [Single-precision](#), [Double-precision](#) and [Extended-precision arrays](#).

Matrix * and INV operations support all numeric types.

It is the programmer's responsibility to ensure that arrays used with MAT are of the appropriate size and type. All operations involving two or more arrays require that they be of exactly the same size and type, without exception. Failure to adhere causes undefined results. In the interest of execution speed, no error checking is performed at run-time.

Every scalar value denoted here as 'expr' must be enclosed in parentheses. Although Matrix operations tend to imply a [two-dimensional array](#), unless otherwise noted (such as with MAT IDN, *, TRN), MAT may be used with arrays of one to eight dimensions. It is permissible to specify one array for multiple MAT parameters.

Example

```
MAT array1() = IDN
```

This establishes *array1* as an identity matrix, with all diagonal elements as 1 and all others as zero. This produces undefined results if *array1* is not a "square" matrix.

```
MAT array1() = (expr) * array2()
```

Each element of *array2* is multiplied by the scalar value of the *expr*, then assigned to *array1*.

```
MAT array1() = TRN(array2())
```

Transposes the row and columns from *array2* to *array1*. Arrays must be equivalent: *array1*(5,2) and *array2*(2,5). Only a square matrix may be

transposed to itself.

```
MAT array1() = INV(array2())
```

Inverts the array from *array2* to *array1*. Only a square matrix may be inverted. Proof: If *array1* is then multiplied by *array2*, the resulting "*array3*" will be equal to an Identify Matrix, (MAT *array3* = *array1* * *array2* ' *array3* should now be equal to "MAT array3 IDN").

```
MAT a() = b() * c()
```

Array multiplication occurs as follows:

```
' Row Column assumption:
'   array [a]l,n = [b]l,m * [c]m,n

FOR i = 1 TO l           ' Row      [a]l = Row      [b]l
  FOR j = 1 TO n         ' Column [a]n = Column [c]n
    a(i,j) = 0#          ' # if Double-precision
    FOR k = 1 TO m       ' Column [b]m = Row      [c]m
      a(i,j) = a(i,j) + b(l,k) * c(k,j)
    NEXT
  NEXT
NEXT
```

MAX function

[Top](#) [Previous](#) [Next](#)

Purpose Return the argument with the largest (maximum) value.

Syntax

```
y = MAX(arg [, arg] ...)  
y& = MAX&(arg& [, arg&] ...)  
y$ = MAX$(arg$ [, arg$] ...)
```

Remarks These functions take any number of arguments and return the argument with the largest (maximum) value. MAX handles arguments of any numeric type.

MAX& handles arguments which evaluate to [Long-integers](#) (MAX& is more efficient than MAX).

MAX\$ handles string arguments.

If any arguments of MAX& are outside of the range of Long-integers, the result is undefined. Any floating point arguments of MAX& will be rounded to Long-integers before the comparison begins.

MAX% is recognized as a valid synonym for MAX&.

See also [CHOOSE](#), [CHOOSE&](#), [CHOOSE\\$](#), [IIF](#), [IIF&](#), [IIF\\$](#), [MIN](#), [MIN&](#), [MIN\\$](#), [SWITCH](#), [SWITCH&](#), [SWITCH\\$](#)

Example

```
x% = MAX&(A, B, C, D)  
x$ = MAX$("abacadabra", "cad", A$, B$(4), C$+D$+LEFT$(E$,5))  
x## = MAX(1.1@@, A%/B!, C#(x)^D, E##, SIN(F&))
```

MCASE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a mixed case version of its string argument.
Syntax	<code>s\$ = MCASE\$(string_expression [,ANSI OEM])</code>
Remarks	<p>MCASE\$ returns a string equivalent to <i>string_expression</i>, except that the first letter of each word is capitalized, while the remaining characters are forced to lowercase. A word is considered to be a consecutive series of letters. The optional ANSI or OEM parameter specifies whether the conversion is made using the ANSI charset for the system, or the original IBM OEM charset. If no charset is specified, Classic PowerBASICfor Windows uses the system ANSI charset, while PB/CC uses the IBM OEM charset. Only "International" characters in the range of CHR\$(128) to CHR\$(255) are affected by this parameter.</p> <p>The OEM charset is based upon the original IBM OEM charset to ensure compatibility with programs written for all previous versions of the Classic PowerBASIC compiler.</p>
See also	LCASE\$, UCASE\$
Example	<code>x\$ = MCASE\$("Cats aren't AL.WAYS good.")</code>
Result	<code>Cats Aren'T Al.Ways Good.</code>

Purpose	A pseudo object variable to reference the current object.
Syntax	<code>ME.Method1(<i>param</i>)</code>
Remarks	<p>ME is a pseudo-variable, which Classic PowerBASIC automatically defines in every Method and Property. It is treated as a reference to the current object. Using ME, it's possible to call any other Method or Property which is a member of the class: <code>var = ME.Method1(<i>param</i>)</code></p> <p>ME can also be assigned to an appropriate object variable, or used as a Sub/Function/Method/Property parameter.</p>
See also	CLASS , INTERFACE (Direct) , INTERFACE (IDBind) , METHOD , PROPERTY , What is an object, anyway?

MENU ADD POPUP statement

[Top](#) [Previous](#) [Next](#)

Purpose	Add a popup child menu to an existing menu.
Syntax	<code>MENU ADD POPUP, <i>hMenu</i>, <i>txt\$</i>, <i>hPopup</i>, <i>state&</i> [, AT [BYCMD] <i>position&</i>]</code>
Remarks	A popup menu is a small window that "pops up" when a menu item is highlighted. This allows nesting, and gives the user an opportunity to choose from "sub-menu" items.
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the parent menu to which you are adding the popup child menu
<i>txt\$</i>	Text displayed in the parent menu. An ampersand (&) may be used in the string to make the following letter into a control accelerator (hot-key). The letter appears underscored to signify that it is an accelerator.
<i>hPopup</i>	Double-word or Long-integer containing the handle of the child popup menu you are adding to the parent menu.
<i>state&</i>	The initial state of the menu item. It can be one of the following: %MF_DISABLED Disable the item so that it cannot be selected. %MF_ENABLED Enable the item so that it can be selected.
<i>position&</i>	Indicates the position in the parent menu where the popup child menu is to be inserted. If the BYCMD option is used, the popup menu is inserted prior to the menu item ID specified by <i>position&</i> . Otherwise, the popup menu is inserted at the physical <i>position&</i> within the parent menu, where <i>position&</i> = 1 for the first position, <i>position&</i> = 2 for the second, and so on. If position is not specified then the popup menu is appended to the end of the menu.
See also	Dynamic Dialog Tools , ACCEL ATTACH , Menus , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT
Example	See Menu Example .

Purpose	Add a string or separator to an existing menu .										
Syntax	<pre>MENU ADD STRING, hMenu, txt\$, id&, state& [, AT [BYCMD] position&] [, CALL callback]</pre>										
Remarks	A string may contain an optional command accelerator key, and also describe an equivalent keyboard accelerator combination.										
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the parent menu to which the string should be added.										
<i>txt\$</i>	<p>Text to display in the parent menu. An ampersand (&) may be used in the string to make the following letter into a command accelerator (hot-key). The letter is underscored to signify that it is an accelerator. To create a horizontal separator instead of a text string, set <i>txt\$</i> = "-", <i>id&</i> = 0, <i>state&</i> = 0.</p> <p>Keyboard accelerators, as described in the ACCEL ATTACH statement, can be indicated in the text of a menu item, for the reference of the user. To include a keyboard accelerator description in a menu string, separate it from the menu item text with a \$TAB {CHRS\$(9)} character. For example:</p> <pre>MENU ADD STRING, hMenu, "Cu&t" & \$TAB & "CTRL+X", id&, mstate&</pre>										
<i>id&</i>	The unique numeric identifier for the menu item. When a menu item is selected, <i>id&</i> is sent to the parent dialog Callback Function to notify the dialog which option was selected.										
<i>state&</i>	<p>The initial state of the menu item. It can be one or more of the following, combined together with the OR operator to form a bitmask:</p> <table><tr><td>%MF_CHECKED</td><td>Place a checkmark next to the item.</td></tr><tr><td>%MF_DISABLED</td><td>Disable the menu item so that it cannot be selected.</td></tr><tr><td>%MF_ENABLED</td><td>Enable the menu item so that it can be selected.</td></tr><tr><td>%MF_GRAYED</td><td>Disable the menu item so that it cannot be selected, and draw it in a "grayed" state to indicate this.</td></tr><tr><td>%MF_UNCHECKED</td><td>Do not place a checkmark next to the item.</td></tr></table>	%MF_CHECKED	Place a checkmark next to the item.	%MF_DISABLED	Disable the menu item so that it cannot be selected.	%MF_ENABLED	Enable the menu item so that it can be selected.	%MF_GRAYED	Disable the menu item so that it cannot be selected, and draw it in a "grayed" state to indicate this.	%MF_UNCHECKED	Do not place a checkmark next to the item.
%MF_CHECKED	Place a checkmark next to the item.										
%MF_DISABLED	Disable the menu item so that it cannot be selected.										
%MF_ENABLED	Enable the menu item so that it can be selected.										
%MF_GRAYED	Disable the menu item so that it cannot be selected, and draw it in a "grayed" state to indicate this.										
%MF_UNCHECKED	Do not place a checkmark next to the item.										
<i>position&</i>	Optional position in the parent menu, where the menu item should be inserted. If the BYCMD option is used, the menu item is inserted prior to the menu item ID specified by <i>position&</i> . Otherwise, the menu item is inserted at the physical <i>position&</i> within the parent menu, where <i>position&</i> = 1 for the first position, <i>position&</i> = 2 for the second, and so on. If position is not specified then the popup menu is appended to the end of the menu.										

<i>callback</i>	Optional name of a Callback Function that will be called when the menu item is selected.
Restrictions	The application must call the MENU DRAW BAR statement whenever a menu changes, whether or not the menu is in a displayed dialog.
See also	Dynamic Dialog Tools , ACCEL ATTACH , Menus , MENU ADD POPUP , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT
Example	See Menu Example .

MENU ATTACH statement

[Top](#) [Previous](#) [Next](#)

Purpose	Attach a menu to a given dialog .
Syntax	<code>MENU ATTACH <i>hMenu</i>, <i>hDlg</i></code>
Remarks	Attach a menu to a dialog, replacing any existing menu that is attached to the dialog. The dialog is redrawn to accommodate the new menu.
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the menu to attach to the dialog.
<i>hDlg</i>	Double-word or Long-integer variable containing the handle of the target dialog.
See also	Dynamic Dialog Tools , ACCEL ATTACH , Menus , MENU ADD POPUP , MENU ADD STRING , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT
Example	See Menu Example .

MENU DELETE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Delete a menu item from an existing menu.
Syntax	<code>MENU DELETE <i>hMenu</i>, [BYCMD] <i>position</i>&</code>
Remarks	If the menu item is a popup child menu, the menu is destroyed and its memory is released.
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the menu containing the item you are deleting.
<i>position</i> &	Position of the item within the menu. If BYCMD is specified, <i>position</i> & refers to the unique menu identifier. Otherwise, <i>position</i> & is the position of the menu item, where <i>position</i> & = 1 for the first position, <i>position</i> & = 2 for the second, and so on.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT

Purpose	Redraw the menu bar for a given dialog .
Syntax	<code>MENU DRAW BAR <i>hDlg</i></code>
Remarks	This operation should be performed when a menu is altered dynamically after the dialog has been initially created, without regard to the visible state of the dialog.
<i>hDlg</i>	Handle of the dialog that owns the menu to be redrawn.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT

MENU GET STATE statement

[Top](#) [Previous](#) [Next](#)

Purpose Return the state of a specified [menu](#) item.

Syntax `MENU GET STATE hMenu, [BYCMD] position& TO state&`

Remarks Retrieves the menu flags associated with the specified menu item. If the menu item activates a pop-up menu, this function returns the number of items in that pop-up menu.

hMenu [Double-word](#) or [Long-integer](#) variable containing the handle of the menu containing the item to examine.

position& Position within the menu of the menu item to examine. If the BYCMD option is specified, *position&* specifies the unique menu item identifier of the item to examine. Otherwise, *position&* indicates the physical position of the menu item within the menu, where *position&* = 1 for the first position, *position&* = 2 for the second position, and so on.

state& Long-integer variable where the menu state will be placed. If the specified menu item does not exist, the result is -1. If the menu item is a popup child menu, the [LO](#)(BYTE, *state&*) of the return value contains the menu flags for the item, and the [HI](#)(BYTE, *state&*) contains the number of items in the popup child menu. Otherwise the result is a bitmask containing one or more of the following, combined together with the [OR](#) operator to form the bitmask:

<code>%MF_CHECKED</code>	Menu item has a checkmark next to it.
<code>%MF_DISABLED</code>	Menu item is disabled and cannot be selected.
<code>%MF_ENABLED</code>	Menu item is enabled and can be selected.
<code>%MF_GRAYED</code>	Menu item is disabled and cannot be selected, and is drawn in a "grayed" state.
<code>%MF_HILITE</code>	Menu item is highlighted.
<code>%MF_SEPARATOR</code>	Menu item is a separator (horizontal line).
<code>%MF_UNCHECKED</code>	Menu item does not have a checkmark next to it.

See also [Dynamic Dialog Tools](#), [Menus](#), [MENU ADD POPUP](#), [MENU ADD STRING](#), [MENU ATTACH](#), [MENU DELETE](#), [MENU DRAW BAR](#), [MENU GET TEXT](#), [MENU NEW BAR](#), [MENU NEW POPUP](#), [MENU SET STATE](#), [MENU SET TEXT](#)

Purpose	Return the text associated with a given menu item.
Syntax	<code>MENU GET TEXT <i>hMenu</i>, [BYCMD] <i>position&</i> TO <i>txt\$</i></code>
Remarks	Return the text displayed in the menu item identified by <i>position&</i> .
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the menu that contains the menu item to be examined.
<i>position&</i>	Position within the menu, of the menu item to examine. If BYCMD is specified, <i>position&</i> refers to the unique menu item identifier of the item to examine. Otherwise, <i>position&</i> indicates the physical position of the menu item within the menu, where <i>position&</i> = 1 for the first position, <i>position&</i> = 2 for the second position, and so on.
<i>txt\$</i>	String variable where the text from the menu item will be placed.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT

Purpose	Create a new menu bar.
Syntax	<code>MENU NEW BAR TO <i>hMenu</i></code>
Remarks	Items may be added to the menu using the MENU ADD POPUP and MENU ADD STRING statements.
<i>hMenu</i>	Double-word or Long-integer variable where the handle of the new menu will be placed.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW POPUP , MENU SET STATE , MENU SET TEXT
Example	See Menu Example .

Purpose	Create a new popup menu .
Syntax	<code>MENU NEW POPUP TO <i>hPopup</i></code>
Remarks	Once created, items may be added to the popup menu using the MENU ADD POPUP and MENU ADD STRING statements.
<i>hPopup</i>	Double-word or Long-integer variable where the handle of the new popup menu will be placed.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU SET STATE , MENU SET TEXT
Example	See Menu Example .

Purpose	Set the state of a specified menu item.												
Syntax	<code>MENU SET STATE <i>hMenu</i>, [BYCMD] <i>position&</i>, <i>state&</i></code>												
Remarks	Change the state of the menu item identified by <i>position&</i> .												
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the menu that contains the item to change.												
<i>position&</i>	Position within the menu, of the menu item to be changed. If the BYCMD option is specified, <i>position&</i> refers to the unique menu item identifier of the item. Otherwise, <i>position&</i> indicates the physical position of the menu item within the menu, where <i>position&</i> = 1 for the first position, <i>position&</i> = 2 for the second position, and so on.												
<i>state&</i>	<p>The new state of the menu item. This must be one or more of the following items, combined together with the OR operator to form a bitmask:</p> <table><tr><td>%MF_CHECKED</td><td>Place a checkmark next to the item.</td></tr><tr><td>%MF_DISABLED</td><td>Disable the menu item so that it cannot be selected.</td></tr><tr><td>%MF_ENABLED</td><td>Enable the menu item so that it can be selected.</td></tr><tr><td>%MF_GRAYED</td><td>Disable the menu item so that it cannot be selected and draw it in a "grayed" state to indicate this.</td></tr><tr><td>%MF_HILITE</td><td>Highlight the menu item.</td></tr><tr><td>%MF_UNCHECKED</td><td>Remove any checkmark next to the item.</td></tr></table>	%MF_CHECKED	Place a checkmark next to the item.	%MF_DISABLED	Disable the menu item so that it cannot be selected.	%MF_ENABLED	Enable the menu item so that it can be selected.	%MF_GRAYED	Disable the menu item so that it cannot be selected and draw it in a "grayed" state to indicate this.	%MF_HILITE	Highlight the menu item.	%MF_UNCHECKED	Remove any checkmark next to the item.
%MF_CHECKED	Place a checkmark next to the item.												
%MF_DISABLED	Disable the menu item so that it cannot be selected.												
%MF_ENABLED	Enable the menu item so that it can be selected.												
%MF_GRAYED	Disable the menu item so that it cannot be selected and draw it in a "grayed" state to indicate this.												
%MF_HILITE	Highlight the menu item.												
%MF_UNCHECKED	Remove any checkmark next to the item.												
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET TEXT												

Purpose	Set the text of a given menu item.
Syntax	<code>MENU SET TEXT <i>hMenu</i>, [BYCMD] <i>position</i>&, <i>txt</i>\$</code>
Remarks	Set the text of the menu item identified by <i>position</i> &.
<i>hMenu</i>	Double-word or Long-integer variable containing the handle of the menu that contains the menu item to change.
<i>position</i> &	Position within the menu, of the menu item to be changed. If the BYCMD option is used, <i>position</i> & specifies the unique menu item identifier of the item to change. Otherwise, <i>position</i> & indicates the physical position of the menu item within the menu, where <i>position</i> & = 1 for the first position, <i>position</i> & = 2 for the second position, and so on.
<i>txt</i> \$	The new text for the menu item.
See also	Dynamic Dialog Tools , Menus , MENU ADD POPUP , MENU ADD STRING , MENU ATTACH , MENU DELETE , MENU DRAW BAR , MENU GET STATE , MENU GET TEXT , MENU NEW BAR , MENU NEW POPUP , MENU SET STATE

Purpose	Define a METHOD procedure within a class .
Syntax	<pre>[CLASS OVERRIDE] METHOD <i>name</i> [<DispID>] [ALIAS "<i>altname</i>"] (var AS <i>type</i>...) [AS <i>type</i>] [<i>statements</i>] METHOD = <i>expression</i> END METHOD</pre>
Remarks	METHOD/END METHOD is used to define a METHOD procedure within a class. Standard methods can only be called through a virtual function table on a valid object .

```
[CLASS] METHOD name [ALIAS "altname"] (var AS type...) [AS type]  
    [statements]  
    METHOD = expression  
END METHOD
```

A METHOD is a block of code, very similar to a user-defined function. Optionally, it can return a value, like a [FUNCTION](#), or merely act as a subroutine, like a [SUB](#). If the optional "AS type" is included, the method returns a value set by "*Method=expr*", or defaults to a return value of zero (0) or nul string, depending upon the type. METHOD parameters may be any [variable](#) type, including [VARIANT](#) variables. Methods may be called using any of the five following forms:

```
DIM ObjVar AS MyInterface  
LET ObjVar = NEWCOM Progid$  
1. ObjVar.Method1(param)  
2. CALL ObjVar.Method1(param)  
3. ObjVar.Method1(param) TO var  
4. CALL ObjVar.Method1(param) TO var  
5. var = ObjVar.Method1(param)
```

Forms 1 and 2 assume that the Method does not return a value, or you simply wish to discard it. Forms 3, 4, and 5 require that the Method return a value compatible with the type of variable specified as *var*. Parentheses enclosing parameters are optional in forms 1 and 3.

Methods may be declared (using AS *type*...) to return a string, any of the numeric types, a specific class of object variable (AS MyClass), a Variant, or a [user defined Type](#).

Type Libraries only support the following data types: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). If any Methods or Properties use data types not supported by Type Libraries, you will receive a [Error 581 - Type Library creation error](#), when using the [#COM TLIB ON](#) metastatement.

In addition to the explicit return value which you declare, all Methods and [Properties](#) on an [IAutomation](#) or [IDispatch](#) interface have another "Hidden Return Value", which is cryptically named `hResult`. While the name would imply a handle for a result, it's really not a handle at all, but just a [long integer](#) value, used to indicate success or failure of the Method. After calling a Method or Property, you can retrieve the `hResult` value with the Classic PowerBASIC function [OBJRESULT](#). The most significant bit of the value is known as the severity bit. That bit is 0 (value is positive) for success, or 1 (value is negative) for failure. The remaining bits are used to convey error codes and additional status information. If you call any object Method/Property (either [Dispatch](#) or [Direct](#)), and the severity bit of `hResult` is set, Classic PowerBASIC generates [Run-Time error 99](#): Object error. When you create a Method or Property, Classic PowerBASIC automatically returns an `hResult` of zero, which implies success. You can return a non-zero `hResult` value by executing a `METHOD OBJRESULT = expr` within a Method, or `PROPERTY OBJRESULT = expr` within a Property.

Class Methods

A CLASS METHOD is one which is private to the class in which it is located. That is, it may only be called from a METHOD or PROPERTY in the same class. The CLASS METHOD must be located within a CLASS block, but outside of any [INTERFACE](#) blocks. This shows it is a direct member of the class, rather than a member of an [interface](#).

```
CLASS MyClass
    INSTANCE MyVar AS LONG

    CLASS METHOD MyClassMethod(BYVAL param AS LONG) AS STRING
        METHOD = "My" + STR$(param + MyVar)
    END METHOD

    INTERFACE MyInterface
        INHERIT IUNKNOWN
        METHOD MyMethod()
            Result$ = ME.MyClassMethod(66)
        END METHOD
    END INTERFACE
END CLASS
```

In the above example, `MyClassMethod()` is a CLASS METHOD, and is always accessed using the pseudo-object [ME](#) (in this case `ME.MyClassMethod`). Class methods are never accessible from outside a class, nor are they ever described or published in a type library. By definition, there is no reason to have a private PROPERTY, so Classic PowerBASIC does not offer a CLASS PROPERTY structure.

Constructors and Destructors

There are two special class methods which you may optionally add to a

class. They meet a very specific need: automatic initialization when an object is created, and cleanup when an object is destroyed. Technically, they are known as [constructor and destructor](#) methods, and can perform almost any functionality needed by your object: initialization of variables, reading/writing data to/from disk, etc. You do not call these methods directly from your code. If they are present in your class, Classic PowerBASIC automatically calls them each time an object of that class is created or destroyed. If you choose to use them, these special class methods must be named CREATE and DESTROY. They may take no parameters, and may not return a result. They are defined at the class level, so they may never appear within an INTERFACE definition.

```
CLASS MyClass
  INSTANCE MyVar AS LONG

  CLASS METHOD CREATE()
    ' Do initialization
  END METHOD

  CLASS METHOD Destroy()
    ' Do cleanup
  END METHOD

  INTERFACE MyInterface
    INHERIT IUNKNOWN
    METHOD MyMethod()
      ' Do things
    END METHOD
  END INTERFACE
END CLASS
```

As displayed above, CREATE and DESTROY must be placed at the class level, but outside of any interface block. You should note that it's not possible to name any standard method (one that's accessible through an interface) as CREATE or DESTROY. That's just to help you remember the rules for a constructor or destructor. However, you may use these names as needed to describe a method external to your program.

A very important caution: You must never create an object of the current class in a CREATE method. To do so will cause CREATE to be executed again and again until all available memory is consumed. This is a fatal error, from which recovery is impossible.

Override Methods

You can add to, or replace, the functionality of a particular method or property of an inherited [base class](#) by coding a replacement which is preceded by the word OVERRIDE. The overriding method must have the same name and signature (parameters, return value, etc.) as the one it replaces.

Dispatch ID

Every method and property in a [dual interface](#) needs a positive, long integer value to identify it. That integer value is known as a Dispid (Dispatch ID), and it's used internally by [COM](#) services to call the correct function on a [Dispatch](#) interface. You can optionally specify a particular Dispid by enclosing it in angle brackets immediately following the Method/Property name:

```
METHOD MethodOne <76> ()
```

If you don't specify a Dispid, Classic PowerBASIC will assign a random value for you. This is fine for internal objects, but may cause a failure for [published](#) COM objects, as the Dispid could change each time you compile your program. It is particularly important that you specify a Dispid for each Method/Property in a COM [Event Interface](#).

BYREF and BYVAL attributes

Just like a SUB or FUNCTION, Classic PowerBASIC uses BYREF parameters as the default form, unless you specify a BYVAL override. Either key word can be placed before the parameter name, along with IN, OUT, and INOUT, as described later.

BYVAL A copy of the data value is placed on the [stack](#) as a parameter. The copy is destroyed when the METHOD ends. BYVAL parameters default to an IN parameter, if no explicit direction is specified.

BYREF A [pointer](#) to the data is placed on the stack as a parameter. If the data is a variable, any changes to the parameter are passed back to the caller in the variable. If the data is an expression, it is destroyed when the METHOD ends. BYREF parameters default to an INOUT parameter, if no explicit direction is specified.

Direction attributes

METHOD parameters may also specify the direction in which data is passed between the caller and callee:

IN Data is passed from the caller to the METHOD. Generally speaking, you'll find that almost all IN parameters are passed BYVAL, and that is highly recommended. However, it is possible to pass them BYREF if necessary.

OUT Data is passed from the METHOD back to the caller. All OUT parameters must be passed BYREF.

INOUT Data is passed from the caller to the METHOD, and results are returned to the caller in the same parameter. All INOUT parameters must be passed BYREF.

In many cases, the direction of a parameter can be inferred directly from the BYVAL/BYREF attribute (BYVAL=IN, BYREF=OUT). However, we recommend that you include the direction attribute as an added means of self-documentation. Each METHOD parameter name may be preceded by one of BYVAL/BYREF, and one of IN/OUT/INOUT, in any sequence.

You should note an interesting rule of COM objects: **IN parameters are read-only. They may not be altered.**

IN parameters are considered by COM rules to be "constant" which may not be altered, because they are values which are not returned to the caller. However, since this is not a rule normally applied to a standard SUB or FUNCTION, it can allow programming bugs which are most difficult to find and correct. For this reason, Classic PowerBASIC automatically protects you from this issue with no action needed on your part. When writing METHOD or PROPERTY code in Classic PowerBASIC, you may freely assign new values to BYVAL/IN parameters. They will simply be discarded when the METHOD exits. Of course, not every programming language protects you in this way, so you must use caution if you create a COM METHOD in another compiler.

Using OPTIONAL/OPT

METHOD statements may specify one or more parameters as optional by preceding the parameter with either the keyword OPTIONAL (or the abbreviation OPT). When a parameter is declared optional, all subsequent parameters in the declaration are optional as well, whether or not they specify an explicit OPTIONAL or OPT directive.

VARIANT variables are particularly well suited for use as an optional parameter. If the calling code omits an optional VARIANT parameter, (BYVAL or BYREF), Classic PowerBASIC (and most other compilers) substitute a variant of type %VT_ERROR which contains an error value of %DISP_E_PARAMNOTFOUND (&H80020004). In this case, you can check for this value directly, or use the [ISMISSING](#) function to determine whether the parameter was physically passed or not.

When optional parameters (other than VARIANT) are omitted from the calling code, the stack area normally reserved for those parameters is zero-filled.

If the parameter is defined as a BYVAL parameter, it will have the value zero. For [TYPE](#) or [UNION](#) variables passed BYVAL, the compiler will pass a string of binary zeroes of length [SIZEOF](#)(*Type_or_union_var*).

If the parameter is defined as a BYREF parameter, [VARPTR](#)(*Varname*) will equal zero; when this is true, any attempt to use *Varname* in your code will result in a General Protection Fault or memory corruption. You should

use the ISMISSING() function first to determine whether it is safe to access the parameter.

See also

[#COM](#), [CLASS](#), [INSTANCE](#), [INTERFACE \(Direct\)](#), [ISMISSING](#), [Just what is COM?](#), [ME](#), [PROPERTY](#), [What is an object, anyway?](#)

Purpose Return a portion of a string.

Syntax `s$ = MID$(string_expression, start& [, length&])`

Remarks *start&* and *length&* are numeric variables or expressions. As a function, MID\$ returns a sub-string of [string_expression](#) that is *length&* characters long and starts at the *start&* character of *string_expression*. For example:

```
a$ = MID$("PowerBASIC", 1, 2)
```

returns "Po". If *length&* is omitted, or there are fewer than *length&* characters to the right of the *start&* character of *string_expression*, all remaining characters of *string_expression*, including the *start&* character, are returned. If *start&* is greater than the length of *string_expression*, MID\$ returns an empty string. If *start&* is negative, the starting position is assumed to be *start&* characters from the end of the string. If *length&* is negative, it is interpreted as [LEN](#)(*string_expression*)-[ABS](#)(*length&*).

Restrictions If *start&* evaluates to a position outside of the string on either side, or if *start&* is zero, an empty string is returned.

See also [EXTRACT\\$](#), [INSTR](#), [LEFT\\$](#), [LTRIM\\$](#), [MID\\$ statement](#), [RIGHT\\$](#), [RTRIM\\$](#), [TALLY](#), [TRIM\\$](#), [VERIFY](#)

Example

```
a$ = MID$("PowerBASIC", 4, 2) ' returns "er"
a$ = MID$("PowerBASIC", 4)   ' returns "erBASIC"
a$ = MID$("PowerBASIC", 4, 20) ' returns "erBASIC"
a$ = MID$("PowerBASIC", 20)  ' returns a null string
a$ = MID$("1234567890", 3, -4) ' returns "345678"
a$ = MID$("abcde", -3, 2)    ' returns "cd"
a$ = MID$("abcde", -3)       ' returns "cde"
```

MID\$ statement

[Top](#) [Previous](#) [Next](#)

Purpose	Replace characters in a string with characters from another string.
Syntax	<code>MID\$(string_var, start& [, length&]) = replacement</code>
Remarks	<p><i>start&</i> and <i>length&</i> are numeric variables or expressions. As a statement, MID\$ replaces <i>length&</i> characters of <i>string_var</i>, beginning at character position <i>start&</i>, with the contents of <i>replacement</i> string.</p> <p>If <i>length&</i> is included, it determines how many characters of <i>replacement</i> string are inserted into <i>string_var</i>. If <i>length&</i> is omitted, all of <i>replacement</i> string is used. If <i>length&</i> is negative, it is interpreted as LEN(<i>string_var</i>)-ABS(<i>length&</i>). For example, MID\$("ABCDEFGHIIJK",3,-7) = "*****" yields "AB****GHIJK".</p> <p>If <i>start&</i> is negative, the starting position is assumed to be <i>start&</i> characters from the end of the string. MID\$("abcde", -3, 2) = "123" yields "ab12e", while the statement MID\$("abcde", -3) = "123" yields "ab123".</p> <p>The replacement will never extend past the end of the original <i>string_var</i>; that is, MID\$ never alters the length of a string. For a similar function that can alter the length of a string, please refer to the REPLACE statement.</p>
Restrictions	<p>If <i>start&</i> evaluates to a position outside of the string on either side, or if <i>start&</i> is zero, no operation is performed.</p> <p>The MID\$ statement cannot be used to extend the length of a string. If <i>start</i> is beyond the end of a string, the statement is ignored.</p>
See also	BUILD\$, INSTR , LTRIM\$, MID\$ function , REMOVE\$, REPLACE , RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>DummyString\$ = "1234567890" FOR M = 1 TO 10 TestString\$ = DummyString\$ MID\$(TestString\$,1,M) = "PowerBASIC" NEXT M</pre>
Result	<pre>P234567890 Po34567890 Pow4567890 Powe567890 ... PowerBAS90 PowerBASIO PowerBASIC</pre>

MIN function

[Top](#) [Previous](#) [Next](#)

Purpose Return the argument with the smallest (minimum) value.

Syntax

```
y = MIN(arg, arg [, arg] ...)  
y& = MIN&(arg&, arg& [, arg&] ...)  
y$ = MIN$(arg$, arg$ [, arg$] ...)
```

Remarks These functions take any number of arguments and return the argument with the smallest (minimum) value. MIN handles arguments of any numeric type.

MIN& handles arguments that evaluate to [Integers](#) and [Long-integers](#) (MIN& is more efficient than MIN). MIN\$ handles string arguments.

If any arguments of MIN& are outside of the range of Long-integers, the result is undefined. Any floating point arguments of MIN& will be rounded to Long-integers before the comparison begins.

MIN% is recognized as a valid synonym for MIN&.

See also [CHOOSE](#), [CHOOSE&](#), [CHOOSE\\$](#), [IIF](#), [IIF&](#), [IIF\\$](#), [MAX](#), [MAX&](#), [MAX\\$](#), [SWITCH](#), [SWITCH&](#), [SWITCH\\$](#)

Example

```
x& = MIN&(A, B, C, D)  
x$ = MIN$("abacadabra", "cad", A$, B$(4), C$ + D$ + LEFT$(E$, 5))  
x## = MIN(1.1@@, A%/B!, C#(x)^D, E##, SIN(F&))
```

MKBYT\$, MKCUR\$, MKCUX\$, MKD\$, MKDWD\$, MKE\$, MKI\$, MKL\$, MKQ\$, MKS\$ and MKWRD\$ functions

Purpose Convert numeric data into strings, ensuring that storage for a given numeric data type is consistent, regardless of its [absolute value](#).

Syntax

```

DataTypeString$ = MKBYT$(byte_expr)
DataTypeString$ = MKCUR$(currency_expr)
DataTypeString$ = MKCUX$(extended_currency_expr)
DataTypeString$ = MKD$(double_precision_expr)
DataTypeString$ = MKDWD$(double_word_expr)
DataTypeString$ = MKE$(extended_precision_expr)
DataTypeString$ = MKI$(integer_expr)
DataTypeString$ = MKL$(long_integer_expr)
DataTypeString$ = MKQ$(quad_integer_expr)
DataTypeString$ = MKS$(single_precision_expr)
DataTypeString$ = MKWRD$(word_expr)

```

Remarks The MKx functions return the binary representations of a number as a string value. Do not confuse these functions with the [STR\\$](#) and [FORMAT\\$](#) functions (which return a printable ASCII representation of a numeric expression as a string (e.g., "-42.75")).

The [CVx](#) functions are complementary to the MKx functions. They convert the binary representation in a string to an actual numeric value:

Function	Converts to	From
MKBYT\$	1-byte string	Byte
MKCUR\$	8-byte string	Currency
MKCUX\$	8-byte string	Extended-currency
MKD\$	8-byte string	Double-precision
MKDWD\$	4-byte string	Double-word
MKE\$	10-byte string	Extended-precision
MKI\$	2-byte string	Integer
MKL\$	4-byte string	Long-integer
MKQ\$	8-byte string	Quad-integer
MKS\$	4-byte string	Single-precision
MKWRD\$	2-byte string	Word

See also

[CVI and associated functions](#)

Purpose Create a subdirectory/folder (like the DOS MKDIR command).

Syntax `MKDIR path$`

Remarks *path*\$ is a [string expression](#) describing the directory to be created. MKDIR (make directory) creates the subdirectory specified by *path*\$. If you try to create a directory that already exists, a run-time [Error 75](#) occurs ("Path/file access error"). If *path*\$ includes an parent folder that does not exist, a run-time [Error 76](#) occurs ("Path not found").

MKDIR can use Long File Names (LFNs).

See also [CHDIR](#), [RMDIR](#)

Example `MKDIR "C:\Program Files\Company\Application Data"`

Purpose Return the remainder of the division between two numbers.

Syntax $p \text{ MOD } q$

Remarks The MOD operator divides the two operands, p and q , and returns the remainder of that division. The result of the initial division is truncated to an integer class value, before the remainder is calculated. See the example below.

The remainder may be a floating point value. MOD is often considered to complement integer division.

See Also [LET](#)

Example

```
lResult1& = 10& MOD 3&      ' Returns 1&
fResult2! = 13! MOD 2.7!    ' Returns 2.2!

iStack&   = 1023&
HiStack& = iStack& \ 256&   ' Returns 3&
LoStack& = iStack& MOD 256 ' Returns 255&

' c! and d! are calculated equivalently
a! = 13
b! = 2.7
c! = a! MOD b!
d! = a! - FIX(a! / b!) * b!

CurrentLine = 1
WHILE CurrentLine < Lines
  PrintLine txt$(CurrentLine)
  IF (CurrentLine MOD 55) = 0 THEN DoFormFeed
  INCR CurrentLine
WEND
```


Purpose Change the mouse pointer (cursor) to a new shape.

Syntax `MOUSEPTR style [TO var&]`

Remarks If the optional TO clause is included, the handle of the previous cursor is assigned to *var&*. If the operation fails, the value zero is assigned to *var&*. Normally, the [Long integer](#) or [DWORD](#) value style should be in the range of 1 through 13 to choose one of the stock cursor shapes as follows:

***style&* Definition**

- 0 Hide mouse pointer **
- 1 Arrow
- 2 Cross
- 3 I-Beam
- 4 Arrow
- 5 Sizing pointer (all directions)
- 6 Sizing pointer (NE-SW diagonal)
- 7 Sizing pointer (vertical)
- 8 Sizing pointer (NW-SE diagonal)
- 9 Sizing pointer (horizontal)
- 10 Up arrow
- 11 Hourglass ("Busy" or "Wait" pointer)
- 12 No mouse pointer
- 13 App Starting (arrow with an hourglass)

** If *style&* = 0 then the OS may restore the cursor if it is moved.

If style is outside the range 0 through 13, it must contain a valid handle to a cursor, such as the value which was returned by a prior invocation of MOUSEPTR. This allows the programmer to restore a previous cursor style.

The mouse pointer is only changed for dialogs and windows in your application. If the mouse pointer is moved over another application or the desktop, the pointer will change to the default for that application/process. In GUI applications, MOUSEPTR can be useful in %WM_SETCURSOR message handler routines that override the default cursor handling.

Purpose	Display a message box containing a text string and an optional title, using one or more styles, and returning the button selected by the user.
Syntax	<code>lResult& = MSGBOX(txt\$ [, [style&], title\$])</code>
Remarks	The MSGBOX function is comprised of the following elements:
txt\$	Indicates the text to display within the message box.
style&	Optional parameter which determines the appearance of the message box. Some styles may be combined (OR'ed together) to specify the button and icon displayed in the message box. If style& is omitted, Classic PowerBASIC substitutes %MB_OK. The following styles are defined in WIN32API.INC , in the form of numeric equates:
%MB_OK	Display OK button (default)
%MB_OKCANCEL	Display OK and Cancel buttons
%MB_ABORTRETRYIGNORE	Display Abort, Retry and Ignore
%MB_YESNOCANCEL	Display Yes, No and Cancel
%MB_YESNO	Display Yes and No buttons
%MB_RETRYCANCEL	Display Retry and Cancel buttons
%MB_ICONERROR	Display Error icon (stop sign)
%MB_ICONINFORMATION	Display Information icon ("i")
%MB_ICONQUESTION	Display Query icon (question mark)
%MB_ICONWARNING	Display Warning icon (exclamation)
%MB_DEFBUTTON1	Default to 1st button (default)
%MB_DEFBUTTON2	Default to 2nd button
%MB_DEFBUTTON3	Default to 3rd button
%MB_APPLMODAL	Application Modal - Despite the name, the user can continue to interact with other dialogs without dismissing the MSGBOX.
%MB_SYSTEMMODAL	System Modal - Operates identically to %MB_APPLMODAL, except the MSGBOX is given the %WS_EX_TOPMOST style so that it remains above all other windows and dialogs.
%MB_TASKMODAL	Task Modal - All top-level windows belonging to the current application are disabled until the MSGBOX is dismissed.

%MB_TASKMODAL is commonly used to display a truly modal MSGBOX. (default)

title\$ Optional title to be displayed in the caption of the message box. If *title\$* is not specified, "PowerBASIC" is used automatically.

!Result& Identifies the Button selected by the user. This will be equal to one of the following equates:

%IDOK	OK button
%IDCANCEL	Cancel button
%IDABORT	Abort button
%IDRETRY	Retry button
%IDIGNORE	Ignore button
%IDYES	Yes button
%IDNO	No button

Additional styles may be found in WIN32API.INC in the section prefixed with %MB_. If you are not interested in which button the user selects, use a [MSGBOX statement](#) rather than a MSGBOX function.

A question mark may be used as an abbreviation for the MSGBOX statement.

Restrictions Strings displayed by the MSGBOX function are displayed only up to the first [\\$NUL](#) character, if any.

See also [INPUTBOX\\$](#), [MSGBOX statement](#)

Example

```
!Result& = MSGBOX("Overwrite registry file?", %MB_OKCANCEL OR
%MB_DEFBUTTON2 _
OR %MB_TASKMODAL, "Critical Warning")
```

Purpose	Display a message box containing a text string and optional title, using one or more styles.														
Syntax	<code>MSGBOX txt\$ [, [style%], title\$]</code> <code>? txt\$ [, [style%], title\$]</code>														
Remarks	The MSGBOX statement comprises the following elements:														
txt\$	Text to display within the message box.														
style&	Optional parameter which determines the appearance of the message box. Some styles may be combined (OR 'ed together) to specify the button and icon displayed in the message box. If <i>style&</i> is omitted, Classic PowerBASIC substitutes %MB_OK. The following are some of the more common styles used with the MSGBOX statement (also see the MSGBOX function for more information): <table><tr><td>%MB_OK</td><td>Display OK button (default)</td></tr><tr><td>%MB_ICONERROR</td><td>Display Error icon (stop sign)</td></tr><tr><td>%MB_ICONINFORMATION</td><td>Display Information icon ("i")</td></tr><tr><td>%MB_ICONWARNING</td><td>Display Warning icon (exclamation)</td></tr><tr><td>%MB_APPLMODAL</td><td>Application Modal - Despite the name, the user can continue to interact with other dialogs without dismissing the MSGBOX.</td></tr><tr><td>%MB_SYSTEMMODAL</td><td>System Modal - Operates identically to %MB_APPLMODAL, except the MSGBOX is given the %WS_EX_TOPMOST style so that it remains above all other windows and dialogs.</td></tr><tr><td>%MB_TASKMODAL</td><td>Task Modal - All top-level windows belonging to the current application are disabled until the MSGBOX is dismissed. %MB_TASKMODAL is commonly used to display a truly modal MSGBOX. (default)</td></tr></table> Additional styles may be found in WIN32API.INC , in the section prefixed with %MB_. If you are interested in which button the user selects, use a MSGBOX function rather than a MSGBOX statement.	%MB_OK	Display OK button (default)	%MB_ICONERROR	Display Error icon (stop sign)	%MB_ICONINFORMATION	Display Information icon ("i")	%MB_ICONWARNING	Display Warning icon (exclamation)	%MB_APPLMODAL	Application Modal - Despite the name, the user can continue to interact with other dialogs without dismissing the MSGBOX.	%MB_SYSTEMMODAL	System Modal - Operates identically to %MB_APPLMODAL, except the MSGBOX is given the %WS_EX_TOPMOST style so that it remains above all other windows and dialogs.	%MB_TASKMODAL	Task Modal - All top-level windows belonging to the current application are disabled until the MSGBOX is dismissed. %MB_TASKMODAL is commonly used to display a truly modal MSGBOX. (default)
%MB_OK	Display OK button (default)														
%MB_ICONERROR	Display Error icon (stop sign)														
%MB_ICONINFORMATION	Display Information icon ("i")														
%MB_ICONWARNING	Display Warning icon (exclamation)														
%MB_APPLMODAL	Application Modal - Despite the name, the user can continue to interact with other dialogs without dismissing the MSGBOX.														
%MB_SYSTEMMODAL	System Modal - Operates identically to %MB_APPLMODAL, except the MSGBOX is given the %WS_EX_TOPMOST style so that it remains above all other windows and dialogs.														
%MB_TASKMODAL	Task Modal - All top-level windows belonging to the current application are disabled until the MSGBOX is dismissed. %MB_TASKMODAL is commonly used to display a truly modal MSGBOX. (default)														
title\$	Determines the title to be displayed in the caption of the message box. If <i>title\$</i> is not specified, "PowerBASIC" is used automatically. The MSGBOX statement may be represented by the query (?) character as a shortcut. This is similar to the behavior in the Classic PowerBASIC														

[Console Compiler](#) (PB/CC), where the query character is recognized as a synonym for the PRINT statement. This can simplify the creation of certain test code, since the query character provides similar functionality in both compilers.

Restrictions Strings displayed by the MSGBOX function are displayed only up to the first [\\$NUL](#) character, if any.

See also [INPUTBOX\\$](#), [MSGBOX function](#)

Example

```
MSGBOX "Got here, hit OK to continue",, "Title of subroutine 123"  
MSGBOX "Current value of X% is: " & STR$(x%)  
MSGBOX "Paused, click OK to continue!"  
MSGBOX "Click OK to reboot the universe", %MB_TASKMODAL OR  
%MB_ICONERROR, "Reality has crashed!"
```

Purpose	A pseudo object variable to reference the inherited parent object.
Syntax	<code>MYBASE.Method1 (param)</code>
Remarks	<p>MYBASE is a pseudo-variable, which Classic PowerBASIC automatically defines in every Method and Property on an inherited interface. It is treated as a reference to the original, inherited object. Using MYBASE, it's possible to call the original Methods and Properties so you can modify or build upon them in the derived interface.</p> <p>MYBASE may not be assigned to an object variable, nor may it be used as a Sub/Function/Method/Property parameter.</p>
See also	CLASS , INTERFACE (Direct) , INTERFACE (IDBind) , ME , METHOD , PROPERTY , What is an object, anyway? , What is inheritance?

Purpose	Rename a file or a directory (like the DOS REN command).
Syntax	<code>NAME <i>filespec1</i>\$ AS <i>filespec2</i>\$</code>
Remarks	<p>The NAME statement comprises the following elements:</p> <p><i>filespec1</i>\$ The current name of a file or directory. The file must not be currently opened or locked. <i>filespec1</i>\$ may be either a Short File Name (SFN) or a Long File Name (LFN).</p> <p><i>filespec2</i>\$ The desired name of the file or directory, and may use Long File Name (LFN) naming conventions.</p> <p>Each filespec may contain drive and path specifications as well as a file or directory name.</p> <p>If <i>filespec1</i> does not exist, run-time Error 53 ("File not found") occurs. If <i>filespec2</i> already exists, run-time Error 58 occurs ("File already exists"). If <i>filespec1</i> has been opened or locked by your application and not closed, an Error 51 can occur ("Internal system error"). You should never rename a file that has been opened by your code and not (yet) closed.</p>
Restrictions	<p>It is possible to move a file from one directory, drive, or partition, to another. It is not possible to move directories between drives or partitions. Wildcard characters are not permitted in the file names.</p>
Example	<pre>OldName\$ = "MYFILE.EXE" NewName\$ = "YOURFILE.EXE" NAME OldName\$ AS NewName\$</pre>

Purpose The NOT operator works as a bitwise [arithmetic operator](#).

Syntax `NOT p`

Remarks Classic PowerBASIC's NOT operator returns the one's-complement of an integer class expression. When dealing with the absolute values 0 and -1, the NOT operator "reverses" the two values, performing a Boolean-like operation.

Classic PowerBASIC accepts any non-zero value as a logical TRUE value; therefore, subtle logic problems can arise in a program when the NOT operator is used to perform Boolean logic tests with operand values that are not limited to just 0 and -1.

Consider the following two test conditions:

```
test1 = 0           ' test1 is FALSE (zero)
IF NOT test1 THEN   ' TRUE (-1 is non-zero)

test2 = 1           ' test2 is TRUE (1 is non-zero)
IF NOT test2 THEN   ' still TRUE (-2 is non-zero)
```

Because NOT performs a bitwise operation on *test2*, it does not reverse the logical TRUE/FALSE value of *test2*, rather, it returns -2 (the one's-complement of 1) and this is evaluated as a logical TRUE value.

In cases where a proper logical (Boolean) evaluation is required, and the operand may be a value other than 0 and -1, the [ISFALSE](#) operator should be used in place of the NOT operator:

```
test3 = 1           ' test3 is TRUE (non-zero)
IF ISFALSE test3 THEN ' ISFALSE detects test3 is
[statements]         ' TRUE so the IF test fails
```

The two's-complement of a value can be obtained with the following algorithm:

```
y = (NOT x) + 1
Using NOT as a logical operator
```

NOT returns 0 (FALSE) if *and only if* its operand is *exactly* -1 (TRUE).

Generally, you should use the ISFALSE operator instead of NOT, when you are testing for logical falsity.

Truth table	
x	NOT x
0	-1
-1	0

[circuit evaluation](#), [XOR](#)

NUL\$ function

Purpose	Return a string consisting of a specified number of \$NUL (CHR\$(0)) characters.
Syntax	<i>sResult\$</i> = NUL\$(<i>count</i>)
Remarks	NUL\$ returns a <i>count</i> character string of \$NUL characters.
See also	CHR\$, REPEAT\$, SPACE\$, STRING\$

Purpose	Return TRUE/FALSE as an indication of the running state of an initialized COM object (EXE based).
Syntax	<code>lResult& = OBJECTIVE(<i>progid</i>)</code>
Remarks	OBJECTIVE can provide information that may prove useful in determining whether to use the NEWCOM or GETCOM options with the LET (with Objects) statement. OBJECTIVE may only be used on COM objects that are in EXE format, not DLL/OCX/etc. In the latter case, OBJECTIVE returns FALSE (0).
<i>progid</i>	The registered program ID string for the COM object. For example, "Word.Application.8", or a version-independent program ID such as "Word.Application". A valid program ID string can be obtained from a 16-byte class ID string using the PROGID\$ function, or derived from a 38 character GUID string using PROGID\$ and GUID\$.
See also	DIM , CLSID\$, GUID\$, GUIDTXT\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISNOTHING , ISOBJECT , Just what is COM? , LET (with Objects) , OBJECT , OBJPTR , OBJRESULT , PROGID\$, What is a COM component?
Example	<pre>' Create a reference to the MSWORD object LOCAL oWord AS IDISPATCH ' use late-binding LOCAL i AS LONG IF OBJECTIVE("Word.Application") THEN ' Word is already active, use the existing instance oWord = GETCOM "Word.Application" ELSE ' Word is not active, create a new instance oWord = NEWCOM "Word.Application" END IF ' Set MS Word to a normal visible state i = 0 OBJECT LET oWord.WindowState = i ' more code here</pre>

Purpose	Communicate with a COM object through the dispatch interface.										
Syntax	<pre>OBJECT GET <i>interface.member</i>[<i>.member</i>.] [([<i>paramname</i> =] <i>param1</i> [, ...])] TO <i>ResultVar</i> OBJECT LET <i>interface.member</i>[<i>.member</i>.] [([<i>paramname</i> =] <i>param1</i> [, ...])] = <i>ValueVar</i> OBJECT SET <i>interface.member</i>[<i>.member</i>.] [([<i>paramname</i> =] <i>param1</i> [, ...])] = <i>ValueVar</i> OBJECT CALL <i>interface.member</i>[<i>.member</i>.] [([<i>paramname</i> =] <i>param1</i> [, ...])] [TO <i>ResultVar</i>] OBJECT RAISEEVENT [<i>interface.</i>]<i>member</i>[([<i>paramname</i> =] <i>param1</i> [, ...])]</pre>										
Remarks	<p>There are five general forms of the OBJECT statement which are used to communicate through a Dispatch interface to an object.</p> <table><tr><td>OBJECT GET</td><td>Retrieve or read the value of an Interface member Property. This is similar to retrieving the value of a variable.</td></tr><tr><td>OBJECT LET</td><td>Assign or write a value to an Interface member Property. This is similar to assigning a value to a variable.</td></tr><tr><td>OBJECT SET</td><td>Assign or write a value to an Interface member Property that contains a reference to an object. For example, a reference to another Interface.</td></tr><tr><td>OBJECT CALL</td><td>Call or execute a member Method of an Interface. This is equivalent to calling a Sub or Function.</td></tr><tr><td>OBJECT RAISEEVENT</td><td>Call or execute a member Method of a Dispatch event Interface. Because the Dispatch event interface is pre-defined, you are not required to specify the interface name in this form of the statement. However, including it aids in self-documentation of your program. If your program is using a Direct, V-Table event handler you should use the RAISEEVENT statement instead. See the EVENT SOURCE statement for an OBJECT RAISEEVENT example.</td></tr></table>	OBJECT GET	Retrieve or read the value of an Interface member Property . This is similar to retrieving the value of a variable .	OBJECT LET	Assign or write a value to an Interface member Property. This is similar to assigning a value to a variable.	OBJECT SET	Assign or write a value to an Interface member Property that contains a reference to an object. For example, a reference to another Interface.	OBJECT CALL	Call or execute a member Method of an Interface. This is equivalent to calling a Sub or Function .	OBJECT RAISEEVENT	Call or execute a member Method of a Dispatch event Interface . Because the Dispatch event interface is pre-defined, you are not required to specify the interface name in this form of the statement. However, including it aids in self-documentation of your program. If your program is using a Direct, V-Table event handler you should use the RAISEEVENT statement instead. See the EVENT SOURCE statement for an OBJECT RAISEEVENT example.
OBJECT GET	Retrieve or read the value of an Interface member Property . This is similar to retrieving the value of a variable .										
OBJECT LET	Assign or write a value to an Interface member Property. This is similar to assigning a value to a variable.										
OBJECT SET	Assign or write a value to an Interface member Property that contains a reference to an object. For example, a reference to another Interface.										
OBJECT CALL	Call or execute a member Method of an Interface. This is equivalent to calling a Sub or Function .										
OBJECT RAISEEVENT	Call or execute a member Method of a Dispatch event Interface . Because the Dispatch event interface is pre-defined, you are not required to specify the interface name in this form of the statement. However, including it aids in self-documentation of your program. If your program is using a Direct, V-Table event handler you should use the RAISEEVENT statement instead. See the EVENT SOURCE statement for an OBJECT RAISEEVENT example.										

All parameters, return values, and assignment values must be in the form of [COM](#)-compatible variables. [Literals](#) and expressions are not allowed. COM-compatible variables include [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), string, and [VARIANT](#). You should use caution passing string data since COM Objects require that [unicode](#) format be used. When string data is contained in a VARIANT variable, conversion to/from unicode is automatic, and no intervention is needed from the programmer. However, if you pass data in a [dynamic](#)

[string](#) variable, you must use the [ACODE\\$\(\)](#) and [UCODE\\$\(\)](#) functions to convert the data to an appropriate format. For this reason, we recommend that string data be passed using VARIANT variables.

Dispatch OBJECT Method calls may be bound at run-time using [late binding](#), which requires no declaration of Properties and [Methods](#).

However, for this very reason, the validity of these references can not be verified by Classic PowerBASIC at the time the program is compiled.

The OBJECT statement can use both positional and named parameters, but you should keep in mind that not all COM Dispatch Servers support named parameters. Positional parameters are universally supported.

A positional parameter is simply a variable containing an appropriate value. It is identified by its position in the parameter list, just as in a traditional SUB or FUNCTION. A named parameter consists of a parameter identifier (a name), an equal (=) sign, and a variable containing an appropriate value. Positional parameters must precede any and all named parameters, but named parameters may be specified in any sequence.

Each time you call a Method or Property using the OBJECT statement, a status code is returned in a hidden parameter to indicate the success or failure of the operation. You can retrieve information about this status code with the [OBJRESULT](#) function, and also by using the [IDISPINFO](#) Dispatch Information Object. If the failure was severe, then a Classic PowerBASIC [error 99](#) (Object Error) is also generated and the [ERR](#) system variable is set. You can find more information about these items by referring to OBJRESULT, IDISPINFO, and ERR. This information can be very useful for both debugging and handling [run-time errors](#).

Restrictions All parameters, return values, and assignment values must be in the form of COM-compatible variables. Use of the wrong member mode (GET/LET/SET/CALL/RAISEEVENT) can sometimes result in unexpected and fatal run-time errors. So, it's usually prudent to test the result code in OBJRESULT after every OBJECT statement.

See also [ACODE\\$](#), [DIM](#), [CLASS](#), [CLSID\\$](#), [EVENT SOURCE](#), [IDISPINFO](#), [GUID\\$](#), [GUIDTXT\\$](#), [ID Binding](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [ISINTERFACE](#), [ISNOTHING](#), [ISOBJECT](#), [Just what is COM?](#), [Late Binding](#), [LET \(with Objects\)](#), [METHOD](#), [OBJACTIVE](#), [OBJPTR](#), [OBJRESULT](#), [PROGID\\$](#), [PROPERTY](#), [UCODE\\$](#), [What is an object, anyway?](#), [What is DISPATCH?](#)

Example

```
' Assumes Interface definitions have been
' declared for the Microsoft Agent Control
LOCAL AgentCtrlEx AS IAgentCtrlEx
LOCAL StartX      AS LONG
LOCAL StartY      AS LONG
LOCAL CharW       AS LONG
LOCAL CharH       AS LONG
```

```
LOCAL Connected AS LONG
LOCAL AgentName AS STRING
LOCAL AgentFile AS STRING
```

```
' Create a new instance of the COM Object
AgentCtrlEx = NEWCOM $PROGID_Agent2
IF ISFALSE(ISOBJECT(AgentCtrlEx)) THEN EXIT FUNCTION
```

```
' Set the connected property
Connected = 1
OBJECT LET AgentCtrlEx.Connected = Connected
```

```
' Load the Merlin Agent Character
AgentName = UCODE$("Merlin")
AgentFile = UCODE$("Merlin.acs")
OBJECT CALL AgentCtrlEx.Characters.Load(AgentName, AgentFile)
```

```
' Display the Merlin Agent Character on the screen
OBJECT CALL AgentCtrlEx.Characters.Character(AgentName).Show
```

```
' Find the center of the screen for the Character Agent
OBJECT GET AgentCtrlEx.Characters.Character(AgentName).Width TO CharW
OBJECT GET AgentCtrlEx.Characters.Character(AgentName).Height TO CharH
DESKTOP GET CLIENT TO StartX, StartY
StartX = (StartX - CharW)\2
StartY = (StartY - CharH)\2
```

```
' Move the Character to the center of the screen
OBJECT CALL AgentCtrlEx.Characters.Character(AgentName).MoveTo(StartX,
StartY)
' more code here
```

OBJPTR function

[Top](#) [Previous](#) [Next](#)

Purpose	Return an object pointer contained in the specified object variable.
Syntax	<i>ObjectPointer??? = OBJPTR(objectvar)</i>
Remarks	OBJPTR returns the object pointer as a Double-word (DWORD) value.
See also	DIM , CLSID\$, GUID\$, GUIDTXT\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISINTERFACE , ISNOTHING , ISOBJECT , LET (with Objects) , OBJECT , OBJECTIVE , OBJRESULT , PROGID\$, What is an object, anyway?

Purpose	Returns a status code (hResult) to describe the success or failure of the most recent METHOD or PROPERTY procedure.
Syntax	<code>lResult& = OBJRESULT</code>
Remarks	<p>An Automation procedure is a METHOD or PROPERTY on an IAUTOMATION, IDISPATCH, or DUAL interface. By definition, an Automation procedure always returns a hidden result code which is cryptically called an hResult. OBJRESULT the most recent hResult generated by the program, and can be used to identify the success or failure of an operation.</p> <p>If an Automation procedure fails with a severe error, the ERR system variable is set to an appropriate Classic PowerBASICerror code. This is usually Error 99 ("Object error"). In such cases, you can use the OBJRESULT function to return the result (hResult&) of the last run-time OBJECT statement or direct METHOD/PROPERTY reference.</p> <p>Numeric equates for most OBJRESULT errors can be found in the WIN32API.INC file, and are mostly prefixed with %E_, %CO_, %OLE_, and %DISP_. The following list includes the most common codes that may be returned by a direct call of a Method or Property:</p>

%S_OK	= &H0
%S_FALSE	= &H1
%E_UNEXPECTED	= &H8000FFFF&
%E_NOTIMPL	= &H80004001&
%E_NOINTERFACE	= &H80004002&
%E_POINTER	= &H80004003&
%E_ABORT	= &H80004004&
%E_FAIL	= &H80004005&
%E_ACCESSDENIED	= &H80070005&
%E_HANDLE	= &H80070006&
%E_OUTOFMEMORY	= &H8007000E&
%E_INVALIDARG	= &H80070057&

This list tells the most common status codes which may be returned by a [DISPATCH](#) call using the OBJECT statement:

%S_OK	= &H0
%DISP_E_ARRAYISLOCKED	= &H8002000D
%DISP_E_BADINDEX:	= &H8002000B
%DISP_E_BADPARAMCOUNT	= &H8002000E
%DISP_E_BADVARTYPE	= &H80020008
%DISP_E_EXCEPTION	= &H80020009
%DISP_E_MEMBERNOTFOUND	= &H80020003
%DISP_E_NONAMEDARGS	= &H80020007
%DISP_E_OVERFLOW	= &H8002000A
%DISP_E_PARAMNOTFOUND	= &H80020004
%DISP_E_TYPERISMATCH	= &H80020005
%DISP_E_UNKNOWNINTERFACE	= &H80020001
%DISP_E_UNKNOWNLCID	= &H8002000C
%DISP_E_UNKNOWNNAME	= &H80020006

`%DISP_E_PARAMNOTOPTIONAL = &H8002000F`

If the status code `%DISP_E_EXCEPTION` is returned, you can use the [IDISPINFO](#) object to secure much additional information about the status. This includes a more specific error code, a description, help file information, etc. If the status code `%DISP_E_PARAMNOTFOUND` or `%DISP_E_TYPEMISMATCH`, you can use [IDISPINFO.PARAM](#) to determine which parameter actually caused the problem. Please refer to the [IDISPINFO](#) section for more details.

As can be seen from the above lists, a large numeric status code can be cryptic. However, you can translate the `OBJRESULT` code into a descriptive message using the [OBJRESULT\\$](#) function. This can be most helpful, especially during application development and [debugging](#).

Restrictions Methods and Properties on a custom interface (a direct interface based upon `IUnknown` rather than `IDispatch`) do not support OLE Automation, and do not return an `OBJRESULT` (`hResult`).

See also [DIM](#), [CLASS](#), [CLSID\\$](#), [IDISPINFO](#), [GUID\\$](#), [GUIDTXT\\$](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [ISINTERFACE](#), [ISNOTHING](#), [ISOBJECT](#), [LET \(with Objects\)](#), [METHOD](#), [PROPERTY](#), [OBJECT](#), [OBJACTIVE](#), [OBJPTR](#), [OBJRESULT\\$](#), [PROGID\\$](#), [What is an hResult?](#), [What is an object, anyway?](#)

OBJRESULT\$ function New!

[Top](#) [Previous](#) [Next](#)

Purpose	Returns a string which describes an OBJRESULT (hResult) code.
Syntax	<code>text\$ = OBJRESULT\$([nexp&])</code>
Remarks	This function returns a text string which describes the hResult code specified by nexp&. If the parameter nexp& is omitted, it is replaced by the most recent OBJRESULT value. That is, OBJRESULT\$() is identical to OBJRESULT\$(OBJRESULT).
See also	DIM , CLASS , CLSID\$, IDISPINFO , GUID\$, GUIDTXT\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISINTERFACE , ISNOTHING , ISOBJECT , LET (with Objects) , METHOD , PROPERTY , OBJECT , OBJACTIVE , OBJPTR , OBJRESULT , PROGID\$, What is an hResult? , What is an object, anyway?

Purpose	Return a string that is the octal (base 8) representation of its argument.
Syntax	<code>s\$ = OCT\$(<i>numeric_expression</i> [, <i>digits</i>])</code>
Remarks	<p><i>numeric_expression</i> must be in the range -2,147,483,648 to +4,294,967,295. Any fractional part of <i>numeric_expression</i> is rounded before the string is created.</p> <p>If <i>digits</i> is specified, the result will be of the length <i>digits</i>. If the value is shorter than <i>digits</i>, leading zero will be supplied to pad the string. If the value is longer than <i>digits</i>, the result will be truncated from the left. <i>digits</i> may range from 1 to 11. If <i>digits</i> is zero, no padding is used. If <i>digits</i> is less than zero, Classic PowerBASIC pads the string to 11 digits.</p> <p>Octal is a number system that uses base 8, rather than the base 10 used by the ordinary decimal system. A single octal digit represents three bits. Octal notation was commonly used in programming, because it was a convenient way of representing the binary bit patterns used internally by computers. Octal has largely been replaced by hexadecimal on modern computers, due to the more convenient size of hexadecimal notation. However, octal is still often used in Unix-oriented operating systems, and in C code, typically for historical reasons.</p> <p>By convention, octal numbers are unsigned, since they represent bit patterns. If you are not familiar with two's-complement arithmetic, the result of using OCT\$ on a negatively signed value may surprise you. See the BIN\$ function entry for information about two's-complement arithmetic.</p> <p>Octal strings can be converted to numeric values with the VAL function by prefixing the string with "&O", "&Q" or just "&". If the string has a leading zero, the result is always unsigned. For example:</p> <pre>a\$ = OCT\$(65535) ' a\$ contains "177777" x& = VAL("&Q" + a\$) ' Signed result (-1) y& = VAL("&Q0" + a\$) ' Unsigned result (65535)</pre>
See also	BIN\$, FORMAT\$, HEX\$, STR\$, USING\$, VAL

Purpose	Specify an error handling routine; enable or disable error trapping .
Syntax	<pre>ON ERROR GOTO {<i>label</i> <i>line_number</i>} ON ERROR RESUME NEXT ON ERROR GOTO 0</pre>
Remarks	<p>label or line_number identifies the first line of the error trapping routine. Once error handling has been turned on with this statement, all run-time errors result in a jump to your error handling code. You must always use the RESUME statement to continue execution once the error has been handled.</p> <p>To disable error trapping, use ON ERROR GOTO 0 or ON ERROR RESUME NEXT. You can use this technique if an error occurs for which you have not defined a recovery path; you can also choose to display the contents of ERR or ERRCLEAR at this time.</p> <p>The default for error trapping is disabled. If an error occurs while error trapping is disabled, the error code is placed into the ERR system variable, and execution continues. Errors can still be trapped by checking the value of the ERR or ERRCLEAR variable with IF ERR THEN or SELECT CASE ERR statements.</p> <p>Error trapping is local to each Sub, Function, Method, and Property. Classic PowerBASIC does not support global error trapping.</p> <p>Numeric errors such as Divide-by-zero, Overflow and Underflow are not trapped. Array out-of-bounds and null-pointer trapping are only enabled if #DEBUG ERROR ON is used.</p> <p>If you're running a program with error trapping turned off, a run-time error may cause a General Protection Fault (GPF). A GPF cannot be trapped with ON ERROR.</p> <p>It is not possible to branch to an error handler from within an expression. The compiler tests for an error only after the statement is completed. This means that a statement such as:</p> <pre>ON ERROR GOTO ErrorHandlerLabel IF GETATTR(<i>sFile</i>) THEN ' Do something END IF</pre> <p>will generate an Error 53 when <i>sFile</i> does not exist, but will not branch to the <i>ErrorHandlerLabel</i>. This is because the GETATTR(sFile) is an expression in the IF/ END IF block. You could do a</p> <pre>ON ERROR GOTO ErrorHandlerLabel a& = GETATTR(<i>sFile</i>) IF a& THEN ' Do something END IF</pre>

which will branch to the *ErrorHandlerLabel* if *sFile* does not exist. You could also check the value of the [ERR](#) variable or use a [TRY/ END TRY](#) block to see if an error occurred when checking for errors that occur during an expression.

See also

[#DEBUG DISPLAY](#), [#DEBUG ERROR](#), [ERR](#), [ERRCLEAR](#), [ERROR](#), [Error Overview](#), [ERROR\\$](#), [Errors and Error Trapping](#), [RESUME](#)

Purpose	Call one of several subroutines according to the value of a numeric expression.
Syntax	<code>ON <i>n</i> GOSUB {<i>label</i> <i>line_number</i>} [, {<i>label</i> <i>line_number</i>}] ...</code>
Remarks	<p><i>n</i> is a numeric expression ranging from 1 to 255, and each label or line number identifies a statement to branch to. When this statement is encountered, the <i>n</i>th label in the list is branched to; for example, if <i>n</i> equals 4, the fourth label in the list receives control. If <i>n</i> is less than one or greater than the number of labels, no branch occurs, and Classic PowerBASIC continues execution with the statement immediately following the ON GOSUB statement.</p> <p>Each subroutine should end with RETURN, which causes execution to resume with the statement immediately following the ON GOSUB statement. ON GOSUB can only branch to labels or line numbers that have the same scope as the ON GOSUB statement.</p> <p>The SELECT and IF blocks also perform multiple branching and are more flexible than ON GOSUB.</p> <p>Note that ON GOSUB (and ON GOTO) have been internally optimized to produce greater run-time performance than was possible with previous versions of Classic PowerBASIC.</p>
See also	GOSUB , FUNCTION/END FUNCTION , IF block , METHOD , ON GOTO , PROPERTY , RETURN , SELECT , SUB/END SUB
Example	<pre>FOR I& = 1 TO 3 ON I& GOSUB OneHandler, TwoHandler, ThreeHandler NEXT I& OneHandler: Message\$ = "Handler number" + STR\$(I&) RETURN TwoHandler: Message\$ = "Handler number" + STR\$(I&) RETURN ThreeHandler: Message\$ = "Handler number" + STR\$(I&) RETURN</pre>
Result	<pre>Handler number 1 Handler number 2 Handler number 3</pre>

Purpose	Send program flow to one of several possible destinations based on the value of a numeric expression.
Syntax	<code>ON <i>n</i> GOTO {<i>label</i> <i>line_number</i>} [, {<i>label</i> <i>line_number</i>}] ...</code>
Remarks	<p><i>n</i> is a numeric expression ranging from 1 to 255, and label or line_number identifies a statement in the program to branch to. The <i>n</i>th label is branched to; for example, if <i>n</i> equals 4, the fourth label in the list receives control. If <i>n</i> is less than one or greater than the number of labels in the list, program execution continues with the statement that immediately follows the ON GOTO statement.</p> <p>ON GOTO behaves exactly like ON GOSUB, except that it performs a GOTO rather than a GOSUB. This means that the program retains no memory of where the branch originated. ON GOTO can only branch to labels or line numbers that have the same scope as the ON GOTO statement.</p> <p>The SELECT and IF blocks also perform multiple branching, and are more flexible than ON GOTO. See the GOTO entry for a discussion of ways to avoid using GOTOs in your programs.</p> <p>Note that ON GOTO (and ON GOSUB) have been internally optimized to produce greater run-time performance than was possible with previous versions of Classic PowerBASIC.</p>
See also	GOTO , IF block , ON GOSUB , SELECT
Example	<pre>SUB Main FOR I& = 1 TO 3 ON I& GOTO OneHandler, TwoHandler, ThreeHandler Back: NEXT I& EXIT SUB OneHandler: Message\$ = "Handler number" + STR\$(I&) GOTO Back TwoHandler: Message\$ = "Handler number" + STR\$(I&) GOTO Back ThreeHandler: Message\$ = "Handler number" + STR\$(I&) GOTO Back END SUB</pre>
Result	<pre>Handler number 1 Handler number 2 Handler number 3</pre>

Purpose Prepare a [file](#) or device for reading or writing.

Syntax

```
OPEN filespec [FOR mode] [ACCESS access] [LOCK lock] AS _  
    [#] filenum& [LEN = record_size] [BASE = base]  
OPEN HANDLE filehandle [FOR mode] [ACCESS access] [LOCK lock] AS _  
    [#] filenum& [LEN = record_size] [BASE = base]
```

Remarks The OPEN statement comprises the following elements:

filespec A [string expression](#) specifying the name of the file to be opened, and may optionally include a drive and/or path specification. *filespec* may be either a Short File Name (SFN) or a Long File Name (LFN). *filespec* has a limit of 259 characters (%MAX_PATH - 1), although the file name portion of *filespec* may be no more than 255 characters (%MAX_FNAME - 1).

mode Specifies the file organization and style of access ([sequential](#), [random access](#), or [binary](#)) for reading, writing (or both), or appending. If *mode* is not specified, the default is RANDOM access.

Mode	File type	Action
INPUT	Sequential	Read from
OUTPUT	Sequential	Write to
APPEND	Sequential	Append to
BINARY	Binary	Reading or writing
RANDOM	Random	Reading or writing (default)

access Specifies the type of access this process will have to the file. By default, the file may be written to and read from.

Access	Description
READ	Only read operations allowed
WRITE	Only write operations allowed
READ WRITE *	Both read and write operations allowed (default)

*** If you explicitly specify an ACCESS clause in an OPEN statement for a file opened in APPEND mode, the ACCESS clause must be READ WRITE.**

lock Specifies the type of access other processes will have to the file. If a LOCK clause is not specified in the OPEN statement, the default LOCK READ WRITE mode is applied. This mode ensures exclusive access to the file, and enables Classic PowerBASIC to optimize its internal buffering for utmost I/O performance. If other processes or threads are to be permitted WRITE access to the file (LOCK SHARED or LOCK READ), internal buffering is disabled. Whilst performance may be marginally lower, it ensures that data read from the file is completely up-to-date.

Lock	Description
LOCK SHARED	Both read and write operations allowed
LOCK WRITE	Prevent write operations
LOCK READ	Prevent read operations
LOCK READ WRITE	Neither read nor write operations allowed (default)

To open a text file for OUTPUT and allow other processes to only read the file, use the following:

```
OPEN "MYFILE.TXT" FOR OUTPUT LOCK WRITE AS #1
```

It is possible for an application to open more than one copy of a given file at the same time. In this case, each OPEN statement must use a unique file number, and LOCK READ WRITE mode should not be used.

filenum& A unique [integer](#) value identifying the file, in the range 1 to 32767. Typically, this value is obtained from the [FREEFILE](#) function.

record_size Specifies the size of each record of a random access file. The default record length is 128 if not specified. If *record_size* is specified for a sequential file, it instructs Classic PowerBASIC to use internal buffering to improve I/O performance. A random access file is limited to 32768 bytes per record, to ensure consistent behavior across all Win32 platforms.

base Specifies the number of the first record in a random access file, or the number of the first byte in a sequential or binary file. It can be either zero (0) or one (1). The default value for *base* is 1, if not specified.

The main function of OPEN is to associate a file number (*filenum&*) with a file or physical device and to prepare that device for reading and/or writing. This file number is then used, rather than its name, in every statement that refers to the file. The FREEFILE function can be used to determine the next available file number, or you can pick one yourself. The OPEN

statement contains information on the mode of the file; that is, the methods by which the file will be accessed: sequential (for input/output to a new file, or output to an existing file), random access, and binary. An OPEN statement is usually balanced by a matching [CLOSE](#) statement.

HANDLE The HANDLE option allows you to access files that have already been opened by another process, DLL, or API function. The *filehandle* specified here must be a valid Win32 operating system file handle.

When Classic PowerBASIC closes a file opened with OPEN HANDLE, the Win32 handle is simply detached from the internal Classic PowerBASIC handle table. The file is not physically closed since Classic PowerBASIC did not originally open it. In Classic PowerBASIC, the [FILEATTR](#) function can be used to obtain the operating system file handle for a file opened with the OPEN statement.

Restrictions Attempting to OPEN a file for INPUT that does not exist causes a run-time [Error 53](#) ("File not found"). Attempting to open a file that is locked can result in either an [Error 70](#) ("Permission denied"), or an [Error 75](#) ("Path/file access error").

Similarly, attempting to OPEN a file using a file number that is already in use will result in a run-time [Error 55](#) ("File is already open "). For this reason, programs that use hard-coded file numbers should take special care to close files before the file number is used again. In addition, code that may be used by more than one thread should use FREEFILE and avoid hard-coded file numbers.

If you try to open a nonexistent file for OUTPUT, APPEND, RANDOM, or BINARY operations, a new file is automatically created. For this reason, files on Read-only network drives may only be opened in INPUT mode.

See also [CLOSE](#), [FILEATTR](#), [FILENAME\\$](#), [FILESCAN](#), [FREEFILE](#), [TCP OPEN](#), [UDP OPEN](#)

Example This program is divided into five procedures. The difference between each procedure is the mode in which the file is opened, and the way the data in the file is manipulated:

```
SUB SequentialOutput
' The file is opened for sequential output,
' and some data is written to it. If the file
' exists, it is over-written.
OPEN "OPEN.DTA" FOR OUTPUT AS #1
IntegerVar% = 12345
TempStr$ = "History is made at night."
WRITE #1, TempStr$, IntegerVar%*2, TempStr$, IntegerVar% \ 2
CLOSE #1
END SUB ' end procedure Sequential Output
```

```

SUB SequentialAppend
' The file is opened for sequential output, and
' data in this case is added to the end of file.
' If the file does not exist, it is created.
OPEN "OPEN.DTA" FOR APPEND AS #1
IntegerVar% = 32123
TempStr$ = "I am not a number!"
WRITE #1, TempStr$, IntegerVar% * 0.2
CLOSE #1
END SUB ' end procedure Sequential Append

```

```

SUB SequentialInput
' The file is opened for sequential input,
' and data is read from the file.
DIM a$
OPEN "OPEN.DTA" FOR INPUT AS #1
LINE INPUT #1, TempStr$
TempStr$ = ""
WHILE ISFALSE EOF(1) ' check if at end of file
    LINE INPUT #1, a$
    TempStr$ = TempStr$ + a$
WEND
CLOSE #1
END SUB ' end procedure SequentialInput

```

```

SUB BinaryIO
' The file is opened for binary I/O. Data is
' read 'using GET$. SEEK explicitly moves the
' file pointer to 'the end of file, and the
' same data is written back to 'the file.
OPEN "OPEN.DTA" FOR BINARY AS #1
TempStr$ = ""
WHILE ISFALSE EOF(1)
    GET$ #1, 1, Char$
    TempStr$ = TempStr$ + Char$
WEND
SEEK #1, LOF(1)
FOR I& = 1 TO LEN(TempStr$)
    PUT$ #1, MID$(TempStr$,I&,1)
NEXT I&
CLOSE 1
END SUB ' end procedure BinaryIO

```

```

SUB RandomIO
' Open file for random I/O. GET and PUT read
' and write the data.
OPEN "OPEN.DTA" FOR RANDOM AS #1 LEN = 1
TempStr$ = ""
TempSize& = LOF(1) ' save file size
' using GET, read in the entire file
FOR I& = 1 TO TempSize&
    GET #1, I&, Char$
    TempStr$ = TempStr$ + Char$
NEXT I&
' PUT copies the data in reverse into the
' random access file.
SEEK #1, 1
FOR I& = TempSize& TO 1 STEP -1
    LSET Char$ = MID$(TempStr$,I&,1)
    PUT #1,, Char$
NEXT I&

```

```
CLOSE #1  
END SUB ' end procedure RandomIO
```

OPTION EXPLICIT statement

[Top](#) [Previous](#) [Next](#)

Purpose	Force explicit declaration of all variables .
Syntax	<code>OPTION EXPLICIT</code>
Remarks	<p>Using OPTION EXPLICIT in a program has the same effect as using the #DIM ALL metastatement. That is, it requires that all variables be declared before they are used.</p> <p>When this option is used, the compiler generates a compile-time error if a variable or array is used without being explicitly declared.</p>
See also	#DIM

- Purpose

The OR operator works as both a logical and a bitwise [arithmetic operator](#).
- Syntax

$p \text{ OR } q$
- Remarks

Using OR as a logical operator

OR returns TRUE (non-zero) if and only if either or both of its operands is TRUE. Here is OR's truth table:

Truth table		
x	y	x OR y
T	T	T
T	F	T
F	T	T
F	F	F

- See also

[Arithmetic Operators](#), [AND](#), [EQV](#), [IMP](#), [ISFALSE](#), [ISTRUE](#), [LET](#), [NOT](#), [XOR](#)

Purpose	Parse an entire string and extract all delimited fields into an array .
Syntax	<code>PARSE start\$, target\$() [, {[ANY] delim\$ BINARY}]</code>
Remarks	<p>PARSE parses the entire string or string expression specified by <i>start\$</i>, assigning each delimited sub-string to successive elements of <i>target\$</i>. The array specified by <i>target\$</i> may be a dynamic string array, a fixed-length string array, an ASCIIZ string array, or a user defined type.</p> <p>The field delimiter is defined by <i>delim\$</i>, which may be one or more characters long. To be valid, the entire delimiter must match exactly, but the delimiter itself is never assigned as a part of the delimited field.</p> <p>If <i>delim\$</i> is not specified or is null (zero-length), standard comma-delimited (optionally quoted) fields are presumed. In this case only, the following parsing rules apply. If a standard field is enclosed in optional quotes, they are removed. If any characters appear between a quoted field and the next comma delimiter, they are discarded. If no leading quote is found, any leading or trailing blank spaces are trimmed before the field is returned.</p>
ANY	If the ANY option is chosen, each appearance of any single character comprising <i>delim\$</i> is considered a valid delimiter.
BINARY	<p>The BINARY option presumes that the <i>string_expr</i> was created with the JOIN\$/BINARY function, or its equivalent, which creates a string as a binary image or in the Classic PowerBASIC and/or Visual Basic packed string format: If a string is shorter than 65535 bytes, it starts with a 2-byte length WORD followed by the string data. Otherwise it will start a 2-byte value of 65535, followed by a DWORD indicating the string length, then finally the string data itself.</p> <p>It is usually advantageous to dimension <i>target\$</i> to the correct size with the use of the PARSECOUNT function. The PARSE statement is typically much more efficient, as a whole, than repeated use of the PARSE\$ function when it is necessary to parse an entire string expression.</p> <p>The JOIN\$ function is the natural complement to the PARSE statement.</p>
See also	JOIN\$, PARSE\$, PARSECOUNT , PATHNAME\$, PATHSCAN\$
Example	<pre>a\$ = "Trevor, Bob, Bruce, Dan, Simon, Jenny" DIM b\$(1 TO PARSECOUNT(a\$)) PARSE a\$, b\$() ARRAY SORT b\$()</pre>
Result	<pre>b\$(1) = "Bob" b\$(2) = "Bruce" b\$(3) = "Dan" b\$(4) = "Jenny"</pre>


```
b$(5) = "Simon"  
b$(6) = "Trevor"
```

Purpose	Return a delimited field from a string expression .
Syntax	<code>a\$ = PARSE\$(string_expr [, {[ANY] string_delimiter BINARY}], index&)</code>
Remarks	PARSE\$ uses the following parameters:
<i>string_expr</i>	The string to parse. If <i>string_expr</i> is empty (a null string) or contains no delimiter character(s), the string is considered to contain exactly one field. In this case, PARSE\$ will return <i>string_expr</i> .
<i>string_delimiter</i>	<p>Contains delimiter character(s). A delimiter is a character, list of characters, or string, that is used to mark the end of a field in <i>string_expr</i>. For example, if you consider a sentence to be a list of words, the delimiter between the words is a space (or perhaps punctuation). Text files typically consist of lines that are delimited by CR/LF (\$CRLF or CHR\$(13,10)) characters; a database file may consist of items separated by commas; etc. A delimiter is not considered part of a field, but as the divider between fields, so the delimiter is never returned by PARSE\$.</p> <p>If <i>delim\$</i> is not specified or is null (zero-length), standard comma-delimited (optionally quoted) fields are presumed. In this case only, the following parsing rules apply. If a standard field is enclosed in optional quotes, they are removed. If any characters appear between a quoted field and the next comma delimiter, they are discarded. If no leading quote is found, any leading or trailing blank spaces are trimmed before the field is returned.</p> <p>Delimiters are case-sensitive, so capitalization may be a consideration.</p>
ANY	If the ANY keyword is used, <i>string_delimiter</i> contains a set of characters, any of which may act as a delimiter character. If the ANY keyword is omitted, the entire <i>string_delimiter</i> string acts as a single delimiter.
BINARY	The BINARY option presumes that <i>string_expr</i> was created with the JOIN\$/BINARY function, or its equivalent, which creates a string as a binary image or in the Classic PowerBASIC and/or Visual Basic packed string format: If a string is shorter than 65535 bytes, it starts with a 2-byte length WORD followed by the string data. Otherwise it will start a 2-byte value of 65535, followed by a DWORD indicating the string length, then finally the string data itself.
<i>index&</i>	An numeric variable or expression that specifies the delimited field number to return. The first field is 1, and so on up to the maximum number of fields contained in <i>string_expr</i> , which may be determined with the PARSECOUNT function. If <i>index&</i> is negative, <i>string_expr</i> is parsed from

right to left. In this case, index& = -1 returns the last field in string_expr, -2 returns the second to last, etc. If index& evaluates to zero, or is outside of the actual field count, an empty string is returned.

See also

[JOIN\\$](#), [PARSE](#), [PARSECOUNT](#), [PATHNAME\\$](#), [PATHSCAN\\$](#)

Example

```
a$ = PARSE$ ("one,two,three", 2)      ' returns "two"
a$ = PARSE$ ("one;two,three", 2)      ' returns "three"
a$ = PARSE$ ("one",2)                  ' returns ""
a$ = PARSE$ ("xyz",1)                  ' returns "xyz"
a$ = PARSE$ ("xx1x","x",3)              ' returns "1"
a$ = PARSE$ ("1;2,3", ANY " , ; ", 2) ' returns "2"
```

Purpose	Return the count of delimited (sub) fields in a string expression .
Syntax	<code>x& = PARSECOUNT(string_expr [, {[ANY] string_delimiter BINARY}])</code>
Remarks	<p>PARSECOUNT uses the same rules as PARSE\$ in the determination of fields within <i>string_expr</i>. Individual fields within <i>string_expr</i> are evaluated, and the tally of the fields forms the result value.</p> <p><i>string_expr</i> is the string to parse. If <i>string_expr</i> is empty (a null string) or contains no delimiter character(s), the string is considered to contain exactly one field. In this case, PARSECOUNT returns the value 1.</p> <p><i>string_delimiter</i> defines one or more characters to use as a delimiter. To be valid, the entire delimiter must match exactly, but the delimiter itself is never returned as part of the field.</p> <p>If <i>string_delimiter</i> is not specified, or contains an empty string, special rules apply. The delimiter is assumed to be a comma. Fields may optionally be enclosed in quotes, and are ignored before the result string is returned. Any characters that appear between a quote mark and the next comma delimiter character are discarded. If no leading quote is found, any leading or trailing quotes are trimmed before the result string is returned.</p> <p>If <i>string_expr</i> is empty (a null string), PARSECOUNT will return 1, indicating that one empty (null) field exists. Similarly, if <i>string_expr</i> does not contain any delimiters, <i>string_expr</i> is considered to contain one field.</p>
ANY	If the ANY keyword is used, <i>string_delimiter</i> contains a set of characters, any of which may act as a delimiter character. If the ANY keyword is omitted, the entire <i>string_delimiter</i> string acts as a single delimiter.
BINARY	The BINARY option returns the number of sub-fields and presumes that <i>string_expr</i> was created with the JOIN\$/BINARY function, or its equivalent, which creates a string as a binary image or in the Classic PowerBASIC and/or Visual Basic packed string format: If a string is shorter than 65535 bytes, it starts with a 2-byte length WORD followed by the string data. Otherwise it will start a 2-byte value of 65535, followed by a DWORD indicating the string length, then finally the string data itself.
See also	JOIN\$, PARSE , PARSE\$, PATHNAMES\$, PATHSCAN\$
Example	<pre>a& = PARSECOUNT("one,two,three") ' returns 3 a& = PARSECOUNT("one;two,three") ' returns 2 a& = PARSECOUNT("") ' returns 1 a& = PARSECOUNT("xx1x","x") ' returns 4</pre>

```
a& = PARSECOUNT("1;2,3", ANY ",;") ' returns 3
```

Purpose Parse a path/file name to extract component parts

Syntax `fil$ = PATHNAME$(director, filespec$)`

Remarks The PATHNAME\$ function evaluates a path/file text name, and returns a requested part of the name. The functionality is strictly one of string parsing alone. No attempt is made to find the file on disk. If you wish to scan for a particular file on disk, you should use the companion function [PATHSCAN\\$](#).

director This is one of the following words which is used to specify the requested part:

- FULL Return the full path/file name, just as given in the *filespec\$* parameter. This is really a non-operation, but is included for symmetry with the companion function PATHSCAN\$.
- PATH Return the path portion of the path/file name. That is the text up to and including the last backslash (\).
- NAME Return the name portion of the path/file name. That is the text to the right of the last backslash (\), ending just before the last period (.
- EXTN Return the extension portion of the path/file name. That is the last period (.) in the string plus the text to the right of it.
- NAMEX Return the NAME and the EXTN parts combined.

filespec\$ A path/file name which does not necessarily exist on disk.

See also [DIR\\$](#), [EXE](#), [PATHSCAN\\$](#), [PARSE](#), [PARSE\\$](#), [PARSECOUNT](#)

Example

```
PATHNAME$(PATH, "C:\PB\XXX.TXT") ' returns "C:\PB\  
PATHNAME$(NAME, "C:\PB\XXX.TXT") ' returns "XXX"  
PATHNAME$(NAMEX, "C:\PB\XXX.TXT") ' returns "XXX.TXT"  
PATHNAME$(EXTN, "C:\PB\XXX.TXT") ' returns ".TXT"
```

Purpose	Find a file on disk and return the path and/or file name parts.										
Syntax	<code>fil\$ = PATHSCAN\$(director, filespec[, pathspec\$])</code>										
Remarks	The PATHSCAN\$ function scans specified directories to find a particular file. If the file is found, it returns either the full path/file name, or a selected part of it. If the file is not found, a nul (zero-length) string is returned. If you wish to simply parse a text file name, without regard to its validation on disk, you should use the companion function PATHNAMES\$.										
<i>director</i>	<p>This is one of the following words which is used to specify the requested part:</p> <table><tr><td>FULL</td><td>Return the full drive/path/file name.</td></tr><tr><td>PATH</td><td>Return the path portion of the path/file name. That is the text up to and including the last backslash (\).</td></tr><tr><td>NAME</td><td>Return the name portion of the path/file name. That is the text to the right of the last backslash (\), ending just before the last period (.) in the string.</td></tr><tr><td>EXTN</td><td>Return the extension portion of the path/file name. That is the last period (.) in the string plus the text to the right of it.</td></tr><tr><td>NAMEX</td><td>Return the NAME and the EXTN parts combined.</td></tr></table>	FULL	Return the full drive/path/file name.	PATH	Return the path portion of the path/file name. That is the text up to and including the last backslash (\).	NAME	Return the name portion of the path/file name. That is the text to the right of the last backslash (\), ending just before the last period (.) in the string.	EXTN	Return the extension portion of the path/file name. That is the last period (.) in the string plus the text to the right of it.	NAMEX	Return the NAME and the EXTN parts combined.
FULL	Return the full drive/path/file name.										
PATH	Return the path portion of the path/file name. That is the text up to and including the last backslash (\).										
NAME	Return the name portion of the path/file name. That is the text to the right of the last backslash (\), ending just before the last period (.) in the string.										
EXTN	Return the extension portion of the path/file name. That is the last period (.) in the string plus the text to the right of it.										
NAMEX	Return the NAME and the EXTN parts combined.										
<i>filespec\$</i>	A file name which is expected to exist on disk. It must not be an ambiguous name -- that is, it may not include a query (?) or an asterisk (*) character.										
<i>pathspec\$</i>	<p>An optional path string which includes one or more paths to be searched to find filespec\$. If multiple path names are included in this string, they must each be separated by a semicolon (;) delimiter. If pathspec\$ is not given, or it is a nul (zero-length) string, the following directories are searched:</p> <ol style="list-style-type: none">1. The directory from which the application was loaded.2. The current directory.3. The Windows System32 directory.4. The Windows System16 directory.5. The Windows directory.6. The directories in the PATH environment variable.										
See also	DIR\$, EXE , PATHNAMES\$, PARSE , PARSE\$, PARSECOUNT										
Example	The following information assumes that the file named "MyFile.txt" currently										

exists in the directory "C:\PB".

```
f$ = "MyFile.txt" : p$ = "C:\MyDir;C:\PB"
```

```
PATHSCAN$(FULL, f$, p$) ' returns "C:\PB\MyFile.txt"
```

```
PATHSCAN$(PATH, f$, p$) ' returns "C:\PB\"
```

```
PATHSCAN$(NAME, f$, p$) ' returns "MyFile"
```

```
PATHSCAN$(EXTN, f$, p$) ' returns ".txt"
```

```
PATHSCAN$(NAMEX, f$, p$) ' returns "MyFile.txt"
```


PBLIBMAIN function

[Top](#) [Previous](#) [Next](#)

Purpose

PBLIBMAIN performs a similar task to [DLLMAIN](#) and [LIBMAIN](#), except that PBLIBMAIN takes no parameters.

In 32-bit Windows, PBLIBMAIN is called each time a [DLL](#) is loaded or unloaded by an application or process, and (usually) if a thread is started and stopped. Your code should never call PBLIBMAIN.

Syntax

```
FUNCTION PBLIBMAIN [()] [AS LONG]
```

Remarks

See [LIBMAIN](#) / [DLLMAIN](#) for more information.

See also

[DLLMAIN](#), [LIBMAIN](#), [PBMAIN](#), [THREAD CREATE](#), [WINMAIN](#)

PBMAIN function

[Top](#) [Previous](#) [Next](#)

Purpose	PBMAIN is a user-defined function called by Windows to begin running an executable application. Every Classic PowerBASICexecutable (EXE) must contain either a PBMAIN or a WINMAIN function.
Syntax	<code>FUNCTION PBMAIN [()] [AS LONG]</code>
Remarks	<p>Either a PBMAIN or WINMAIN function is required in every Classic PowerBASIC application (.EXE). If you use PBMAIN, no parameters are passed, and you cannot directly obtain the instance handle of your application or the pointer to any command-line parameters.</p> <p>However, you can use COMMAND\$ to get the command-line passed to your program, and the GetModuleHandle API function to get the application instance handle.</p>
Return	The return value of PBMAIN has an effective range of 0 to 255. Batch files may act on the result through the IF [NOT] ERRORLEVEL batch command.
Restrictions	DLLs created with Classic PowerBASIC should contain a DLLMAIN , LIBMAIN , or PBLIBMAIN function instead of PBMAIN/WINMAIN.
See also	COMMAND\$, DLLMAIN , LIBMAIN , PBLIBMAIN , WINMAIN
Example	<pre>#COMPILE EXE FUNCTION PBMAIN MSGBOX "This is my program!" 'Return an error level of 15 FUNCTION = 15 ' or you can use PBMAIN = 15 END FUNCTION</pre>

PEEK and PEEK\$ functions

[Top](#) [Previous](#) [Next](#)

Purpose	Return the byte (PEEK) or sequence of bytes (PEEK\$) at a specified memory location.
Syntax	<pre>numvar = PEEK([datatype,] address???) strvar = PEEK\$([ASCIIZ,] address???, count&)</pre>
Remarks	<p>The PEEK functions and complementary POKE statements are low-level methods of accessing individual bytes in memory. The data is retrieved from memory, starting at the specified 32-bit <i>address???</i>.</p> <p>PEEK retrieves a numeric value starting at a specified memory address. PEEK\$ retrieves <i>count&</i> consecutive bytes and returns them as a string, where the ASCII code of the first character of the string is the value of the first byte retrieved, the next ASCII code is the next byte retrieved, and so on. If ASCIIZ is specified, PEEK\$ reads successive characters from an ASCIIZ buffer of the specified size, until a terminating \$NUL (CHR\$(0)) byte is found. Since ASCIIZ strings must contain a terminating \$NUL, the maximum length of the returned string is 1 character less than the specified size.</p>
<i>datatype</i>	The numeric data type to retrieve, which may be any one of BYTE , WORD , DWORD , INTEGER , LONG , QUAD , SINGLE , DOUBLE , EXT , CUR , CUX . If a data type is not specified, BYTE is assumed.
<i>address???</i>	A valid 32-bit memory address specifying the location in memory where data retrieval should begin.
<i>count&</i>	A numeric expression that specifies the number of consecutive bytes to be read from memory starting at <i>address???</i> .
Restrictions	If <i>address???</i> (or any byte in the range covered by <i>count&</i>) references an invalid address (memory that is not allocated to the application), Windows will generate a General Protection Fault (GPF) and terminate the application. GPFs cannot be trapped with an ON ERROR error handler.
See also	POKE , STRPTR , VARPTR
Example	<p>One common application for PEEK\$ and POKE\$ is to perform fast array and memory block copy operations by simply copying the entire block of memory which contains the array data, rather than storing each element individually with an assignment statement:</p> <pre>Elements& = 2000 ' 2000 elements in each array DIM OriginalArray%(1 TO Elements&) DIM NewArray%(1 TO Elements&) 'Method 1: assign each element individually FOR Index& = 1 TO Elements&</pre>

```
    NewArray%(Index&) = OriginalArray%(Index&)  
NEXT Index&
```

```
'Method 2: block copy with PEEK$ and POKE$ (faster)  
Source&    = VARPTR(OriginalArray%(1))  
Dest&      = VARPTR(NewArray%(1))  
ArrayLen& = Elements& * 2      'byte length of array  
POKE$ Dest&, PEEK$(Source&, ArrayLen&) 'copy block
```

Purpose	Store the byte (POKE), or sequence of bytes (POKE\$) at a specified memory location.
Syntax	<pre>POKE [<i>datatype</i>,] <i>address</i>???, <i>datavalue</i> POKE\$ [<i>ASCIIZ</i>,] <i>address</i>???, <i>string_expr</i></pre>
Remarks	<p>The POKE statements and complementary PEEK functions are low-level methods of accessing individual bytes at a specific address in memory. The data is stored to memory starting at location <i>address</i>???, which is a full 32-bit address.</p> <p>In its classic form, the POKE statement stores a single byte (8 bits) whose value ranges from 0 to 255. In its enhanced form, POKE provides the functionality of a dynamic pointer: the <i>datatype</i> parameter specifies the data type and hence the size of the target data to write to the target memory address. <i>datatype</i> can be any one of BYTE, WORD, DWORD, INTEGER, LONG, QUAD, SINGLE, DOUBLE, EXT, CUR, CUX.</p> <p>POKE\$ stores a string in consecutive bytes where the ASCII code of the first character of the string is stored in the first byte of memory, the next ASCII code is stored in the next byte of memory, and so on, until all bytes in <i>string_expr</i> are stored. If ASCIIZ is specified, POKE\$ writes successive characters to a target address in memory until a terminating \$NUL byte is found in the source string. If no \$NUL is found in the string, one is automatically appended to the target buffer. It is the programmer's responsibility to ensure that POKE\$ does not overrun the target memory area to avoid data corruption or protection faults.</p>
<i>address</i> ???	A valid 32-bit memory address, specifying the location in memory where the byte or sequence of bytes should be stored. Specifying an invalid address can result in a General Protection Fault (GPF).
<i>datavalue</i>	The data value to be stored at <i>address</i> ???
<i>string_expr</i>	A string constant , literal or string expression that specifies the sequence of bytes to be stored in memory starting at the byte referenced by <i>address</i> ???
Restrictions	If <i>address</i> ???
See also	GLOBALMEM ALLOC , PEEK , STRPTR , VARPTR

Purpose

Write data to a [device](#) or [sequential file](#).

Syntax

```
PRINT # fNum&  
PRINT # fNum&, [ExpList] [SPC(n)] [TAB(n)] [,] [;] [...]  
PRINT # fNum&, array$()
```

Remarks

The first form of the PRINT# statement (with or without a trailing comma) outputs a blank line to the file (i.e. a CR/LF only).

The second form of the PRINT# statement has the following parts, which may occur in any order and quantity, within a single PRINT# statement:

fNum& Number used in an [OPEN](#) statement to open a sequential file. It can be any numeric expression that evaluates to the number of an open file. Note that the Number symbol (#) preceding *fNum*& is not optional.

ExpList Numeric and/or [string expression](#)(s) to be written to the file.

SPC(*n*) An optional function used to insert *n* spaces into the printed output. Multiple use of the SPC argument is permitted in the PRINT statement, for example, between expressions. Values of *n* less than 1 are ignored.

TAB(*n*) An optional function used to tab to the *n*th column before printing *ExpList*. Multiple use of the TAB argument is permitted in the PRINT, for example, to position arguments in columns. Values of *n* less than 1 are ignored.

{*;*,*,*}

Character that determines the position of the next character printed. A semicolon (;) means the next character is printed immediately after the last character; a comma (,) means the next character is printed at the start of the next print zone. Print zones begin every 14 columns.

If the final argument of a PRINT# statement is a semicolon or comma, PRINT# will not append the (default) CR/LF byte pair to the data as it is written to the file. For example:

```
PRINT #1, "Hello";  
PRINT #1, " world!"
```

...produces the contiguous string "Hello world!" in the disk file.

If you omit all arguments, the PRINT# statement prints a blank line in the file (i.e., a CR/LF pair only), but you must include the comma after the file number. Because PRINT# writes an image of the data to the file, you must delimit the data so it is printed correctly. If you use commas as delimiters, PRINT# also writes the blanks between print fields to the file. Also, remember that spacing of data displayed on a text screen using

monospaced characters may not work well when the data is redisplayed in a graphical environment using proportionally spaced characters.

If you are not careful, you can waste a lot of disk space with unnecessary spaces, or worse, put fields so close together that you cannot tell them apart when they are later input with INPUT#. For example:

```
PRINT #1,1,2,3
```

sends:

```
1           2           3
```

to file #1. Because of the 14-column print zones between characters, superfluous spaces are sent to the file. On the other hand:

```
PRINT #1,1;2;3
```

sends:

```
1 2 3
```

to the file, and you cannot read the separate numeric values from this record because INPUT# requires commas as delimiters. The best way to delimit fields is to put a comma between each field, like so:

```
PRINT #1, 1 ", " 2 ", " 3
```

which writes:

```
1, 2, 3
```

to the file, and wastes the least possible space and is easy to read with an [INPUT#](#) statement. The [WRITE#](#) statement delimits fields with commas automatically.

PRINT# is advantageous when writing a single number or string on each line in a file. Use PRINT# followed by a comma but no arguments to write a blank line (carriage return/linefeed) to a file:

```
PRINT #1, 'writes a blank line to file #1
```

array\$() When PRINT# specifies an [array](#) name with empty parentheses, the entire array is written to the disk file as text strings, with each element delimited by a CR/LF ([\\$CRLF](#) or [CHR\\$\(13,10\)](#)). Numeric [arrays](#) are converted to the ASCII text equivalent.

Restrictions Arrays of [User-Defined Types](#) (UDTs) may not be used with the array form of the PRINT# statement.

See also [GET](#), [GET\\$](#), [INPUT#](#), [LINE INPUT#](#), [PUT](#), [PUT\\$](#), [WRITE#](#)

Example

```
' Classic PRINT# statement example
SUB MakeFile
' opens a sequential file for output. Using PRINT #,
' it writes lines of different data types to the file.
x& = FREEFILE
OPEN "INPUT#.DTA" FOR OUTPUT AS #x&
StringVariable$ = "I'll be back."
IntegerVar% = 1000
FloatingPoint! = 30000.12
' Write a line of text to the sequential file.
PRINT #x&, StringVariable$
```



```

PRINT #x&, IntegerVar%
PRINT #x&, FloatingPoint!
CLOSE #x& ' close file variable
END SUB ' end procedure MakeFile

```

```

SUB ReadFile
' Opens a sequential file for input. Using INPUT #,
' reads lines of different types of data from the file.
x& = FREEFILE
OPEN "INPUT#.DTA" FOR INPUT AS #x&
RESET StringVariable$
RESET IntegerVar%
RESET FloatingPoint!
' Read a line of text from the sequential file.
INPUT #x&, StringVariable$
INPUT #x&, IntegerVar%
INPUT #x&, FloatingPoint!
CLOSE #x& ' close file variable
END SUB ' end procedure ReadFile

```

```

' Array mode PRINT# statement example
a$ = "Trevor, Bob, Bruce, Dave, Simon, Jenny"
DIM b$(1 TO PARSECOUNT(a$))
PARSE a$, b$()
ARRAY SORT b$()
OPEN "filename.txt" FOR OUTPUT AS #1
PRINT #1, b$()
CLOSE #1

```

PRINTER\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve printer names and printer port names.
Syntax	<code>device\$ = PRINTER\$([NAME PORT], <i>printernum</i>&)</code>
Remarks	<code>printernum&</code> specifies the printer number, from 1 to PRINTERCOUNT . If the NAME option is specified in the first position, the printer name is returned. If the PORT option is specified instead, the port name (e.g., LPT1) is returned.
See also	LPRINT ATTACH , PRINTERCOUNT , XPRINT ATTACH

PRINTERCOUNT function

[Top](#) [Previous](#) [Next](#)

Purpose Retrieve the number of available (installed) printers.

Syntax *ncPrinters*& = PRINTERCOUNT

See also [LPRINT ATTACH](#), [PRINTER\\$](#), [XPRINT ATTACH](#)

Example

```
FUNCTION PBMAIN
  LOCAL ix AS LONG, sPrinters AS STRING
  FOR ix = 1 TO PRINTERCOUNT
    sPrinters = sPrinters & PRINTER$(NAME, ix) & $CRLF
  NEXT
  MSGBOX sPrinters
END FUNCTION
```

Purpose	Retrieve the Priority Value for the current process.
Syntax	<code>PROCESS GET PRIORITY TO <i>lResult</i>&</code>
Remarks	<p>PROCESS GET PRIORITY retrieves the priority value for the current process. The retrieved priority value is assigned to the long or dword variable designated by <i>lResult</i>&.</p> <p>The process priority value is one of the following:</p> <p><code>%IDLE_PRIORITY_CLASS</code> = &H00000040</p> <p>Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.</p> <p><code>%NORMAL_PRIORITY_CLASS</code> = &H00000020</p> <p>Indicates a normal process with no special scheduling needs.</p> <p><code>%HIGH_PRIORITY_CLASS</code> = &H00000080</p> <p>Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes. An example is Windows Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.</p> <p><code>%REALTIME_PRIORITY_CLASS</code> = &H00000100</p> <p>Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.</p>
See also	PROCESS SET PRIORITY , THREAD GET PRIORITY , THREAD SET PRIORITY

Purpose	Sets the Priority Value for the current process.
Syntax	<code>PROCESS SET PRIORITY <i>Priority</i>&</code>
Remarks	<p>PROCESS SET PRIORITY assigns a new priority value to the current process.</p> <p>The process priority value must be one of the following:</p> <p><code>%IDLE_PRIORITY_CLASS</code> = &H00000040</p> <p>Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.</p> <p><code>%NORMAL_PRIORITY_CLASS</code> = &H00000020</p> <p>Indicates a normal process with no special scheduling needs.</p> <p><code>%HIGH_PRIORITY_CLASS</code> = &H00000080</p> <p>Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes. An example is Windows Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.</p> <p><code>%REALTIME_PRIORITY_CLASS</code> = &H00000100</p> <p>Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.</p>
See also	PROCESS GET PRIORITY , THREAD GET PRIORITY , THREAD SET PRIORITY

Purpose Capture a profile report detailing total execution times of the [Subs](#) , [Functions](#), [Methods](#), and [Properties](#) in a program and write it to a disk file.

Syntax `PROFILE diskfilename$`

Remarks At the time the PROFILE statement is executed, a standard [sequential file](#) of the specified file name *diskfilename*\$ is created. For the best results in executable files, the PROFILE statement should be the last statement executed in the [PBMAIN/WINMAIN](#) function.

The profile report contains a list of every procedure within the same module (EXE or [DLL](#)), the number of times it was called, and the total elapsed time (in milliseconds) spent executing all instances of the procedure. These statistics appear in the disk file in that specific order on each line:

```
<Procedure Name>, <Call Count>, <Time mSec>
```

The profile report only describes procedures that physically reside within the module (EXE or DLL) where the PROFILE statement is located. Procedures in an external EXE or DLL are not profiled individually; however, the time taken to call other procedures and DLL/API functions is included in the accumulated execution time of the calling procedure.

It is highly recommended that you close all other applications when profiling a Classic PowerBASIC application. When an application is being profiled, Classic PowerBASIC must generate a considerable amount of extra code to gather all of the needed information. This extra code is generated whenever a valid PROFILE statement appears in your program, regardless of whether it is actually executed.

For final production code, use the [#TOOLS OFF](#) metastatement is used to ensure the highest performance levels.

Interpreting a profile report

The execution time of nested procedures needs to be understood in order to obtain a clear "picture" of the execution times. For example, consider the following results:

Procedure calls time

```
MySubA,    1,  11016
```

```
MySubB,   100, 10014
```

At first glance, these results may suggest a "bottleneck" in *MySubA* since it took *MySubA* 11016 milliseconds to execute just one call, whereas the average time for *MySubB* was only about 100 milliseconds per call (10014 mSec / 100 calls = 100.14 mSec).

However, if *MySubB* is actually **called by** *MySubA*, the results need to be assessed differently. For example, we could say: "*MySubB took 10014 milliseconds of the 11016 milliseconds of the time spent in MySubA*". Or to put it another way: "*Of the 11016 milliseconds MySubA took to execute, 10014 milliseconds of that time was spent executing MySubB*".

Interpolating these results, it can be easily calculated that the code in *MySubA* only took 1002 milliseconds to run, yet this blossomed to 11016 milliseconds because of its dependence on *MySubB*.

Therefore, improving the performance of *MySubB* would clearly improve the overall speed of *MySubA*, and the profile results of both functions would be improved accordingly.

Restrictions Profiling is "enabled" when the first procedure that contains a PROFILE statement begins execution. All procedures *subsequently* executed from within that procedure are profiled.

It is not possible to profile the actual PBMAIN or WINMAIN functions. If a PROFILE statement occurs within PBMAIN/WINMAIN, all procedures that are called from PBMAIN/[LIBMAIN](#) are profiled normally.

Therefore, if PBMAIN/WINMAIN contains code that requires profiling, simply rename the function and create a new PBMAIN/WINMAIN function that immediately calls the renamed function and then executes a PROFILE statement. See the example below.

For application code with nested and lengthy procedure calls, adding up the total number of milliseconds in the last column of a PROFILE disk file will usually produce a number that is far larger than the actual time it took your program to execute.

The time resolution of the profile report is limited by the *Quantum* supported by the operating system (Win95/98 is 54 mSec, and WinNT/2000/XP is 10 mSec), and can be influenced by any other applications which run concurrently. Nonetheless, PROFILE can offer a great insight as to which code may be consuming the most CPU time, and where optimization efforts should be concentrated.

See also [#TOOLS](#), [CALLSTK](#), [CALLSTK\\$](#), [CALLSTKCOUNT](#), [FUNCNAME\\$](#), [TRACE](#)

Example

```
SUB B1
  SLEEP 1000
END SUB
```

```
SUB A1
  SLEEP 250
  CALL B1
END SUB
```

```
FUNCTION PBMAIN
```

```
CALL A1
PROFILE "Profile Results.txt"    ' Profile at end
END FUNCTION
```

```
A1,      1,1252
B1,      1,1002
PBMAIN, 1, 0
```

Result

Purpose	Return the unique alphanumeric PROGID string (text) associated with a unique CLSID string of a COM object or component. A COM object/component must include an alphanumeric PROGID string in order to be used by Classic PowerBASIC (and Visual Basic).
Syntax	<code>a\$ = PROGID\$(ClassID\$)</code>
Remarks	<p>A PROGID string is the unique alphanumeric text name associated with a given COM object/component. For example, "Word.Application.8".</p> <p>You convert the 16-byte (128-bit) binary class ID of a COM object/component into a PROGID string with the PROGID\$ function.</p> <p>PROGID\$ takes the (16-byte) binary string <i>ClassID\$</i> representing the GUID or UUID of a COM object/component, and examines the system registry in order to determine the PROGID string associated with the <i>ClassID\$</i> string. <i>ClassID\$</i> may be a dynamic string or fixed-length string of at least 16 bytes, or (typically) a GUID variable.</p> <p>If the <i>ClassID\$</i> cannot be found, or any error occurs in the lookup process, PROGID\$ will not set the ERR system variable, but will return an empty string.</p> <p>PROGID\$ is the complement to the CLSID\$ function. Using these two functions together, it is possible to extract the precise capitalization of the PROGID from the system registry. See the example below.</p>
See also	DIM , CLSID\$, GUID\$, GUIDTXT\$, INTERFACE (Direct) , INTERFACE (IDBind) , ISINTERFACE , ISNOTHING , ISOBJECT , Just what is COM? , LET (with Objects) , METHOD , OBJECT , OBJACTIVE , OBJPTR , OBJRESULT , PROPERTY , What is an object, anyway?
Example	<pre>DIM MSWordClassID AS GUID MSWordClassID = CLSID\$("Word.Application") IF TRIM\$(MSWordClassID, \$NUL) <> "" THEN 'Success getting the CLSID\$ of MSWord a\$ = PROGID\$(MSWordClassID) 'a\$ now contains "Word.Application.8" b\$ = GUIDTXT\$(MSWordClassID) 'b\$ holds "{000209FF-0000-0000-C000-000000000046}" END IF</pre>

Purpose	Manipulate a PROGRESSBAR control. A ProgressBar is a rectangle that is gradually filled, left to right, as some work progresses.
Syntax	<pre>PROGRESSBAR GET POS <i>hDlg</i>, <i>id</i>& TO <i>datav</i>& PROGRESSBAR GET RANGE <i>hDlg</i>, <i>id</i>& TO <i>LoDatav</i>&, <i>HiDatav</i>& PROGRESSBAR SET POS <i>hDlg</i>, <i>id</i>&, <i>position</i>& PROGRESSBAR SET RANGE <i>hDlg</i>, <i>id</i>&, <i>lolimit</i>&, <i>hilimit</i>& PROGRESSBAR SET STEP <i>hDlg</i>, <i>id</i>&, <i>stepval</i>& PROGRESSBAR STEP <i>hDlg</i>, <i>id</i>& [, <i>incramt</i>&]</pre>
<i>hDlg</i>	Handle of the dialog that owns the ProgressBar.
<i>id</i> &	The control identifier assigned with CONTROL ADD PROGRESSBAR .
Remarks	In each of the following samples and descriptions, the PROGRESSBAR control that is the subject of the statement is identified by the handle of the dialog that owns the ProgressBar (<i>hDlg</i>), and the unique control identifier you gave it upon creation in CONTROL ADD PROGRESSBAR. To alter the color of the bar or the background, use CONTROL SET COLOR .

PROGRESSBAR GET POS *hDlg*, *id*& TO *datav*&

The current position of the ProgressBar is retrieved and assigned to the variable designated by *datav*&.

PROGRESSBAR GET RANGE *hDlg*, *id*& TO *LoDatav*&, *HiDatav*&

The current range of the ProgressBar is retrieved and assigned to the variables designated by *LoDatav*& and *HiDatav*&. Upon ProgressBar creation, the default range is 0 to 100.

PROGRESSBAR SET POS *hDlg*, *id*&, *position*&

The current position of the ProgressBar is set to the value of the parameter *position*&, and the bar is redrawn to reflect the new position.

PROGRESSBAR SET RANGE *hDlg*, *id*&, *lolimit*&, *hilimit*&

The range for the ProgressBar is specified to be from *lolimit*& to *hilimit*&. If *lolimit*& is greater than *hilimit*&, the results are undefined.

PROGRESSBAR SET STEP *hDlg*, *id*&, *step*&

The default increment value to be used by PROGRESSBAR STEP is specified by the *stepval*& parameter.

PROGRESSBAR STEP *hDlg*, *id*& [, *incramt*&]

The ProgressBar is "stepped". The current position is advanced by the step increment, and the bar is redrawn to reflect the new position. If the optional *incramt*& expression is included, the position is advanced by that amount instead. The default step increment is 10, and the default range is from 0 to 100.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD PROGRESSBAR](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#)

PROPERTY/END PROPERTY statements New!

[Top](#) [Previous](#)
[Next](#)

Purpose	Define a PROPERTY procedure within a class .
Syntax	<pre>[VERRIDE] PROPERTY GET SET <i>name</i> [<DispID>] [ALIAS "<i>altname</i>"] (var AS <i>type</i>...) [AS <i>type</i>] [<i>statements</i>] PROPERTY = <i>expression</i> END PROPERTY</pre>
Remarks	<p>PROPERTY/END PROPERTY is used to define a PROPERTY procedure within a class. Properties can only be called through a virtual function table on an active object. A PROPERTY is a special type of METHOD, which is only used to set or retrieve data in an object. While the work of a PROPERTY could readily be accomplished with a standard METHOD, this distinction is convenient to emphasize the concept of encapsulation of instance data within an object. There are two forms of PROPERTY procedures: PROPERTY GET and PROPERTY SET. As implied by the names, the first form is used to retrieve a data value from the object, while the second form is used to assign a value. Properties must be defined within a CLASS Block, and may only be declared within a DECLARE CLASS Block. Properties are defined:</p>

```
PROPERTY GET name [ALIAS "altname"] (BYVAL var AS type...) [AS type]  
  [statements]  
  PROPERTY = expression  
END PROPERTY
```

```
PROPERTY SET name [ALIAS "altname"] (BYVAL var AS type...) [AS type]  
  [statements]  
  variable = value  
END PROPERTY
```

When you use PROPERTY SET, the last (or only) parameter is used to pass the value to be assigned. A PROPERTY may be considered "Read-Only" or "Write-Only" by simply omitting one of the definitions. However, if both GET and SET forms are defined for a particular property, parameters and the property must be identical in both forms, and they must be paired. That is, the PROPERTY SET must immediately follow the PROPERTY GET.

Properties can only be called through a virtual function table on an active object. Property parameters may be of any variable type. You can access a PROPERTY GET with:

```
DIM ObjVar AS MyInterface  
LET ObjVar = NEWCOM Progid$  
1. ObjVar.Prop1(param) TO var  
2. CALL ObjVar.Prop1(param) TO var  
3. var = ObjVar.Prop1(param)
```

You can access a PROPERTY SET with:

```
DIM ObjVar AS MyInterface
LET ObjVar = NEWCOM Progid$
1. ObjVar.Prop1(param) = expr
2. CALL ObjVar.Prop1(param) = expr
```

Note that the choice of Property procedure is syntax directed. In other words, depending upon the way you use the name, Classic PowerBASIC will automatically decide whether the GET or SET PROPERTY should be called.

In every Method and Property, Classic PowerBASIC automatically defines a pseudo-variable named [ME](#), which is treated as a reference to the current object. Using ME, it's possible to call any other Method or Property which is a member of the class: *var = ME.Method1(param)*

Methods may be declared (using AS type...) to return a string, any of the numeric types, a specific class of object variable (AS MyClass), a [Variant](#), or a [user defined Type](#).

Type Libraries only support the following data types: [BYTE](#), [WORD](#), [DWORD](#), [INTEGER](#), [LONG](#), [QUAD](#), [SINGLE](#), [DOUBLE](#), [CURRENCY](#), [OBJECT](#), [STRING](#), and [VARIANT](#). If any Methods or Properties use data types not supported by Type Libraries, you will receive a [Error 581 - Type Library creation error](#), when using the [#COM TLIB ON](#) metastatement.

In addition to the explicit return value which you declare, all [COM](#) Methods and Properties have another "Hidden Return Value", which is cryptically named [hResult](#). While the name would imply a handle for a result, it's really not a handle at all, but just a [long integer](#) value, used to indicate success or failure of the Method. After calling a Method or Property, you can retrieve the hResult value with the Classic PowerBASIC function [OBJRESULT](#). The most significant bit of the value is known as the severity bit. That bit is 0 (value is positive) for success, or 1 (value is negative) for failure. The remaining bits are used to convey error codes and additional status information. If you call any object Method/Property (either [Dispatch](#) or [Direct](#)), and the severity bit in the returned hResult is set, Classic PowerBASIC generates Run-Time [error 99](#): Object error. When you create a Method or Property, Classic PowerBASIC automatically returns an hResult of zero, which implies success. You can return a non-zero hResult value by executing a `METHOD OBJRESULT = expr` within a Method, or `PROPERTY OBJRESULT = expr` within a Property.

Every method and property in a [dual interface](#) needs a positive, long integer value to identify it. That integer value is known as a *DispID* (Dispatch ID), and it's used internally by COM services to call the correct function on a Dispatch interface. You can optionally specify particular

DispID by enclosing it in angle brackets immediately following the Method/Property name:

```
METHOD MethodOne <76> ()
```

If you don't specify a *DispID*, Classic PowerBASIC will assign a random value for you. This is fine for internal objects, but may cause a failure for published [COM objects](#), as the *DispID* could change each time you compile your program. It is particularly important that you specify a *DispID* for each Method/Property in a COM [Event Interface](#).

Override Properties

You can add to, or replace, the functionality of a particular method or property of an [inherited base class](#) by coding a replacement which is preceded by the word OVERRIDE. The overriding method must have the same name and signature (parameters, return value, etc.) as the one it replaces.

BYREF and BYVAL parameters

- BYVAL A copy of the data value is placed on the [stack](#) as a parameter. The copy is destroyed when the PROPERTY ends. BYVAL parameters default to an IN attribute, if no explicit direction is specified.
- BYREF A pointer to the data is placed on the stack as a parameter. This option may not be used with an internal PROPERTY parameter.

Direction attributes

PROPERTY parameters also specify the direction in which data is passed between the caller and callee:

- IN Data is passed from the caller to the PROPERTY. Generally speaking, you'll find that almost all IN parameters are passed BYVAL, and that is highly recommended. However, it is possible to pass them BYREF if necessary.
- OUT Data is passed from the PROPERTY back to the caller. All OUT parameters must be passed BYREF.
- INOUT Data is passed from the caller to the PROPERTY, and results are returned to the caller in the same parameter. All INOUT parameters must be passed BYREF.

In many cases, the direction of a parameter can be inferred directly from the BYVAL/BYREF attribute (BYVAL=IN, BYREF=OUT). However, we recommend that you include the direction attribute as an added means of self-documentation. Each METHOD parameter name may be preceded by

one of BYVAL/BYREF, and one of IN/OUT/INOUT, in any sequence.

You should note an interesting rule of COM objects: **IN parameters are read-only. They may not be altered.**

IN parameters are considered by COM rules to be "constant" which may not be altered, because they are values which are not returned to the caller. However, since this is not a rule normally applied to a standard [SUB](#) or [FUNCTION](#), it can allow programming bugs which are most difficult to find and correct. For this reason, Classic PowerBASIC automatically protects you from this issue with no action needed on your part. When writing METHOD or PROPERTY code in Classic PowerBASIC, you may freely assign new values to BYVAL/IN parameters. They will simply be discarded when the METHOD exits. Of course, not every programming language protects you in this way, so you must use caution if you create a COM METHOD in another compiler.

Using OPTIONAL/OPT

PROPERTY statements may specify one or more parameters as optional by preceding the parameter with either the keyword OPTIONAL (or the abbreviation OPT). When a parameter is declared optional, all subsequent parameters in the declaration are optional as well, whether or not they specify an explicit OPTIONAL or OPT directive.

VARIANT variables are particularly well suited for use as an optional parameter. If the calling code omits an optional VARIANT parameter, (BYVAL or BYREF), Classic PowerBASIC (and most other compilers) substitute a variant of type %VT_ERROR which contains an error value of %DISP_E_PARAMNOTFOUND (&H80020004). In this case, you can check for this value directly, or use the [ISMISSING](#) function to determine whether the parameter was physically passed or not.

When optional parameters (other than VARIANT) are omitted from the calling code, the stack area normally reserved for those parameters is zero-filled.

If the parameter is defined as a BYVAL parameter, it will have the value zero. For [TYPE](#) or [UNION](#) variables passed BYVAL, the compiler will pass a string of binary zeroes of length [SIZEOF](#)(*Type_or_union_var*).

If the parameter is defined as a BYREF parameter, [VARPTR](#)(*Varname*) will equal zero; when this is true, any attempt to use *Varname* in your code will result in a General Protection Fault or memory corruption. You should use the ISMISSING() function first to determine whether it is safe to access the parameter.

See also

[#COM](#), [CLASS](#), [INSTANCE](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [ISINTERFACE](#), [ISNOTHING](#), [ISMISSING](#), [ISOBJECT](#), [Just what is](#)

[COM?](#), [LET \(with Objects\)](#), [ME](#), [METHOD](#), [OBJECTIVE](#), [OBJPTR](#),
[OBJRESULT](#), [What is an object, anyway?](#)

Purpose	Write a record to a random-access file or a variable to a binary file .
Syntax	<p><i>Random-Access and Binary files:</i></p> <pre>PUT [#] fNum&, [RecPos], [ABS] VarName</pre> <p><i>Binary files:</i></p> <pre>PUT [#] fNum&, [RecPos], Arr()</pre>
Remarks	The PUT statement has the following parts:
<i>fNum&</i>	A numeric literal or variable containing the file number used in the OPEN statement to open the file. The Number symbol (#) before <i>fNum&</i> is optional, but recommended for source code clarity.
<i>RecPos</i>	<p>Identifies the position in the file to write the data. If <i>RecPos</i> is greater than the number of existing records or bytes in the file, the file is extended to the appropriate length, and the record is written at the specified position.</p> <p>For random access files, <i>RecPos</i> is the record to be written, in the range 1 to $(2^{63})-1$. If <i>RecPos</i> is omitted, the next record in sequence (following the one specified by the most recent GET, PUT or SEEK) is written. If the file was only just opened, the first record is written.</p> <p>For binary files, <i>RecPos</i> is the starting byte position where <i>VarName</i> should be written. The default byte position is 1, unless the <code>BASE = 0</code> clause was used in the OPEN statement. <i>RecPos</i> may be no larger than $2^{63}-1$. <i>RecPos</i> is optional. If it is omitted, Classic PowerBASIC uses the current file pointer position.</p>
<i>VarName</i>	<p>The name of a variable to write to the file. <i>VarName</i> can specify a simple variable, an element in an array, or a variable of User-Defined Type (UDT).</p> <p>When writing a dynamic string to a random access file, PUT writes a 2-byte descriptor containing the string's length, before the actual string data. This descriptor reduces the available space in a record by two bytes. The descriptor is written with the low-order byte (<i>record length</i> MOD 256) occurring before the high-order byte (<i>record length</i> \ 256). If <i>VarName</i> contains more characters than record, <i>VarName</i> is truncated at <i>record length</i> less two bytes, and the descriptor is written to reflect the truncated string size.</p> <p>When writing a dynamic string to a binary file, PUT only writes the actual string data: no length descriptor is written.</p> <p>PUT is complementary to GET; it writes one record to a file. It is possible to PUT to records out of contiguous order, as in:</p> <pre>PUT #1, 1, MyVar PUT #1, 100, MyVar</pre>

which creates a random-access file 100 records long. The data in records 2 through 99, however, are undefined until you explicitly PUT something there. PUT writes the contents of *VarName* to the specified record or byte positions.

(no *VarName*) When the second form of PUT is used (without a *VarName* source string), PUT writes the data from an internal buffer into the file at the point where the file pointer indicates. This data must first be assigned to the file buffer using [FIELD string](#) variables.

ABS When PUT is used to write a dynamic string to a random file, it normally precedes the actual data with a two-byte binary length [Word](#) to define the number of valid bytes in the record. If you precede the variable name with ABS (i.e., PUT #1, , ABS x\$), no length Word is written: only the actual data, subject to the defined random record length. This offers greater compatibility with the actual operation of other versions of BASIC, such as [Classic PowerBASIC for DOS](#).

The record length in a random access file is limited to 32768 bytes, in order to ensure consistent behavior across all Win32 platforms.

Arr() When PUT is used on a binary file, the entire [array](#) specified by *Arr()* is written to the file. With dynamic strings, the file is written in the Classic PowerBASIC and/or VB packed string format. If the string is shorter than 65535 bytes, a 2-byte length Word is followed by the string data. Otherwise, a 2-byte value of 65535 is followed by a length [Double-word](#) (DWORD), then finally the string data.

With other data types, the entire data area is written as a single block. In either case, it is presumed the file will be read with the complementary GET Array statement.

See also [CSET](#), [CSET\\$](#), [FIELD](#), [GET](#), [LOF](#), [LSET](#), [PUT\\$](#), [RSET](#), [SETEOF](#), [TYPE](#), [WRITE#](#)

Example

```
' Random-access PUT example
TYPE TestRec
    uName    AS STRING * 10
    uNumber  AS INTEGER
END TYPE

DIM Rec AS TestRec, Record AS QUAD

OPEN "RANDOM.DTA" FOR RANDOM AS #1 LEN = LEN(TestRec)

FOR Record = 1 TO 100
    Rec.uName = "Joe" + STR$(Record)
    Rec.uNumber = Record
    PUT #1,Record, Rec
NEXT Record
```

```
CLOSE #1
```

```
' Binary PUT Array example
```

```
DIM TheData$(1 TO count&)
```

```
TheData$(1) = "text"
```

```
' Assign more array values...
```

```
OPEN "Data file to write.dat" FOR BINARY AS #1
```

```
PUT #1, 1, TheData$()
```

```
CLOSE #1
```

PUT\$ statement

[Top](#) [Previous](#) [Next](#)

Purpose	Write a string to a file opened in binary mode.
Syntax	<code>PUT\$ [#] <i>filenum</i>&, <i>string_expression</i></code>
Remarks	<p>PUT\$ writes the contents of <i>string_expression</i> to file <i>filenum</i>& at the file's current file pointer position. File <i>filenum</i>& must have been opened in binary mode.</p> <p>If the file pointer position is at the end of the file, PUT\$ appends (adds) <i>string_expression</i> to the file, increasing its length by <code>LEN(<i>string_expression</i>)</code> bytes. If the file pointer is before the end of the file, PUT\$ overwrites existing data with <i>string_expression</i>. In either case, the file pointer position following a PUT\$ is at the end of the just-written string. You can use SEEK to retrieve or change the file pointer position.</p>
See also	GET , GET\$, OPEN , PUT , SEEK function , SEEK statement , SETEOF , WRITE#
Example	<pre>' Open a binary file and write the alphabet to it OPEN "SEEK.DTA" FOR BINARY AS #1 BASE = 1 FOR I& = ASC("A") TO ASC("Z") ' 65 TO 90 PUT\$ #1, CHR\$(I&) NEXT CLOSE #1</pre>

Purpose Call [Event](#) Handler code.

Syntax `RAISEEVENT ObjVar.Method()`

Remarks The RAISEEVENT statement is used to call event handler code from an Event Source. RAISEEVENT may only appear within a [class](#) which declares the [Event Source](#) interface. The concept of RAISEEVENT is very similar to the [CALL](#) statement, but it may only be used to execute event procedures:

```
RaiseEvent Status.Progress(10) ' advise the code is 10% done
```

It should be noted that RAISEEVENT does not reference an [object](#) variable at all, because it calls any and all Direct, V-Table event handlers which are currently subscribed to these events. Instead, it references the interface name (in this case "Status"), followed by the name of the Event Method to be executed (in this case "Progress"). If your program is using a Dispatch event handler you should use the [OBJECT RAISEEVENT](#) statement instead.

See also [CLASS](#), [EVENT SOURCE](#), [EVENTS](#), [INTERFACE \(Direct\)](#), [INTERFACE \(IDBind\)](#), [Just what is COM?](#), [EVENTS](#), [OBJECT RAISEEVENT](#), [What is an object, anyway?](#), [What are Connection Points?](#)

Example See the [EVENT SOURCE](#) statement for an example of RAISEEVENT.

RANDOMIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose Seed the random number generator.

Syntax `RANDOMIZE [number]`

Remarks *number* is a seed value that may be any numeric type. If *number* is not specified, the value returned by the [TIMER](#) function is used.

Values returned by the random number generator ([RND](#)) depend on an initial seed value. For a given seed value, RND always returns the same sequence of values, yielding a predictable pseudo-random number sequence. Thus, any program that depends on RND will run exactly the same way each time unless a different seed is given.

The default seed can be duplicated with the following statement:

```
RANDOMIZE CVS (CHR$ (255,255,255,255))
```

Note that each [thread](#) has its own, independent random number seed.

See also [RND](#), [TIMER](#)

Example

```
' Seed generator and get 5 random values
RANDOMIZE 1.5!
FOR I& = 1 TO 5
    Table(I&) = RND(1,100)
NEXT I&

' Reseeding with the same starting value
' means you get the same sequence of values!
RANDOMIZE 1.5!
FOR I& = 1 TO 5
    Table(I&) = RND(1,100)
NEXT I&

' Now reseed from the TIMER and we get
' a completely different set of values:
RANDOMIZE TIMER
FOR I& = 1 TO 5
    Table(I&) = RND(1,100)
NEXT I&
```

READ\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve string data from a local DATA list.
Syntax	<code>value\$ = READ\$(n%)</code>
Remarks	<p>The READ\$ function is used to retrieve a specified string data item from a local DATA list, and returns the data in string format. READ\$ offers a simple technique for random-access of local DATA.</p> <p><i>n%</i> An integral expression, constant, or variable, which specifies an index position in the local DATA list. <i>n%</i> = 1 for the first data item, <i>n%</i> = 2 for the second, and so on. READ\$ accesses the DATA statements in the order in which they appear in the source program, from left to right.</p> <p>If <i>n%</i> is greater than DATACOUNT, READ\$ returns an empty string, but no run-time error occurs.</p> <p><i>value\$</i> READ\$ places the DATA string into <i>value\$</i>.</p> <p>If the target DATA statement is enclosed in quotes, READ\$ preserves any leading or trailing spaces that it may contain; otherwise, READ\$ trims leading and trailing spaces and returns a trimmed string. See DATA for more information on data item formatting.</p> <p>If numeric data needs to be stored in DATA statements and retrieved with READ\$, the VAL function can be used to convert the return values from READ\$ into numeric values.</p>
Restrictions	There is a limit of 64 Kilobytes and 16384 separate data items per Sub , Function , Method , or Property , and it is not possible to read DATA from outside of the scope of current procedure. Restrictions apply to using colon and underscore characters in DATA statements - see DATA for more information.
See also	DATA , DATACOUNT
Example	<pre>' The following returns the day of the week string. FUNCTION WeekDayName\$(BYVAL DayNum%) IF DayNum% < 1 OR DayNum% > DATACOUNT THEN WeekDayName\$ = "" ELSE WeekDayName\$ = READ\$(DayNum%) END IF DATA Sun, Mon, Tue, Wed, Thu, Fri, Sat END FUNCTION</pre>

Purpose	Used at the procedure level to declare dynamic array variables and allocate, deallocate, or reallocate storage space.
Syntax	<code>REDIM [PRESERVE] array[(subscripts)] [AS type] [AT address] [, ...]</code>
Remarks	<p>The REDIM statement allows dynamic arrays (including string arrays) to be erased and re-dimensioned. It is really just a shortcut for the two-step process ERASE x(), followed by DIM x(). REDIM uses the same basic syntax as the DIM statement.</p> <p>array is the name of the array, and subscripts is either a group of single integers (one per dimension of a particular array), or a group of ranges (REDIM arr1(5 TO 25, 1 TO 4, 3 TO 8)), separated by commas.</p> <p>Use the TO keyword instead of the colon (:) syntax, as the colon syntax may not be supported in future versions of Classic PowerBASIC.</p>
AS type	The AS type clause is optional, but recommended for the purposes of clarity.
AT address	The AT address clause indicates the array is to be an absolute array. Absolute arrays are not reset by the REDIM statement, nor are they reset when the Sub/Function/Method/Property exits, but they can be reset with the RESET statement. See the discussion in the DIM topic for more information on absolute arrays.
PRESERVE	<p>The PRESERVE keyword tells the compiler to preserve the values of all existing elements in the array. For example, if you REDIM PRESERVE an array with 10 elements to 20 elements, the first 10 elements will retain their original value. The remaining 10 elements will be initialized to zero (or null/empty in the case of a string array). If the array is resized to be smaller, the specified number of elements is preserved, and the remaining elements are discarded. When PRESERVE is specified, you can resize only the upper boundary of the last (outer) dimension of the array. Arrays of only one dimension can always be resized.</p> <p>In a procedure, you can use REDIM to re-dimension an array that was passed as an argument. That is, when the complete array was passed to the procedure:</p> <pre>CALL RemoveDuplicates(CustomerNames\$()) ' more code here SUB RemoveDuplicates(a\$()) ' Remove duplicate array values REDIM PRESERVE a\$(1 TO NewCount&) END SUB</pre>

REDIM may also be used to alter the size of [Static](#), [Global](#), and [Instance](#) arrays.

When used with no subscript parameters, REDIM will erase all contents of an array and deallocate the memory used:

```
REDIM xyz&() ' Equivalent to ERASE xyz&()
```

Restrictions When **PRESERVE** is specified, *only the upper bound of the last (outer) dimension may be redefined*.

When a REDIM statement is executed, the location of the array elements always moves in memory; however, the array's *Descriptor* location ([VARPTR](#)(arrayname())) will remain fixed at the original location. When using REDIM, your code must be sure to refresh any pointers that target the array data memory locations ([STRPTR](#)(arrayname(subscript)) for [dynamic string](#) arrays, and [VARPTR](#)(arrayname(subscript)) for all other array types).

While Classic PowerBASIC supports lower boundary values that are non-zero, Classic PowerBASIC generates the most efficient code if the lower boundary parameter is omitted (i.e., the array uses the default lower boundary of zero).

See also [ARRAYATTR](#), [DIM](#), [ERASE](#), [RESET](#)

Example

```
DIM MyData(40), Names$(100)
REDIM MyData(5 TO 50), Names$(10)
```

Purpose	Scan a string for a matching "wildcard" or regular expression.
Syntax	<code>REGEXPR mask\$ IN target\$ [AT start&] TO iPos& [, iLen&]</code>
Remarks	<p>REGEXPR scans <i>target\$</i> for a matching expression specified in <i>mask\$</i>. If found, it returns the position of the match in the <i>iPos&</i> variable (indexed to the first character position), and optionally, the length of the matching expression in <i>iLen&</i>.</p> <p>If a match is made, the <i>iPos&</i> and <i>iLen&</i> results can be immediately used with subsequent string operations such as MID\$ to extract the matched portion of <i>target\$</i>, and/or to continue the search through the remainder of <i>target\$</i>. If no matching expression is found, both <i>iPos&</i> (and <i>iLen&</i> if specified) are set to zero.</p> <p>If specified, the search begins at the character position <i>target&</i> in <i>target\$</i>; however, <i>start&</i> must be between 1 and the length of <i>target\$</i>. If <i>start&</i> is less than 1, the <i>start&</i> parameter is ignored.</p> <p>While it is possible for more than one match to be found in a particular target string, REGEXPR first selects one or more matches which start at the leftmost possible position, then returns the longest of those. Use the <code>\s</code> special escape operator to force a match on the shortest match (see below).</p> <p>The <code>^</code> and <code>\$</code> operators match on both the actual string start/end, or the previous/next embedded line-delimiter characters (CHR\$(13,10) or \$CRLF) in <i>target\$</i>. This enables REGEXPR to treat the <i>target\$</i> string as containing a set of "logical lines" of text. In this situation, the <i>start&</i> character position plays a crucial role in identifying which logical delimited line that should be examined by REGEXPR.</p> <p>By default, search expressions are assumed to be case-insensitive, so capitalization is ignored.</p> <p><i>mask\$</i></p> <p>The regular (<i>wildcard</i>) expression specified in <i>mask\$</i> may contain a combination of standard text characters and/or the <i>metacharacters</i> which are defined as follows:</p>

Char	Definition
.	(period) Matches any character, <i>except</i> the end-of-line.
^	(caret) Matches the actual beginning-of-line position or the preceding line-delimiter character pair (CHR\$(13,10) or \$CRLF), as taken from the <i>start&</i> character position. The line-delimiter characters themselves are not included in the <i>iLen&</i> result. (also

	see [^] below for usage within a character class definition).
\$	(dollar) Matches the end-of-line position, which may be either the first line-delimiter character pair (CHR\$(13,10) or \$CRLF) that is encountered in the search to the right of the <i>start</i> & position, or the actual end of the <i>target</i> \$ string, whichever occurs first. The line-delimiter characters themselves are not included in the <i>iLen</i> & result.
	(stile) Specifies alternation (the OR operator), so that an expression on either side can match. Precedence is from left-to-right, as encountered in the expression.
?	(question mark) Specifies that zero or one match of the preceding sub-pattern is allowed. Cannot be used with a Tag.
+	(plus) Specifies that one or more matches of the preceding sub-pattern are allowed. Cannot be used with a Tag.
*	(asterisk) Specifies that zero or more matches of the preceding sub-pattern are allowed. Cannot be used with a Tag.

Character classes

[]	(square brackets) Identifies a user-defined class of characters, any of which will match: [abc] will match a, b, or c. Only three special metacharacters are recognized within a class definition, the caret ^ for complemented characters, the hyphen - for a range of characters, or one of the following \ backslash escape sequences:
	\\ \- \] \e \f \n \q \r \t \v \x##
	Any other use of a backslash within a class definition yields an undefined operation that should be avoided.
[-]	(hyphen) The hyphen identifies a range of characters to match. For example, [a-f] will match a, b, c, d, e, or f.
	Characters in an individual range must occur in the natural order as they appear in the character set. For example, [f-a] will match nothing.
	Lists of characters, and one or more ranges of characters, may be intermixed in a single class definition. The start and end of a range may be specified by a literal character, or one of the \ backslash escape sequences:
	\\ \- \] \e \f \n \q \r \t \v \x##
	Any other use of a backslash within a class definition yields an

	undefined operation.
	Multiple ranges in a class are valid. For example, [a-d2-5] matches a, b, c, d, 2, 3, 4, or 5.
	When the hyphen is escaped, it is treated as a literal. For example, [a\-c] is a list, not a range, and matches a, -, or c due to the \ backslash escape sequence.
[^]	(caret) When the caret appears as the first item in a class definition, it identifies a complemented class of characters, which will not match. For example, [^abc] matches any character <i>except</i> a, b, or c.
	A range can also be specified for the complemented class. For example, [^a-z] matches any character except a through z.
	A caret located in any position other than the first is treated as a literal character.

Tags/sub-patterns

()	(parentheses) Parentheses are used to match a Tag, or sub-pattern, within the full search pattern, and remember the match. The matched sub-pattern can be retrieved later in the mask (or in a replace operation with REGREPL), with \01 through \99, based upon the left-to-right position of the opening parentheses.
	Parentheses may also be used to force precedence of evaluation with the alternation operator. For example, "(Begin) (End)File" would match either "BeginFile" or "EndFile", but without the Tag designations, "Begin EndFile" would only match either "BeginndFile" or "BegiEndFile".

Escaped characters

\	(backslash). The escape operator (single-character quote). The following character will be treated as a literal value rather than being interpreted as a special character. Note that the character following the backslash must actually be a special character, as follows:
\b	A word boundary. The start or end of a word, where a word is defined as one or more characters that include an alphabetic character (A-Z or a-z), a numeric character (0-9), and an underscore. For example, "abc_123" is considered a single word and "abc-123" is considered two words.
\c	Case-sensitive search. Without the \c operator, the default is to ignore case when matching. Unlike some other implementations of regular expressions, case-insensitivity is recognized in all

	operations, even a range of characters such as "[6-Z]". The \c operator may appear at any position in the mask.
\e	Escape character: CHR\$(27) or \$ESC .
\f	Formfeed character: CHR\$(12) or \$FF .
\n	Linefeed (or new-line) character: CHR\$(10) or \$LF .
\q	Double-quote mark ("): CHR\$(34) or \$DQ . \q is included for ease of inclusion within a literal string. For example: "\qHello\q".
\r	Carriage-return character: CHR\$(13) or \$CR .
\s	Shortest match character: The \s flag causes the shortest matching string to be returned, rather than the longest (the default). For example, when searching for the mask "abc.*abc" in "abcdabcabc", the default setting would return position 1 and length 10. With the \s switch set, it returns position 1 and length 7. This option may cause a slight increase in processing time. The \s flag must appear at the beginning of the mask string.
\t	Horizontal tab character: CHR\$(9) or \$TAB .
\v	Vertical tab character: CHR\$(11) or \$VT .
\x##	Hex character code: Indicates that an ASCII code follows, given by two hexadecimal digits. For example, \xFF = CHR\$(&HFF) (which is equivalent to CHR\$(255)). XX must be in the range 0 through 255.

Restrictions To maximize performance, avoid overuse of the *, + and ? metacharacters.

See also [REGREPL](#), [Online Regular Expression Tester](#)

Example

```
a$ = "please send email to support@Classic PowerBASIC.com"
b$ = "([a-z0-9._/+ -] +) ([a-z0-9.-] +)"
REGEXPR b$ IN a$ TO position&, length&
email_address$ = MID$(a$, position&, length&)
```

```
a$ = "Amount owed: $42.75 and is overdue!"
b$ = "\$[0-9.,] +"
REGEXPR b$ IN a$ TO position&, length&
amount$ = MID$(a$, position&, length&)
```

```
a$ = "Open 24 Hours"
b$ = "[^a-z ] +"
REGEXPR b$ IN a$ TO position&, length&
hours$ = MID$(a$, position&, length&)
```

```
a$ = "Line 1" + $CRLF + "Line 2" + $CRLF
b$ = "([0-9])$"
RESET position& : RESET length&
DO
    position& = position& + length&
    REGEXPR b$ IN a$ AT position& TO _
        position&, length&
```

```
    c$ = "Match at " + STR$(position&)  
LOOP WHILE position&
```

Purpose	To define Register variables, which are local to a Sub , Function , Method , or Property . The REGISTER statement provides an optimization hint to the compiler.
Syntax	<code>REGISTER <i>variable</i> [AS <i>type</i>] [, <i>variable</i> [AS <i>type</i>]]</code>
Remarks	<p>The REGISTER statement is used to define certain local variables as Register variables - that is, variables which are stored directly in specific CPU registers, rather than in application memory. Since data in a CPU register can be accessed much faster, and with less code, Register variables are valuable optimization tools.</p> <p>Register variables are always local to the procedure where they appear. In the current version of Classic PowerBASIC, there may be up to two integer-class variables (Word/Dword/Integer/Long) and up to four Extended-precision floats. It is possible that future versions of the compiler will change these limits, so you may declare an unlimited number of them. Any "extra" Register variables are automatically reclassified as locals during compilation.</p> <p>The REGISTER statement allows you to choose which variables will be classified as Register variables. If you do not make the choice in a particular procedure, the compiler will attempt to choose for you. By default, the compiler will always assign any integer-class local variables available. Extended-precision float variables will be automatically assigned only in Functions that contain no external Function calls.</p> <p>Integer class Register variables are most efficient for variables that are updated or used often, such as For/Next loop counter variables, and variables that are used repeatedly as array indexes.</p> <p>Floating-point Register variables should generally be chosen with a bit more caution, since the compiler must generate code to save and restore them to conventional memory around each call to a procedure. In some rather rare cases, it is possible that floating-point Register variables could actually reduce execution speed. However, they are extremely valuable with intensive floating-point calculations in Functions that have few references to other procedures.</p> <p>Due to the design of FPUs (floating point units), and the instruction sets available, the first float register variable declared in your program has far more optimization possibilities than the others do. Use care in choosing the variable which is used most within floating-point expressions (that is, on the right side of the '=' assignment operator), in order to gain the greatest advantage in execution speed. Also, remember it is typically valuable to</p>

assign floating-point [constants](#) to Register variables when they are used in repetitive or intensive calculations.

You must use care with [Inline Assembler](#) floating-point [opcodes](#) in Functions that enable Register variables. Floating-point Register variables may occupy up to four of the [FPU](#) registers, so you must limit your use of the x87 registers to the remaining four. Further, floating-point Register variables may never be [referenced by name](#) from Inline Assembler code, as the compiler cannot always track the register locations with absolute certainty.

Restrictions [VARPTR](#) cannot be used on a Register variable.

Classic PowerBASIC transparently prevents the *automatic* register conversion of the variable used in the TO clause of the [DIALOG SHOW MODAL](#) and [DIALOG SHOW MODELESS](#) statements. If the target variable is *explicitly* declared as a register variable, Classic PowerBASIC raises a compile-time [Error 491](#) ("Invalid register variable"). This is necessary as the result values stored in such variables may be assigned from the context of other procedures, and this may only occur with a memory variable.

See also [#REGISTER](#), [Optimizing your code](#)

Example

```
SUB ReindexDatabase() AS LONG
  #REGISTER NONE      ' I'll choose my own register vars.
  REGISTER i AS LONG
  REGISTER fVar AS EXT
  ' do something
END FUNCTION
```


Purpose	Scan a string for a matching "wildcard" or regular expression, and replace it with a new value.
Syntax	<code>REGREPL mask\$ IN target\$ WITH repl\$ [AT start&] TO iPos&, newtarget\$</code>
Remarks	<p>REGREPL scans <i>target\$</i> for a matching regular expression specified in <i>mask\$</i>. If a match is made, REGREPL replaces the matched text with the contents of <i>repl\$</i>, and assigns the new text to <i>newtarget\$</i>. Additionally, REGREPL sets <i>iPos&</i> to reflect the character position immediately following the matched text in <i>newtarget\$</i>, so the operation can be repeated, if desired.</p> <p>If no matching expression is found, <i>iPos&</i> will be set to zero, and <i>newtarget\$</i> receives a direct copy of <i>target\$</i>. In either case, <i>target\$</i> remains unchanged.</p> <p><i>mask\$</i> may contain literal characters and <i>metacharacters</i> (wildcards) to form the regular expression, and <i>repl\$</i> may only contain literal characters and tags specified by \##. Each tag from \01 through \99 is replaced by the text actually matched for that tag. \00 is replaced by the entire matched text.</p> <p>If specified, the search begins at the character position <i>start&</i> in <i>target\$</i>; however, <i>start&</i> must be between 1 and the length of <i>target\$</i>. If <i>start&</i> is less than 1, the <i>start&</i> parameter is ignored.</p> <p>While it is possible for more than one match to be found in a particular target string, REGREPL first selects one or more matches which start at the leftmost possible position, then returns the longest of those. Use the \s special escape operator to force a match on the shortest match (see below).</p> <p>The ^ and \$ operators match on both the actual string start/end, or the previous/next embedded line-delimiter characters (CHR\$(13,10) or \$CRLF) in <i>target\$</i>. This enables REGREPL to treat the <i>target\$</i> string as containing a set of "logical lines" of text. In this situation, the <i>start&</i> character position plays a crucial role in identifying which logical delimited line that should be examined by REGREPL.</p> <p>By default, search expressions are assumed to be case-insensitive, so capitalization is ignored.</p>
mask\$	The regular (<i>wildcard</i>) expression specified in <i>mask\$</i> may contain a combination of standard text characters and/or the <i>metacharacters</i> which are defined as follows:

--	--

Char	Definition
.	(period) Matches any character, <i>except</i> the end-of-line.
^	(caret) Matches the actual beginning-of-line position or the preceding line-delimiter character pair (CHR\$(13,10) or \$CRLF), as taken from the <i>start&</i> character position. The line-delimiter characters themselves are not replaced by <i>rep/\$</i> . (also see [^] below for usage within a character class definition).
\$	(dollar) Matches the end-of-line position, which may be the either the first line-delimiter character pair (CHR\$(13,10) or \$CRLF) that is encountered in the search to the right of the <i>start&</i> position, or the actual end of the <i>target\$</i> string, whichever occurs first. The line-delimiter characters themselves are not replaced by <i>rep/\$</i> .
	(stile) Specifies alternation (the OR operator), so that an expression on either side can match. Precedence is from left-to-right, as encountered in the expression.
?	(question mark) Specifies that zero or one match of the preceding sub-pattern is allowed. Cannot be used with a Tag.
+	(plus) Specifies that one or more matches of the preceding sub-pattern are allowed. Cannot be used with a Tag.
*	(asterisk) Specifies that zero or more matches of the preceding sub-pattern are allowed. Cannot be used with a Tag.

Character classes

[]	(square brackets) Identifies a user-defined class of characters, any of which will match: [abc] will match a, b, or c. Only three special metacharacters are recognized within a class definition, the caret (^) for complemented characters, the hyphen (-) for a range of characters, or one of the following \ backslash escape sequences:
	\\ \- \] \e \f \n \q \r \t \v \x##
	Any other use of a backslash within a class definition yields an undefined operation that should be avoided.
[-]	(hyphen) The hyphen identifies a range of characters to match. For example, [a-f] will match a, b, c, d, e, or f.
	Characters in an individual range must occur in the natural order as they appear in the character set. For example, [f-a] will match nothing.
	Lists of characters, and one or more ranges of characters, may

	be intermixed in a single class definition. The start and end of a range may be specified by a literal character, or one of the \ backslash escape sequences:
	<code>\\ \- \] \e \f \n \q \r \t \v \x##</code>
	Any other use of a backslash within a class definition yields an undefined operation.
	Multiple ranges in a class are valid. For example, <code>[a-d2-5]</code> matches a, b, c, d, 2, 3, 4, or 5.
	When the hyphen is escaped, it is treated as a literal. For example, <code>[a\-c]</code> is a list, not a range, and matches a, -, or c due to the \ backslash escape sequence.
<code>[^]</code>	(caret) When the caret appears as the first item in a class definition, it identifies a complemented class of characters, which will not match. For example, <code>[^abc]</code> matches any character <i>except</i> a, b, or c.
	A range can also be specified for the complemented class. For example, <code>[^a-z]</code> matches any character except a through z.
	A caret located in any position other than the first is treated as a literal character.

Tags/sub-patterns

<code>()</code>	(parentheses) Parentheses are used to match a Tag, or sub-pattern, within the full search pattern, and remember the match. The matched sub-pattern can be retrieved later in the mask, or in a replace operation, with <code>\01</code> through <code>\99</code> , based upon the left-to-right position of the opening parentheses.
	Parentheses may also be used to force precedence of evaluation with the alternation operator. For example, <code>"(Begin) (End)File"</code> would match either <code>"BeginFile"</code> or <code>"EndFile"</code> , but without the Tag designations, <code>"Begin EndFile"</code> would only match either <code>"BeginndFile"</code> or <code>"BegEndFile"</code> .
	Note: Parentheses may not be used with <code>?</code> <code>+</code> <code>*</code> as any match repetition could cause the tag value to be ambiguous. To match repeated expressions, use parentheses followed by <code>\01*</code> .

Escaped characters

<code>\</code>	(backslash). The escape operator (single-character quote). The following character will be treated as a literal value rather than being interpreted as a special character. Note that the character following the backslash must actually be a special character, as follows:
----------------	---

\b	A word boundary. The start or end of a word, where a word is defined as one or more characters that include an alphabetic character (A-Z or a-z), a numeric character (0-9), and an underscore. For example, "abc_123" is considered a single word and "abc-123" is considered two words.
\	Case-sensitive search. Without the \c operator, the default is to ignore case when matching. Unlike some other implementations of regular expressions, case-insensitivity is recognized in all operations, even a range of characters such as "[6-Z]". The \c operator may appear at any position in the mask.
\e	Escape character: CHR\$(27) or \$ESC .
\f	Formfeed character: CHR\$(12) or \$FF .
\n	Linefeed (or newline) character: CHR\$(10) or \$LF .
\q	Double-quote mark ("): CHR\$(34) or \$DQ . \q is included for ease of inclusion within a literal string. For example: "\qHello\q".
\r	Carriage-return character: CHR\$(13) or \$CR .
\s	Shortest match character: The \s flag causes the shortest matching string to be returned, rather than the longest (the default). For example, when searching for the mask "abc.*abc" in "abcdabcbabc", the default setting would return position 1 and length 10. With the \s switch set, it returns position 1 and length 7. This option may cause a slight increase in processing time. The \S flag must appear at the beginning of the mask string.
\t	Horizontal tab character: CHR\$(9) or \$TAB .
\v	Vertical tab character: CHR\$(11) or \$VT .
\x##	Hex character code: Indicates that an ASCII code follows, given by two hexadecimal digits. For example, \xFF = CHR\$(&HFF) (which is equivalent to CHR\$(255)). XX must be in the range 0 through 255.
\##	Tag number: Evaluated as the characters matched by tag number ## (where ## is in the range 01 through 99, in decimal). Tags are implicitly numbered from 01 through 99, based upon the left-to-right position of the left parenthesis. "()(w\01" would match "abcwabc" or "456w456".

Tags cannot be forward-referenced - that is, if a reference is made to any Tag that is not yet defined, a non-match is presumed.

Restrictions To maximize performance, avoid overuse of the *, + and ? metacharacters.

See also

[REGEXPR](#)

Example

```
#COMPILE EXE
FUNCTION PBMAIN
  a$ = "please email support@Classic PowerBASIC.com"
  b$ = "([a-z0-9._/+-]+) (@[a-z0-9.-]+) "
  c$ = "sales\02"
  REGREPL b$ IN a$ WITH c$ TO position&, d$
  ' d$ -> "please email sales@Classic PowerBASIC.com"

  a$ = "Line 1" + $CRLF + "Line 2" + $CRLF
  b$ = "([0-9])$"
  c$ = "\01.0"
  position& = 1
  DO
    REGREPL b$ IN a$ WITH c$ AT position& TO position&, a$
  LOOP WHILE position&
  ' a$ -> " Line 1.0" + $CRLF + "Line 2.0" + $CRLF
END FUNCTION
```

Purpose	Indicate that the remainder of a line in source code files is to be regarded as a Remark or Comment, and excluded from the compiled code.
Syntax	<pre>REM <i>comment text</i> ' <i>comment text</i> ; <i>comment in an Inline Assembler statement</i></pre>
Remarks	<p>The Classic PowerBASIC compiler ignores Remarks; they do not take up space in your generated code, so use them abundantly - useful comments greatly increase the readability and maintainability of source code.</p> <p><i>Comment text</i> is any sequence of characters. A comment can appear on a line with other statements, but it must be the last thing on that line, and a colon must precede it. For example, the assignment below will not be compiled or executed because the compiler cannot tell where the comment ends and the statement begins:</p>

```
REM now add the numbers: a = b + c
```

The following works:

```
a = b + c : REM now add the numbers
```

The apostrophe (') is an alternate form of REM. When you use an apostrophe, you do not need a colon to separate the remark from the other statements on the same line.

When using the [Inline Assembler](#), use the semi-colon (;) to indicate that the remainder of the line should be ignored. An apostrophe (') can still be used for comments, however.

In addition, the compiler treats text that appears after the line continuation character as a remark. However, we still recommend that such comments are preceded by a REM or an apostrophe (') symbol to clearly distinguish remarks from the actual code. For example:

```
DECLARE FUNCTION Call32& _      The prototype
LIB "CALL32.DLL" _             The DLL name
ALIAS "Call32" _               ' The exported name
(Param1 AS ANY, _              ' 1st parameter
BYVAL id&) ' 2nd parameter
```

For situations where a large section of code needs to be REMmed out (yet preserved within the source code file), it is often easier to enclose the code with [#IF 0/#ENDIF](#) metastatements. For example:

```
#IF 0                          ' Exclude the following lines
Code and text in between the #IF 0 and #ENDIF
metastatements is ignored by the compiler.
DIM a$(1 TO 1000) ' This line is ignored too.
INCR x&           ' As is this line!
#ENDIF
```

Since the #IF expression evaluates to false (zero), this forces the compiler to exclude the enclosed block of code from the compilation process, in

exactly the same way as if a REM statement had been prefixed to each line.

See also

[Long Lines](#)

Example

```
x% = 10           : REM This is a comment
y% = 20           ' This is another form of comment
! MOV EAX,"ABCD" ; An Inline Assembler comment
```

REMAIN\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose	Return the portion of a string following the first occurrence of a specified character or string.
Syntax	<code>a\$ = REMAIN\$([<i>position</i>&], <i>main</i>\$, [ANY] <i>match</i>\$)</code>
Remarks	REMAIN\$ is a complement to the EXTRACT\$ function. REMAIN\$ uses the following parameters:
<i>main</i> \$	The <i>main</i> \$ string is searched for the character or string specified in <i>match</i> \$. If found, all characters after <i>match</i> \$ are returned in <i>a</i> \$. If <i>match</i> \$ is not found in <i>main</i> \$ then an empty string is returned in <i>a</i> \$.
<i>position</i> &	An optional starting position within <i>main</i> \$. If not specified, the first position is used.
<i>match</i> \$	The search string. If ANY is specified with <i>match</i> \$, REMAIN\$ will match the first instance of any single character in <i>match</i> \$, not the whole string.
Restrictions	If <i>position</i> & evaluates to a position outside of the string on either side, or if <i>position</i> & is zero, an empty string is returned in <i>a</i> \$.
See also	EXTRACT\$, LEFT\$, LTRIM\$, MID\$, REMOVES\$, REPLACE , RIGHT\$, RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>a\$ = REMAIN\$("I think, therefore I am hungry", ",") Result " therefore I am hungry"</pre>

REMOVE\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose	Return a copy of a string with characters or strings removed.
Syntax	<code>x\$ = REMOVE\$(MainString, [ANY] MatchString)</code>
Remarks	The REMOVE\$ function has the following parts:
<i>MainString</i>	The string expression from which to remove characters.
<i>MatchString</i>	The string expression to remove all occurrences of. If <i>MatchString</i> is not present in <i>MainString</i> , all of <i>MainString</i> is returned intact.
ANY	If the ANY keyword is included, <i>MatchString</i> specifies a list of single characters to be searched for individually, a match on any one of which will cause that character to be removed from the result.
Restrictions	REMOVE is case-sensitive.
See also	EXTRACT\$, INSTR , LTRIM\$, MID\$, REPLACE , RETAIN\$, RIGHT\$, RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>' The following returns "aadabra", ' removing the string "bac" x\$ = REMOVE\$("abacadabra", "bac") ' The following returns "dr", ' removing all "b", "a", and "c" x\$ = REMOVE\$("abacadabra", ANY "bac")</pre>

REPEAT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose Return a string consisting of multiple copies of the specified string.

Syntax `s$ = REPEAT$(count&, string_expr)`

Remarks The REPEAT\$ function has the following parts:

count& Is an integral expression, [constant](#) or [variable](#), specifying the number of copies of *string_expr* to be included in the result. REPEAT\$ is very similar to [STRING\\$](#) (which makes multiple copies of a single character).

string_expr The string to be duplicated.

See also [BUILD\\$](#), [CHR\\$](#), [GUID\\$](#), [NUL\\$](#), [SPACE\\$](#), [STRING\\$](#)

Example `x$ = REPEAT$(5, "<*> ")`

Result `<*> <*> <*> <*> <*>`

REPLACE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Within a specified string, replace all occurrences of one string with another string.
Syntax	<code>REPLACE [ANY] <i>MatchString</i> WITH <i>NewString</i> IN <i>MainString</i></code>
Remarks	<p>The REPLACE statement replaces all occurrences of <i>MatchString</i> in <i>MainString</i> with <i>NewString</i>. The replacement can cause <i>MainString</i> to grow or shrink in size. <i>MainString</i> must be a string variable; <i>MatchString</i> and <i>NewString</i> may be string expressions. REPLACE is case-sensitive. When a match is found, the scan for the next match begins at the position immediately following the prior match.</p>
ANY	<p>If you use the ANY option, within <i>MainString</i>, each occurrence of each character in <i>MatchString</i> will be replaced with the corresponding character in <i>NewString</i>. <u>In this case, <i>MatchString</i> and <i>NewString</i> must be the same length</u>, because there is a one-to-one correspondence between their characters.</p>
See also	EXTRACT\$, INSTR , LTRIM\$, MID\$, REMOVES\$, RETAINS\$, RIGHT\$, RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>A\$ = "abacadabra" 'now replace "bac" with "----bac----" REPLACE "bac" WITH "----bac----" IN A\$ A\$ = "abacadabra" 'now replace all "b", "a", and "c" with "*" REPLACE ANY "bac" WITH "***" IN A\$</pre>

RESET statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set a scalar (non-array) variable , Variant , User-Defined Type , individual array element (or an entire array) to zero or null/empty. RESET does not deallocate the actual memory used (with the exception of dynamic string array data, which is automatically deallocated).
Syntax	<pre>RESET <i>variable</i> [, ...] RESET <i>array</i>() [, ...] RESET <i>array</i>(<i>index</i>) [, ...]</pre>
Remarks	<p>If <i>variable</i> is numeric, it is set to zero. If <i>variable</i> is a dynamic string, it is set to null (""; an empty string). If <i>variable</i> is an ASCIIZ string, the first byte is set to nul (\$NUL). If <i>variable</i> is a fixed-length string or User-Defined Type/Union, all bytes in <i>variable</i> are set to nul, or CHR\$(0). If <i>variable</i> is a Variant, it is cleared and set to data type %VT_EMPTY.</p> <p>If <i>array</i>() is numeric, all elements are set to zero; otherwise all elements are set to zero/null. If an array index value is specified within the parentheses, just that array element is set to zero/null, as if it were a scalar (non-array) variable.</p> <p>RESET also works with <i>absolute arrays</i>, clearing the contents to zeroes or empty strings. For more information on absolute arrays, please refer to the DIM statement.</p>
See also	ARRAYATTR , DIM , ERASE , LET , LET (with Types) , LET (with Variants) , REDIM

RESUME statement

[Top](#) [Previous](#) [Next](#)

Purpose	Restart program execution after error handling with ON ERROR GOTO .
Syntax	<code>RESUME [<i>label</i> NEXT]</code>
Remarks	<p>The RESUME statement is used to continue execution of a program after a run-time error has been trapped and processed with an ON ERROR handler. If a label is specified, execution continues at the specified label. If NEXT is specified, execution continues on the line after the error occurred. If you do not specify a label or NEXT, execution continues where the error occurred, so make sure you've fixed the cause of the error first!</p>
Restrictions	ON ERROR and RESUME may not be used within a TRY/END TRY block.
See also	ERL , ERR , ERROR , Error Overview , ERROR\$, Error Trapping , ON ERROR
Example	See the examples in Error Trapping .

RETAIN\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string containing only the characters contained in a specified match string. All other characters are removed..
Syntax	<code>sResult\$ = RETAIN\$(main\$, [ANY] match\$)</code>
Remarks	RETAIN\$ returns a string consisting of zero or more copies of the complete expression <i>match\$</i> which are found in <i>main\$</i> . All other characters are removed.
ANY	If the ANY option is included, <i>match\$</i> specifies a list of single characters to be retained, if they are found in <i>main\$</i> .
Restrictions	If <i>match\$</i> is an empty string, RETAIN\$ returns an empty string.
See also	EXTRACT\$, REMAINS\$, REMOVES\$, REPLACE
Example	<pre>a\$ = "<p>1234567890<ak;l;l>1234567890</p>" b\$ = RETAIN\$(a\$, ANY "<:/p>") c\$ = RETAIN\$(a\$, ANY "0123456789")</pre>
Result	<pre>b\$ contains "<p><;;></p>" c\$ contains "12345678901234567890"</pre>

RETURN statement

[Top](#) [Previous](#) [Next](#)

Purpose	Return from a (GOSUB) subroutine to its caller.
Syntax	<code>RETURN</code>
Remarks	<p>RETURN terminates the execution of a subroutine, and passes control to the statement directly following the calling GOSUB statement.</p> <p>Performing a RETURN without a corresponding GOSUB can cause unexpected behavior and difficult-to-track errors, including the possibility of a General Protection Fault (GPF) in Windows.</p>
See also	CALL , GOSUB , GOTO , ON ERROR , SUB/END SUB
Example	See the example in GOSUB .

Purpose	Create an RGB color value from 3 primary color values or from a BGR value.
Syntax	<pre>result& = RGB(<i>red&</i>, <i>green&</i>, <i>blue&</i>) result& = RGB(<i>bgrexpr&</i>)</pre>
Remarks	<p>An RGB value is a long integer value in the range of 0 to &H00FFFFFF. It is used to specify a very precise color to various Classic PowerBASIC functions and Windows API functions. The lowest three bytes of the value each specify the intensity of a primary color which combine to form the resultant color. Byte 1 (lowest) represents the red component, byte 2 the green, and byte 3 the blue. They can each take on a value in the range of 0 to 255. Byte 4 (highest) is always 0. When used with 3 parameters, the RGB() function creates an RGB value from the three component values.</p> <p>Some Windows API functions, namely those which reference Device Independent Bitmaps (DIB), require that the colors be specified in the reverse sequence (Blue-Green-Red instead of Red-Green-Blue). In order to maximize performance and execution speed, Classic PowerBASIC statements and functions which reference these structures also use the BGR format. These include GRAPHIC GET BITS and GRAPHIC SET BITS.</p> <p>When used with one parameter, this function translates a BGR value to its RGB equivalent by swapping the first byte with the third byte, and returning the result.</p> <p>For example, the BGR value of red is &HFF0000. RGB() translates it to &H0000FF. Calling BGR() with that value converts it back to &HFF0000.</p>
See also	Built In RGB Color Equates , BGR

RIGHT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the rightmost <i>n</i> characters of a string.
Syntax	<i>s\$</i> = RIGHT\$(<i>string_expression</i> , <i>n&</i>)
Remarks	<p>If <i>n&</i> is positive, RIGHT\$ returns the indicated number of characters from the string, starting from the right and working left. If <i>n&</i> greater than the length of <i>string_expression</i>, all of <i>string_expression</i> is returned.</p> <p>If <i>n&</i> is 0, RIGHT\$ returns an empty string. If <i>n&</i> is negative, it is interpreted as (LEN(<i>string_expression</i>) - ABS(<i>n&</i>)). For example, RIGHT\$("1234567890", -2) returns "34567890".</p>
See also	EXTRACT\$, INSTR , LEFT\$, LTRIM\$, MID\$, REMOVE\$, REPLACE , RTRIM\$, TALLY , TRIM\$, VERIFY
Example	<pre>' Demonstrate LEFT\$ and RIGHT\$ functions DIM aString\$, x\$, n AS LONG aString\$ = "ABCDEFGHJKLMNOP" FOR n = 1 TO 14 STEP 2 x\$ = LEFT\$(aString, n) + SPACE\$(28 - n * 2) + RIGHT\$(aString, n) NEXT n</pre>

Purpose	Delete a disk directory (like the DOS RMDIR command).
Syntax	<code>RMDIR <i>path</i></code>
Remarks	<p><i>path</i> is a directory path, which may include a drive specification. RMDIR deletes the directory indicated by <i>path</i>.</p> <p>This statement works like the DOS "RMDIR" or "RD" commands. As with the DOS commands, the <i>path</i> must specify a valid, empty directory, other than the default (current) directory. Otherwise, a run-time Error 75 occurs ("Path/file access error").</p> <p style="text-align: center;">RMDIR can use Long File Names (LFNs).</p>
See also	CHDIR , KILL , MKDIR
Example	<pre>DirectoryName\$ = "\TEMP" RMDIR DirectoryName\$</pre>

Purpose	Return a random number.
Syntax	<pre>y = RND y = RND (a, b) y = RND (numeric_expression)</pre>
Remarks	<p>Floating point mode: RND returns a random value that is less than 1, but greater than or equal to 0. Numbers generated by RND aren't really random, but are the result of applying a pseudo-random transformation algorithm to a starting ("seed") value. Given the same seed, Classic PowerBASIC's RND algorithm always produces the same sequence of "random" numbers. The pseudo-random value is calculated internally as a single precision value, but returned as an extended precision representation so it can be readily used in any situation.</p> <p>Integer Range mode: RND(a, b) returns a Long-integer in the range of a to b inclusive. a and b can each be a numeric literal or a numeric expression that evaluates within the range of a Long-integer (-2,147,483,648 to 2,147,483,647).</p> <p>Special effects mode: When used with a single numeric expression argument, the value returned by RND depends on the optional numeric value you supply as the argument, as follows:</p> <p>With no argument, or with a positive argument, RND generates the next number in sequence based on the initial seed value. With an argument of 0, RND repeats the last number generated. A negative argument causes the random number generator to be re-seeded, so subsequent uses of RND with no argument or with a positive argument result in a new sequence of values.</p> <p>Do not use 0 or negative value arguments in special effects mode unless you are looking for the special effects those argument values produce.</p> <p>The random number generator can be reset back to the default seed using the following statement:</p> <pre>RANDOMIZE CVS (CHR\$(255,255,255,255))</pre> <p>Note that each thread has its own, independent random number seed. See the discussion under RANDOMIZE for additional information on seeding the random number generator.</p>
Example	See the example under RANDOMIZE .

ROTATE statement

Purpose	Rotate the bits in an integer class variable.
Syntax	<code>ROTATE {LEFT RIGHT} <i>ivar</i>, <i>count</i></code>
Remarks	<i>ivar</i> must be one of the integer-class variable types: Byte , Word , Integer , Double-word , Long-integer , or Quad-integer . <i>count</i> is the number of bits by which to rotate <i>ivar</i> . ROTATE rotates all the bits in <i>ivar</i> without special regard to the sign bit of a signed integer-class variable.
See also	BIT function , BIT statement , BITS , SHIFT
Example	<pre>i? = 221 ' binary: 1 1 0 1 1 1 0 1 ROTATE RIGHT i?, 1 ' binary: 1 1 1 0 1 1 1 0</pre>

ROUND function

[Top](#) [Previous](#) [Next](#)

Purpose Round a numeric value to a specified number of decimal places.

Syntax `x = ROUND(numeric_expression, n)`

Remarks *n* is an integer class expression specifying the number of decimal places required in the result. ROUND is especially useful in cases where you have a variable in [Single](#), [Double](#), or [Extended-precision](#), and you want to put it into a [Currency](#) variable or display it, rounded to a specific number of decimal places.

Rounding is done according to the "banker's rounding" principle: if the fractional digit being rounded off is exactly five, with no trailing digits, the number is rounded to the nearest even number. This provides better results, on average, than the simple "round up at five" approach.

A%	=	ROUND (0.5, 0)		'	0
A%	=	ROUND (1.5, 0)		'	2
A%	=	ROUND (2.5, 0)		'	2
A%	=	ROUND (2.51, 0)		'	3

See also [CEIL](#), [FIX](#), [FORMAT\\$](#), [INT](#), [USING\\$](#)

Purpose	Right justify a string into the space of a string variable or User-Defined Type .
Syntax	<code>RSET [ABS] <i>result_var</i> = <i>string_expression</i> [USING <i>ustring_expression</i>]</code>
Remarks	RSET right-aligns a string within the space of another string, or within a variable of a User-Defined Type (UDT).
ABS	If ABS is specified, or <i>ustring_expression</i> is null (empty), RSET leaves the padding positions unchanged from their original content, rather than replacing them with spaces.
USING	<p>If <i>string_expression</i> is shorter than <i>result_var</i>, RSET right-justifies <i>string_expression</i> within <i>result_var</i>, and pads remaining character positions on the left side using the first character in <i>ustring_expression</i> or spaces if not specified or is null (empty).</p> <p>If <i>string_expression</i> is longer than <i>result_var</i>, RSET truncates <i>string_expression</i> from the right until it fits in <i>result_var</i>.</p> <p>RSET can be used to assign the content of a User-Defined Type to a User-Defined Type variable of a different class, or assign a dynamic string to a User-Defined Type. For example:</p> <pre>RSET MyUDT = STRING\$(LEN(MyUDT), 0) RSET MyUDT = b\$</pre> <p>LSET works in a similar manner, but left-aligns <i>string_expression</i>; CSET performs center-justification.</p>
See also	CSET , CSET\$, GET , LET , LET (with Types) , LSET , LSET\$, PUT , RESET , RESET\$, STRINSERT\$, TYPE SET
Example	<pre>a\$ = SPACE\$(20) RSET a\$ = "Right-align" ' result " Right-align" RSET a\$ = "Right-align" USING "*" ' result "*****Right-align"</pre>

RSET\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string containing a right-justified (padded) string.
Syntax	<code>a\$ = RSET\$(string_expression, strlen& [USING ustring_expression])</code>
Remarks	RSET\$ right-aligns the string <i>string_expression</i> into a string of <i>strlen&</i> characters.
USING	<p>If <i>ustring_expression</i> is null (empty) or is not specified, RSET\$ pads <i>string_expression</i> with space characters. Otherwise, RSET\$ pads the string with the first character of <i>ustring_expression</i>.</p> <p>If <i>string_expression</i> is shorter than <i>strlen&</i>, RSET\$ right-justifies <i>string_expression</i> within the assigned string variable (<i>a\$</i>), padding the left side as described above; otherwise, RSET\$ returns the left-most <i>strlen&</i> bytes of <i>string_expression</i>.</p>
See also	CSET , CSET\$, GET , LET , LSET , LSET\$, PUT , RESET , RSET , STRINSERT\$, TYPE SET
Example	<pre>a\$ = RSET\$("Classic PowerBASIC", 20) ' result: " Classic PowerBASIC" a\$ = RSET\$("Classic PowerBASIC",20 USING "*") ' result: "*****Classic PowerBASIC"</pre>

RTRIM\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a copy of a string with trailing characters or strings removed.
Syntax	<code>x\$ = RTRIM\$(MainString [, [ANY] MatchString])</code>
Remarks	<p><i>MainString</i> is the string expression from which to remove characters, and <i>MatchString</i> is the string expression specifying the characters that should be removed from the right hand side of <i>MainString</i>.</p> <p>If <i>MatchString</i> is not specified, RTRIM\$ removes trailing spaces. RTRIM\$ returns a sub-string of <i>MainString</i>, from the beginning of the string to the character preceding the consecutive occurrences of <i>MatchString</i> (or space), which continues to the end of the original string. If <i>MatchString</i> (or a space) is not present at the end of <i>MainString</i>, all of <i>MainString</i> is returned.</p> <p>RTRIM\$ is case-sensitive.</p>
ANY	If the ANY keyword is included, <i>MatchString</i> specifies a list of single characters to be searched for individually - a match on any one of which as a trailing character will cause the character to be removed from the result.
See also	EXTRACT\$, INSTR , LEFT\$, LTRIM\$, MID\$, REMOVE\$, REPLACE , RIGHT\$, STRDELETE\$, STRINSERT\$, STRREVERSE\$, TALLY , TRIM\$, VERIFY
Example	<pre>' returns "abacadabra" (match on spaces) x\$ = RTRIM\$("abacadabra ") ' returns "abacadabra " (no match on " cad") x\$ = RTRIM\$("abacadabra ", " cad") ' returns "abacadabr" (match on " " and "a") x\$ = RTRIM\$("abacadabra ", ANY " cad")</pre>

Purpose	Manipulate a SCROLLBAR control. A ScrollBar is a control that allows the user to scroll a data object to bring into view portions of the object that extend beyond the borders of the window.
Syntax	<pre>SCROLLBAR GET PAGE_SIZE hDlg, id& TO datav& SCROLLBAR GET POS hDlg, id& TO datav& SCROLLBAR GET RANGE hDlg, id& TO LoDatav&, HiDatav& SCROLLBAR GET TRACKPOS hDlg, id& TO datav& SCROLLBAR SET PAGE_SIZE hDlg, id&, pagesize& SCROLLBAR SET POS hDlg, id&, position& SCROLLBAR SET RANGE hDlg, id&, lolimit&, hilimit&</pre>
<i>hDlg</i>	Handle of the dialog that owns the ScrollBar.
<i>id&</i>	The control identifier assigned with CONTROL ADD SCROLLBAR .
Remarks	In each of the following samples and descriptions, the SCROLLBAR control that is the subject of the statement is identified by the handle of the dialog that owns the ScrollBar (<i>hDlg</i>), and the unique control identifier you gave it upon creation in CONTROL ADD SCROLLBAR. To alter the color of the bar or the background, use CONTROL SET COLOR .

SCROLLBAR GET PAGE_SIZE hDlg, id& TO datav&

The current page size of the ScrollBar is retrieved and assigned to the [variable](#) designated by *datav&*. Upon ScrollBar creation, the default page size is 10.

SCROLLBAR GET POS hDlg, id& TO datav&

The current position of the ScrollBar is retrieved and assigned to the variable designated by *datav&*. Upon ScrollBar creation, the default position is 0.

SCROLLBAR GET RANGE hDlg, id& TO LoDatav&, HiDatav&

The current range of the ScrollBar is retrieved and assigned to the variables designated by *LoDatav&* and *HiDatav&*. Upon ScrollBar creation, the default ScrollBar range is 0 to 100.

SCROLLBAR GET TRACKPOS hDlg, id& TO datav&

The current position of the scroll box, being dragged by the user, is retrieved and assigned to the variable designated by *datav&*. This is normally read while responding to the %SB_THUMBPOSITION or the %SB_THUMBTRACK messages. The TRACKPOS is then used to move

the scroll position with SCROLLBAR SET POS.

SCROLLBAR SET PAGESIZE *hDlg, id&, pagesize&*

The current page size of the ScrollBar is set to the value of the parameter *pagesize&*, and the bar is redrawn to reflect the new position.

SCROLLBAR SET POS *hDlg, id&, position&*

The current position of the ScrollBar is set to the value of the parameter *position&*, and the bar is redrawn to reflect the new position.

SCROLLBAR SET RANGE *hDlg, id&, lolimit&, hilimit&*

The range for the ScrollBar is specified to be from *lolimit&* to *hilimit&*. If *lolimit&* is greater than *hilimit&*, the results are undefined.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD SCROLLBAR](#), [CONTROL SET COLOR](#)

SEEK function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the location within a file where the next I/O operation will take place.
Syntax	<code>position&& = SEEK([#] <i>filenum</i>&)</code>
Remarks	<p>If file <i>filenum</i>& was opened in random-access mode, SEEK returns the record number of the next record to be written or read as a Quad-integer (64-bit) value. If the file was opened in any other mode, SEEK returns the byte position of the next byte to be written or read, as a Quad-integer (64-bit) value. The Number symbol (#) is optional, but recommended for clarity. The beginning byte position (for binary and sequential files) or record position (for random-access files) may be 0 or 1, depending on the BASE option used when the file was initially Opened. The default, if no BASE is specified, is a starting position of 1.</p> <p>Classic PowerBASIC recommends using the SEEK function over the (more complex) LOC function used in prior versions of Classic PowerBASIC. LOC remains supported for compatibility with older versions of BASIC, but it is likely that LOC may be removed in future versions of Classic PowerBASIC.</p>
See also	EOF , FILEATTR , GET\$, LOC , LOF , OPEN , PUT\$, SEEK statement
Example	<pre>RANDOMIZE TIMER OPEN "OUTPUT.TXT" FOR OUTPUT AS #1 PRINT #1, STRING\$(RND * 80, RND * 255); position&& = SEEK(1) CLOSE #1</pre>

Purpose	Set the position in a file for the next input or output operation.
Syntax	<code>SEEK [#] <i>filenum</i>&, <i>position</i>&&</code>
Remarks	<p>SEEK sets the file pointer position of file <i>filenum</i>& to <i>position</i>&&. <i>position</i>&& is a Quad-integer variable, constant, or expression.</p> <p>The next GET\$ or PUT\$ performed on the file <i>filenum</i>& will occur <i>position</i>&& bytes (or records) deep into the file. If file <i>filenum</i>& was opened in binary or sequential mode, <i>position</i>&& indicates the new file position in bytes; for random-access files, <i>position</i> is in records.</p> <p>The first byte position (for binary and sequential files) or record position (for random-access files) may be 0 or 1, depending on the BASE option used when the file was initially Opened. If no BASE was specified, the default position is 1.</p> <p>Use the SEEK function to determine a binary file's current pointer position, and LOF to determine its length. Seeking past the end of a file does not produce an error, but no data can be read from beyond the true end of the file.</p>
See also	EOF , FILEATTR , GET\$, LOC , LOF , OPEN , PUT\$, SEEK function , SETEOF
Example	<pre>SUB CreateFile ' Open a binary file and writes 75 chars to it. LOCAL I& OPEN "SEEK.DTA" FOR BINARY AS #1 FOR I& = 48 TO 122 PUT\$ 1, CHR\$(I&) NEXT I& END SUB FUNCTION ReadIt\$(Start&&, qSize&&) ' SEEK to the correct position in the file, ' which was previously opened in the CreateFile SUB. SEEK 1, Start&& I&& = 1 TempStr\$ = "" ' Read in the indicated data - don't read past end of file. WHILE (ISFALSE EOF(1)) AND (I&& <= qSize&&) GET\$ 1, 1, Char\$ TempStr\$ = TempStr\$ + Char\$ INCR I&& WEND ReadIt\$ = TempStr\$ ' assign function's result END FUNCTION</pre>

Purpose Control program flow based on the value of an expression.

Syntax

```
SELECT CASE [AS] [LONG | CONST | CONST$] expression
CASE [IS] testlist
    [statements]
[CASE [IS] testlist
    [statements]]
[CASE ELSE
    [statements]]
END SELECT
```

Remarks

testlist is one or more tests, separated by commas, to be performed on *expression*. *expression* can be either string or numeric.

When a SELECT statement is encountered, *expression* is evaluated using the *testlist* in the first CASE clause. If the evaluation is FALSE, the evaluation is repeated using the next *testlist*. As soon as an evaluation is TRUE (non-zero), the statements following that CASE clause are executed, up to the next CASE clause.

Execution then passes to the statement following the END SELECT statement. If none of the evaluations is TRUE, the statements following the optional CASE ELSE clause are executed.

The tests that may be performed by a CASE clause include: equality, inequality, greater than, less than, and range ("from-to") testing. The SELECT CASE block can do string or numeric tests, but these cannot be interchanged.

Examples of numeric CASE clause tests include:

```
SELECT CASE numeric_expression
CASE > b          ' relational; is expression > b?
CASE 14           ' equality (= is assumed); is expression equal to
14?
CASE b TO 99      ' range; is expression between the value of the
                  ' variable b and 99 (inclusive)?
CASE 14, b        ' two equality tests; is expression equal to
                  ' 14 or equal to b?
CASE 25 TO 99,14  ' combination range and equality; is expression
                  ' between 25 and 99 (inclusive) or equal to 14?
```

Examples of string CASE clause tests include:

```
SELECT CASE string_expression
CASE > b$         ' relational; is expression > b$?
CASE "X"          ' equality (= is assumed); is expression equal
                  ' to "X"?
CASE "A" TO "C"   ' range; is expression between "A" and
                  ' "C" (inclusive)?
CASE "Y", b$      ' two equality tests; is expression equal to
                  ' "Y" or equal to b$?
CASE "A" TO "C", "Q" ' combination range and equality; is expression
                  ' between "A" and "C" (inclusive) or equal
                  ' to "Q"?
```

When a CASE clause contains multiple tests separated by commas, a logical [OR](#) is performed. That is, if any one (or more) of the tests is TRUE, the entire clause is deemed to be TRUE.

Use [EXIT SELECT](#) to jump out of a SELECT block prematurely.

Classic PowerBASIC now offers three optional modifiers to provide highly optimized code generation for specific circumstances. By default, numeric expressions are evaluated either as floating point values (to offer the widest range of compatibility for any possible circumstance) or as integer class (for example, if Classic PowerBASIC can establish that all case clauses are integer class values, etc). Further, [string expressions](#) are evaluated dynamically to allow virtually any data. However, if limits on the type and range of the data used for CASE comparison are restricted, performance can be dramatically enhanced with the LONG, CONST or CONST\$ clauses.

AS LONG

In this case, the controlling expression and the CASE expressions must evaluate in the range of a [Long-integer](#). Each of these expressions are calculated dynamically, so all of the normal operators are still available. Performance is enhanced by the integer class of code generation, rather than floating-point. For example, [DWORD](#) values are treated as Long-integer values, so &H0FFFFFFF??? and -1& would be considered equal values. This can help eliminate the need to use functions such as [BITS???](#) when performing comparisons between signed and unsigned values.

AS CONST

In this case, the controlling expression must evaluate in the range of a Long-integer. However, each of the case values must be strictly specified by a [numeric literal](#) (or [numeric equate](#)) in the range of a Long-integer. Multiple case values may be given (CASE 2,3,7), but operators and ranges of values are not allowed. CASE ELSE is permitted. Performance is enhanced by the internal creation of a vector [jump](#) table, one entry for each number from the smallest to the largest case value.

While this form of the structure offers the utmost performance possible, the execution speed must be carefully weighed against the increased program size, particularly when using sparse case values. For example, with just two CASE values of 2 and 1000, the generated jump table would need 999 table entries (3996 bytes in size). The largest allowed jump table for this form is approximately 3200 entries (12K bytes). If exceeded, an [Error 402](#) is generated ("Statement too long/complex").

AS CONST\$

In this case, the controlling expression must evaluate to a [dynamic \(variable length\) string](#) of length zero through 255 bytes. However, each of the case values must be strictly specified by a [string literal](#) (a quoted string, or a [string equate](#)). Multiple case values may be given (CASE "a","Bob",\$value), but operators and ranges of values are not allowed.

Performance is enhanced by the internal creation of a vectored scan table, eight bytes for each case value specified.

See also

[CHOOSE](#), [CHOOSE&](#), [CHOOSE\\$](#), [EXIT SELECT](#), [IF](#), [IF block](#), [IIF](#), [IIF&](#), [IIF\\$](#), [MAX](#), [MAX&](#), [MAX\\$](#), [MIN](#), [MIN&](#), [MIN\\$](#), [ON GOTO](#), [ON GOSUB](#), [SWITCH](#), [SWITCH&](#), [SWITCH\\$](#)

Example

```
DIM Dwrđ AS DWORD
DIM Lint AS LONG

Dwrđ = &H0FFFFFFFF???
Lint = -1&

SELECT CASE Lint
CASE Dwrđ
a$ = "A Match!"
CASE ELSE
a$ = "*No Match"
END SELECT

SELECT CASE AS LONG Lint
CASE Dwrđ
a$ = "*A Match!"
CASE ELSE
a$ = "No Match"
END SELECT

SELECT CASE AS CONST Dwrđ
CASE -1&
a$ = "*A Match!"
CASE 0
a$ = "No Match"
END SELECT

*No Match
*A Match!
*A Match!
```

Result

- Purpose

Set the file system attribute(s) of a disk file or directory.
- Syntax

`SETATTR filespec$, attribute`
- Remarks

filespec\$ specifies a filename (optionally including a drive letter and directory path). *attribute* is a standard operating system attribute code:

Attribute	Description	Equate
0	Normal	%NORMAL
1	Read-only	%READONLY
2	Hidden	%HIDDEN
4	System	%SYSTEM
32	Archived	%ARCHIVE

The attribute code of a given file or directory may be constructed from a combination of individual attribute values. For example, if you use an *attribute* of 0, *filespec\$* will be a regular file: not read-only, not hidden, not system, and not archived.

See also

[DIR\\$](#), [FILEATTR](#), [GETATTR](#)

Example

```
Files$ = "MYTEST.DAT"
SETATTR Files$, %HIDDEN + %SYSTEM
IF ISFALSE ERR THEN a$ = Files$ + " has been hidden!"
```


SETEOF statement

[Top](#) [Previous](#) [Next](#)

Purpose	Truncate or extend an open file to its current file pointer (read/write) position.
Syntax	<code>SETEOF [#] <i>filenum</i>&</code>
Remarks	<p>SETEOF will truncate or extend an open file to its current file pointer (read/write) position, which may be set explicitly with the SEEK statement. Unlike 16-bit Windows and DOS BASIC, Win32 will not truncate a file if you simply write an empty string to it, so the SETEOF statement is provided to cater for this need.</p>
See also	CLOSE , FILEATTR , FLUSH , OPEN , SEEK function , SEEK statement
Example	<pre>FUNCTION PBMAIN OPEN "Temp.dat" FOR BINARY AS #1 BASE = 1 A\$ = SPACE\$(50) PUT\$ #1, A\$ ' File is now 50 bytes SEEK #1, 15 ' Move to the 15th byte and truncate there SETEOF #1 ' File is now 14 bytes CLOSE #1 END FUNCTION</pre>

Purpose	Return the sign of a numeric expression.
Syntax	<code>y = SGN(<i>numeric_expression</i>)</code>
Remarks	<p>If <i>numeric_expression</i> is positive, SGN returns 1. If <i>numeric_expression</i> is zero, SGN returns 0. If <i>numeric_expression</i> is negative, SGN returns -1.</p> <p>In conjunction with the ON GOTO and ON GOSUB statements, SGN can produce a FORTRAN-like three-way branch:</p> <pre>ON SGN(balance) + 2 GOTO InTheRed, BreakingEven, InTheMoney</pre>
See also	ABS , IF , ON GOSUB , ON GOTO , SELECT
Example	<pre>' ON SGN value, GOSUB appropriate subroutine ON SGN (value) + 2 GOSUB Minus, Zero, Plus ' more code here Minus: x\$ = "The product is negative" : RETURN Zero: x\$ = "The product is zero" : RETURN Plus: x\$ = "The product is positive" : RETURN</pre>

Purpose Run an executable program asynchronously (as a separate process), while execution of the original application continues uninterrupted.

Syntax `ProcessId??? = SHELL([HANDLES,] CmdString [, WndStyle])`

Remarks The SHELL function has the following parts:

CmdString The name of the program to execute ("child process"), along with and any required arguments or command-line switches.

WndStyle A number corresponding to the style of the window in which the child process is to be executed. If *WndStyle* is omitted, the program is opened *normal with focus*, the same as *WndStyle* = 1.

The following table identifies the values for *WndStyle* and the resulting style of window:

WndStyle	Window style
0	Hide window
1	Normal with focus (default)
2	Minimized with focus
3	Maximized with focus
4	Normal without focus
6	Minimized without focus

SHELL returns the *process id* of the child process. The process id is a 32-bit [LONG](#) or [DWORD](#) value that identifies the child process, if it's a 32-bit or 64-bit process. If the process id is zero, the child process is not a 32-bit or 64-bit process, or an error occurred. Use [ERR](#) to detect the success of the SHELL function. The HANDLES option allows the child process to inherit the file handles opened by your program. This affects only Windows handles, not Classic PowerBASIC file identifiers. It is an advanced option, for those who know it works and why they need it.

Restrictions Child processes run asynchronously, or independently of the program that SHELLs. So, the child process is, probably, still running after control returns from SHELL to your program. Also, if your program ends before the child process, the child process will continue to run.

To use internal DOS commands like DIR and COPY, you must run the DOS command processor, passing the DOS command as a parameter. See the example below.

If the program name in *CmdString* does not include an explicit path,

Windows will search for the file in the following paths: the directory where the current program is located, the default directory, the 32-bit Windows system directory, the 16-bit Windows system directory, the Windows directory, and any directories listed in the PATH environment variable.

See also

[ERR](#), [SHELL statement](#)

Example

```
pid??? = SHELL(MyApp$,1)
pid??? = SHELL(ENVIRON$("COMSPEC") + " /C DIR *.* > filename.txt")
```

Purpose

Run an executable program synchronously. The SHELLing thread of the calling program is suspended until the SHELLED program ends.

Syntax

`SHELL [HANDLES,] CmdString [, WndStyle, EXIT TO exitcode&]`

Remarks

The SHELL statement has the following parts:

HANDLES

This option, if present, allows the child process to inherit (and access) the Windows file handles of all open files in the parent process. These are not Classic PowerBASIC file numbers, but system file handles and you must use [OPEN HANDLE](#) to access them.

CmdString

The name of the program to execute ("child process"), along with and any required arguments or command-line switches.

WndStyle

A number corresponding to the style of the window in which the program is to be executed. If *WndStyle* is omitted, the program is opened *normal with focus*, the same as *WndStyle* = 1.

The following table identifies the values for *WndStyle* and the resulting style of window:

WndStyle	Window style
0	Hide window
1	Normal with focus (default)
2	Minimized with focus
3	Maximized with focus
4	Normal without focus
6	Minimized without focus

exitcode&

The exit code of the child process (the value returned by the WinMain function) is assigned to the [long integer](#) variable specified by *exitcode&*.

Restrictions

The SHELL statement executes the child process synchronously. That is, SHELL will not return control to your program until the child program finishes.

To use internal DOS commands like DIR and COPY, you must run the DOS

command processor, passing the DOS command as a parameter. See the example below.

If the program name in *CmdString* does not include an explicit path, Windows will search for the file in the following paths: the directory where the current program is located, the default directory, the 32-bit Windows system directory, the 16-bit Windows system directory, the Windows directory, and any directories listed in the PATH environment variable.

If the SHELL statement is not executed successfully, an appropriate error is generated. The [ERR](#) function can be used to detect it.

See also

[ERR](#), [SHELL function](#)

Example

```
SHELL MyApp$,1, EXIT TO exitvar&  
SHELL ENVIRON$("COMSPEC") + " /C DIR *.* > filename.txt"
```

Purpose	Shift the bits in an integer class variable .
Syntax	SHIFT [SIGNED] {LEFT RIGHT} <i>ivar</i> , <i>countexpr</i>
Remarks	<p><i>ivar</i> must be one of the integer-class variable types: Byte, Word, Integer, Double-word, Long-integer, or Quad-integer. <i>countexpr</i> is an integer-class expression specifying the number of bits by which to shift <i>ivar</i>.</p> <p>SHIFT shifts all the bits in <i>ivar</i> without special regard to the sign bit of a signed integer-class variable.</p>
SIGNED	The SIGNED option shifts everything, but does not allow the sign (positive or negative) of the value to change.
LEFT RIGHT	The LEFT or RIGHT option determines the direction of the bit SHIFT operation. SHIFT LEFT shifts the bits toward the high-order end of <i>ivar</i> , and SHIFT RIGHT shifts bits toward the low-order end of <i>ivar</i> .
See also	AND , BIT function , BIT statement , BITS functions , NOT , OR , ROTATE , XOR
Example	<pre>DIM i AS BYTE n = 221 SHIFT LEFT n, 1 ' binary 1 1 0 1 1 1 0 1 ' n = 186 ' binary 1 0 1 1 1 0 1 0 n = 221 SHIFT RIGHT n, 1 ' binary 1 1 0 1 1 1 0 1 ' n = 110 ' binary 0 1 1 0 1 1 1 0 n = 221 SHIFT SIGNED RIGHT n, 1 ' binary 1 1 0 1 1 1 0 1 n = 238 ' binary 1 1 1 0 1 1 1 0</pre>

Purpose	Return the sine of its argument.
Syntax	<code>y = SIN(numeric_expression)</code>
Remarks	<p><i>numeric_expression</i> is an angle specified in radians. SIN returns an Extended-precision value between -1 and +1.</p> <p>To convert radians to degrees, multiply by 57.29577951308232##. To convert degrees to radians, multiply by 0.0174532925199433##. For more information on radians, see the ATN function.</p> <p>The Inverse Sine (ARCSIN) of a value can be calculated as follows:</p> $ArcSin = ATN(Value / SQRT(1 - Value * Value))$ <p>The Hyperbolic Sine (SINH) of a value can also be calculated:</p> $SinH = (EXP(Value) - EXP(-Value)) / 2$ <p>The Inverse Hyperbolic Sine (ARCSINH) of a value can also be calculated:</p> $ArcSinH = LOG(Value + SQRT(Value * Value + 1))$ <pre>' Useful Macro functions MACRO Pi = 3.141592653589793## MACRO DegreesToRadians(dpDegrees) = (dpDegrees * 0.0174532925199433##) MACRO RadiansToDegrees(dpRadians) = (dpRadians * 57.29577951308232##)</pre> <p>See also ATN, COS, TAN</p> <p>Example</p> <pre>pi## = 3.141592653589793## FOR I& = 0 TO 360 STEP 45 x\$ = "The Sine of " + FORMAT\$(I&,"* 0") + _ " degrees =" + FORMAT\$(SIN(pi## / 180 * _ I&),"* 0.00") NEXT I&</pre> <p>Result</p> <pre>The Sine of 0 degrees = 0.00 The Sine of 45 degrees = 0.71 The Sine of 90 degrees = 1.00 The Sine of 135 degrees = 0.71 The Sine of 180 degrees = 0.00 The Sine of 225 degrees = -0.71 The Sine of 270 degrees = -1.00 The Sine of 315 degrees = -0.71 The Sine of 360 degrees = 0.00</pre>

Purpose	Return the total or physical length of any Classic PowerBASIC variable .
Syntax	<code>x& = SIZEOF(<i>target</i>)</code>
Remarks	<p>Particularly useful for determining the maximum length of a fixed-length string, ASCIIZ string, or User-Defined Type. It provides similar functionality to LEN, which returns the current length of a data item.</p> <p><i>target</i> can be the name of any variable type (fixed-length string, ASCIIZ string, User-Defined Type (UDT) variable or definition, etc).</p> <p>When measuring the size of a padded (aligned) UDT variable or definition with the SIZEOF (or LEN) statement, the measured length includes any padding that was added to the structure. For example, the following UDT structure:</p>

```
TYPE LengthTestType DWORD
    a AS INTEGER
END TYPE
' more code here
DIM abc AS LengthTestType
x& = SIZEOF(abc) ' or use SIZEOF(LengthTestType)
```

Returns a length of 4 bytes in x&, since the UDT was padded with 2 additional bytes to enforce DWORD alignment. Note that the SIZEOF of individual UDT members returns the true size of the member without regard to padding or alignment. In the previous example, SIZEOF(abc.a) returns 2.

When used on a dynamic (variable length) string, SIZEOF returns 4, which is the size of the string handle. To obtain the length of the string data in the dynamic string, use the LEN function. SIZEOF also returns 4 for [pointer](#) variables, since a pointer is always stored as a [DWORD](#).

Pointers

When used with a dereferenced pointer (i.e., SIZEOF(@p), SIZEOF returns the size of the pointer target variable type, as defined in the DIM x AS y PTR [* pSize] statement.

For example, with a dynamic string pointer, SIZEOF returns 4. If the pointer target is a fixed-length string, UDT, [Union](#), or ASCIIZ string, SIZEOF returns the size of the target data structure. However, if the pointer is declared to reference an ASCIIZ with no specific target size (i.e., DIM a AS ASCIIZ PTR), SIZEOF returns 0.

Likewise, if SIZEOF is used on a BYREF ASCIIZ string that does not have an explicit length specification, SIZEOF will also return 0. For example:

```
SUB ProcessData(BYREF szText AS ASCIIZ)
' Within this Sub, SIZEOF(szText) will return 0 because there is no
explicit length specification
```

See also

[DIM](#), [LEN](#)

Example

```
DIM Strval AS ASCIIZ * 10
Strval = "test"
' SIZEOF(Strval) = 10, LEN(Strval) = 4

DIM Intval AS QUAD
Intval = 1
' SIZEOF(Intval) = 8, LEN(Strval) = 8

DIM CustName AS STRING
CustName = "Fred Dagg"
' SIZEOF(CustName) = 4, LEN(CustName) = 9

UNION Arrs
  m1(1 TO 1024) AS BYTE
END UNION
DIM p1 AS STRING PTR
DIM p2 AS STRING PTR * 1024
DIM p3 AS Arrs PTR
DIM p4 AS ASCIIZ PTR
DIM p5 AS ASCIIZ PTR * 64
' Results of SIZEOF on these pointers:
' SIZEOF(p1) = 4, SIZEOF(@p1) = 4
' SIZEOF(p2) = 4, SIZEOF(@p2) = 1024
' SIZEOF(p3) = 4, SIZEOF(@p3) = 1024
' SIZEOF(p4) = 4, SIZEOF(@p4) = 0
' SIZEOF(p5) = 4, SIZEOF(@p5) = 64
' SIZEOF(Arrs) = 1024
```

Purpose	Pause the current thread of the application for a specified number of milliseconds (<i>mSec</i>), allowing other processes (or threads) to continue.
Syntax	<code>SLEEP <i>m&</i></code>
Remarks	<p><i>m&</i> is the number of milliseconds (1 millisecond = 1/1000th of a second) to pause the application. Only the current thread pauses. If other threads are present, they will continue to execute. During the SLEEP period, all time-slices for the current thread are given to other threads and processes.</p> <p>If <i>m&</i> is zero, the remainder of the current time-slice is relinquished. If there are no other threads of equal priority, execution continues immediately.</p> <p>The time-slice duration (also known as the Quantum) can vary from version to version of Windows, ranging from 20 mSec to 120 mSec. Therefore, the Quantum can affect the performance of applications when SLEEP 0 is overused. That is, excessive use of SLEEP 0 can cause an application to cede much of its available processor time, causing a significant drop in application performance.</p> <p>When code is running in a tight loop, it is quite possible to use up 100% of the available CPU time, so the occasional use of SLEEP 0 within a tight loop is often beneficial to overall performance of the target PC. For example, it may not be necessary to use SLEEP 0 for <i>every</i> iteration of a loop, but every second or third instead.</p>
See also	THREAD CREATE , TIMER
Example	<pre>' Pause for 5 seconds SLEEP 5000 ' Release time-slice every 256 iterations FOR x& = 0 TO &H0FFFFFFFF& ' code goes here IF x& MOD 256 = 0 THEN SLEEP 0 NEXT x&</pre>

SPACE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string consisting of a specified number of spaces.
Syntax	<code>s\$ = SPACE\$(<i>numeric_expression</i>)</code>
Remarks	<i>numeric_expression</i> is a non-negative expression that specifies how many spaces the function is to return. SPACE\$ can be useful for formatting or prefilling strings.
See also	BUILD\$, CHR\$, CSET , CSET\$, LSET , NUL\$, REPEAT\$, RSET , STRING\$
Example	<code>A\$ = SPACE\$(1000000) ' fill A\$ with 1000000 spaces</code>

SQR function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the square root of its argument.
Syntax	$y = \text{SQR}(\text{numeric_expression})$
Remarks	<p><i>numeric_expression</i> must be greater than or equal to zero. SQR calculates square roots using an optimized algorithm. That is, $y = \text{SQR}(x)$ takes less time to execute than $y = x^{0.5}$.</p> <p>Attempting to take the square root of a negative number does not produce any run-time errors, but the results of such an operation are undefined.</p> <p>SQR returns an Extended-precision result.</p>
See also	EXP , EXP2 , EXP10 , LOG , LOG2 , LOG10

Purpose Declare [static](#) variables inside a [Sub](#), [Function](#), [Method](#), or [Property](#). Static variables retain their values as long as the program is running.

Syntax `STATIC variable[()] [AS type] [, variable[()]]`
`STATIC variable[()] [, variable[()]] [, ...] AS type`

Remarks The STATIC statement is valid only inside a procedure. Static variables retain their values even after the procedure ends. A static variable is local to its procedure, and can have the same name as other variables in other parts of the program without conflict.

To declare an [array](#) as a static variable, use an empty set of parentheses in the variable list: You can then use the [DIM](#) statement to dimension the array.

```
STATIC MyArray%()  
STATIC StringArray() AS STRING
```

The STATIC statement may, optionally, accept a list of variables, all of which are defined by the type descriptor keyword that follows them. For example:

```
STATIC aaa, bbb, ccc AS INTEGER  
STATIC vptr, aptr() AS LONG PTR
```

Restrictions [DEFtype](#) has no effect on variables defined by a STATIC statement.

See also [DIM](#), [GLOBAL](#), [LOCAL](#), [THREADED](#)

Example

```
#COMPILE EXE  
#DIM ALL  
#INCLUDE "WIN32API.INC"  
  
DECLARE SUB DoMessage ()  
  
FUNCTION PBMAIN  
    DIM z%  
    FOR z% = 1 TO 5  
        DoMessage  
    NEXT z%  
END FUNCTION  
  
SUB DoMessage ()  
    STATIC x AS INTEGER  
    STATIC Message() AS ASCIIZ * 256  
    DIM Message(1 TO 5) AS STATIC ASCIIZ * 256  
  
    INCR x          'add one to x  
    Message(x) = "X =" + STR$(x)  
  
    #IF %DEF(%PB_CC32)  
        PRINT Message(x)  
    #ELSE  
        MSGBOX Message(x)  
    #ENDIF  
END SUB
```


Purpose	Manipulate a STATUSBAR control. A StatusBar is a horizontal window, typically at the bottom of a dialog client area, which displays various kinds of status information. It can be divided into parts to display multiple items.
Syntax	<pre>STATUSBAR SET PARTS <i>hDlg</i>, <i>id&</i>, <i>x&</i> [,<i>x&</i>...] STATUSBAR SET TEXT <i>hDlg</i>, <i>id&</i>, <i>item&</i>, <i>style&</i>, <i>text\$</i></pre>
<i>hDlg</i>	Handle of the dialog that owns the status bar.
<i>id&</i>	The control identifier assigned with CONTROL ADD STATUSBAR .
<i>item&</i>	Position of data on the STATUSBAR. First item=1, second=2...
<i>style&</i>	Style bits which specify the appearance of the status bar.
<i>text\$</i>	A string expression passed as a parameter.
Remarks	<p>In each of the following samples and descriptions, the STATUSBAR control which is the subject of the statement is identified by the handle of the dialog that owns the STATUSBAR (<i>hDlg</i>), and the unique control identifier you gave it upon creation in CONTROL ADD STATUSBAR.</p> <p>The value <i>item&</i> refers to the position of the text data item on the STATUSBAR, and is always indexed to one. The first string is position 1, the second is position 2, and so forth.</p>

STATUSBAR SET PARTS *hDlg*, *id&*, *x&* [,*x&*...]

The STATUSBAR control is partitioned into as many as 32 sections, each of which can be used to display some particular status data to the user. The statement contains from 1 to 32 width parameters (*x&*), which specify the pixel or dialog unit size of that section. You can use a very large number for the last parameter to signify that the section should extend all the way to the right side of the window.

```
STATUSBAR SET PARTS hDlg, id&  
    , 50, 50, 9999
```

For example, the above statement would create a status bar with 2 sections of 50 pixels each, and a third section of the remaining width.

STATUSBAR SET TEXT *hDlg*, *id&*, *item&*, *style&*, *text\$*

The text for the data item specified by *item&* is replaced with the new text in *text\$*. The value of *item&* = 1 for the first item, 2 for the second item, etc. The status bar style value can be the default value of zero (0), or one of the other style values formed as a bitmask:

Zero (0) default	Text with a border to appear lower than the window.
------------------	---

%SBT_NOBORDERS The text is drawn without any borders.

%SBT_POPOUT Text with a border to appear higher than the window.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD STATUSBAR](#), [CONTROL SET FONT](#)

Purpose	Return the string representation of a number in printable form.
Syntax	<code>s\$ = STR\$(numeric_expression [, digits])</code>
Remarks	<p>STR\$ returns the string form of a numeric variable or expression in printable text form. <code>digits</code> is an optional integer class expression specifying the maximum total number of digits to appear in the result. If <code>numeric_expression</code> is greater than or equal to zero, STR\$ adds a leading space character; if <code>numeric_expression</code> is less than zero, STR\$ adds a leading negation (minus) character.</p> <p>For example, STR\$(14) returns a three-character string, of which the first character is a space, and the second and third are the ASCII characters "1" and "4". LTRIM\$ can be used to remove leading space characters.</p> <p><code>digits</code> specifies the maximum number of significant digits (1 to 18) desired in the result.</p> <p>STR\$ can also be used to convert numeric values with more than 16 significant digits (i.e., Extended-precision floating-point and Quad-integers) to a printable form. For example:</p> <pre>a## = 2.0/3.0 x\$ = STR\$(a##,18)</pre> <p>returns 0.666666666666666667.</p> <p>The complementary function is VAL, which takes a string argument and returns the numeric equivalent. Thus, <code>number = VAL(STR\$(number))</code>.</p> <p>An integer-class numeric value may also be converted to a string with the FORMAT\$ and USING\$ functions; however, FORMAT\$ can easily return a string that is free from leading space characters. For example, <code>x\$ = FORMAT\$(a&)</code>.</p>
See also	BIN\$, FORMAT\$, HEX\$, LTRIM\$, OCT\$, ROUND , USING\$, VAL
Example	<pre>x\$ = STR\$(-1,5) x\$ = STR\$(5/3,2) x\$ = STR\$(5/3,8) x\$ = STR\$(100000## / 3##, 18)</pre>
Result	<pre>-1 1.7 1.6666667 33333.33333333333333</pre>

STRDELETE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Delete a specified number of characters from a string expression .
Syntax	<code>s\$ = STRDELETE\$(string_expression, start&, count&)</code>
Remarks	Returns a string based on copying string_expression, but with count& characters deleted starting at position start&. The first character in the string is position 1, etc.
See also	STRINSERT\$, STRREVERSE\$
Example	<code>a\$ = STRDELETE\$("Classic PowerBASIC", 4, 2)</code>
Result	<code>PowBASIC</code>

STRING\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string consisting of multiple copies of the specified character.
Syntax	<code>s\$ = STRING\$(count, {code string_expression})</code>
Remarks	<p>STRING\$ with a numeric argument returns a string of <i>count</i> copies of the character with the ASCII code of <i>code</i>, where <i>code</i> is between 0 and 255, inclusive.</p> <p>STRING\$ with a string argument returns a string of <i>count</i> copies of the first character in <i>string_expression</i>.</p> <p>The following functions all return a string of 8 spaces:</p> <pre>A\$ = STRING\$(8, 32) A\$ = STRING\$(8, " ") A\$ = STRING\$(8, \$SPC) A\$ = SPACE\$(8) A\$ = REPEAT\$(8, " ") A\$ = REPEAT\$(8, \$SPC)</pre> <p>Use REPEAT\$ to make multiple copies of a multiple-character string, and SPACE\$ to return a string of space characters.</p>
See also	ASC , BUILD\$, CHR\$, NUL\$, REPEAT\$, SPACE\$

STRINSERT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Insert a string at a specified position within another string expression .
Syntax	<code>s\$ = STRINSERT\$(Main\$, sNew\$, position&)</code>
Remarks	Returns a string consisting of the string expression <i>Main\$</i> , with the string expression <i>sNew\$</i> inserted at <i>position&</i> . If <i>position&</i> is greater than the length of <i>Main\$</i> , <i>sNew\$</i> is appended to <i>Main\$</i> . The first character in the string is position 1, etc.
See also	BUILD\$, CSET , CSET\$, LSET , RSET , STRDELETE\$, STRREVERSE\$
Example	<code>a\$ = STRINSERT\$("Classic PowerBASIC", "ful", 6)</code>
Result	PowerfulBASIC

Purpose	Return the 32-bit DWORD address of the memory block used to store the data held by a dynamic (variable length) string .
Syntax	<code>xPtr = STRPTR(StringVar)</code>
Remarks	<p><i>StringVar</i> is the name of a string variable. STRPTR returns the 32-bit address in memory, where the contents of <i>StringVar</i> are stored.</p> <p><i>Note</i> that STRPTR differs from VARPTR. When used with a string variable, VARPTR returns the address of the string's <i>handle</i>, while STRPTR returns the address of the actual string <i>data</i>. Similarly, a STRING POINTER (or STRING PTR) is a pointer to a string handle - this important distinction should be recognized when working with the VARPTR and STRPTR operations.</p> <p>STRPTR may not be used with fixed-length or ASCIIZ strings, because they do not use string handles. Use VARPTR with fixed-length and ASCIIZ strings.</p> <p>When a dynamic string content is changed, the address of the string data will also change, but the string handle will remain in the same location. Therefore, it is important that your code refresh any pointers that target the string data memory locations directly.</p>
See also	CODEPTR , POKE\$, PEEK\$, VARPTR
Example	<pre>DIM x AS ASCIIZ PTR, A\$ A\$ = "Classic PowerBASIC" x = STRPTR(A\$) ' address of the string handle Message\$ = A\$ ' returns A\$ Message\$ = @x ' returns A\$ as the target of x A\$ = "The power of BASIC!" x = STRPTR(A\$) ' Update string pointer address Message\$ = @x ' returns the target of x</pre>
Result	<p>As the code above runs, Message\$ is assigned the following strings:</p> <pre>Classic PowerBASIC Classic PowerBASIC The power of BASIC!</pre>

STRREVERSE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Reverse the contents of a string expression.
Syntax	<code>s\$ = STRREVERSE\$(Main\$)</code>
Remarks	Reverses the contents of <i>Main\$</i> and returns the result.
See also	STRDELETE\$, STRINSERT\$
Example	<code>a\$ = STRREVERSE\$("Classic PowerBASIC")</code>
Result	CISABrewoP

SUB/END SUB statements

[Top](#) [Previous](#) [Next](#)

Purpose	Define a Sub block.
Syntax	<pre>SUB ProcName [BDECL CDECL SDECL] [ALIAS "AliasName"] [([arguments])] [EXPORT PRIVATE] [STATIC] [LOCAL variable_list] [STATIC variable_list] [statements] [EXIT SUB] [statements] END SUB</pre>
Remarks	<p>All executable code must reside in a Sub (Function, Method, or Property) block. You cannot define a procedure inside another procedure.</p> <p>SUB and END SUB define a subroutine-like block of statements called a procedure (or subprogram), which is invoked with the CALL statement, and may be passed parameters by value or by reference.</p> <p>A Sub may also be invoked without the use of the CALL statement. If the CALL statement is omitted, the parentheses around the <i>arguments</i> list must also be omitted.</p>
STATIC	Specifies the default storage class for variables declared inside the Sub. If not specified, the default storage class is LOCAL . The STATIC keyword must appear after the list of <i>arguments</i> .
ProcName	The name of the Sub. <i>ProcName must be unique</i> : no variable, Function, Sub, Method, Property or label can share the same name.
BDECL	<p>Specifies that the declared procedure uses the BASIC/Pascal calling convention. When a procedure calls a BDECL procedure, it passes its parameters on the stack from left to right.</p> <p>It is the responsibility of the called procedure to clean up the stack before returning to the calling procedure. Therefore, all Classic PowerBASIC procedures that specify the BDECL convention automatically clean up the stack before execution returns to the calling code.</p> <p>In the event the called procedure is imported or exported, Classic PowerBASIC will automatically capitalize the procedure name unless an explicit ALIAS clause is specified. See ALIAS below.</p>
CDECL	<p>Specifies that the declared procedure uses the C calling convention. When a CDECL procedure is called, it passes its parameters on the stack from right to left.</p> <p>The calling procedure removes any passed parameters from the stack as part of the return process. When Classic PowerBASIC code calls procedures using the CDECL convention, the stack is cleaned</p>

automatically after execution returns from the called code.

In the event that the called procedure is imported or exported, Classic PowerBASIC will automatically create a C style ALIAS for the procedure name. This alias will be prefixed with an underscore, followed by the original function name converted to lowercase.

The following two declarations are equivalent, indicating how the default ALIAS name would be created by Classic PowerBASIC:

```
DECLARE SUB C_Function CDECL ()  
DECLARE SUB C_Function CDECL ALIAS "_c_function" ()
```

SDECL

This is the default if neither BDECL nor CDECL are specified. SDECL (and its synonym STDCALL) specifies that the declared procedure uses the "Standard Calling Convention" as defined by Microsoft. When calling an SDECL procedure, parameters are passed on the stack right to left.

Classic PowerBASIC procedures that use the SDECL/STDCALL convention automatically clean up the stack before execution returns to the calling code.

In the event the called procedure is imported or exported, Classic PowerBASIC will automatically capitalize the procedure name unless an explicit ALIAS clause is specified.

ALIAS

[String literal](#) that identifies an case-sensitive alternative name for the procedure. This lets you export a procedure by a name other than what it is called and referenced within the source code.

This can be useful if you want to abbreviate a long name, provide a more descriptive name, or if the exported name needs to contain characters that are illegal in Classic PowerBASIC. *AliasName* is the routine's actual name as it appears in the export table, and *ProcName* is the title that you can use in Classic PowerBASIC. For example:

```
SUB ShortName ALIAS "LongProcName" () EXPORT STATIC
```

The ALIAS clause is very important when exporting procedures. Omitting the ALIAS clause or incorrectly capitalizing the alias name are common causes of "Missing Export" errors. Please refer to the DECLARE, FUNCTION, METHOD, and PROPERTY sections for more information.

Procedure definitions and program flow

The position of procedure definitions is mostly immaterial. They are usually grouped together in one region of the source code, but you cannot nest procedure definitions. That is, you cannot define a procedure within another procedure (although a procedure definition can contain *calls* to other procedures).

Unlike subroutines (see [GOSUB](#)), program execution cannot accidentally

"fall into" a procedure, even if it is located before the [PBMAIN](#) or [WINMAIN](#) Function in your code. For example:

```
#COMPILE EXE

SUB DisplayInfo(a$)
    ' Code goes here
END SUB

FUNCTION PBMAIN
    ' Main program code goes here
END FUNCTION
```

When this program is executed, the code in *DisplayInfo* is only executed if the procedure is explicitly called, even though it is located *earlier* in the source code file.

Procedure definitions should be treated like isolated islands of code; do not jump in or out of them with [GOTO](#), GOSUB or [RETURN](#). Within a procedure block, such statements are legal.

Parameters

arguments

An optional, comma-delimited sequence of formal parameters. The parameters used in the *arguments* list serve only to define the procedure; they have no relationship to other variables in the program (outside of the procedure) with the same name. Including a type class keyword (STATIC, or LOCAL) at the end of the SUB header can specify the default variable type within the SUB body. If no keyword is included, the default is LOCAL. Normally, Classic PowerBASIC passes parameters to a procedure either by reference or by copy. Either way, the address of a variable is passed, and the procedure has to look at that address to get the value of the parameter. If you do not need to modify the parameter (true in many cases), you can speed up your procedures by passing the parameter by value using the BYVAL keyword with your parameter name.

The type of the parameter is specified either by including a type-declaration character at the end of the name or using an AS clause. For example:

```
SUB Test(A AS INTEGER) ' integer passed by reference
SUB Test(A%)           ' integer passed by reference
SUB Test(BYREF A%)     ' integer passed by reference
SUB Test(BYVAL A%)     ' integer passed by value
```

Classic PowerBASIC compilers have a limit of 32 parameters per SUB. To pass more than 32 parameters to a SUB, construct a [User-Defined Type](#) (UDT) and pass the address of the UDT by reference (BYREF) instead.

[Fixed-length strings](#), [ASCIIZ strings](#), and User-Defined

[Types/Unions](#) may also be passed as BYVAL or OPTIONAL

parameters, now. Try to avoid passing large items BYVAL, as it's terribly inefficient, and there is a maximum size limit of 64 Kb for

a given parameter list.

When a Sub definition specifies either a BYREF parameter or a pointer variable parameter, the calling code may freely pass a BYVAL [DWORD](#) or a [pointer](#) instead. While the use of the explicit BYVAL override in the calling code is optional, it is recommended for clarity. It is necessary to explicitly declare all pointer parameters as BYVAL. Failure to do so will generate a compile-time [Error 549](#) ("BYVAL required with pointers"). For example:

```
SUB DoPtrMath(BYVAL x AS BYTE PTR)
```

A Sub may be imported and exported within the same module. That is, a procedure in the module may be stated as EXPORT, while a DECLARE in the same module specifies it as an imported SUB by the option LIB "filename.dll", provided FILENAME.DLL is the name of the module. This may be particularly valuable when you wish to build an [#INCLUDE](#) file with all of the [DECLARE](#) statements for a project.

Additional information on BYVAL/BYREF/BYCOPY parameter passing can be found in the [CALL statement](#) topic.

Optional parameters

Classic PowerBASIC now supports two syntax formats for optional parameters: the classic optional parameter syntax using brackets "[.]", and the new syntax using the OPTIONAL (or OPT) keyword. We'll discuss each one in turn.

Using OPTIONAL/OPT

SUB statements may specify one or more parameters as optional by preceding the parameter with either the keyword OPTIONAL (or the abbreviation OPT). Optional parameters are only allowed with CDECL or SDECL calling conventions, not BDECL.

When a parameter is declared optional, all subsequent parameters in the declaration are optional as well, whether or not they specify an explicit OPTIONAL or OPT directive. The following two lines are equivalent, with both second and third parameters being optional:

```
SUB sABC(a&, OPTIONAL BYVAL b&, OPTIONAL BYVAL c&)  
SUB sABC(a&, OPT BYVAL b&, BYVAL c&)
```

[VARIANT](#) variables are particularly well suited for use as an optional parameter. If the calling code omits an optional VARIANT parameter, (BYVAL or BYREF), Classic PowerBASIC (and most other compilers) substitute a variant of type %VT_ERROR which contains an error value of %DISP_Ee_PARAMNOTFOUND (&H80020004). In this case, you can check for this value directly, or use the [ISMISSING\(\)](#) function to determine whether the parameter was physically passed or not.

When optional parameters (other than VARIANT) are omitted in the calling code, the stack area normally reserved for those parameters is zero-filled. This allows you to test if an optional parameter was passed or not:

If the parameter is defined as a BYVAL parameter, it will have the value zero. For TYPE or UNION variables passed BYVAL, the compiler will pass a string of binary zeroes of length [SIZEOF](#)(Type_or_union_var).

If the parameter is defined as a BYREF parameter, [VARPTR](#) (varname) will equal zero; when this is true, any attempt to use Var_name in your code will result in General Protection Fault or memory corruption. You should use the ISMISSING() function first to determine whether it is safe to access the parameter.

The OPTIONAL directive provides the same functionality as the older syntax using square brackets "[.]". See below.

Using classic optional parameters

When declaring a CDECL procedure, you can specify trailing parameters as optional, using a set of brackets "[.]":

```
SUB KerPlunk CDECL (x%, y% [, z%])
```

Note that the comma separating the y% parameter from the optional z% parameter is inside the brackets. The following calling sequences would then be valid:

```
CALL KerPlunk (x%, y%)  
CALL KerPlunk (x%, y%, z%)
```

Optional parameters must be the last parameters designated in the list. The following is invalid:

```
SUB KerPlunk CDECL ([x%,] y%, z%)
```

Because the SUB (or procedure) being called does not know how many parameters are being passed at the time it is called, you should pass the number of parameters as one of the required parameters in the list.

Classic PowerBASIC continues to support the use of classic optional parameter syntax using brackets ([.]) but this will not be the case in future versions of Classic PowerBASIC. Existing code should be changed to the new OPTIONAL syntax as soon as possible to ensure compatibility with future versions of Classic PowerBASIC.

Local variables

By default, all undeclared variables in a procedure are LOCAL (unless you have specified STATIC in the procedure header). You can also use the LOCAL statement before any executable statements in the Sub definition, to explicitly declare local variables. Since this default behavior is subject to change, you should make an effort to declare every variable used in a

procedure. For example:

```
SUB MySub()  
    LOCAL a%, b#, BigArray%()  
    [statements]  
END SUB
```

creates three local variables: scalar variables *a%* and *b#* ([Integer](#) and [Double-precision](#), respectively), and the Integer [array](#) *BigArray%*. The array must then be dimensioned appropriately:

```
DIM BigArray%(1000)
```

Local variables and arrays variables are automatically deallocated when the procedure terminates. LOCAL scalar variables (except [dynamic strings](#)) are stored on the stack, so creating large ASCIIZ or fixed-length strings (say, approaching 1 MB) may cause a stack overflow. In this case, use STATIC or [GLOBAL](#) ASCIIZ or fixed-length strings, or change the code to use LOCAL dynamic (variable-length) strings.

Static variables

Static variables retain their values as long as the program is running. To declare static variables within a procedure, use the STATIC statement before any executable statements in the definition.

Use the GLOBAL statement to declare variables that are global to the rest of the program.

You must terminate a procedure definition with END SUB, which returns control to the statement directly after the invoking CALL. Use the EXIT SUB statement to return from a procedure definition before reaching the END SUB statement.

Exported procedures

The EXPORT attribute makes a procedure available from another program (DLL or EXE). This is similar to the "PUBLIC" keyword used by some other programming languages.

EXPORT	Subs are private by default. The EXPORT keyword can be used to make a Sub accessible from another module (a DLL). Exported procedures can only be imported from a DLL, not from another .EXE.
PRIVATE	Subs are private by default. The PRIVATE keyword is not necessary, but may be used to help make your code more readable.
See also	CALL , DECLARE , EXIT SUB , FUNCNAME\$, FUNCTION/END FUNCTION , GLOBAL , GOSUB , ISMISSING , LOCAL , RETURN , STATIC
Example	<pre>SUB TestProcedure(I%, L&, S!, D#, E##, A()) ' Code to process parameters END SUB ' end procedure TestProcedure</pre>

```
DIM MyArray(20)      ' declare array of numbers
IntegerVar% = 1
LongInt&      = 2
SinglePre!    = 3
DoublePre#    = 4
MyArray(3)    = 5
CALL TestProcedure(IntegerVar%, LongInt&, SinglePre!, DoublePre#,
IntegerVar%^2, MyArray())
```

Purpose	Exchange the values of two variables of the same data type.
Syntax	<code>SWAP var1, var2</code>
Remarks	<p>var1 and var2 are two variables of the same type. If you try to swap variables of differing types (for example, string and integer, or Single-precision and Double-precision), a compile-time Error 482 occurs ("Data type mismatch").</p> <p>SWAP is handy because a simple trading of values in two consecutive assignment statements does not get the job done:</p> <pre>a = b b = a</pre> <p>By the time you make the second assignment, variable <i>a</i> does not contain the value it used to. To do this without the SWAP statement requires a temporary variable and a third assignment:</p> <pre>temp = a a = b b = temp</pre> <p>SWAP <i>can</i> be used to swap the <i>target</i> values of pointers. In addition, SWAP can also be used to swap the values of pointers themselves.</p>

SWITCH function

[Top](#) [Previous](#) [Next](#)

Purpose	Return one of a series of values based upon a TRUE/FALSE evaluation of a corresponding series of expressions.
Syntax	<pre>var = SWITCH(expr1, val1 [, expr2, val2], ...) var& = SWITCH&(expr1, val1& [, expr2, val2&], ...) var\$ = SWITCH\$(expr1, val1\$ [, expr2, val2\$], ...)</pre>
Remarks	<p>SWITCH expects values of any numeric type. SWITCH& expects values optimized for Long-integer type. SWITCH\$ expects values of string type. If <i>expr1</i> evaluates TRUE, <i>val1</i> is returned, if <i>expr2</i> evaluates TRUE, <i>val2</i> is returned, etc.</p> <p>Each control expression in the series is evaluated as a typical Classic PowerBASIC Boolean expression, which offers short-circuit expression evaluation as needed. To force a bitwise evaluation of an expression, enclose it in parentheses. The value parameters may be expressions, literals or variables of the appropriate data type for the SWITCH function in use.</p> <p>SWITCH returns the matching value parameter from the first TRUE evaluation of the control expressions, evaluated from left to right in the list. Therefore, it would be wise to place the most likely selections at the front of the SWITCH list to achieve the utmost efficiency. If no expressions evaluate to TRUE, then zero (0) is returned.</p>
Restrictions	Contrary to the implementation in some other languages, only the chosen value (one of <i>val1</i> , <i>val2</i> , <i>val3</i>) is evaluated at run-time; the other value parameters are not. This ensures optimum execution speed, as well as the elimination of unanticipated side effects.
See also	CHOOSE , IF , IIF , SELECT
Example	<pre>' SWITCH with simple expressions A\$ = SWITCH\$(x%=1, "Bob", x%=20, "Bruce", x% > 20, "Dan", x% < 1, "Nobody!") ' SWITCH with complex expressions FUNCTION z(i&) AS LONG INCR i& FUNCTION = i& END FUNCTION FUNCTION PBMAIN x& = -1 Choice& = SWITCH&(z(x&), 1, z(x&), 2, z(x&), 3) ' Choice& will equal 2 END FUNCTION</pre>

TAB\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a string with embedded TAB (\$TAB) characters expanded with spaces to a given tab stop.
Syntax	<code>sResult\$ = TAB\$(strtotab\$, tabstop&)</code>
Remarks	All TAB (CHR\$(9) or \$TAB) characters in <i>strtotab\$</i> are replaced with spaces to pad the resulting string to the tab stop position specified in <i>tabstop&</i> . <i>strtotab\$</i> and <i>tabstop&</i> may be variables , literals , or expressions .
Restrictions	If the tab stop specified in <i>tabstop&</i> is less than 1 or greater than 256, the original string is returned unchanged.
See also	PARSE\$, REPLACE
Example	<pre>a\$ = "Hello" & \$TAB & "World" & \$TAB & _ "From PB, Inc." b\$ = TAB\$(a\$,8) b\$ contains "Hello World From PB, Inc."</pre>
Result	

Purpose A [Tab Control](#) is analogous to the dividers in a notebook. It displays one particular page, selecting it from multiple pages, when the user chooses the corresponding tab. The TAB statement is used to manipulate a TAB control.

Syntax

```
TAB DELETE hDlg, id&, tabpage&
TAB GET COUNT hDlg, id& TO datav&
TAB GET DIALOG hDlg, id&, tabpage& TO Hndlv&
TAB GET SELECT hDlg, id& TO datav&
TAB INSERT PAGE hDlg, id&, tabpage&, image&, text$ [CALL CallBack] TO Hndlv&
TAB RESET hDlg, id&
TAB SELECT hDlg, id&, TabPage&
TAB SET IMAGELIST hDlg, id&, hLst
```

hDlg Handle of the [dialog](#) that owns the Tab Control.

id& The control identifier assigned with [CONTROL ADD TAB](#).

Remarks In each of the following samples and descriptions, the Tab Control that is the subject of the statement is identified by the handle of the dialog that owns the Tab Control (*hDlg*), and the unique control identifier you gave it upon creation in CONTROL ADD TAB.

TAB DELETE *hDlg*, *id*&, *tabpage*&

The page specified by the index parameter *tabpage*& is deleted from the Tab Control. The index is 1 for the first item, 2 for the second item, etc.

TAB GET COUNT *hDlg*, *id*& TO *datav*&

The number of pages in the TAB Control is retrieved, and assigned to the [long integer](#) variable specified by *datav*&.

TAB GET DIALOG *hDlg*, *id*&, *tabpage*& TO *Hndlv*&

The handle of a child dialog attached to the TAB Control is retrieved and assigned to the variable designated by *Hndlv*&. The specific dialog to be returned is determined by the value of the parameter *tabpage*& (1=first, 2=second, etc.). If that page/dialog does not exist, the value zero is assigned to *Hndlv*&.

TAB GET SELECT *hDlg*, *id*& TO *datav*&

The index of the currently selected page in the Tab Control is retrieved, and assigned to the variable specified by *datav*&. The index is 1 for the first item, 2 for the second item, etc. If there is no current selection, the value zero (0) is assigned.

TAB INSERT PAGE *hDlg, id&, tabPage&, image&, text\$* **[CALL *CallBack*] TO *Hndlv&***

A page is added to this TAB Control. The parameter *TabPage&* specifies the position of the page to be inserted (1=first, 2=second, etc.). An optional image to be displayed on the tab area is selected from the attached [IMAGELIST](#), based upon the parameter *image&* (1=first, 2=second, etc.). Set *image&* to 0 if no image is desired. The *text\$* parameter specifies the text to be displayed on the tab area. *CallBack* is the name of a [callback](#) procedure to be used for the page dialog. The handle of the newly created dialog is assigned to the variable designated by *Hndlv&*.

TAB RESET *hDlg, id&*

All pages in the specified Tab Control are deleted.

TAB SELECT *hDlg, id&, tabpage&*

The page specified by the *tabpage&* parameter is chosen as the selected page for the TAB control, and the associated dialog is displayed. The value of *item&* = 1 for the first item, 2 for the second item, etc.

TAB SET IMAGELIST *hDlg, id&, hLst*

The IMAGELIST specified by *hLst* is attached to this TAB control. The graphical images contained in the IMAGELIST are displayed on the tabs of this control. The image to be displayed is determined by the specification made in TAB INSERT PAGE. When the TAB control is destroyed, any attached IMAGELIST is automatically destroyed.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD TAB](#), [CONTROL SET FONT](#), [DIALOG FONT](#), [DIALOG SET COLOR](#), [IMAGELIST](#)

Purpose	Count the number of occurrences of specified characters or strings within a string.
Syntax	<code>x& = TALLY(MainString, [ANY] MatchString)</code>
Remarks	<p><i>MainString</i> is the string expression in which to count characters. <i>MatchString</i> is the string expression to count all occurrences of. If <i>MatchString</i> is not present in <i>MainString</i>, zero is returned. When a match is found, the scan for the next match begins at the position immediately following the prior match.</p>
ANY	<p>If the ANY keyword is included, <i>MatchString</i> specifies a list of single characters to be searched for individually: a match on any one of which will cause the count to be incremented for each occurrence of that character. Note that repeated characters in <i>MatchString</i> will not increase the tally. For example:</p> <pre>x = TALLY("ABCD", ANY "BDB") ' returns 2, not 3</pre>
Restrictions	TALLY is case-sensitive, so be wary of capitalization.
See also	INSTR , JOIN\$, LCASE\$, LTRIM\$, MID\$, PARSE , PARSE\$, PARSECOUNT , REMOVE\$, REPLACE , RIGHT\$, RTRIM\$, TRIM\$, UCASE\$, VERIFY
Example	<pre>' Returns 1, counting the string "bac" x& = TALLY("abacadabra", "bac") ' returns 8, counting all "b", "a", and "c" characters x& = TALLY("abacadabra", ANY "bac")</pre>

TAN function

[Top](#) [Previous](#) [Next](#)

Purpose Return the tangent of its argument.

Syntax `y = TAN(numeric_expression)`

Remarks *numeric_expression* is an angle specified in radians. To convert radians to degrees, multiply by 57.29577951308232###. To convert degrees to radians, multiply by 0.0174532925199433###. For more information on radians, see [ATN](#).

TAN returns an [Extended-precision](#) result.

TAN is approximated with the expression:

$$\text{TAN} = \text{SIN}(\text{Value}) / \text{COS}(\text{Value})$$

The Inverse Tangent (ARCTAN) of a value can be easily calculated with the ATN function.

The Hyperbolic Tangent (TANH) of a value can be calculated:

$$\text{TanH} = (\text{EXP}(2 * \text{Value}) - 1) / (\text{EXP}(2 * \text{Value}) + 1)$$

The Inverse Hyperbolic Tangent (ARCTANH) of a value can be calculated:

$$\text{ArcTanH} = \text{LOG}((1 + \text{Value}) / (1 - \text{Value})) / 2$$

' Useful Macro functions

MACRO Pi = 3.141592653589793##

MACRO DegreesToRadians(dpDegrees) = (dpDegrees * 0.0174532925199433##)

MACRO RadiansToDegrees(dpRadians) = (dpRadians * 57.29577951308232##)

See also [ATN](#), [COS](#), [SIN](#)

Example

```
pi# = 3.141592653589793##
FOR I& = 5 TO 45 STEP 5
  x$ = "The Tangent of " + FORMAT$(I&,"* ") + _
    " degrees = " + FORMAT$(TAN(pi## / 180 * _
    I&),"0.00")
NEXT I&
```

Result

```
The Tangent of 5 degrees = 0.09
The Tangent of 10 degrees = 0.18
The Tangent of 15 degrees = 0.27
The Tangent of 20 degrees = 0.36
The Tangent of 25 degrees = 0.47
The Tangent of 30 degrees = 0.58
The Tangent of 35 degrees = 0.70
The Tangent of 40 degrees = 0.84
The Tangent of 45 degrees = 1.00
```

Purpose	Accept an incoming request for communication from a specified TCP/IP port.
Syntax	<code>TCP ACCEPT [#] <i>fNum</i>& AS <i>newfNum</i>&</code>
Remarks	<p>Accept an incoming connection request to the <i>fNum</i>& socket, and create a <i>newfNum</i>& socket handle to communicate with the new connection.</p> <p>TCP ACCEPT is only valid with sockets opened using TCP OPEN SERVER.</p>
See also	TCP and UDP communications , TCP CLOSE , TCP LINE INPUT , TCP NOTIFY , TCP OPEN , TCP PRINT , TCP RECV , TCP SEND , UDP OPEN

TCP CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Close a previously opened TCP/IP port.
Syntax	<code>TCP CLOSE [#] <i>fNum</i>&</code>
Remarks	Close the previously opened TCP/IP port specified by <i>fNum</i> &.
See also	TCP and UDP communications , TCP ACCEPT , TCP LINE INPUT , TCP NOTIFY , TCP OPEN , TCP PRINT , TCP RECV , TCP SEND , UDP CLOSE

Purpose	Receive a line of text from a specified TCP/IP port.
Syntax	<code>TCP LINE [INPUT] [#] <i>fNum</i>&, <i>Buffer</i>\$</code>
Remarks	<p>Receive a line of text from the <i>fNum</i>& TCP/IP port, and place the data in <i>Buffer</i>\$. If no bytes are available, <i>Buffer</i>\$ will be empty (a null string). If TCP LINE did not receive a complete line of text (terminated by a \$CRLF character pair), EOF(<i>fNum</i>&) will return TRUE (non-zero).</p> <p>If a time-out occurs, ERR will be set to indicate a run-time Error 24 ("Device timeout"). See TCP OPEN to specify the TCP socket timeout value.</p> <p>The EOF function may also be used with TCP LINE (and COMM LINE) to detect that an incomplete line was received. Normally, the TCP LINE statement reads data until a \$CRLF character pair is found, and in that case, EOF will return false (zero). However, even if no \$CRLF has been found, TCP LINE will return if no additional data is available. In that case, TCP LINE will return whatever data has been accumulated, and set EOF to logical TRUE (non-zero).</p> <p>In many cases, it would be prudent to test EOF after every TCP LINE statement to verify that a full line has been received. In some cases, you may wish to execute the statement one or more additional times, combining the data, in order to obtain a full line of text.</p>
See also	TCP and UDP communications , EOF , TCP ACCEPT , TCP CLOSE , TCP NOTIFY , TCP OPEN , TCP PRINT , TCP RECV , TCP SEND , UDP RECV

- Purpose

Designate which [TCP/IP](#) events will generate a notification message.
- Syntax

```
TCP NOTIFY [#] fNum&, {SEND | RECV | ACCEPT | CONNECT | CLOSE} TO hWnd&  
AS wMsg&
```
- Remarks

Designates which events (SEND, RECV, ACCEPT, CONNECT and CLOSE) will generate a *wMsg*& notification message to the window procedure (callback) of the GUI window or dialog whose window handle is contained in *hWnd*&.

Your program defines the *wMsg*& value, and this value should be equal or larger than %WM_USER + 500 to avoid conflict with other (common) callback message values.

When the nominated callback function receives the *wMsg*& notification, the *wParam*& parameter identifies the operating system's handle of the socket (see [FILEATTR](#)), the low-order Word of *lParam*& specifies the code of the event (see table below), and the high-order Word of *lParam*& contains the error code (if any).

LO(WORD, <i>lParam</i> &)	Definition
%FD_READ	Data is available to be read from the socket.
%FD_WRITE	The socket is ready for data to be written.
%FD_ACCEPT	The socket is able to accept a new connection.
%FD_CONNECT	The connection has been established.
%FD_CLOSE	The socket has been closed.

Notification messages do not arrive in unabated or continuous streams. That is, once a particular notification message arrives, it will not be sent again until the initial message is acted upon. For example, if a %FD_READ notification is received, it will not be resent until after a [TCP RECV](#) statement is executed.

The [Winsock](#) error codes are listed in WS2_32.INC, prefixed with %WSAE.

INPUT, TCP OPEN, TCP PRINT, TCP RECV, TCP SEND, UDP NOTIFY

Purpose	Enable an application to communicate with a TCP/IP server or client using the TCP protocol over Winsock .
Syntax	<p><i>As a client:</i></p> <pre>TCP OPEN {PORT <i>p&</i> <i>svrc\$</i>} AT <i>address\$</i> AS [#] <i>fNum&</i> [TIMEOUT <i>timeoutval&</i>]</pre> <p><i>As a server:</i></p> <pre>TCP OPEN SERVER [ADDR <i>ip&</i>] {PORT <i>p&</i> <i>svrc\$</i>} AS [#] <i>fNum&</i> [TIMEOUT <i>timeoutval&</i>]</pre>
Remarks	Open a TCP/IP port or service for communication, either as a client or as a server.
SERVER	If the keyword server is included, the TCP port is opened as a TCP/IP server; otherwise, it is opened as a TCP/IP client.
ADDR <i>ip&</i>	As a server, if you specify the optional ADDR <i>ip&</i> , the TCP server monitors connections at the specified <i>ip&</i> address. Otherwise, the primary IP address for the computer is used by default.
PORT <i>p&</i>	As a client, PORT identifies the server port that the client attempts to connect to. As a server, PORT identifies the port the server will monitor for connection requests. You may specify either a port number or a service name, but not both.
<i>svrc\$</i>	If the port number is not specified, a service name must be specified instead. A service name takes the form of "http", "smtp", or "ftp", etc. You may specify either a port number or a service name, but not both.
AT <i>address\$</i>	As a client, <i>address\$</i> identifies the address to connect with. <i>address\$</i> can be a domain such as "Classic PowerBASIC.com", or a dotted IP address in string form, such as "127.0.0.1".
<i>fNum&</i>	A file number such as #1, or a variable with a value obtained using the FREEFILE function.
TIMEOUT	The optional TIMEOUT value allows you to specify how long a TCP SEND , RECV , PRINT , or LINE operation should wait for completion, in milliseconds (<i>mSec</i>). If the specified number of milliseconds elapses without a response, the TCP operation will fail, and the ERR system variable will be set to indicate a run-time Error 24 ("Device timeout"). The default timeout is 60000 milliseconds (60 seconds).
See also	TCP and UDP communications , FREEFILE , TCP ACCEPT , TCP CLOSE , TCP LINE INPUT , TCP NOTIFY , TCP PRINT , TCP RECV , TCP SEND , UDP OPEN

Example

```
#COMPILE EXE

FUNCTION PBMAIN() AS LONG
    LOCAL Buffer$, Site$, File$, Entire_page$
    LOCAL Length&

    Site$ = "www.PowerBASIC.com"
    File$ =
"http://www.PowerBASIC.com/support/forums/Forum2/HTML/000031.html"

    ' Connecting...
    TCP OPEN "http" AT Site$ AS #1 TIMEOUT 60000

    ' Could we connect to site?
    IF ERR THEN
        BEEP
        EXIT FUNCTION
    END IF

    ' Send the GET request...
    TCP PRINT #1, "GET " & File$ & " HTTP/1.0"
    TCP PRINT #1, "Referer: http://www.PowerBASIC.com/"
    TCP PRINT #1, "User-Agent: TCP OPEN Example (www.PowerBASIC.com)"
    TCP PRINT #1, ""

    ' Retrieve the page...
    DO
        TCP RECV #1, 4096, Buffer$
        Entire_page = Entire_page + Buffer$
    LOOP WHILE ISTRUE LEN(Buffer$) AND ISFALSE ERR

    ' Close the TCP/IP port...
    TCP CLOSE #1
END FUNCTION
```

Purpose	Write a string to a nominated TCP/IP port.
Syntax	<code>TCP PRINT [#] <i>fNum</i>&, <i>string_expression</i>[:]</code>
Remarks	<p>Write the data in <i>string_expression</i> to the <i>fNum</i>& TCP/IP port. If the optional semi-colon is not specified, a carriage-return and linefeed pair (\$CRLF or CHR\$(13,10)) is also sent.</p> <p>The TCP PRINT statement does not return until <i>string_expression</i> has been sent, or an error occurs. That is, TCP PRINT is a synchronous or "blocking" statement. If a time-out occurs, ERR will be set to indicate a run-time Error 24 ("Device timeout"). See TCP OPEN to specify the TCP socket timeout value.</p>
See also	TCP and UDP communications , TCP ACCEPT , TCP CLOSE , TCP LINE INPUT , TCP NOTIFY , TCP OPEN , TCP RECV , TCP SEND , UDP OPEN

Purpose	Receive data from a specified TCP/IP port.
Syntax	<code>TCP RECV [#] <i>fNum</i>&, <i>count</i>&, <i>Buffer</i>\$</code>
Remarks	<p>Receive <i>count</i>& bytes from the <i>fNum</i>& TCP/IP port and place them in <i>Buffer</i>\$. If <i>count</i>& bytes are not available, <i>Buffer</i>\$ will receive whatever bytes are available and EOF(<i>fNum</i>&) will return TRUE (non-zero).</p> <p>Typically used in a loop to retrieve a stream of data, a TCP RECV loop should be terminated if <i>Buffer</i>\$ returns an empty string, or if EOF(<i>fNum</i>&) returns TRUE, or if ERR becomes set. If a time-out occurs, ERR will be set to indicate a run-time Error 24 ("Device timeout"). See TCP OPEN to specify the TCP socket timeout value.</p>
See also	TCP and UDP communications , EOF , TCP ACCEPT , TCP CLOSE , TCP LINE INPUT , TCP NOTIFY , TCP OPEN , TCP PRINT , TCP SEND , UDP RECV

Purpose	Write a string to a nominated TCP/IP port.
Syntax	<code>TCP SEND [#] <i>fNum</i>&, <i>string_expression</i></code>
Remarks	<p>Write the specified <i>string_expression</i> to the TCP/IP port specified by <i>fNum</i>&.</p> <p>The TCP SEND statement does not return until <i>string_expression</i> has been sent, or an error occurs. That is, TCP SEND is a synchronous or "blocking" statement. If a time-out occurs, ERR will be set to indicate a run-time Error 24 ("Device timeout"). See TCP OPEN to specify the TCP socket timeout value.</p>
See also	TCP and UDP communications , TCP ACCEPT , TCP CLOSE , TCP LINE INPUT , TCP NOTIFY , TCP OPEN , TCP PRINT , TCP RECV , UDP SEND

THREAD CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Release the handle of a running thread.
Syntax	<code>THREAD CLOSE <i>hThread</i> TO <i>lResult&</i></code>
Remarks	<p>THREAD CLOSE releases the <i>thread handle</i> of the thread identified by the DWORD value <i>hThread</i> (see THREAD CREATE).</p> <p>If successful, <i>lResult&</i> is TRUE (non-zero); otherwise, it is FALSE (zero). If a thread is not closed once it has completed, it will continue to take up memory and CPU resources. Note that THREAD CLOSE does not stop a thread if it is still running; it simply releases the thread's handle (i.e., the resources used to track the thread), and the thread itself will continue to run.</p> <p>Once a thread handle is released, the value stored in <i>hThread</i> becomes undefined. On this basis, thread handles should not be released until there is no further need to test the thread status or change the suspend count for a thread. If a thread does not need to be monitored, its handle can be released immediately after the THREAD CREATE statement, and the threads resources will be freed automatically when the thread terminates naturally. Best practice suggests that after releasing a thread handle, the thread handle variable should be set to 0, to set it apart from other valid thread handle variables.</p> <p>Once a thread has exited, it is not possible to restart the same thread (as identified by <i>hThread</i>). However, a fresh thread can be executed, using the same target thread Function, and resulting in a new thread handle which will identify the new thread.</p>
Restrictions	<p>THREAD CLOSE will always execute successfully provided <i>hThread</i> contains a valid thread handle value. THREAD CLOSE generates no run-time errors; all exceptions are reported in the return value <i>lResult&</i>.</p> <p>The <i>WaitForSingleObject</i> API function can be used wait until a nominated thread has finished executing. Similarly, the <i>WaitForMultipleObjects</i> API can be used to wait for one, two, or all secondary threads (to a maximum of 64 or %MAXIMUM_WAIT_OBJECTS) to complete before continuing on. Such functions can be very useful when a program creates a set of "worker" threads to process data, and the primary thread can then sit idle until all the worker threads have completed all their work. At that point, the primary thread may gather the results of the worker threads, etc.</p> <p>It is also useful to understand that these kind of <i>wait</i> functions are very efficient and use almost no CPU time or resources while they are waiting; however, care must be exercised to avoid a deadlock or circular</p>

suspension. For example, a deadlock condition could occur if thread A is halted while it waits for thread B, which in turn has a suspend count that might only be adjustable by Thread A. Similarly, an infinite loop in one thread may also halt any other thread that is waiting for it to terminate.

[THREADCOUNT](#) continues to report a thread tally that will include threads whose handle has already been released. A thread ID value may not be used interchangeably with a thread handle value.

See also

[FUNCTION/END FUNCTION](#), [THREAD CREATE](#), [THREAD RESUME](#), [THREAD STATUS](#), [THREAD SUSPEND](#), [THREADCOUNT](#), [THREADED variables](#), [THREADID](#)

Purpose	Create a Windows thread, which is a smaller "program-within-a-program", that runs concurrently with the main thread and other threads in the same application program. Threads provide powerful ways for an application to perform several tasks at the same time.
Syntax	<code>THREAD CREATE <i>FuncName</i> (<i>param</i>) [<i>StackSize</i>,] [SUSPEND] TO <i>hThread</i></code>
Remarks	<p>THREAD CREATE creates and begins execution of a new thread Function identified by <i>FuncName</i>. <i>FuncName</i> is specified without quotation marks. This function must take exactly one Long-integer or Double-word (DWORD) parameter by value (BYVAL). For example:</p> <pre>THREAD FUNCTION MyThreadFunction(BYVAL x AS LONG) AS LONG ' Thread code goes here END FUNCTION ' more code here THREAD CREATE MyThreadFunction(var&) TO hThread???</pre> <p>The 32-bit parameter passed to the thread may be used to pass a value such as a programmer-defined ID or window handle to <i>post</i> "progress" messages back to a GUI window/dialog running in another thread. A more common use for the parameter is to pass the address to a UDT or other data structure. Passing an address this way can enable the thread to use a pointer to access large volumes of data that reside outside of the thread. For example:</p> <pre>THREAD FUNCTION MyThread(BYVAL y AS DWORD) AS DWORD DIM x AS MyUDT POINTER x = y ' Set the pointer from the DWORD param ' From here we can access all of the UDT member elements ' using the standard @x pointer syntax END FUNCTION ' more code here DIM x AS MyUDT, hThread??? ' Initialize the members of x here THREAD CREATE MyThread(VARPTR(x)) TO hThread???</pre> <p>Note that data passed this way is subject to the notes (below) concerning GLOBAL and STATIC variables, in order to avoid synchronization problems during context-switching.</p> <p>The return value of the thread Function is retrieved with the THREAD STATUS statement (once the thread has completed execution).</p>
StackSize	A long integer expression to specify the requested size of the stack for this newly created thread. This value should always be specified in increments of 64K (65536). If this parameter is omitted, the size of the stack for the main thread will be used.
SUSPEND	Execution of the thread begins immediately unless the SUSPEND option is

included. In that case, the *suspend count* for the thread will be initially set to 1, and the thread will be initially suspended. The [THREAD RESUME](#) statement is used to decrease the suspend count of a thread by 1, and when the suspend count reaches 0, the thread will start (resume) execution. Controlling the suspend state of a thread requires the thread handle value be retained until such time as the thread can be closed or left to run unmonitored.

hThread If successful, THREAD CREATE returns a Double-word (or Long-integer) handle in *hThread*, or zero (0) if the thread was not started. This handle is used with the other thread statements to control the suspend count, and to release the thread handle, etc. Also see [THREAD CLOSE](#) for more information on monitoring, closing, and waiting for threads to complete.

FuncName The name of the thread function to execute as a thread. A thread Function must comply exactly with the following syntax:

```
[THREAD] FUNCTION ThreadFuncName (BYVAL param AS {LONG | DWORD}) AS  
{LONG | DWORD}
```

Restrictions The THREAD CREATE statement generates no [run-time errors](#); all exceptions are reported as a zero stored in the return value *hThread*. However, the target thread Function must be located in the same compiled module as the THREAD CREATE statement. That is, a thread Function may not be an imported Function.

Additionally, a thread Function may not be directly called or executed, *except* by a THREAD CREATE statement. This restriction is imposed to ensure that Classic PowerBASIC run-time library can maintain a thread-safe state at all times, correctly allocate and deallocate internal [thread-local storage](#), and the various thread functions (such as THREADCOUNT) can return accurate values.

One situation that can arise is where a Function may need to be invoked both directly and used as a thread Function. The easiest solution is to create a small *wrapper* Function for the Function, then use THREAD CREATE with the wrapper Function when a thread is required, or continue to call the original Function directly when a separate thread is not required. For example:

```
FUNCTION WorkerFunc (BYVAL x AS LONG) AS LONG  
    ' code here  
END FUNCTION  
  
THREAD FUNCTION WorkerThread (BYVAL x AS LONG) AS LONG  
    FUNCTION = WorkerFunc (x)  
END FUNCTION  
  
' more code here  
  
' Execute the worker function directly, thus:  
lResult& = WorkerFunc (var&)
```

```
' Execute the worker thread as a thread, using  
' the wrapper function:  
THREAD CREATE WorkerThread(var&) TO hThread???
```

A thread can determine its own ID with the [THREADID](#) function. Note: a thread ID is not interchangeable with a thread handle.

Threads are initialized and started asynchronously, so it is wise to give the operating system a small amount of time to perform thread initialization before using the [THREADCOUNT](#) function to monitor the thread.

Once a thread has exited, it is not possible to restart the same thread as identified by *hThread* - however, a new thread can be initiated using the same Function (which naturally provides a new *hThread* handle value). In addition, the same thread Function can be launched multiple times to create a set of identical threads executing the same code.

As each thread is created, it is assigned its own "private" stack frame. Therefore, [LOCAL](#) and [REGISTER](#) variables are private to each thread, and are automatically "*thread-safe*".

Exercise care when using GLOBAL and STATIC variables that may be accessed by more than one thread at the same time. If one thread is part way through storing data at the point where another thread begins to read the same memory block, it can result in the second thread reading only partially updated (i.e., invalid) data. The point where one thread is suspended so that another can run is called a "*context-switch*". In these situations, the use of Windows' synchronization functions (such as *Critical Sections* and *Mutexes*) may be employed to create thread-safe code.

Thread-safe code is deemed to be unaffected by context-switching, regardless of when context-switching occurs. Local variables, being stored in a "private" stack frame, are not affected by context-switching.

Local variable storage created by each thread is automatically freed when the thread Function terminates, in the same manner as a normal [Sub](#), Function, [Method](#), or [Property](#). However, the thread handle must be explicitly freed with a THREAD CLOSE statement. The THREAD CLOSE can occur at any time, since it only frees the thread handle and has no other impact on the running thread. If the thread result value is not required (or the thread state does not need to be altered), THREAD CLOSE can be used immediately after the THREAD CREATE statement, leaving the thread to run its course.

For more information on threading and synchronization techniques, please refer to MSDN <http://msdn.microsoft.com>.

The Classic PowerBASIC run-time library is thread-safe and reentrant.

See also

[FUNCTION/END FUNCTION](#), [THREAD CLOSE](#), [THREAD GET PRIORITY](#), [THREAD RESUME](#), [THREAD STATUS](#), [THREAD SUSPEND](#), [THREADCOUNT](#), [THREADED variables](#), [THREADID](#)

Example

```
SUB SpawnThreads()  
  LOCAL x AS LONG  
  LOCAL s AS LONG  
  DIM hThread(10) AS LOCAL DWORD  
  
  FOR x = 1 TO 10  
    THREAD CREATE MyThread(x) TO hThread(x)  
    SLEEP 50  
  NEXT  
  
  DisplayText "10 Threads Started! " + _  
    "Wait for them to finish!"  
  
  DO  
    FOR x = 1 TO 10  
      SLEEP 0  
      THREAD STATUS hThread(x) TO s  
      IF s <> &H103 AND s <> 0 THEN ITERATE DO  
    NEXT  
  LOOP WHILE s  
  
  FOR x = 1 TO 10  
    THREAD CLOSE hThread(x) TO s  
  NEXT x  
  
  DisplayText "Finished!"  
END SUB  
  
' The following is executed as a thread Function!  
THREAD FUNCTION MyThread (BYVAL x AS LONG) AS LONG  
  LOCAL n AS LONG  
  LOCAL t AS SINGLE  
  
  DisplayText "Begin Thread" + STR$(x)  
  t = TIMER  
  
  FOR n = 1 TO 10  
    SLEEP 100 + 100 * x  
  NEXT n  
  
  t = TIMER - t  
  DisplayText "End Thread" + STR$(x) + _  
    " Elapsed time = " + STR$(t,5)  
  
END FUNCTION
```

New!

Purpose	Retrieve the Priority Value for a thread.														
Syntax	<code>THREAD GET PRIORITY <i>hThread</i> TO <i>lResult</i>&</code>														
Remarks	<p>THREAD GET PRIORITY retrieves the priority value for the thread specified by the thread handle (<i>hThread</i>). The thread handle is returned by the THREAD CREATE statement at the time the thread is created. If <i>hThread</i> is zero (0), the thread which is currently executing is presumed. The retrieved priority value is assigned to the long or dword variable designated by <i>lResult</i>&. A thread ID cannot be used in place of a thread handle.</p> <p>The thread priority value is one of the following:</p> <table><tr><td><code>%THREAD_PRIORITY_IDLE</code></td><td><code>= -15</code></td></tr><tr><td><code>%THREAD_PRIORITY_LOWEST</code></td><td><code>= -2</code></td></tr><tr><td><code>%THREAD_PRIORITY_BELOW_NORMAL</code></td><td><code>= -1</code></td></tr><tr><td><code>%THREAD_PRIORITY_NORMAL</code></td><td><code>= 0</code></td></tr><tr><td><code>%THREAD_PRIORITY_ABOVE_NORMAL</code></td><td><code>= +1</code></td></tr><tr><td><code>%THREAD_PRIORITY_HIGHEST</code></td><td><code>= +2</code></td></tr><tr><td><code>%THREAD_PRIORITY_TIME_CRITICAL</code></td><td><code>= +15</code></td></tr></table>	<code>%THREAD_PRIORITY_IDLE</code>	<code>= -15</code>	<code>%THREAD_PRIORITY_LOWEST</code>	<code>= -2</code>	<code>%THREAD_PRIORITY_BELOW_NORMAL</code>	<code>= -1</code>	<code>%THREAD_PRIORITY_NORMAL</code>	<code>= 0</code>	<code>%THREAD_PRIORITY_ABOVE_NORMAL</code>	<code>= +1</code>	<code>%THREAD_PRIORITY_HIGHEST</code>	<code>= +2</code>	<code>%THREAD_PRIORITY_TIME_CRITICAL</code>	<code>= +15</code>
<code>%THREAD_PRIORITY_IDLE</code>	<code>= -15</code>														
<code>%THREAD_PRIORITY_LOWEST</code>	<code>= -2</code>														
<code>%THREAD_PRIORITY_BELOW_NORMAL</code>	<code>= -1</code>														
<code>%THREAD_PRIORITY_NORMAL</code>	<code>= 0</code>														
<code>%THREAD_PRIORITY_ABOVE_NORMAL</code>	<code>= +1</code>														
<code>%THREAD_PRIORITY_HIGHEST</code>	<code>= +2</code>														
<code>%THREAD_PRIORITY_TIME_CRITICAL</code>	<code>= +15</code>														
See also	PROCESS GET PRIORITY , PROCESS SET PRIORITY , THREAD CREATE , THREAD SET PRIORITY														

Purpose	Resume execution of a Windows thread.
Syntax	<code>THREAD RESUME <i>hThread</i> TO <i>lResult</i>&</code>
Remarks	<p>THREAD RESUME decreases the <i>suspend count</i> of the thread identified by the 32-bit DWORD value stored in <i>hThread</i> (see THREAD CREATE). If it succeeds, the <i>lResult</i>& value is the thread's previous suspend count; otherwise, it is -1.</p> <p>Execution of a suspended thread resumes when the suspend count of a thread is decremented to zero. If the SUSPEND option is included in the associated THREAD CREATE statement, the thread will have an initial suspend count of 1. In that case, execution of the thread will only begin when a THREAD RESUME statement is executed, using the thread handle stored in <i>hThread</i> to identify the thread.</p>
Restrictions	The THREAD RESUME statement generates no run-time errors ; all exceptions are reported in the return value <i>lResult</i> &. A thread ID cannot be used interchangeably with a thread handle. A thread can suspend itself by incrementing its own suspend count, but logically, cannot decrement its own suspend count.
See also	FUNCTION/END FUNCTION , THREAD CLOSE , THREAD CREATE , THREAD STATUS , THREAD SUSPEND , THREADCOUNT , THREADED variables , THREADID

THREAD SET PRIORITY statement

[Top](#) [Previous](#) [Next](#)

New!

Purpose Sets the Priority Value for a thread.

Syntax `THREAD SET PRIORITY hThread, Priority&`

Remarks THREAD SET PRIORITY assigns a new priority value to the thread specified by the thread handle (*hThread*). The thread handle is returned by the [THREAD CREATE](#) statement at the time the thread is created. If *hThread* is zero (0), the thread which is currently executing is presumed. A thread ID cannot be used in place of a thread handle.

The thread priority value must be one of the following:

```
%THREAD_PRIORITY_IDLE           = -15
%THREAD_PRIORITY_LOWEST          = -2
%THREAD_PRIORITY_BELOW_NORMAL   = -1
%THREAD_PRIORITY_NORMAL          = 0
%THREAD_PRIORITY_ABOVE_NORMAL   = +1
%THREAD_PRIORITY_HIGHEST         = +2
%THREAD_PRIORITY_TIME_CRITICAL  = +15
```

See also [PROCESS GET PRIORITY](#), [PROCESS SET PRIORITY](#), [THREAD CREATE](#), [THREAD GET PRIORITY](#)

Purpose	Retrieve the Status of a Windows thread.
Syntax	<code>THREAD STATUS <i>hThread</i> TO <i>lResult</i>&</code>
Remarks	<p>THREAD STATUS assigns the status of the thread identified by the DWORD value in <i>hThread</i> (see THREAD CREATE) to <i>lResult</i>&.</p> <p>If the function fails, <i>lResult</i>& is set to zero. If the thread is still running, the system value &H103 is assigned. If the thread has terminated and the thread handle has not yet been closed, the return value from the thread Function is assigned to <i>lResult</i>&. To wait for one or more threads to complete execution, use the WaitForSingleObject or WaitForMultipleObjects API functions - see THREAD CLOSE for more information.</p> <p>The number of currently running threads in a module can be determined with the THREADCOUNT function.</p>
Restrictions	The THREAD STATUS statement generates no run-time errors ; all exceptions are reported in the return value <i>lResult</i> &. A thread ID cannot be used in place of a thread handle.
See also	FUNCTION/END FUNCTION , THREAD CLOSE , THREAD CREATE , THREAD RESUME , THREAD SUSPEND , THREADCOUNT , THREADED variables , THREADID

Purpose	Suspend execution of a Windows thread.
Syntax	<code>THREAD SUSPEND <i>hThread</i> TO <i>lResult</i>&</code>
Remarks	<p>THREAD SUSPEND adds 1 to the <i>suspend count</i> of the thread specified by <i>hThread</i> (see THREAD CREATE). If it succeeds, the <i>lResult</i>& value is the thread's previous suspend count; otherwise, it is -1. A thread is always suspended if it has a suspend count of 1 or higher.</p> <p>To decrement the suspend count of a thread, use the THREAD RESUME statement. A suspended thread will only resume execution when its suspend count is decremented to 0.</p>
Restrictions	The THREAD SUSPEND statement generates no run-time errors ; all exceptions are reported in the return value <i>lResult</i> &. A thread ID cannot be used interchangeably with a thread handle. A thread can suspend itself by incrementing its own suspend count.
See also	FUNCTION/END FUNCTION , THREAD CLOSE , THREAD CREATE , THREAD RESUME , THREAD STATUS , THREADCOUNT , THREADED variables , THREADID

Purpose	Return the number of Classic PowerBASIC-created active threads that exist in a module.
Syntax	<i>lCount&</i> = THREADCOUNT
Remarks	<p>Applications will return a THREADCOUNT of at least 1, which is attributed to the "primary" application thread. Additional threads created by the application or module with the THREAD CREATE function will also be included in the tally returned by THREADCOUNT.</p> <p>THREADCOUNT can be useful for when a "controlling thread" needs to poll the state of a collection of "worker threads" as they complete a set of tasks. However, care should be exercised if other (unrelated) threads may also be running in the same module - in such cases, using THREAD STATUS is the preferred solution. If polling is not desired, the WaitForMultipleObjects API function can also be useful - see THREAD CLOSE for more information.</p>
Restrictions	<p>THREADCOUNT includes threads that have had their thread handle released with THREAD CLOSE, yet are still running.</p> <p>Threads are initialized and started asynchronously, so it is wise to give the operating system a small amount of time to perform thread initialization before using the THREADCOUNT function to monitor the thread.</p> <p>A thread Function may not be directly called or executed, except by a THREAD CREATE statement. This restriction is imposed to ensure that Classic PowerBASIC run-time library can maintain a thread-safe state at all times, correctly allocate and deallocate internal thread-local storage, and functions such as THREADCOUNT can return accurate values. See THREAD CREATE for more information and solutions.</p>
See also	FUNCTION/END FUNCTION , THREAD CLOSE , THREAD CREATE , THREAD RESUME , THREAD STATUS , THREAD SUSPEND , THREADED variables , THREADID
Example	<pre>THREAD FUNCTION tZ(BYVAL x&) AS LONG ' Wait for a random time SLEEP x& * RND(1,1000) FUNCTION = 1 END FUNCTION FUNCTION PBMAIN ' Create 10 threads FOR x& = 1 TO 10 THREAD CREATE tZ(x&) TO hThread??? THREAD CLOSE hThread??? TO lResult& NEXT x&</pre>

```
'Wait until the threads are all done  
DO  
    SLEEP 100  
LOOP WHILE THREADCOUNT > 1  
END FUNCTION
```

THREADED statement

[Top](#) [Previous](#) [Next](#)

Purpose	Declare thread-local variables .
Syntax	<pre>THREADED variable[()] [AS type] [, variable[()]] THREADED variable[()] [, variable[()]] [, ...] AS type</pre>
Remarks	<p>Threaded variables are global to every Sub, Function, Method, and Property but are not shared across threads. Each thread has its own independent set of thread-local variables.</p> <p>To declare an array as a threaded variable, use an empty set of parentheses in the variable list: You can then use the DIM statement to dimension the array.</p> <pre>THREADED MyArray%() THREADED StringArray() AS STRING</pre> <p>The THREADED statement may, optionally, accept a list of variables, all of which are defined by the type descriptor keyword that follows them. For example:</p> <pre>THREADED aaa, bbb, ccc AS INTEGER THREADED vptr, aptr() AS LONG PTR</pre>
Restrictions	DEFtype has no effect on variables defined by a THREADED statement.
See also	DIM , GLOBAL , INSTANCE , LOCAL , STATIC , THREADED variables
Example	<pre>THREADED xxx, yyy, zzz AS INTEGER THREADED vptr, aptr() AS LONG PTR</pre>

THREADID function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a Long-integer thread identifier of the current thread.
Syntax	<i>thrdID&</i> = THREADID
Remarks	The thread ID value is returned for the thread that is currently executing. The Thread ID is intended for use with the various (advanced) thread-related API functions provided by Windows.
Restrictions	The thread ID value cannot be used interchangeably with the thread handle returned by THREAD CREATE .
See also	FUNCTION/END FUNCTION , THREAD CLOSE , THREAD CREATE , THREAD RESUME , THREAD STATUS , THREAD SUSPEND , THREADED variables

TIME\$ system variable

[Top](#) [Previous](#) [Next](#)

Purpose Read and/or set the system time.

Syntax To read the time:

```
s$ = TIME$
```

To set the time:

```
TIME$ = string_expression
```

Remarks The system variable TIME\$ contains an eight-character string that represents the time of the system clock in the form "hh:mm:ss", where hh is hours (in 24-hour military form), mm is minutes, and ss is seconds.

Assigning *string_expression* to TIME\$ resets the system clock.

string_expression must contain time information in military (24-hour) format. Minutes and seconds information can be omitted. For example:

```
TIME$ = "12"           'set clock to 12 noon
TIME$ = "13:01"        'set clock to 1:01 PM
TIME$ = "13:01:30"     'set clock to 30 sec after 1:01 PM
TIME$ = "0:01"         'set clock to 1 min after midnight
```

Use the [TIMER](#) function to return the number of seconds that have elapsed since midnight.

See also [DATE\\$](#), [TIMER](#), [TIX](#)

TIMER function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the number of seconds that have elapsed since midnight.
Syntax	<i>y</i> = TIMER
Remarks	TIMER returns the number of seconds since midnight as a Double-precision floating-point value. The resolution is about 1/100 of a second on NT-based platforms, or 1/18th of a second on earlier platforms.
See also	DATE\$, TIME\$, TIX
Example	<pre>OldTime\$ = TIME\$ ' Current time TIME\$ = "12" ' Noon NoonSec\$ = FORMAT\$(TIMER, "#,") x\$ = "Noon is " + NoonSec\$ + " seconds past midnight" TIME\$ = OldTime\$ ' Restore time</pre>
Result	Noon is 43,200 seconds past midnight

Purpose Measures elapsed CPU cycles.

Syntax `TIX QuadVar`
`TIX END QuadVar`

Remarks The TIX statement offers you the ability to measure elapsed CPU cycles, the smallest timing increment possible. Modern processors typically execute billions of cycles per second. This can be beneficial for comparing the execution speed of various styles of coding in Classic PowerBASIC.

TIX QuadVar

The first form of the TIX statement retrieves the current value of the cycle counter and assigns it to the Quad Integer variable.

TIX END QuadVar

The second form of the TIX statement retrieves the current value of the cycle counter. The value in the QuadVar is subtracted from it, and the result is assigned to QuadVar.

To measure the total cycle count for a particular set of statements, you would write:

```
TIX CycleCount&&  
  ' statements to measure go here  
TIX END CycleCount&&
```

At this point, CycleCount&& contains the elapsed number of CPU cycles.

See also [#ALIGN](#), [TIMER](#)

Purpose	A ToolBar control contains one or more buttons which act as shortcuts to menu items. The TOOLBAR statement is used to manipulate a TOOLBAR control.
Syntax	<pre>TOOLBAR ADD BUTTON <i>hDlg</i>, <i>id&</i>, <i>image&</i>, <i>cmd&</i>, <i>style&</i>, <i>text\$</i> [AT <i>item&</i>] [CALL <i>callback</i>] TOOLBAR ADD SEPARATOR <i>hDlg</i>, <i>id&</i>, <i>size&</i> [AT <i>item&</i>] TOOLBAR DELETE BUTTON <i>hDlg</i>, <i>id&</i>, [BYCMD] <i>item&</i> TOOLBAR GET STATE <i>hDlg</i>, <i>id&</i>, [BYCMD] <i>item&</i> TO <i>datav&</i> TOOLBAR GET COUNT <i>hDlg</i>, <i>id&</i> TO <i>datav&</i> TOOLBAR SET IMAGELIST <i>hDlg</i>, <i>id&</i>, <i>hLst</i>, <i>type&</i> TOOLBAR SET STATE <i>hDlg</i>, <i>id&</i>, [BYCMD] <i>item&</i>, <i>state&</i></pre>
<i>hDlg</i>	Handle of the dialog that owns the ToolBar.
<i>hLst</i>	Handle of the ImageList to be used for graphical items.
<i>id&</i>	The control identifier assigned with CONTROL ADD TOOLBAR .
<i>cmd&</i>	Command id number associated with this button.
<i>image&</i>	Image number selected (1=first, 2=second, etc.)
<i>item&</i>	A data item number. First=1, second=2...
<i>size&</i>	Size of the item expressed in pixels.
<i>state&</i>	A state descriptor to define specific attributes.
<i>style&</i>	Style descriptor bits for this button.
<i>text\$</i>	A text to be displayed on this button.
<i>type&</i>	A type descriptor to define specific attributes.
<i>callback</i>	A callback function which receives messages for the control.
<i>datav&</i>	A long integer variable to which result data is assigned.
<i>txtv\$</i>	A string variable to which result text is assigned.
Remarks	<p>A TOOLBAR control contains one or more buttons, each of which normally corresponds to a menu item. It is generally placed at the top of the client area of a dialog. When the user "presses" a tool bar button, the program reacts in the same way as if the command had been selected from a menu. It simply acts as a shortcut to common menu commands.</p> <p>In each of the following samples and descriptions, the TOOLBAR control which is the subject of the statement is identified by the handle of the</p>

dialog that owns the TOOLBAR (*hDlg*), and the unique control identifier (*id&*) you gave it upon creation in CONTROL ADD TOOLBAR.

On many forms of the TOOLBAR statement, a specific button is chosen with the *item&* parameter. If the BYCMD option is included, *item&* specifies the command id number of the button to be used. If not, *item&* describes the button by its position on the TOOLBAR. Since separators are considered to be a special class of button by the operating system, they must be counted when you calculate a position item number. Position parameters are always indexed to one (1=first, 2=second, and so on).

TOOLBAR ADD BUTTON *hDlg*, *id&*, *image&*, *cmd&*, *style&*, *text\$* [AT *item&*]

A button is added to this TOOLBAR. The image to be displayed is selected from the attached IMAGELIST based upon the parameter *image&* (1=first, 2=second, etc.). The *cmd&* parameter specifies the command id number to be executed (with %WM_COMMAND) when the button is pressed. The *style&* parameter describes the style of the button from the following most often used attributes:

%TBSTYLE_BUTTON	The button behaves like a standard push button.
%TBSTYLE_CHECK	The button is dual-state which toggles between the pressed and non-pressed state each time it's clicked.
%TBSTYLE_GROUP	Defines a group of buttons. When combined with the check style, it creates a button that stays pressed until another button in the group is pressed. This is similar to an option button or radio button .

%TBSTYLE_CHECKGROUP A combination of check and group styles.

The *text\$* parameter specifies the text to be displayed on the button

If the optional "AT *item&*" clause is included, the button is inserted at the designated position (1=first, 2=second, etc.). Otherwise, it is added to the end of the list.

If the optional "CALL *callback*" clause is included, it specifies the name of a Callback Function that receives %WM_COMMAND messages when the button is clicked. If not specified, these command messages are sent to the dialog callback specified in string. Message routing by button allows you to easily determine which button generated the event.

If the Callback Function processes a message, it should return [TRUE](#) (non-zero) to prevent the message being passed unnecessarily to the dialog

callback (if one exists). The dialog callback should also return TRUE if the notification message is processed by that Callback Function. Otherwise, the [DDT](#) engine processes unhandled messages.

TOOLBAR ADD SEPARATOR *hDlg, id&, size& [AT *item&]**

A separator is added to this TOOLBAR. It separates two buttons by the number of pixels specified in *size&*. It may be used to separate and distinguish two adjacent button groups (%TBSTYLE_GROUP), or to just enhance the visual appearance. If the "AT *item&*" clause is included, the separator is inserted at the designated position (1=first, 2=second, etc.). Otherwise, it is added to the end of the list.

TOOLBAR DELETE BUTTON *hDlg, id&, [BYCMD] *item&**

A BUTTON or SEPARATOR, specified by *item&*, is deleted from the TOOLBAR. The parameter *item&* may be positional, or it may represent a command id number with BYCMD.

TOOLBAR GET COUNT *hDlg, id& to *datav&**

The number of buttons (and separators) on the TOOLBAR is retrieved and assigned to the long integer variable specified by *datav&*.

TOOLBAR GET STATE *hDlg, id&, [BYCMD] *item&* TO *datav&**

The state descriptor bits for a specific button are retrieved and assigned to the variable designated by *datav&*. The parameter *item&* tells which button to check -- it may be positional, or it may be the command id number when used with BYCMD. The descriptor bits may consist of one or more of:

%TBSTATE_DISABLED	The button is disabled and grayed. (value=0)
%TBSTATE_CHECKED	The button is checked.
%TBSTATE_PRESSED	The button is pressed.
%TBSTATE_ENABLED	The button is enabled.
%TBSTATE_HIDDEN	The button is hidden.
%TBSTATE_INDETERMINATE	The button is indeterminate and grayed.
%TBSTATE_MARKED	The button is highlighted.

TOOLBAR SET IMAGELIST *hDlg, id&, *hLst*, *type&**

The IMAGELIST specified by *hLst* is attached to this TOOLBAR control. The value of *type&* specifies the type of IMAGELIST:

0 Default images

- 1 Disabled images
- 2 Hot images

The graphical images contained in the IMAGELIST are displayed on the TOOLBAR buttons. Up to three IMAGELIST structures may be attached to each TOOLBAR control. The image to be displayed is determined by the specification made in TOOLBAR ADD BUTTON, and the current state of the button. When the TOOLBAR control is destroyed, any attached IMAGELIST is automatically destroyed.

TOOLBAR SET STATE *hDlg*, *id*&, [BYCMD] *item*&, *state*&

The state descriptor bits for the specified button are applied from the expression *state*&. The parameter *item*& tells which button to set -- it may be positional, or it may be the command id number when used with BYCMD. The descriptor bits *state*& may consist of:

%TBSTATE_DISABLED	The button is disabled and grayed. (value=0)
%TBSTATE_CHECKED	The button is checked.
%TBSTATE_PRESSED	The button is pressed.
%TBSTATE_ENABLED	The button is enabled.
%TBSTATE_HIDDEN	The button is hidden.
%TBSTATE_INDETERMINATE	The button is indeterminate and grayed.
%TBSTATE_MARKED	The button is highlighted.

See also

[DIALOG SHOW MODAL](#), [DIALOG SHOW MODELESS](#), [Dynamic Dialog Tools](#), [CONTROL ADD TOOLBAR](#), [CONTROL SET FONT](#), [IMAGELIST](#)

Purpose	Capture a representation of the precise flow of execution in a module.
Syntax	<pre>TRACE NEW <i>fname</i>\$ TRACE ON TRACE PRINT <i>string_expr</i> TRACE OFF TRACE CLOSE</pre>
Remarks	<p>The TRACE statement is used to generate a <i>trace file</i> detailing program flow as execution passes through Labels, plus entry and exit of all Subs, Functions, Methods, and Properties, along with details of passed parameters, etc. All trace details are written to a named disk file <i>fname</i>\$. TRACE also logs Classic PowerBASIC run-time errors as they occur, to assist with locating program errors. TRACE can be dynamically started and stopped with the TRACE ON and TRACE OFF statements to enable the programmer to check specific portions of a program without generating volumes of irrelevant trace data.</p> <p>The five general forms of the TRACE statement are described as follow:</p> <p>TRACE NEW <i>fname</i>\$</p> <p>TRACE NEW causes a standard sequential <i>trace file</i> (of the specified file name <i>fname</i>%) to be created, deleting any previous file of the same name.</p> <p>TRACE ON</p> <p>When a subsequent TRACE ON is then executed, Classic PowerBASIC begins to write pertinent trace information to the trace file. It will contain a chronological list of every call to an internal procedure, the associated parameter values, and the point at which it was exited. Further, it will list a label name each time that program execution flows through the label position.</p> <p>In a test or debugging situation, TRACE, CALLSTK, and CALLSTK\$ allow you to easily answer that age-old programming question, "How did I get here?". TRACE details the entry and exit of every procedure in your program, while CALLSTK simply lists the stack frames that exist above the current level. TRACE is particularly valuable in pinpointing the area of a program where a fatal machine crash occurs.</p> <p>TRACE PRINT <i>string_expr</i></p> <p>TRACE PRINT writes the value of <i>string_expr</i> to the trace file. It can be used to record the value of important variables or other information of importance.</p> <p>TRACE OFF</p> <p>TRACE OFF temporarily stops output to the trace file. The trace can be</p>

subsequently restarted with another TRACE ON statement. An implied TRACE OFF is performed when you exit the procedure in which the current TRACE ON was executed.

TRACE CLOSE

TRACE CLOSE permanently detaches the trace file from the stream of trace data.

The TRACE statement can easily create a huge trace file, so caution must be exercised. Use TRACE ON at the lowest procedure level possible, to keep the output size within reason.

If [PBMAIN](#) contains TRACE NEW and TRACE ON statements, and subsequently calls SUB AAA(x&), which in turn calls SUB BBB(y&,a\$), which then calls SUB CCC(z&), which encounters a run-time [error 5](#), the trace file might look something like this:

```
Trace Begins...
AAA(3)
  BBB(4,string data)
    CCC(5)
      TRACE PRINT printed this user data from CCC()
      ERROR 151 was generated in this thread
    CCC Exit
  BBB Exit
AAA Exit
```

Numeric parameters are displayed in decimal, while pointer and array parameters display a decimal representation of the offset of the target value.

Restrictions TRACE can be invaluable during [debugging](#), but it generates substantial additional code that should be avoided in the final release version of an application. If the source code contains [#TOOLS OFF](#), all TRACE statements which remain in the program are ignored by the compiler, and the parameters and expressions are excluded from the compiled program. To conserve memory requirements in the code, long labels are truncated to 13 characters; however, procedure names are not truncated.

The TRACE statement is "Thread-Aware", displaying only Sub, Function, Method, Property, or Label details from the thread in which it was executed. You can execute TRACE multiple times, or even in multiple concurrent threads. However, you must use caution to ensure that each thread uses a unique name for its own trace file.

See also [#TOOLS](#), [CALLSTK](#), [CALLSTK\\$](#), [CALLSTKCOUNT](#), [FUNCNAME\\$](#), [PROFILE](#)

Example

```
#TOOLS ON
FUNCTION PBMAIN
  TRACE NEW "tracelog.txt"
  TRACE ON
  x& = 3
```

```
CALL AAA(x&)  
TRACE OFF  
TRACE CLOSE  
END FUNCTION
```

```
SUB AAA(x&)  
  INCR x&  
  CALL BBB(x&,"string data")  
  ' More code  
END SUB
```

```
SUB BBB(y&,a$)  
  INCR y&  
  CALL CCC(y&)  
END SUB
```

```
SUB CCC(z&)  
  TRACE PRINT "TRACE PRINT printed this " + _  
    "user data from " + FUNCNAME$ + "()"   
  ERROR 151 ' Trigger a run-time error  
END SUB
```


Purpose A [TreeView](#) control displays a set of data items with a parent-child relationship between the items. This creates a hierarchical list of data which can have any number of levels. Each item displays an optional image and a text string. Each time you add an item, you must specify its relationship to existing data items.

Syntax

```
TREEVIEW DELETE hDlg, id&, hItem
TREEVIEW GET BOLD hDlg, id&, hItem TO datav&
TREEVIEW GET CHECK hDlg, id&, hItem TO datav&
TREEVIEW GET CHILD hDlg, id&, hItem TO datav&
TREEVIEW GET COUNT hDlg, id& TO datav&
TREEVIEW GET EXPANDED hDlg, id&, hItem TO datav&
TREEVIEW GET NEXT hDlg, id&, hItem TO datav&
TREEVIEW GET PARENT hDlg, id&, hItem TO datav&
TREEVIEW GET PREVIOUS hDlg, id&, hItem TO datav&
TREEVIEW GET ROOT hDlg, id& TO datav&
TREEVIEW GET SELECT hDlg, id& TO datav&
TREEVIEW GET TEXT hDlg, id&, hItem TO txtv$
TREEVIEW GET USER hDlg, id&, hItem TO datav&
TREEVIEW INSERT ITEM hDlg, id&, hPrnt, hIAftr, image&, simage&, txt$ TO hItem
TREEVIEW RESET hDlg, id&
TREEVIEW SELECT hDlg, id&, hItem
TREEVIEW SET BOLD hDlg, id&, hItem, flag&
TREEVIEW SET CHECK hDlg, id&, hItem, flag&
TREEVIEW SET EXPANDED hDlg, id&, hItem, flag&
TREEVIEW SET IMAGELIST hDlg, id&, hLst
TREEVIEW SET TEXT hDlg, id&, hItem, txt$
TREEVIEW SET USER hDlg, id&, hItem, NumExpr
TREEVIEW UNSELECT hDlg, id&
```

hDlg Handle of the [dialog](#) that owns the Treeview.

id& The control identifier assigned with [CONTROL ADD TREEVIEW](#).

hItem Handle of a Treeview item, used to uniquely identify the item

datav& A [long integer](#) variable to which result data is assigned.

txtv\$ A string variable to which result data is assigned.

hPrnt Handle of the parent item to insert the new item under.

hIAftr Handle of the item to insert the new item after.

image& Image index of the new item

simage& Selected image index of the new item

txt\$ Text to be displayed for the Treeview item

flag& A long integer value to define specific attributes

hLst Handle of the ImageList to be used for graphical items.

Remarks

TREEVIEW_DELETE *hDlg, id&, hItem*

The data item specified by the handle *hItem* is deleted from the TREEVIEW control.

TREEVIEW_GET_BOLD *hDlg, id&, hItem TO datav&*

The bold attribute for the data item *hItem* is retrieved and assigned to the variable *datav&*. If the item is bold, the value [true](#) (-1) is assigned. If not bold, the value [false](#) (0) is assigned.

TREEVIEW_GET_CHECK *hDlg, id&, hItem TO datav&*

The checkmark attribute for the data item *hItem* is retrieved and assigned to the variable *datav&*. If the checkbox is checked, the value true (-1) is assigned. If not checked, the value false (0) is assigned.

TREEVIEW_GET_CHILD *hDlg, id&, hItem TO datav&*

The parent data item specified by *hItem* is scanned for child data items. If any are found, the handle of the first child is assigned to the variable specified by *datav&*. If none are found, the value zero (0) is assigned to *datav&*.

TREEVIEW_GET_COUNT *hDlg, id& TO datav&*

The number of data items in the TREEVIEW is retrieved, and assigned to the long integer variable specified by *datav&*.

TREEVIEW_GET_EXPANDED *hDlg, id&, hItem TO datav&*

The expanded attribute for the data item *hItem* is retrieved and assigned to the variable *datav&*. If the item is expanded, displaying its child data items, the value true (-1) is assigned. If the item is collapsed, the value false (0) is assigned.

TREEVIEW_GET_NEXT *hDlg, id&, hItem TO datav&*

The data item specified by *hItem* is scanned for sibling data items. The handle of the next sibling is assigned to the variable specified by *datav&*. If no next sibling is found, the value zero (0) is assigned to *datav&*.

TREEVIEW_GET_PARENT *hDlg, id&, hItem TO datav&*

The data item specified by *hItem* is scanned for its parent data item. The handle of the parent is assigned to the variable specified by *datav&*. If no parent is found, the value zero (0) is assigned to *datav&*.

TREEVIEW GET PREVIOUS *hDlg, id&, hItem* TO *datav&*

The data item specified by *hItem* is scanned for sibling data items. The handle of the previous sibling is assigned to the variable specified by *datav&*. If no previous sibling is found, the value zero (0) is assigned to *datav&*.

TREEVIEW GET ROOT *hDlg, id&* TO *datav&*

The handle of the very first data item (topmost) in the TREEVIEW is retrieved, and assigned to the variable specified by *datav&*.

TREEVIEW GET SELECT *hDlg, id&* TO *datav&*

The handle of the data item currently selected in the TREEVIEW is retrieved, and assigned to the variable specified by *datav&*. If there is no current selection, the value zero (0) is assigned.

TREEVIEW GET TEXT *hDlg, id&, hItem* TO *txtv\$*

The text of a specific data item (specified by the handle *hItem*) is retrieved from the TREEVIEW control and assigned to the string variable designated by *txtv\$*.

TREEVIEW GET USER *hDlg, id&, hItem* TO *datav&*

Each item in a TREEVIEW may have a long integer user value associated with it at the discretion of the programmer. This user value is assigned with TREEVIEW SET USER and retrieved with TREEVIEW GET USER. The parameter *hItem* specifies the handle of the user item to be accessed. The returned user value is assigned to the long integer variable specified by *datav&*. In addition to these TREEVIEW user values, every [DDT](#) control offers an additional eight user values which can be accessed with [CONTROL GET USER](#) and [CONTROL SET USER](#).

TREEVIEW INSERT ITEM *hDlg, id&, hPrnt, hIAfter, image&, selimage&, txt\$* TO *hItem*

A new data item is added to this TREEVIEW control. The parameter *hPrnt* specifies the parent of this item, or zero if item is to be inserted at the root. The parameter *hIAfter* specifies the handle of the item after which this new item is to be inserted, or %TVI_FIRST (at the beginning), %TVI_LAST (at the end), %TVI_SORT (alphabetical order). If an IMAGELIST has been attached, the parameters *image&* and *selimage&* specify which image should be displayed (1=first, 2=second, etc.) for normal and selected items. If no image is needed, the value(s) 0 should be used. The parameter *txt\$* designates the text string which should be displayed. If the operation is successful, the handle to the new data item is

assigned to the variable designated by *hltem*. If the operation fails, the value zero is assigned to *hltem*.

TREEVIEW RESET *hDlg, id&*

All data items are deleted from the specified TREEVIEW control.

TREEVIEW SELECT *hDlg, id&, hltem*

The data item specified by the handle *hltem* is chosen as selected text for the TREEVIEW control, and the selected text is scrolled into a visible position.

TREEVIEW SET BOLD *hDlg, id&, hltem, flag&*

The bold attribute for the data item specified by *hltem* is set based upon the value of the *flag&* parameter. If *flag&* is true (non-zero), it is displayed in bold format. If *flag&* is false (zero), it is displayed in normal format.

TREEVIEW SET CHECK *hDlg, id&, hltem, flag&*

The optional checkbox for the data item specified by *hltem* is set based upon the value of the *flag&* parameter. If *flag&* is true (non-zero), it is checked. If *flag&* is false (zero), it is unchecked.

TREEVIEW SET EXPANDED *hDlg, id&, hltem, flag&*

The expanded attribute for the data item specified by *hltem* is set based upon the value of the *flag&* parameter. If *flag&* is true (non-zero), it is displayed in expanded format, with its child items visible. If *flag&* is false (zero), it is displayed in collapsed format.

TREEVIEW SET IMAGELIST *hDlg, id&, hLst*

The IMAGELIST specified by *hLst* is attached to this TREEVIEW control. The images it contains are displayed as needed with each data item. When the TREEVIEW control is destroyed, any attached IMAGELIST is automatically destroyed.

TREEVIEW SET TEXT *hDlg, id&, hltem, txt\$*

The text of a specific data item (specified by the handle *hltem*) is replaced by the text in the string expression *txt\$*.

TREEVIEW SET USER *hDlg, id&, hltem, NumExpr*

Each item in a TREEVIEW may have a long integer user value associated with it at the discretion of the programmer. This user value is assigned with TREEVIEW SET USER, and retrieved with TREEVIEW GET USER. The parameter *hltem* specifies the handle of the user item to be accessed,

while *NumExpr* is the user value saved for later retrieval. In addition to these TREEVIEW user values, every DDT control offers an additional eight user values which can be accessed with CONTROL GET USER and CONTROL SET USER.

TREEVIEW UNSELECT *hDlg, id&*

All items in the TREEVIEW control are set to an unselected state.

See also

[Dynamic Dialog Tools](#), [CONTROL ADD TREEVIEW](#), [CONTROL SET COLOR](#), [CONTROL SET FONT](#), [IMAGELIST](#)

TRIM\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return a copy of a string with leading and trailing characters or substrings removed.
Syntax	<i>NewString\$</i> = TRIM\$(<i>OldString\$</i> [, [ANY] <i>CharsToTrim\$</i>])
Remarks	TRIM\$ combines the functionality of LTRIM\$ and RTRIM\$ into a single function . <i>OldString\$</i> is the string expression from which to remove characters, and <i>CharsToTrim\$</i> is the string expression to remove leading and trailing occurrences. If <i>CharsToTrim\$</i> is not specified, TRIM\$ removes leading and trailing spaces.
ANY	If the ANY keyword is included, <i>CharsToTrim\$</i> specifies a list of single characters to be searched for individually, a match on any one of which as a leading or trailing character will cause the character to be removed from the result.
Restrictions	TRIM\$ is case sensitive, so capitalization matters. TRIM\$ is often used to remove the leading space returned by a STR\$ conversion. While this method does work perfectly, it is a common misuse simply because STR\$ will never return any trailing space characters. On this basis, the use of LTRIM\$(STR\$(x&)) is more efficient than TRIM\$(STR\$(x&)). Alternatively, FORMAT\$(x&) may be used, since trimming of space characters is unnecessary.
See also	FORMAT\$, INSTR , LCASE\$, LTRIM\$, MCASE\$, MID\$, REMOVE\$, REPLACE , RIGHT\$, RTRIM\$, TALLY , UCASE\$, VERIFY

Purpose	A structured method of trapping and responding to run-time errors .
Syntax	<pre>TRY [statements] [EXIT TRY] [statements] CATCH [error handling statements] [EXIT TRY] [error handling statements] [FINALLY [statements] [EXIT TRY] [statements]] END TRY</pre>
Remarks	<p>Statements in the TRY section are executed normally. The first time a run-time error occurs, control is transferred to the CATCH section. If no run-time errors are generated in the TRY section, the CATCH section is skipped entirely.</p> <p>Then, regardless of error status, the FINALLY section is executed, if it is present. Error trapping and control transfer are disabled in the CATCH and FINALLY sections, so you would normally use conventional "IF ERR =" tests to check the success of error-prone operations in those sections. However, TRY structures can be nested to any level, so it may be desirable to use another TRY block within these clauses.</p>
Restrictions	<p>CATCH is a mandatory section of this structure, although the FINALLY section is optional.</p> <p>Because of the nesting requirements, the ERR value is local to the TRY structure. Upon exit, the prior ERR value is restored, so be sure to save the value of ERR if it will be needed outside of the TRY structure.</p> <p>To leave the TRY structure, execution must pass normally through END TRY, or by an EXIT TRY statement. Leaving a TRY block any other way is strongly discouraged because error trapping will remain disabled, and the previous ERR value will not be restored. Future versions of Classic PowerBASIC may disallow such practices.</p> <p>ON ERROR GOTO is invalid within a TRY structure, but may be used within the same Sub/Function/Method/Property.</p>
See also	#DEBUG ERROR , ERL , ERR , ERRCLEAR , ERROR , Error Overview , ERROR\$, Error Trapping , ON ERROR
Example	<pre>TRY OPEN "file.dat" FOR INPUT LOCK READ WRITE AS #1 CATCH CALL NotifyUserOfError(ERR)</pre>

```
EXIT TRY
FINALLY
  CALL UpdateDataBase()
  CLOSE #1
END TRY
```


Purpose Define a [User-Defined Data Type](#) (UDT), containing one or more member elements.

Syntax

```
TYPE MyType [BYTE | WORD | DWORD | QWORD] [FILL]
    [MemberName [(subscripts)] AS] TypeName
    [MemberName [(subscripts)] AS] TypeName
    [...]
END TYPE
```

Remarks The TYPE statement has the following parts:

TYPE The beginning of a User-Defined Type definition.

MyType The name of the User-Defined Type, which must conform to standard [variable](#) naming conventions.

Member alignment

TYPE definitions may optionally specify an alignment of [BYTE](#) (the default), [WORD](#), [DWORD](#), or [QWORD](#), as well as FILL characteristics. With standard alignment, each member of a Type Structure will be located on the specified boundary. For example, with DWORD, up to 3 bytes may be skipped between members to accomplish alignment.

However, when a user-defined type is defined as a member of a larger user-defined type, this "sub-type" retains its original size and alignment, just as first declared.

BYTE Each member will be aligned on a BYTE boundary - no padding or alignment is applied to the structure. This is the default alignment method.

WORD Each member will be aligned on a WORD boundary. Any odd byte between members of TYPE will be automatically skipped and ignored. The UDT structure may also be padded with one trailing byte to ensure the total structure size is a multiple of 2 bytes.

DWORD Each member will be aligned on a DWORD boundary. Up to three bytes will be skipped to accomplish this alignment. The UDT structure is also padded with up to three trailing bytes to ensure the total structure size is a multiple of 4 bytes.

QWORD QWORD alignment is included for compatibility with Windows, it cannot be fully implemented in a 32-bit operating system. With QWORD, individual members are 64-bit aligned for the appropriate structure size, but variables of that type may only be aligned on 32-bit boundaries, as stack pointer alignment is not guaranteed.

FILL If the FILL option is specified, such as TYPE xxx DWORD FILL, the

following rules apply:

1. No bytes are skipped if the next member of the Type will fit entirely into that space to be skipped.
2. Fixed-length strings are considered to be an array of bytes, so no bytes are skipped preceding them.
3. The total size of an array is considered to determine if FILL should affect its placement within the structure. For example, with DWORD FILL, an array of two integers would be started on a 4-byte boundary, even if two or three bytes must be skipped.

Microsoft Visual Basic uses an alignment style equivalent to DWORD FILL.

Type members

MemberName The name of a member of the User-Defined Type. This too must follow the standard variable naming conventions.

subscripts The dimensions of a member [array](#). Arrays of one and two dimensions are supported, but must be defined with [constant or numeric literal](#) values. That is, the total size of a UDT must be known at compile-time, so items like [dynamic strings](#), which vary in size, cannot be part of a TYPE structure. A STRING PTR can, however, since a [pointer](#) is implemented as a DWORD. Like conventional arrays, the default lower array boundary is zero, but positive non-zero values may be used to specify a specific range of subscript index values for the array, separated from the upper array boundary [subscript](#) with the TO keyword. Additionally, both the lower and upper subscript index values must be zero or greater (ie, negative subscript values are not permitted). Examples of valid syntax follow:

```
TYPE MYTYPE
  id AS INTEGER           ' Scalar UDT member
  Styles(6)               AS DWORD ' 7 elements (0 TO 6)
  Yrs(1980 TO 2010) AS LONG ' 31 elements
  Team(100 TO 101) AS BYTE ' 2 elements
  Rating(1 TO 10) AS DWORD ' 10 elements
  X(1 TO 5, 0 TO 5) AS EXT ' 30 elements (5x6)
  Y(4,3) AS QUAD ' 20 elements (5x4)
END TYPE
```

Individual UDT structures can be up to 16 MB each. A single member element of a UDT may also occupy the entire 16 MB. For example, arrays within array statements cannot be used directly on a UDT member array. Instead, use [DIM..AT](#) to declare a conventional array at the same memory address as the UDT member array, and the ARRAY statement can then be used on that array.

TypeName One of the supported data types, including User-Defined Types and

[Unions](#), with the exception of arrays.

END TYPE Marks the end of the User-Defined Type definition.

It is often very convenient to be able to refer to several different types of things as a single unit or data structure. For example, in an accounting program, an account number and amount are part of what makes up a single journal entry. The TYPE/END TYPE block statements make it easy to create a single UDT that holds such information.

```
TYPE JournalType DWORD    ' type name and alignment
  AccountNumber AS LONG    ' member name and data type
  Amount          AS CUR    ' this is another one
END TYPE                  ' end of type declaration
```

```
DIM JournalEntry AS JournalType ' declare a record
```

TYPE/END TYPE blocks must be defined outside of a [Sub](#), [Function](#), or [Class](#) and may be defined only once in any program. It is usually easiest to put your TYPE/END TYPE block definitions in an Include file and use the [#INCLUDE](#) metastatement in any module that may need to use them.

TYPE/END TYPE blocks do not declare any variables; instead, they simply define a new type. You can declare variables of that type using the DIM or [REDIM](#) statements, or any statement that lets you use an AS clause:

```
DIM TypeVariable as TypeVariableType
```

Once you have a User-Defined Type variable declared, you can access its member elements using the following format:

```
TypeVariable.Element
```

For example, to change the account number in the *JournalEntryType* type, you might use a statement like:

```
JournalEntry.AccountNumber = 1000
```

A User-Defined Type can be used like any built-in Classic PowerBASIC type. For example, you can define an array of record variables:

```
DIM JournalEntries(1 TO 100) AS JournalEntryType
```

or even create a procedure that accepts a record variable:

```
SUB PrintJournalEntry(aJournalEntry AS JournalEntryType)
  ' Print journal
END SUB
```

You can also use pointers in a TYPE definition. *Note* that the first member in the next example is auto-aligned to start on a DWORD boundary, and three bytes are skipped so that the second member is also aligned on a DWORD boundary:

```
TYPE MyType DWORD
  Count AS BYTE    ' Aligned to a DWORD boundary
  y AS INTEGER PTR ' Aligned to next DWORD boundary
  z AS STRING PTR
END TYPE
```

Since pointers are stored as a DWORD, their length is 4 bytes when used

in a TYPE/END TYPE, regardless of the length of their target. To access the target of a pointer, you must place the at-sign in front of the TYPE/END TYPE member, not the name of the TYPE itself:

```
iResult% = @MyType.y      ' Invalid
iResult% = MyType.@y      ' Valid
```

You can also declare a variable that is a pointer to a TYPE:

```
TYPE MyData
  Val1 AS INTEGER
  Val2 AS INTEGER
  Val3 AS INTEGER
  Val4 AS INTEGER
END TYPE

DIM Info AS MyData PTR
Info = VARPTR(YourData)
Message$ = HEX$(@Info.Val1) + $CRLF + _
          HEX$(@Info.Val2) + $CRLF + _
          HEX$(@Info.Val3) + $CRLF + _
          HEX$(@Info.Val4)
```

Note that the target specifier is in front of the TYPE name since it is the pointer. Val1, Val2, Val3, and Val4 represent offsets from that pointer. Classic PowerBASIC does support a pointer within a structure pointer, but you should be *very* careful in their use. Changing the structure pointer itself could make all member pointers invalid. See the topic on [pointers](#) for more information.

Bit Variables

TYPE structures may contain bit variables, which are named BIT (unsigned values) or SBIT (signed values). Each bit variable may occupy from 1 to 31 bits, and they may be packed one after another up to a total of 32 bits per bit field. The size of a bit variable is defined as follows:

```
var AS BIT * nlit [IN BYTE|WORD|DWORD]
```

where the term "*" nlit" defines the number of bits (1 to 31), and the optional term "IN BYTE|WORD|DWORD", if present, defines the start of a new bit field of 1, 2, or 4 bytes. For example:

```
TYPE ABCD
  Valu2 AS BIT * 31 IN DWORD
  Sign1 AS SBIT * 1
  nybl2 AS BIT * 4 IN BYTE
  nybl1 AS BIT * 4
END TYPE
```

The example TYPE structure above is 5 bytes in size, containing a 4-byte bit field and a 1-byte bit field. In this case, each contains two bit-variables of varying size. The range of values which may be stored depends upon the number of bits available. For example, "BIT * 4" has a range of 0 to 15, "SBIT * 1" has a range of -1 to 0, and "SBIT * 5" has a range of -16 to

Structures within structures

Structures (TYPE/UNION) may be embedded within another structure, for simplification in referencing deeply nested items, by simply stating the structure name alone at the appropriate position. The internal alignment of the member structure is precisely maintained regardless of other alignment specifications, to foster inheritance issues. For example:

TYPE ABCD3 A AS LONG ABCD2 C AS LONG END TYPE	TYPE ABCD2 D AS DWORD E AS DOUBLE ABCD1 END TYPE	UNION ABCD1 F AS DWORD G AS LONG H AS SINGLE END UNION
--	--	---

In this case, you could access the lone [Single-precision float](#) member of this structure very simply. Assuming DIM X AS ABCD3, you could reference the Single-precision Union member with the statement X.H, instead of the extended syntax X.ABCD2.ABCD1.H

For related information, please refer to the [UNION/END UNION](#) and [User-Defined Types](#) and [Unions](#) sections.

Restrictions When measuring the size of a padded (aligned) UDT structure with the [LEN](#) or [SIZEOF](#) statements, the measured length includes any padding that was added to the structure. For example, the following UDT structure:

```
TYPE LengthTestType DWORD
  a AS INTEGER
END TYPE
' more code here
DIM abc AS LengthTestType
x& = LEN(abc)
```

Returns a length of 4 bytes in x&, since the UDT was padded with 2 additional bytes to enforce DWORD alignment. Note that the LEN and SIZEOF of individual UDT members will return the true size of the member without regard to padding or alignment. In the previous example, LEN(abc.a) returns 2.

Individual UDT structures can be up to 16 MB each. Arrays within a UDT, ASCIIZ strings and fixed-length strings may occupy the full 16 MB structure size limit.

[Field strings](#) and [dynamic strings](#) cannot be used in UDT or UNION structures. Attempting to do so results in a compile-time [Error 485](#) ("Dynamic/Field strings not allowed").

See also [DIM](#), [LEN](#), [REDIM](#), [LET \(with Types\)](#), [SIZEOF](#), [TYPE SET](#), [UNION/END](#)

Example

```
TYPE JournalEntryType
  AccountName      AS STRING * 20
  AccountNumber    AS LONG
  Amount           AS CUR
END TYPE

DIM JournalEntry AS JournalEntryType

JournalEntry.AccountName = "Joe Smith"
JournalEntry.AccountNumber = 7467047&
JournalEntry.Amount       = 42.01@
' process journal entry here
JournalEntry.AccountNumber = 705233476&
JournalEntry.Amount        = 69.35@
' process journal entry here
```

Purpose	Assign the value of a User-Defined Type or string expression into another User-Defined Type or string variable .
Syntax	<code>TYPE SET <i>mainvar</i> = {<i>typevar</i> <i>stringexpr</i>\$} [USING <i>ustring_expression</i>]</code>
Remarks	<p>TYPE SET is primarily designed to assign the value of a User-Defined Type (UDT) to a different class of User-Defined Type. Additionally, TYPE SET can be used to assign a string expression (<i>stringexpr</i>\$) to a UDT or a string variable.</p>
USING	<p>Character positions remaining after the assignment are filled (padded) in <i>mainvar</i> as directed by the optional USING clause.</p> <p>If the assigned data is shorter than the string or type variable <i>mainvar</i>, it is padded on the right with the first character of the USING string expression <i>ustring_expression</i>, or binary zeros (\$NUL) if not specified.</p> <p>If the USING string is null (zero length or empty), any trailing padding character positions are left unchanged in <i>mainvar</i>. This is similar to the ABS option in the LSET statement. To perform right-justification, use the RSET statement.</p>
See also	CSET , CSET\$, LET (with Types) , LSET , LSET\$, RSET , RSET\$, TYPE/END TYPE
Example	<pre>TYPE udt1 x AS STRING * 12 y AS LONG z AS INTEGER END TYPE TYPE udt2 a(1 TO 18) AS BYTE END TYPE FUNCTION PBMAIN DIM u1 AS udt1 DIM u2 AS udt2 u1.x = "ABC" TYPE SET u2 = u1 a\$ = CHR\$(u2.a(1), u2.a(2), u2.a(3)) TYPE SET u2 = "1" USING "2" b\$ = CHR\$(u2.a(1), u2.a(2), u2.a(3)) END FUNCTION</pre> <p>a\$ contains "ABC" b\$ contains "122"</p>
Result	

UBOUND function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the largest possible subscript (boundary) for an array's specified dimension.
Syntax	<pre>y = UBOUND(array [(dimension)]) y = UBOUND(array, dimension)</pre>
Remarks	array is the array of interest. dimension is an integer class value or expression from 1 up to the number of dimensions in array; it specifies which dimension's upper bound value will be returned. If you omit dimension, it defaults to 1 (the first dimension). To find the lower bound of an array's dimension, use the LBOUND function. Use LBOUND and UBOUND together to determine an array's size. UBOUND of an undimensioned array returns -1, so that UBOUND(array) - LBOUND(array) + 1 yields zero (0) for such an array.
Restrictions	UBOUND cannot be used on arrays <i>within</i> User-Defined Types .
See also	ARRAYATTR , DIM , LBOUND , REDIM
Example	<pre>' Dimension an array with lower and upper bounds DIM MyArray%(1900 TO 2000,5 TO 10) ' print out the values of the array Message\$ = "The array's first dimension is from" + _ STR\$(LBOUND(MyArray%(1))) + "to" + _ STR\$(UBOUND(MyArray%(1))) Message\$ = "The array's second dimension is from" + _ STR\$(LBOUND(MyArray%(2))) + "to" + _ STR\$(UBOUND(MyArray%(2)))</pre>
Result	<pre>The array's first dimension is from 1900 to 2000 The array's second dimension is from 5 to 10</pre>

UCASE\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return an all-uppercase (capitalized) version of a string.
Syntax	<code>s\$ = UCASE\$(string_expression [,ANSI OEM])</code>
Remarks	<p>UCASE\$ returns a string equivalent to <i>string_expression</i>, except that lowercase letters in <i>string_expression</i> are converted to uppercase. The optional ANSI or OEM parameter specifies whether the conversion is made using the ANSI charset for the system, or the original IBM OEM charset. If no charset is specified, Classic PowerBASIC for Windows uses the system ANSI charset, while PB/CC uses the IBM OEM charset. Only "International" characters in the range of CHR\$(128) to CHR\$(255) are affected by this parameter.</p> <p>The OEM charset is based upon the original IBM OEM charset to ensure compatibility with programs written for all previous versions of the Classic PowerBASIC compiler.</p>
See also	ASC , LCASE\$, MCASE\$
Example	<code>x\$ = UCASE\$("Beware of cats!")</code>
Result	BEWARE OF CATS!

UCODE\$ function

IMPROVED

[Top](#) [Previous](#) [Next](#)

Purpose Translate an ANSI string into a Unicode string.

Syntax `a$ = UCODE$(AnsiStrExpression [, CodePage&])`

Remarks UCODE\$ returns the Unicode equivalent of the ANSI, multi-byte string contained in *AnsiStrExpression*. To convert a Unicode string to an ANSI string, use the [ACODE\\$](#) function.

If the optional parameter *CodePage*& is present, it represents the code page to be used for the conversion process. If not given, the default code page for the locale of the executing computer is used.

Unicode strings require two [bytes](#) to represent a Unicode character, whereas ANSI strings (the native Classic PowerBASICstring format) use one byte (or sometimes more) to represent a character. Therefore, ACODE\$ returns a string that has a lower byte count than the Unicode string, yet represents the same number of characters.

See also [ACODE\\$](#), [UCODEPAGE](#)

Purpose	Set the default codepage used for ANSI / UNICODE conversions.
Syntax	<code>UCODEPAGE <i>numexpr</i> [TO <i>prevpage</i>&]</code>
Remarks	<p>At times, it is necessary to convert strings from the standard ANSI format to UNICODE format and vice-versa. Classic PowerBASIC may do so internally, or you may use ACODE\$() and UCODE\$() explicitly. By default, the system codepage for your computer is used to map the character translation, and this generally works very well, as it represents the usual codepage for your primary language. However, if you need a special codepage for your own purposes, this statement may be used to define which one should be used by default. Of course, if you specify an explicit codepage number in ACODE\$() or UCODE\$(), it will take precedence over this default value.</p> <p>If the optional TO clause is used, the number of the previous default codepage is assigned to the long integer variable specified by <i>prevpage</i>&. By saving the previous codepage, you can later restore it, if that's appropriate.</p> <p>This statement does not change the codepage in use by your computer. It tells what codepage should be used by default for ANSI/UNICODE conversions.</p> <p>Unicode strings require two bytes to represent a Unicode character, whereas ANSI strings (the native Classic PowerBASIC string format) use one byte (or sometimes more) to represent a character.</p>
See also	ACODE\$, UCODE\$

UDP CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Close a previously opened UDP socket that was created with the UDP OPEN statement.
Syntax	<code>UDP CLOSE [#] <i>fNum</i>&</code>
Remarks	Close the previously opened UDP/IP port specified by <i>fNum</i> &.
See also	TCP and UDP Communication , TCP CLOSE , UDP NOTIFY , UDP OPEN , UDP RECV , UDP SEND

- Purpose** Designate which [UDP/IP](#) events will generate a notification message.
- Syntax** `UDP NOTIFY [#] fNum&, {SEND | RECV | CLOSE} TO hWnd& AS wMsg&`
- Remarks** Designates which events (SEND, RECV, and CLOSE) will generate a notification *wMsg*& message, to be sent to the window/dialog procedure (CALLBACK), identified by the window handle *hWnd*&.
- Your program defines the *wMsg*& value, and this value should be equal or larger than %WM_USER + 500, to avoid conflict with common Windows callback message values.
- When the nominated Callback Function receives the *wMsg*& notification, the *wParam*& parameter identifies the operating system's handle of the socket (see [FILEATTR](#)). The low-order Word of *lParam*& specifies the code of the event (see table below), and the high-order [Word](#) of *lParam*& contains the error code (if any).

LO(WORD, <i>lParam</i> &)	Definition
%FD_READ	Data is available to be read from the socket.
%FD_WRITE	The socket is ready for data to be written.
%FD_CLOSE	The socket has been closed.

Notification messages do not arrive in unabated or continuous streams. That is, once a particular notification message arrives, it will not be sent again until the initial message is acted upon. For example, if an %FD_READ notification is received for a particular socket, it will not be resent until after a UDP RECV statement is executed.

The Winsock error codes are listed in WS2_32.INC, prefixed with %WSAE.

See also [FILEATTR](#), [TCP and UDP Communication](#), [TCP NOTIFY](#), [UDP CLOSE](#), [UDP OPEN](#), [UDP RECV](#), [UDP SEND](#)

UDP OPEN statement

[Top](#) [Previous](#) [Next](#)

Purpose	Create a socket for an application to communicate with a UDP server or client using the UDP (connectionless) protocol over Winsock (UDP/IP).
Syntax	<code>UDP OPEN [PORT <i>p&</i>] AS [#] <i>fNum&</i> [TIMEOUT <i>timeoutval&</i>]</code>
Remarks	Open a UDP socket (port or service) for UDP communication. <i>FNum&</i> is a file number such as #1, or a variable with a value obtained using the FREEFILE function.
PORT	If PORT <i>p&</i> is specified, the socket is opened as a server that can receive UDP data. Use the UDP NOTIFY statement to receive server notifications from the socket so that the data can be retrieved. Common port numbers include 7 (Echo, see RFC862); 37 (Time, see RFC868); and 123 (NTP - RFC1305).
TIMEOUT	The TIMEOUT option allows you to specify how long, in milliseconds (<i>mSec</i>), a UDP SEND/RECV operation should wait for completion. If the specified number of milliseconds elapses, the UDP operation will fail, and the ERR system variable will be set to indicate a run-time Error 24 ("Device timeout"). The default timeout is 60000 milliseconds (60 seconds).
See also	TCP and UDP Communication , FREEFILE , TCP OPEN , UDP CLOSE , UDP NOTIFY , UDP RECV , UDP SEND

UDP RECV statement

[Top](#) [Previous](#) [Next](#)

Purpose	Receive data from a previously opened UDP port.
Syntax	<code>UDP RECV [#] <i>fNum</i>&, FROM <i>ip</i>&, <i>pNum</i>&, <i>Buffer</i>\$</code>
Remarks	<p>Receive any bytes from the previously opened UDP port specified by <i>fNum</i>&, and place them into <i>Buffer</i>\$. The IP address that sent the UDP packet is placed into the <i>ip</i>& variable, and the port number is placed into the <i>pNum</i>& variable.</p> <p><i>ip</i>& and <i>pNum</i>& may be subsequently used to send data back in response to data received.</p> <p>UDP RECV is a blocking statement. That is, execution does not continue until either data is retrieved from the socket, or the timeout period expires. If a timeout occurs, a run-time Error 24 ("Device timeout") is generated and placed in the ERR system variable. See UDP OPEN to specify the UDP socket timeout value.</p>
See also	TCP and UDP Communication , TCP RECV , UDP CLOSE , UDP NOTIFY , UDP OPEN , UDP SEND

UDP SEND statement

[Top](#) [Previous](#) [Next](#)

Purpose	Send a string of data through a previously opened UDP socket.
Syntax	<code>UDP SEND [#] <i>fNum</i>&, AT <i>ip</i>&, <i>pNum</i>&, <i>string_expression</i></code>
Remarks	Write the specified <i>string_expression</i> to the UDP/IP port <i>pNum</i> & at the IP address specified in <i>ip</i> &, using the UDP connection specified by <i>fNum</i> &.
See also	TCP and UDP Communication , TCP SEND , UDP CLOSE , UDP NOTIFY , UDP OPEN , UDP RECV

Purpose	Create a new User-Defined Type definition whose member elements overlap in memory.
Syntax	<pre>UNION UnionName MemberName [(subscripts)] AS TypeName [MemberName [(subscripts)] AS TypeName] [...] END UNION</pre>
Remarks	<p>A union is a type - very similar to a User-Defined Type - except that its elements overlap in memory. While this may seem strange at first, it has enormous potential.</p> <p>For example, say you are designing an accounting program. You want to make it general purpose so it has widespread appeal. But everyone does their accounting differently; for example, some people use account numbers that are plain Integers, while others may use alphanumeric account names. Using a Union makes this easy. Another common use of a Union is variable type conversion. The is best described by way of an example:</p>

```
UNION VarConvert
    iLong  AS LONG
    iDword AS DWORD
    sStr   AS STRING * 4
END UNION

DIM x AS VarConvert, y AS DWORD, z AS STRING
x.iLong = 123456&
y       = x.iDword
z       = x.sStr
```

Like a User-Defined Type, a Union may also contain [arrays](#), and these follow the same rules as User-Defined Type member arrays (see [Type Members](#) for syntax rules and additional examples). The following example demonstrates the use of a Union member array:

```
UNION Arrs
    a1(1 TO 1024) AS BYTE
    st AS ASCIIZ * 10
END UNION

FUNCTION PBMAIN
    DIM a AS Arrs
    a.a1(1) = 72
    a.a1(2) = 101
    a.a1(3) = 108
    a.a1(4) = 108
    a.a1(5) = 111
    a.a1(6) = 33
    ' At this point, a.st contains "Hello!"
END FUNCTION
```

Bit Variables

UNION structures may contain bit variables, which are named BIT (unsigned values) or SBIT (signed values). Each bit variable may occupy from 1 to 31 bits, and they may be packed one after another up to a total of 32 bits per bit field. The size of a bit variable is defined as follows:

```
var AS BIT * nlit [IN BYTE|WORD|DWORD]
```

where the term "`* nlit`" defines the number of bits (1 to 31), and the optional term "`IN BYTE|WORD|DWORD`", if present, defines the start of a new bit field of 1, 2, or 4 bytes. For example:

```
UNION ABCDE
  Odd1 AS BIT * 1 IN DWORD
  Value1 AS LONG
END UNION
```

The example UNION structure above is 4 bytes in size, containing a 1-byte bit field and a 4-byte LONG.

```
UNION abcde
  Part1 AS BIT * 8 IN DWORD
  Part2 AS BIT * 16
END UNION
```

The example union above is 4 bytes in size, containing an 8-bit field and an overlapping 16-bit field.

Structures within structures

Structures (TYPE/UNION) may be embedded within another structure, for simplification in referencing deeply nested items, by simply stating the structure name alone at the appropriate position. The internal alignment of the member structure is precisely maintained regardless of other alignment specifications, to foster inheritance issues. For example:

TYPE ABCD3 A AS LONG ABCD2 C AS LONG END TYPE	TYPE ABCD2 D AS DWORD E AS DOUBLE ABCD1 END TYPE	UNION ABCD1 F AS DWORD G AS LONG H AS SINGLE END UNION
--	--	---

In this case, you could access the lone Single-precision float member of this structure very simply. Assuming [DIM](#) X AS ABCD3, you could reference the [Single-precision](#) Union member with the variable name X.H, instead of the extended syntax X.ABCD2.ABCD1.H

Restrictions A Union can contain elements of dissimilar sizes. The size of a Union structure is always determined by the longest member element. This is usually an important consideration when using a Union within another Union or UDT structure, in order to determine the size of the final structure.

For related information, please refer to the [TYPE/END TYPE](#), [User-Defined Types](#) and [Unions](#) sections.

Field strings cannot be used in UDT or UNION structures. Attempting to do so results in a compile-time [Error 485](#) ("Dynamic/Field strings not allowed").

See also [DIM](#), [LEN](#), [LET \(with Types\)](#), [SIZEOF](#), [TYPE/END TYPE](#), [User-Defined Types](#), [Unions](#)

Example

```
UNION AccountUnion
  AccountNumber AS LONG
  AccountName   AS STRING * 16
END UNION

TYPE JournalEntryType
  Account AS AccountUnion
  Amount  AS CUR
END TYPE

DIM JournalEntry AS JournalEntryType

JournalEntry.Account.AccountName = "Smith"
JournalEntry.Amount = 123.01@
' process journal entry here
JournalEntry.Account.AccountNumber = 1001
JournalEntry.Amount = -1.99@
```

UNLOCK statement

[Top](#) [Previous](#) [Next](#)

Purpose	Remove locks placed on a file to permit other threads, processes, and applications to access the locked sections of the file.
Syntax	<code>UNLOCK [#] <i>filenum</i>& [, {<i>record</i>&& <i>start</i>&& TO <i>finish</i>&&}]</code>
Remarks	<p>UNLOCK restores access to a record, range of records, byte, or range of bytes locked by the LOCK statement, in file opened as file number <i>filenum</i>&.</p> <p>If the file was opened in random-access mode, <i>record</i>&&, <i>start</i>&&, and <i>finish</i>&& specify record numbers.</p> <p>When used with binary mode files, <i>record</i>&&, <i>start</i>&&, and <i>finish</i>&& specify byte positions, starting from either one (the default) or zero, depending on the BASE setting given when the file was Opened.</p> <p>If a record is specified, only that record (or byte) is unlocked. Otherwise, a range of records (or bytes) is unlocked, from <i>start</i>&& to <i>finish</i>&&. If no records are specified, or if the file was opened in sequential mode, the entire file is unlocked.</p> <p>All records (or bytes) to be unlocked must have been previously locked using the LOCK statement. Multiple locks may be placed on a file, and locks may be unlocked in any order. However, the parameters used for each UNLOCK statement must exactly match those used for the previous corresponding LOCK statement.</p> <p style="text-align: center;">All locked records (or bytes) must be unlocked using the UNLOCK statement before the file can be closed.</p> <p>If an unlock attempt fails, Classic PowerBASIC sets the ERR system variable to reflect a run-time Error 70 ("Permission denied"), or Error 75 ("Path/file access error").</p>
See also	LOCK , OPEN
Example	See the example for LOCK .

USING\$ function

Purpose	Format one or more string or numeric expressions, based upon the contents of the format mask string.
Syntax	<code>sResult\$ = USING\$(fmtmask\$, expr [, expr [, ...]])</code>
Remarks	<p>The rules of formatting are based upon the PRINT USING statement supported in many DOS versions of BASIC, including Classic PowerBASIC for DOS.</p> <p>However, since it is implemented as a function, it allows far more versatility in that it is not necessary to output a value to gain the benefit of this unique functionality. Also, USING\$ offers a wider range of applications than FORMAT\$ because it can format both numeric and string expressions, and can take multiple arguments.</p>
fmtmask\$	A string expression , string variable or string literal consisting of format characters that will determine how the complete expression should be formatted. This expression is termed the <i>mask</i> . There may be as many format masks in <i>fmtmask\$</i> , arranged in the same order as the <i>expr</i> arguments are specified. See the examples below for more information.
expr	A string or numeric expression, variable, or literal value to be formatted. The mask characters available depend on whether <i>expr</i> is a string or numeric.

Character	Definition
(string expr)	When <i>expr</i> is a string, the following format codes apply within <i>fmtmask\$</i>:
!	The first character of the string is returned.
&	The entire string is returned.
\\	The first two characters are returned.
\\	If backslashes enclose <i>n</i> spaces, <i>n</i> + 2 characters of the string expression are returned.
_	Escape (underscore) character. The following character is interpreted as a literal character instead of a mask format character.
(numeric expr)	When <i>expr</i> is numeric, the following format codes apply within <i>fmtmask\$</i>:
#	A numeric digit position, which is space-filled to the left, and zero-filled to the right of the decimal point. If the number is

	negative, a minus sign occupies a digit position.
.	The decimal point is placed at this position.
,	A numeric digit position, which signifies that whole number digits should be displayed with a comma each three digits.
\$\$	Two numeric digit positions which cause a dollar sign to be inserted immediately before the number.
x	Two numeric digit positions which cause leading blank spaces in the field to be replaced with the character in the second position of the pair "x" (where "x" represents your own choice of character). For example, two asterisks "" will convert leading spaces to asterisks, and "*=" converts leading spaces to equals characters, etc. The *x mask characters also act as two digit (#) placeholders. Your mask must contain at least three characters to use this.
+	A plus at the start of the field causes the sign of the value (+ -) to be inserted before the number. A plus at the end of the field causes the sign of the value (+ -) to be added after the number.
-	A minus at the end of the field causes a minus sign to be added after a negative number, or a space to be added after a positive number. A minus at the start of the field is treated as a literal character, which is always inserted.
^	Numbers can be formatted in scientific notation by including three to six carets (^) in the format string. Each caret corresponds to a numeric digit position in the exponent, one for E, one for the exponent sign, and one to four for the actual digits of the exponent value.
_	Escape (underscore) character. The following character is interpreted as a literal character instead of a mask format character. Therefore, to include a literal underscore character in the format mask, use two underscore characters.
	All characters in the format mask string that are not identified above are copied into the output string just as they are encountered. You can override or <i>escape</i> any special format code by preceding it with an underscore character (_) and it will be copied as any other literal character. This provides the flexibility to include literal string text within the formatted return string.

Restrictions The returned string is limited to an absolute length limit of 1024 [bytes](#). By specifying a single mask in `fmtmask$`, all *expr* arguments are subjected to the single mask. See the examples below.

If there are fewer *expr* arguments than matching format masks in `fmtmask$`, parsing of the *fmtmask\$* halts after the last referenced mask position, and subsequent characters in *fmtmask\$* are ignored. This is consistent with the behavior of `PRINT USING$` in PB/DOS.

If a numeric argument overflows its mask (i.e., there are more digits than digit positions), the resulting string will occupy as many spaces as needed to represent the number. In such cases, PB/DOS includes a leading "%" symbol to indicate the mask overflow; however, Classic PowerBASIC for Windows does not return the additional "%" overflow character.

The semicolon (;) and zero (0) characters are reserved for future use, so it would be prudent to escape such literal characters in `USING$` masks to maintain future compatibility.

See also [GRAPHIC PRINT](#), [XPRINT](#), [FORMAT\\$](#), [STR\\$](#)

Example

```
a$ = USING$("!", "abc")
' returns "a"

a$ = USING$("You owe $$#, .##", 12345.67@)
' returns "You owe $12,345.67"

DIM p AS BYTE PTR
HOST ADDR "localhost" TO ip&
p = VARPTR(ip&)
a$ = USING$("#_#_#_#", @p, @p[1], @p[2], @p[3])
' returns "127.0.0.1"

a$ = USING$("&=#####", "Pi", ATN(1)*4)
' returns "Pi=3.14159265358979"

a$ = USING$("!", "AX", "BX", "CX")
' returns "ABC"

a$ = USING$("$#.##_", 1, 20, 300, 4)
' returns "$1.00,$20.00,$300.00,$4.00,"

a$ = USING$("$*=#####.##_", 1, 20)
' returns "$=====1.00,$=====20.00,"
```

Purpose Return the numeric equivalent of a string argument.

Syntax `y = VAL(string_expression)`

Remarks VAL turns a string argument into a number. If *string_expression* begins with numeric characters (0 to 9 and +, -, or.), but also contains non-numeric characters, VAL returns the number up to the point of the non-numeric character. If *string_expression* does not begin with a numeric character, VAL returns 0. The string argument should not contain any commas, as VAL terminates processing if a comma is encountered. Leading white-space characters (spaces, tabs, and linefeed characters) are ignored. VAL is often used in data entry routines. With it, a program can prompt the user for numeric data (which is usually retrieved in string form), and then convert the legal portions of the string into a numeric result (see the examples below).

VAL also interprets the letters "e" and "d" (and "E" and "D") as the symbols for exponentiation. Both "e" and "d" are supported for compatibility with other BASIC languages, but there is no difference in operation between these symbols. For example:

```
i& = VAL("10.101e3") ' 10101 ~ 10.101*(10^3)
j& = VAL("2D4")      ' 20000 ~ 2 * (10 ^ 4)
```

Hexadecimal, Binary and Octal conversions

VAL can also be used to convert string arguments that are in the form of integer class Hexadecimal, Binary and Octal numbers. Hexadecimal values should be prefixed with "&H" and Binary with "&B". Octal values may be prefixed "&O", "&Q" or just "&". If *string_expression* contains a leading zero, the result is returned as an unsigned value; otherwise, a signed value is returned. For example:

```
i& = VAL("&HF5F3")      ' Hex, returns -2573 (signed)
j& = VAL("&H0F5F3")      ' Hex, returns 62963 (unsigned)
x& = VAL("&B0100101101") ' Binary, returns 301 (unsigned)
y& = VAL("&O4574514")    ' Octal, returns 1243468 (signed)
```

Valid hex characters include 0 to 9, A to F (and a to f). Valid Octal characters include 0 to 7, and binary 0 to 1.

Use the [STR\\$](#) and [FORMAT\\$](#) functions to convert numeric values into decimal strings. Use [BIN\\$](#), [HEX\\$](#) and [OCT\\$](#) to convert numeric values into string representations of Binary, Hexadecimal and Octal number system values.

Restrictions VAL stops analyzing *string_expression* when non-numeric characters are encountered. When dealing with Hexadecimal, Binary, and Octal number systems, the period character is classified as non-numeric. This is because

Classic PowerBASIC only supports floating-point formats for the decimal number system. VAL accepts the period character as a decimal place for all decimal number system values.

VAL does not analyze trailing type-specifiers for decimal strings. For example, VAL("9.1&") is evaluated as 9.1 rather than 9 because the "&" suffix is treated as a non-numeric character, not a [type-specifier](#). However, type suffixes may be used with binary, octal, and hex values.

See also

[BIN\\$](#), [FORMAT\\$](#), [HEX\\$](#), [OCT\\$](#), [STR\\$](#)

Example

```
Price$ = "$ 15,345.92"  
Cost@@ = VAL(REMOVE$(Price$, ANY "$, "))  
15345.92
```

Result

VARIANT# function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the numeric value contained in a Variant variable .
Syntax	<i>numericvar</i> = VARIANT#(<i>vrntvar</i>)
Remarks	<p>The value returned by VARIANT# may be any range from BYTE to DOUBLE/QUAD/CURRENCY, depending upon the internal representation used within the Variant.</p> <p>While Variant variables, by definition, do not offer support for Extended Precision Float data types, you should note that it is possible for a QUAD or CURRENCY value to exceed the precision level offered by a DOUBLE. You should therefore use some judgement in deciding on the numeric variable type to be used as the destination of this function, based upon the expected return values, and the internal representation, which you can obtain with VARIANTVT.</p>
Restrictions	VARIANT# presumes that a valid numeric value is present (not an array); otherwise, the value zero is returned.
See also	DIM , LET , OBJECT , LET (with Variants) , VARIANT\$, VARIANTVT
Example	<pre>DIM vVnt AS VARIANT vVnt = 999& a& = VARIANT#(vVnt)</pre>

VARIANT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the dynamic string value contained in a Variant variable .
Syntax	<i>stringvar</i> = VARIANT\$(<i>vntvar</i>)
Remarks	VARIANT\$ automatically converts the Variant string from Wide/Unicode format to ANSI .
Restrictions	VARIANT\$ presumes that a valid string value is present (not an array); otherwise, an empty string is returned.
See also	DIM , LET , OBJECT , LET (with Variants) , VARIANT# , VARIANTVT
Example	<pre>DIM vVnt AS VARIANT vVnt = "Hello World" a\$ = VARIANT\$(vVnt)</pre>

Purpose Determine the internal data type of the data stored in a [Variant variable](#).

Syntax `numericvar = VARIANTVT(vrntvar)`

Remarks The VARIANTVT function returns the internal VT data type stored in the Variant. The entire range of %VT_ prefixed values are documented by the OLE ([COM](#)) specification and are available in WIN32API.INC.

The most important values in this limited context include %VT_EMPTY (=0) and %VT_BSTR (=8), since most of the others are numeric formats automatically resolved by the [LET \(with Variants\)](#) statement and VARIANT# function.

Result	Equate	Content Type
0	%VT_EMPTY	An Empty Variant
1	%VT_NULL	Null value
2	%VT_I2	Integer
3	%VT_I4	Long-Integer
4	%VT_R4	Single
5	%VT_R8	Double
6	%VT_CY	Currency
7	%VT_DATE	Date
8	%VT_BSTR	Dynamic String
9	%VT_DISPATCH	IDispatch
10	%VT_ERROR	Error Code
11	%VT_BOOL	Boolean
12	%VT_VARIANT	Variant
13	%VT_UNKNOWN	IUnknown
14	%VT_DECIMAL	Decimal
16	%VT_I1	Byte (signed)
17	%VT_UI1	Byte (unsigned)
18	%VT_UI2	Word
19	%VT_UI4	DWORD
20	%VT_I8	Quad (signed)
21	%VT_UI8	Quad (unsigned)

22	%VT_INT	Long-Integer
23	%VT_UINT	DWord
24	%VT_VOID	A C-style void type
25	%VT_HRESULT	COM result code
26	%VT_PTR	Pointer
27	%VT_SAFEARRAY	VB Array
28	%VT_CARRAY	A C-style array
29	%VT_USERDEFINED	User Defined Type
30	%VT_LPSTR	ANSI string
31	%VT_LPWSTR	Unicode string
64	%VT_FILETIME	A FILETIME value
65	%VT_BLOB	An arbitrary block of memory
66	%VT_STREAM	A stream of bytes
67	%VT_STORAGE	Name of the storage
68	%VT_STREAMED_OBJECT	A stream that contains an object
69	%VT_STORED_OBJECT	A storage object
70	%VT_BLOB_OBJECT	A block of memory that represents an object
71	%VT_CF	Clipboard format
72	%VT_CLSID	Class ID
&H1000	%VT_VECTOR	An array with a leading count
&H2000	%VT_ARRAY	Array
&H4000	%VT_BYREF	A reference value

If a Variant contains a complete [array](#), the Variant type is determined by adding the base type to the array modifier. That is, for a [string array](#), it would be %VT_BSTR plus %VT_ARRAY (= &H2008).

[Quad](#) arrays within Variants are not supported by most versions of Windows. The result from VARIANTVT can be used to see whether such an array was created properly.

See also

[DIM](#), [Just what is COM?](#), [OBJECT](#), [LET \(with Variants\)](#), [VARIANT#](#), [VARIANT\\$](#), [What is an object, anyway?](#)

Purpose	Return the 32-bit address of a variable .
Syntax	<code>y = VARPTR(variable)</code>
Remarks	<p>VARPTR returns a complete 32-bit address to the specified variable as a Double-word (DWORD) value. variable is any numeric, string, structure variable (User-Defined Type or Union), or element of an array. VARPTR returns a pointer (32-bit address in memory) where the variable data is stored.</p> <p>VARPTR may also be used to locate an array descriptor, as well as the array data itself. To find the address of an array descriptor, use the array name with empty parentheses: VARPTR(x()).</p> <p>When you use VARPTR to get the address of a dynamic (variable length) string, keep in mind that the value being returned is the address of the string <i>handle</i>, not the actual <i>data</i> in the string. This can be useful for manipulating a dynamic string array using indexed-pointers, For example:</p> <pre>DIM A\$(100), b\$, pA AS STRING PTR, x& ' Assume A\$() is filled here pA = VARPTR(a\$(0)) ' 1st element handle FOR X& = 0 TO 100 B\$ = B\$ + @pA[x&] + ", " NEXT x&</pre> <p>You can use STRPTR to find the address of the string's data. When used with pointers, VARPTR returns the address of the pointer itself.</p>
Restrictions	<p>VARPTR cannot be used on Register variables, because Register variables are stored in internal processor registers rather than application memory. VARPTR can be used on UDT and Union variables, but not the UDT definition name. For example:</p> <pre>TYPE MyType ABC AS LONG END TYPE ' more code here DIM x AS MyType, y& y& = VARPTR(x) ' This is legal y& = VARPTR(MyType) ' This is not</pre>
See also	CODEPTR , PEEK , Pointers , POKE , STRPTR
Example	<pre>DIM x AS INTEGER PTR, a%, b% a% = 55 x = VARPTR(A%) b% = @x CALL DisplayResult("b% contains " + FORMAT\$(b%))</pre>
Result	b% contains 55

VERIFY function

[Top](#) [Previous](#) [Next](#)

Purpose	Determine whether each character of a string is present in another string.
Syntax	<code>x = VERIFY([start&], MainString, MatchString)</code>
Remarks	<p>VERIFY returns zero if each character in <i>MainString</i> is present in <i>MatchString</i>. If not, it returns the position of the first non-matching character in <i>MainString</i>.</p> <p>This function is very useful for determining if a string contains only numeric digits, for example.</p> <p>VERIFY is case-sensitive, so capitalization matters.</p>
Restrictions	If <i>start&</i> evaluates to a position outside of the string on either side, or if <i>start&</i> is zero, VERIFY returns zero.
See also	INSTR , LCASE\$, LTRIM\$, MID\$, REMOVE\$, REPLACE , RIGHT\$, RTRIM\$, TALLY , TRIM\$, UCASE\$
Example	<pre>' returns 4 since "." is not in "0123456789" x& = VERIFY("123.65,22.5", "0123456789") ' returns 7 since 5 starts it past the first non-digit ("." at position 4) x& = VERIFY(5,"123.65,22.5", "0123456789")</pre>

Purpose Manipulate a Window in the program, which may include setting or retrieving data. The target window may be of any class, including a Control or [Dialog](#).

Syntax `WINDOW GET ID hWin TO datav&`
`WINDOW GET PARENT hWin TO datav&`

hWin Handle of the Window to be used.

datav& A [long integer](#) variable to which result data is assigned.

Remarks The WINDOW statement may be used with any type of window in your program, including a Control or Dialog. Generally speaking, the window to be manipulated or tested is identified by its handle (*hWin*), which is often obtained at the time it is created. However, since a control is usually accessed by a "Parent / ID" combination, you must use [CONTROL HANDLE](#) to retrieve its handle for this purpose.

WINDOW GET ID

hWin TO *datav&*

The integer ID of the window *hWin* is retrieved and assigned to the variable designated by *datav&*. Generally, only a CONTROL will have an ID, so windows of other classes will normally return the value zero.

WINDOW GET PARENT *hWin* TO *datav&*

The handle of the PARENT is retrieved and assigned to the variable designated by *datav&*.

See also [CONTROL HANDLE](#)

Purpose	Define a block of program statements that are executed repeatedly for as long as certain conditions are met.
Syntax	<pre>WHILE <i>integer_expression</i> [<i>statements</i>] [EXIT LOOP] [<i>statements</i>] WEND</pre>
Remarks	<p>If <i>integer_expression</i> is TRUE (it evaluates to a non-zero value), all of the statements between the WHILE and the terminating WEND are executed. Classic PowerBASIC then jumps back to the WHILE statement and repeats the test. If it is still TRUE, Classic PowerBASIC executes the enclosed statements again. This process is repeated until the test expression evaluates to zero, or an EXIT statement is encountered. In either case, execution passes to the statement following WEND.</p> <p>If <i>integer_expression</i> evaluates to FALSE (zero) on the first pass, none of the statements in the loop are executed.</p> <p>Loops built with WHILE/WEND statements can be nested (enclosed within each other). Each WEND matches the most recent unmatched WHILE.</p> <p>One use of a WHILE/WEND loop is to input data from a file until the end of the file is reached:</p> <pre>i% = 0 WHILE ISFALSE EOF(1) INCR i% LINE INPUT #1, FileTxt\$(i%) WEND</pre> <p>Although the compiler does not care, it's a good idea to indent the statements between WHILE and WEND, to clarify the structure of the loop you have constructed.</p> <p>Note that the following code creates an infinite loop:</p> <pre>WHILE -1 [<i>statements</i>] WEND</pre> <p>To exit a WHILE/WEND loop prematurely, use the EXIT LOOP statement. Classic PowerBASIC's DO/LOOP construct offers a more flexible way to build conditional loops.</p> <p>Also see the discussion on the IF statement for notes on Classic PowerBASIC's Short-circuit evaluation and its possible side effects.</p>
See also	#OPTIMIZE , DO/LOOP , EXIT , FOR/NEXT , ITERATE , Short-circuit evaluation

Purpose	WINMAIN (or its synonym MAIN) is a user-defined function called by Windows to begin execution of an application.
Syntax	<pre>FUNCTION {WINMAIN MAIN} (_ BYVAL <i>hInstance</i> AS DWORD, _ BYVAL <i>hPrevInst</i> AS DWORD, _ BYVAL <i>lpszCmdLine</i> AS ASCIIZ PTR, _ BYVAL <i>nCmdShow</i> AS LONG) AS LONG</pre>
Remarks	The WINMAIN function is called by Windows when an executable application first loads and begins to run. It is often referred to as the "entry point" for the application. When the execution of WINMAIN is completed, the application is deemed to be finished, and Windows releases the application memory back to the heap. WINMAIN receives the following parameters:
<i>hInstance</i>	The executable's (EXE) <i>instance handle</i> . Each instance of a Windows application has a unique handle. It is used as a parameter to a number of Windows API functions which may need to distinguish between multiple instances of an application.
<i>hPrevInst</i>	Not used by 32-bit Windows. It is present merely for compatibility with existing 16-bit code, and always returns zero in 32-bit applications.
<i>lpszCmdLine</i>	A pointer to an ASCIIZ string that contains a command-line. Note that the string passed in <i>lpszCmdLine</i> is not the same as the string returned by the <i>GetCommandLine</i> API call. The string in <i>lpszCmdLine</i> contains the command-line arguments only (like COMMAND\$), but <i>GetCommandLine</i> returns the program name (including path) followed by the arguments.
<i>nCmdShow</i>	Specifies how to display the application's main window. For example, the calling application can specify %SW_NORMAL or %SW_MINIMIZE, etc. It is up to the programmer to honor this parameter, and to do so is recommended.
Return	<p>The return value assigned to WINMAIN is optional, but by convention, the return value is derived from the <i>wParam</i>& parameter of a %WM_QUIT message.</p> <p>Typically, a GUI-based application uses the WINMAIN function to create the initial GUI application window, and then enters a message loop. This loop should terminate when a %WM_QUIT message is received, and the <i>wParam</i>& parameter of that message should be passed on as the return value for WINMAIN. If WINMAIN terminates before entering the message loop, WINMAIN should return zero.</p> <p>Console applications may use the return value to set an error level that can</p>

be passed back to the calling application, in the range 0 to 255 inclusive. Batch files may act on the result through the IF [NOT] ERRORLEVEL batch command.

If the parameters passed to WINMAIN are not required by the application itself, the [PBMAIN](#) function may be used in place of WINMAIN.

Restrictions [Pointers](#) may not be passed [BYREF](#), so the *lpCmdLine* parameter of WINMAIN must be declared to be passed [BYVAL](#).

See also [PBMAIN](#)

Example

```
#COMPILE EXE
FUNCTION WINMAIN(BYVAL hInst???, BYVAL hPrevInst???, BYVAL pCmdLine AS
ASCIIZ PTR, BYVAL nCmdShow&) AS LONG
    ' more code here
    FUNCTION = 1
END FUNCTION
```

Purpose	Output data to a sequential file in a delimited format.
Syntax	<pre>WRITE #filenum& WRITE #filenum&, [expression [{; ,} expression] ...] [; ,]</pre>
<i>filenum&</i>	The file number used when the file or device was opened.
<i>expression</i>	A string or a string expression representing the data to be written to the file or device.
Remarks	<p>WRITE# is similar to PRINT#, except WRITE# inserts a comma in the output file between each expression. It encloses numeric data within quotation marks, and adds no leading or trailing spaces around numeric values.</p> <p>WRITE# is the preferred method of writing data to a sequential file, since it formats the output to be readable by the INPUT# statement. In other words, INPUT# respects the delimiter characters that separate items in a line of text, as created by WRITE#.</p> <p>WRITE# with a file number and a comma but no expressions, outputs a carriage return to the file.</p> <p>To read a delimited file without regard to the delimiter characters, use the LINE INPUT# statement.</p>
Restrictions	<p>For best results, strings should not contain quotation marks, as these may interfere with the expected output format.</p> <p>Each expression in the WRITE# statement must be separated from other expressions by a comma or semicolon. If you include a trailing comma or semicolon, the final carriage return / line feed is suppressed and replaced with a comma delimiter. This allows you to append data to the sequential record by executing another WRITE statement.</p>
See also	GET , GET\$, INPUT# , LINE INPUT# , OPEN , PRINT# , PUT , PUT\$, SETEOF
Example	<pre>' Open a sequential output file and write to it OPEN "FILE.TXT" FOR OUTPUT AS #1 WRITE #1, "TEST" z& = -12345& info1\$ = "Do not covet" info2\$ = "thy neighbors ox" WRITE #1, z&, info1\$, info2\$ WRITE #1, "TEST" CLOSE #1</pre>
Result	<pre>"TEST" -12345,"Do not covet","thy neighbors ox" "TEST"</pre>

Purpose

The XOR operator works as both a logical and a bitwise [arithmetic operator](#).

Syntax

$p \text{ XOR } q$

Remarks

XOR as a logical operator
XOR returns FALSE (zero) if *and only if* both its operands have the same value. Here is XOR's truth table:

Truth table		
x	y	x XOR y
T	T	F
T	F	T
F	T	T
F	F	F

See also

[Arithmetic Operators](#), [AND](#), [EQV](#), [IMP](#), [NOT](#), [OR](#)

XPRINT\$ function

[Top](#) [Previous](#) [Next](#)

Purpose	Return the name of the attached host printer .
Syntax	<i>device\$</i> = XPRINT\$
Remarks	XPRINT\$ returns the name of the attached host printer, which is the printer that would be used by XPRINT statements. If there is no attached host printer, an empty string is returned. XPRINT\$ is typically used to detect if an XPRINT ATTACH operation was successful.
See also	XPRINT ATTACH , XPRINT CLOSE

Purpose	Output text to a host-printer device.
Syntax	<code>XPRINT [<i>expression</i>] [;]</code>
Remarks	<p>The XPRINT functionality is similar to the PRINT statement, except that the data is sent to a host printer rather than to a display. A host printer is a Windows-Only printer which uses a device driver to form text and graphics. Classic PowerBASIC inserts a carriage return and linefeed at the end of each printed line. A semi-colon between expressions is an optional delimiter which leaves the printer column position unchanged. A trailing semi-colon suppresses the final CR/LF.</p> <p>Before you execute an XPRINT statement, you must explicitly connect to the intended host printer using the XPRINT ATTACH statement. If the connection to the device is unsuccessful, all XPRINT statements are ignored until a valid printer device has been attached.</p> <p>Once all the data has been sent to the printer, detach the printer so other applications can use it., with the XPRINT CLOSE statement.</p> <p>Host-based (Windows-only) printers use proprietary control protocols. Generally speaking, you should avoid the embedding of control codes like \$CR, \$LF, and \$FF in a string expression, as they may not produce the expected results. Instead, use XPRINT without a parameter to move to the next line, or XPRINT FORMFEED to start a new page. If the XPRINT operation is not successful, an appropriate error is generated.</p>
See also	FONT NEW , LPRINT , XPRINT ATTACH , XPRINT CHR SIZE , XPRINT COLOR , XPRINT CLOSE , XPRINT SET FONT , XPRINT GET POS , XPRINT SET POS , XPRINT TEXT SIZE
Example	<pre>' Typical XPRINT printing strategy ERRCLEAR XPRINT ATTACH DEFAULT ' Use default printer IF ERR = 0 AND LEN(XPRINT\$) > 0 THEN XPRINT "This is your printer talking" XPRINT FORMFEED ' Issue a formfeed XPRINT CLOSE ' detach the printer END IF</pre>

Purpose	Draw an arc on a host printer page.
Syntax	<code>XPRINT ARC (x1!, y1!) - (x2!, y2!), arcStart!, arcEnd! [, rgbColor&]</code>
Remarks	<p>An arc is a section of a circle or an ellipse. To specify a particular arc, you would first define the full circle or ellipse of which it is a part, and then specify the points on the ellipse where the arc starts and stops.</p> <p>The full circle or ellipse is defined by its bounding rectangle, which is the smallest rectangle which can be drawn around the circle or ellipse. For example, if the circle is centered at position (400,400), with a radius of 100 pixels, the upper left corner (x1,y1) of the bounding rectangle is (300,300), and the lower right corner (x2,y2) is (500,500).</p> <p>The start point and end point of the arc are specified by their angle, which must be given in radians. A complete circle or ellipse is 2*pi radians. On a 12-hour clock-face, the values 0 and 2*pi both refer to the position of 3 o'clock, while the value 1*pi refers to the position of 9 o'clock. Other positions are specified by a radian value relative to these. In Classic PowerBASIC, arcs are always drawn counter-clockwise from the starting point to the ending point.</p> <p>Prior to any XPRINT operations, a host printer must first be selected with XPRINT ATTACH. The coordinate points are specified in pixels (or world coordinates, if those were chosen with XPRINT SCALE). Line width can be set using XPRINT WIDTH. If line width is set to 1 (the default), the line style can be set with XPRINT STYLE. Because of the nature of an arc, XPRINT ARC neither uses, nor updates, (last point referenced). If executed without a host printer attached, error 57 is generated.</p>
<i>x1!, y1!</i>	The upper left corner of the bounding rectangle of the full circle or ellipse.
<i>x2!, y2!</i>	The lower right corner of the bounding rectangle of the full circle or ellipse.
<i>ArcStart!</i>	The starting angle of the arc, in radians, from 0 to 2*pi.
<i>ArcEnd!</i>	The ending angle of the arc, in radians, from 0 to 2*pi radians. Note that arcs are always drawn counter-clockwise from <i>arcStart!</i> to <i>arcEnd!</i> . Compared with a 12-hour clock-face, 0 or 2*pi radians is at 3 o'clock, and 1*pi radians is at 9 o'clock.
<i>rgbColor&</i>	Optional RGB color for the arc. If omitted (or -1), the current foreground color for the host printer page is used.
See also	Built In RGB Color Equates , XPRINT ATTACH , XPRINT COLOR , XPRINT ELLIPSE , XPRINT PIE , XPRINT STYLE , XPRINT WIDTH

Example

```
' Draw two arcs that combine into a circle.
' The upper half uses the default foreground color.
' The lower half is drawn in red.
LOCAL Pi AS DOUBLE
Pi = 4 * ATN(1)                                ' Calculate Pi
XPRINT ARC (5, 5) - (105, 105), 0,  Pi        ' Upper half
XPRINT ARC (5, 5) - (105, 105), Pi, 0, %RED   ' Lower half
```

Purpose	Connect a host-based (GDI) printer for use with .
Syntax	<code>XPRINT ATTACH {CHOOSE DEFAULT <i>PrinterName\$</i>} [, <i>JobName\$</i>]</code>
Remarks	<p>XPRINT ATTACH connects to a host-based (Windows-only or GDI-based) printer for use with subsequent XPRINT operations. Host-based printing is device-independent and performed through the Windows printing system and printer driver. Device independence can be achieved because the printer driver handles the task of converting text into the manufacturers proprietary binary format used by the printer.</p> <p>To send device-dependent print data (such as plain text) to a line printer device, use the LPRINT ATTACH statement instead.</p> <p>XPRINT ATTACH allows you to change the printer device used by XPRINT operation. When executed, the current connection (if any) is closed and the new connection is established.</p>
DEFAULT	<p>If DEFAULT is specified, the default printer (as set in the Printers applet in Control Panel) is used. For example:</p> <pre>XPRINT ATTACH DEFAULT</pre>
CHOOSE	<p>If CHOOSE is specified, the Choose Printer common dialog is opened, allowing the user to select from the list of installed printers. For example:</p> <pre>XPRINT ATTACH CHOOSE</pre>
<i>PrinterName\$</i>	<p>The name of the printer to attach (as shown in the Printers applet in Control Panel, or returned by the PRINTERS\$ function). <i>printername\$</i> must be a valid device name and cannot exceed 259 characters in length. For example:</p> <pre>XPRINT ATTACH "HP LaserJet 5MP"</pre>
<i>JobName\$</i>	<p>The name of the print job. This will be shown in the print spooler. If you do not supply a name, "Printjob" is used by default.</p> <p>If XPRINT ATTACH is not successful, XPRINT\$ returns an empty string. Error 68 ("device unavailable") is generated if an invalid printer was specified. No error is generated if the user cancels the Choose Printer dialog (with XPRINT ATTACH CHOOSE). Therefore, for host-based printing, applications should always use XPRINT ATTACH to explicitly select the intended host-based printer, then test for a successful selection with the XPRINT\$ and ERR functions to ensure the host-based printer selection was successful.</p> <p>Unlike direct printing (LPRINT ATTACH), host-based printing is handled by a printer driver and the operating system's spooler subsystem. Therefore,</p>

spooler settings such as "work offline" in the Printer Properties dialog will not impede the creation of a spooled print job. Once all the data has been sent to the printer, detach the printer so other applications can use it., with the [XPRINT CLOSE](#) statement.

Host-based printers use proprietary control protocols, unlike line printers, so it is usually not possible to send them printer-dependent control codes. To attach a line printer, use LPRINT ATTACH instead of XPRINT ATTACH.

Note: You can enumerate the available printers with the [PRINTERCOUNT](#) and PRINTER\$ functions.

See also

[LPRINT ATTACH](#), [PRINTER\\$](#), [XPRINT CANCEL](#), [XPRINT CLOSE](#), [XPRINT\\$](#)

Example

```
ERRCLEAR
XPRINT ATTACH "HP DeskJet 960c"
IF ERR = 0 AND LEN(XPRINT$) > 0 THEN
  XPRINT COLOR RGB(0,0,255) ' Blue
  XPRINT "This is your printer talking"
  XPRINT FORMFEED           ' Issue a formfeed
  XPRINT CLOSE              ' Deselect the printer
END IF
```

Purpose	Draw a box with square or rounded corners on a host printer page.														
Syntax	<code>XPRINT BOX (x1!, y1!) - (x2!, y2!) [, [corner&] [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]]</code>														
Remarks	Prior to any XPRINT operations, a host printer must first be selected with XPRINT ATTACH . The coordinate points are specified in pixels (or world coordinates, if those were chosen with XPRINT SCALE). Line width can be set using XPRINT WIDTH . If line width is set to 1 (the default), the line style can be set with XPRINT STYLE . Because of the nature of a box, XPRINT BOX neither uses, nor updates, (last point referenced). If executed without a host printer attached, error 57 is generated.														
x1!, y1!	The upper left corner of the box.														
x2!, y2!	The lower right corner of the box.														
corner&	The percentage of roundness of the corners, in the range of 0 to 100. A value of zero creates square corners, while 100 creates a circle/oval. A value of 20 being most common for a pleasant, rounded appearance. If <i>corner&</i> is omitted, the default is 0, which creates a rectangle with square corners.														
rgbColor&	Optional RGB color of the box edge. If omitted (or -1), the edge color defaults to the current foreground color for the host printer page.														
fillcolor&	Optional RGB color of the box interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the box is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.														
fillstyle&	Optional fill style (pattern) to be used. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>fillcolor&</i> , while the background is specified by the default background color for the host printer page. The optional <i>fillstyle&</i> may be: <table><tr><td>0</td><td>Solid (default)</td></tr><tr><td>1</td><td>Horizontal Lines</td></tr><tr><td>2</td><td>Vertical Lines</td></tr><tr><td>3</td><td>Upward Diagonal Lines</td></tr><tr><td>4</td><td>Downward Diagonal Lines</td></tr><tr><td>5</td><td>Crossed Lines</td></tr><tr><td>6</td><td>Diagonal Crossed Lines</td></tr></table>	0	Solid (default)	1	Horizontal Lines	2	Vertical Lines	3	Upward Diagonal Lines	4	Downward Diagonal Lines	5	Crossed Lines	6	Diagonal Crossed Lines
0	Solid (default)														
1	Horizontal Lines														
2	Vertical Lines														
3	Upward Diagonal Lines														
4	Downward Diagonal Lines														
5	Crossed Lines														
6	Diagonal Crossed Lines														
See also	Built In RGB Color Equates , XPRINT ATTACH , XPRINT COLOR , XPRINT LINE , XPRINT STYLE , XPRINT WIDTH														

Example

' Draw rectangle with square corners and default colors.

```
XPRINT BOX (10, 10) - (100, 80)
```

' Draw a blue rectangle with 20% rounded corners,

' filled with a light-gray, diagonal cross pattern

```
XPRINT BOX (15, 15) - (95, 75), 20, %BLUE, RGB(191,191,191), 6
```

XPRINT CANCEL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Cancel a print job on the host printer .
Syntax	<code>XPRINT CANCEL</code>
Remarks	XPRINT CANCEL deletes the current print job and detaches the host printer, as long as XPRINT CLOSE has not yet been executed. This function is generally used to abort the print process when an error occurs.
See also	XPRINT ATTACH , XPRINT CLOSE

Purpose	Retrieve the character size for the current font on a host printer page.
Syntax	<code>XPRINT CHR SIZE TO <i>ncWidth!</i>, <i>ncHeight!</i></code>
Remarks	<p>The character size is specified in pixels (or world coordinates, if they have been defined with an XPRINT SCALE statement).</p> <p>If the font is a fixed-width font, like Courier New or Lucida Console, the sizes returned are as exact as possible, given the rounding errors possible when converting from pixels to other coordinates. If the font is proportional, like Arial or Times New Roman, the width will be the average for the entire font.</p> <p>If a host printer is not attached, error 57 is generated.</p>
See also	XPRINT , XPRINT ATTACH , XPRINT SET FONT , XPRINT TEXT SIZE

XPRINT CLOSE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Detach a host printer so printing may begin.
Syntax	<code>XPRINT CLOSE</code>
Remarks	XPRINT CLOSE detaches the printer from the current process, and allows printing to a HOST printer to begin. If XPRINT CLOSE is not executed, printed data may be lost.
See also	XPRINT ATTACH , XPRINT CANCEL

Purpose	Set the foreground color (and, optionally, the background color) for various XPRINT statements.
Syntax	<code>XPRINT COLOR <i>foreground&</i> [, <i>background&</i>]</code>
Remarks	<p>Colors are expressed as RGB values, or use -1 for the default color. If the background parameter is -2, the background is not painted, allowing the content behind to become visible. Otherwise, the specified RGB color is used.</p> <p>A host printer must first be connected with XPRINT ATTACH. If a host printer is not attached, error 57 is generated.</p>
See also	Built In RGB Color Equates , XPRINT , XPRINT ATTACH
Example	<pre>' Set colors to red foreground and blue background. XPRINT COLOR %RED, RGB(0,0,191)</pre>

Purpose Copy a bitmap to a [host printer](#) page.

Syntax

```
XPRINT COPY hbmpSource???, id& [, style&]  
XPRINT COPY hbmpSource???, id& TO (x!, y!) [, style&]  
XPRINT COPY hbmpSource???, id&, (x1!, y1!)-(x2!, y2!) TO (x!, y!) [, style%]
```

Remarks You can copy a complete bitmap, or a portion of it, to the host printer page. The expression *hbmpSource*??? specifies the handle of the source bitmap or window. The expression *id*& is the unique control identifier in the range 1 to 65535, as assigned with the [CONTROL ADD GRAPHIC](#) statement. *id*& must be 0 for a [GRAPHIC WINDOW](#) or a GRAPHIC BITMAP. The destination of the copy operation is the host printer page. You must use care that your parameters are valid for the specified bitmaps, or results of the operation are undefined.

The first form of the XPRINT COPY statement copies the complete bitmap, positioning it at (0,0), which is the upper left corner of the destination.

The second form of XPRINT COPY also copies the complete bitmap, but positions it at the point specified by the parameter (*x*!, *y*!).

The third form copies a portion of the bitmap, positioning it at the point specified by the parameter (*x*!, *y*!). If *style*& is included, it is one of the following values:

%mix_NotMergeSrcPixel is the inverse of the MergeSrc color.

%mix_NotCopySrc Pixel is the inverse of the pen color.

%mix_MaskSrcNot Pixel is a combination of the colors common to both the source and the inverse of the pixel.

%mix_XorSrc Pixel is a combination of the colors in the source and in the pixel, but not in both.

%mix_MaskSrc Pixel is a combination of the colors common to both the source and the pixel.

%mix_MergeNotSrcPixel is a combination of the source color and the inverse of the pixel color.

%mix_CopySrc Pixel is the source color (default).

%mix_MergeSrc Pixel is a combination of the source color and the pixel color.

A host printer must first be connected with [XPRINT ATTACH](#). If a host printer is not attached, [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT RENDER](#), [XPRINT STRETCH](#)

Purpose	Draw an ellipse or a circle on a host printer page.														
Syntax	<code>XPRINT ELLIPSE (x1!, y1!) - (x2!, y2!) [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]</code>														
Remarks	<p>A host printer must first be connected with XPRINT ATTACH. The coordinate points are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). Line width can be set using XPRINT WIDTH. If line width is set to 1 (the default), the line style can be set with XPRINT STYLE. Because of the nature of an ellipse, which has no obvious beginning or end, XPRINT ELLIPSE neither uses, nor updates, the last point referenced (POS). If executed without a host printer attached, error 57 is generated.</p> <p><i>x1!, y1!</i> The upper left corner of the bounding rectangle.</p> <p><i>x2!, y2!</i> The lower right corner of the bounding rectangle.</p> <p><i>rgbColor&</i> Optional RGB color of the ellipse edge. If omitted (or -1), the edge color defaults to the current foreground color for the host printer page.</p> <p><i>fillcolor&</i> Optional RGB color of the ellipse interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the ellipse is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.</p> <p><i>fillstyle&</i> Optional fill style (pattern) to be used. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>fillcolor&</i>, while the background is specified by the default background color for the host printer page. The optional <i>fillstyle&</i> may be:</p> <table><tr><td>0</td><td>Solid (default)</td></tr><tr><td>1</td><td>Horizontal Lines</td></tr><tr><td>2</td><td>Vertical Lines</td></tr><tr><td>3</td><td>Upward Diagonal Lines</td></tr><tr><td>4</td><td>Downward Diagonal Lines</td></tr><tr><td>5</td><td>Crossed Lines</td></tr><tr><td>6</td><td>Diagonal Crossed Lines</td></tr></table>	0	Solid (default)	1	Horizontal Lines	2	Vertical Lines	3	Upward Diagonal Lines	4	Downward Diagonal Lines	5	Crossed Lines	6	Diagonal Crossed Lines
0	Solid (default)														
1	Horizontal Lines														
2	Vertical Lines														
3	Upward Diagonal Lines														
4	Downward Diagonal Lines														
5	Crossed Lines														
6	Diagonal Crossed Lines														
See also	Built In RGB Color Equates , XPRINT ARC , XPRINT ATTACH , XPRINT COLOR , XPRINT LINE , XPRINT PIE , XPRINT STYLE , XPRINT WIDTH														
Example	<pre>' Draw a circle, using default colors. XPRINT ELLIPSE (10, 10) - (100, 100) ' Draw a blue ellipse filled with a light-gray,</pre>														

```
' diagonal cross pattern.  
XPRINT ELLIPSE (15, 25) - (95, 50), %BLUE, RGB(191,191,191), 6
```

Purpose	<p>Select a font to be used by the XPRINT statement.</p> <p>XPRINT FONT has been superceded by the XPRINT SET FONT statement, although XPRINT FONT remains supported for a limited period. Existing code should be converted to the new syntax as soon as possible.</p>
Syntax	<pre>XPRINT FONT <i>fontname\$</i> [,<i>points&</i>, <i>style&</i>]</pre>
<i>fontname\$</i>	Name of the font.
<i>points&</i>	Size of the font, in points.
<i>style&</i>	<p>Font style attribute. Any of the following values can be combined or used alone:</p> <ul style="list-style-type: none">0 = normal1 = bold2 = italic4 = underline8 = strikeout <p>For example, a <i>style&</i> value of 3 specifies a combination of both bold and italic attributes.</p>
Remarks	<p>If the requested font is not available on the computer, Windows will search for a substitute font, which is similar to the attributes specified (CharSet, Font Family, etc.).</p> <p>You may use the value zero (0) for any of the numeric parameters to designate that the compiler should use the default for that item. If parameter(s) are missing, the compiler substitutes the default value for all remaining parameters.</p> <p>Prior to any XPRINT operations, a host printer must first be selected with XPRINT ATTACH. If no XPRINT FONT statement is executed, the default font is Courier New with no style attributes. If XPRINT FONT is unsuccessful, an error is generated.</p>
See also	FONT NEW , XPRINT , XPRINT ATTACH , XPRINT CHR SIZE , XPRINT SET FONT , XPRINT TEXT SIZE
Example	<pre>' Set the font for the host printer to: ' to Times New Roman, 18 points, bold + italic + underline (1+2+4). XPRINT FONT "Times New Roman", 18, 7</pre>

XPRINT FORMFEED statement

[Top](#) [Previous](#) [Next](#)

Purpose	Start a new page for the host printer .
Syntax	<code>XPRINT FORMFEED</code>
Remarks	XPRINT FORMFEED causes the current print page to be ejected, and a new page started. If XPRINT FORMFEED is unsuccessful, an error is generated. Note that some printers do not eject a page if it is blank.
See also	XPRINT ATTACH , XPRINT CLOSE

Purpose	Retrieve the size of the client area (printable area) on the host printer page.
Syntax	<code>XPRINT GET CLIENT TO <i>ncWidth!</i>, <i>ncHeight!</i></code>
Remarks	Sizes are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). If executed without a host printer attached, error 57 is generated.
See also	XPRINT ATTACH , XPRINT GET MARGIN , XPRINT GET PPI , XPRINT GET SIZE

XPRINT GET COLLATE statement New! [Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the XPRINT collate status .
Syntax	<code>XPRINT GET COLLATE TO <i>collatestatus</i>&</code>
Remarks	<p>XPRINT allows you to set the collate status, if the printer driver supports both multiple copies and collate capability. XPRINT GET COLLATE retrieves the collate status, assigning the value to the long integer variable specified by <i>collatestatus</i>&. The following equates are predefined in the compiler to symbolically represent the possible collate status:</p> <pre>%DMCOLLATE_FALSE = 0 %DMCOLLATE_TRUE = 1</pre> <p>If this statement is executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT SET COLLATE

XPRINT GET COLORMODE statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose	Retrieve the XPRINT colormode status .
Syntax	<code>XPRINT GET COLORMODE TO <i>colormode&</i></code>
Remarks	<p>XPRINT allows you to set the color or monochrome print mode if the printer driver supports it. XPRINT GET COLORMODE retrieves the colormode status, assigning the value to the long integer variable specified by <i>colormode&</i>. The value zero may be returned if colormode is not supported by the printer driver. The following equates are predefined in the compiler to symbolically represent the possible status:</p> <pre>%DMCOLOR_MONOCHROME = 1 %DMCOLOR_COLOR = 2</pre> <p>If this statement is executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT SET COLORMODE

XPRINT GET COPIES statement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the XPRINT copy count .
Syntax	<code>XPRINT GET COPIES TO <i>copycount</i>&</code>
Remarks	<p>XPRINT allows you to set the number of copies to be automatically printed, if it is supported by the printer driver. XPRINT GET COPIES retrieves the copy count, assigning the value to the long integer variable specified by <i>copycount</i>&. The default value is one (1). If this statement is executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT SET COPIES

XPRINT GET DC statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the handle of the device context (DC) for the host printer page.
Syntax	<code>XPRINT GET DC TO <i>hDC</i>???</code>
Remarks	If no host printer is currently attached , zero is returned. The DC handle may be used with various Windows API functions to perform specialized operations on the host printer page.
See also	XPRINT ATTACH

Purpose Retrieve the XPRINT [duplex status](#).

Syntax XPRINT GET DUPLEX TO *duplexstatus&*

Remarks XPRINT allows you to set the duplex status, if the printer supports printing on both sides of a page. XPRINT GET DUPLEX retrieves the duplex status, assigning the value to the [long integer](#) variable specified by *duplexstatus&*. The following equates are predefined in the compiler to symbolically represent the possible duplex status:

%DMDUP_SIMPLEX	= 1	(single sided printing)
%DMDUP_VERTICAL	= 2	(page flipped on the vertical edge)
%DMDUP_HORIZONTAL	= 3	(page flipped on the horizontal edge)

If the printer does not support duplex printing, the value zero (0) is returned. If this statement is executed without a host printer [attached](#), [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT SET DUPLEX](#)

XPRINT GET LINES statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the number of lines that can be printed.
Syntax	<code>XPRINT GET LINES TO <i>linecount</i>&</code>
Remarks	<p>XPRINT GET LINES retrieves the number of lines of text which can be printed on the host printer page, given the current selected font. Since XPRINT statements do not generate an automatic formfeed when text is printed on the last line, this statement can be used to determine when your program should execute an XPRINT FORMFEED to move to the next printed page on a host printer. If executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT , XPRINT ATTACH , XPRINT SET FONT , XPRINT FORMFEED

Purpose	Retrieve the margin sizes for the host printer .
Syntax	<code>XPRINT GET MARGIN TO <i>nLeft!</i>, <i>nTop!</i>, <i>nRight!</i>, <i>nBottom!</i></code>
Remarks	<p>XPRINT GET MARGIN retrieves the size of the margins (the non-printable area) of the printer page. This is important because some printers do not provide equal margins on each side of the page. This is more common on the vertical coordinate, but could be found in either or both directions. The size of the four margins are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). If executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT GET CLIENT , XPRINT GET PPI , XPRINT GET SIZE

Purpose Retrieve the color mix mode for a [host printer](#) page.

Syntax `XPRINT GET MIX TO mode&`

Remarks Prior to any XPRINT operations, a host printer must first be selected with [XPRINT ATTACH](#). There are 16 mix modes available to use for mixing the drawing color with the color that already exists at the drawing location. The mix mode equates are predefined in Classic PowerBASIC. If executed without a host printer attached, [error 57](#) is generated.

`%mix_Blackness` Pixel is always 0 (black).

`%mix_NotMergeSrc` Pixel is the inverse of the MergeSrc color.

`%mix_MaskNotSrc` Pixel is a combination of the colors common to both the pixel and the inverse of the source.

`%mix_NotCopySrc` Pixel is the inverse of the pen color.

`%mix_MaskSrcNot` Pixel is a combination of the colors common to both the source and the inverse of the pixel.

`%mix_Not` Pixel is the inverse of the pixel color.

`%mix_XorSrc` Pixel is a combination of the colors in the source and in the pixel, but not in both.

`%mix_NotMaskSrc` Pixel is the inverse of the MaskSrc color.

`%mix_MaskSrc` Pixel is a combination of the colors common to both the source and the pixel.

`%mix_NotXorSrc` Pixel is the inverse of the XorSrc color.

`%mix_Nop` Pixel remains unchanged.

`%mix_MergeNotSrc` Pixel is a combination of the source color and the inverse of the pixel color.

`%mix_CopySrc` Pixel is the source color (default).

`%mix_MergeSrcNot` Pixel is a combination of the source color and the inverse of the pixel color.

`%mix_MergeSrc` Pixel is a combination of the source color and the pixel color.

`%mix_Whiteness` Pixel is always 1 (white).

See also [XPRINT ATTACH](#), [XPRINT SET MIX](#)

XPRINT GET ORIENTATION statement [Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the paper orientation for a host printer page.
Syntax	<code>XPRINT GET ORIENTATION TO <i>orient</i>&</code>
Remarks	XPRINT GET ORIENTATION retrieves the orientation of the paper in the host printer, assigning the value to the long integer variable specified by <i>orient</i> &. The value 1 indicates portrait mode, while 2 indicates landscape mode. If the printer does not support paper orientation, 0 is returned. If a host printer is not attached, error 57 is generated.
See also	XPRINT ATTACH , XPRINT SET ORIENTATION

Purpose

Retrieve the [current paper size/type](#).

Syntax

XPRINT GET PAPER TO *papertype*&

Remarks

XPRINT GET PAPER retrieves the paper style for which the [host printer](#) is currently configured. The paper style is identified by an integral value which is assigned to the [long integer](#) variable specified by papertype&. The following equates are predefined in the compiler, and represent the most common paper styles:

%DMPAPER_LETTER	=	1	Letter	8.5	x	11	inches
%DMPAPER_TABLOID	=	3	Tabloid	11	x	17	inches
%DMPAPER_LEDGER	=	4	Ledger	17	x	11	inches
%DMPAPER_LEGAL	=	5	Legal	8.5	x	14	inches
%DMPAPER_STATEMENT	=	6	Statement	5.5	x	8.5	inches
%DMPAPER_EXECUTIVE	=	7	Executive	7.25	x	10.5	inches
%DMPAPER_A3	=	8	A3	297	x	420	mm
%DMPAPER_A4	=	9	A4	210	x	297	mm
%DMPAPER_A5	=	11	A5	148	x	210	mm
%DMPAPER_B4	=	12	B4	250	x	354	mm
%DMPAPER_B5	=	13	B5	182	x	257	mm
%DMPAPER_FOLIO	=	14	Folio	8.5	x	13	inches
%DMPAPER_QUARTO	=	15	Quarto	215	x	275	mm
%DMPAPER_10X14	=	16	10x14	10	x	14	inches
%DMPAPER_11X17	=	17	11x17	11	x	17	inches
%DMPAPER_NOTE	=	18	Note	8.5	x	11	inches
%DMPAPER_ENV_9	=	19	9 Envlp	3.875	x	8.875	inches
%DMPAPER_ENV_10	=	20	10 Envlp	4.125	x	9.5	inches

Other paper style codes may be defined by Windows or printer suppliers. You can use [XPRINT GET PAPERS](#) to obtain a list of all the paper styles supported by the [attached](#) host printer.

If the printer does not support paper style changes, the value zero is returned. If executed without a host printer attached, [error 57](#) is generated.

See also

[XPRINT ATTACH](#), [XPRINT GET PAPERS](#), [XPRINT SET PAPER](#)

Purpose

Syntax

Remarks

Retrieve a list of supported [paper types](#).

XPRINT GET PAPERS TO *papers\$*

XPRINT GET PAPERS retrieves a string which contains a list of all of the paper types supported by the [attached](#) host printer. This string is assigned to the string variable specified by *papers\$*.

The string contains a comma-delimited list of *papertype*, *papername*... repeated as many times as necessary. For example:

```
"1,Letter,5,Legal,7,Executive,20,Envelope #10"
```

You can use [PARSECOUNT](#) to determine the number of delimited fields in the string, and [PARSE\\$\(\)](#) to easily extract the type numbers and names. The following equates are predefined in the compiler, and represent the most common paper styles:

%DMPAPER_LETTER	=	1	Letter	8.5	x	11	inches
%DMPAPER_TABLOID	=	3	Tabloid	11	x	17	inches
%DMPAPER_LEDGER	=	4	Ledger	17	x	11	inches
%DMPAPER_LEGAL	=	5	Legal	8.5	x	14	inches
%DMPAPER_STATEMENT	=	6	Statement	5.5	x	8.5	inches
%DMPAPER_EXECUTIVE	=	7	Executive	7.25	x	10.5	inches
%DMPAPER_A3	=	8	A3	297	x	420	mm
%DMPAPER_A4	=	9	A4	210	x	297	mm
%DMPAPER_A5	=	11	A5	148	x	210	mm
%DMPAPER_B4	=	12	B4	250	x	354	mm
%DMPAPER_B5	=	13	B5	182	x	257	mm
%DMPAPER_FOLIO	=	14	Folio	8.5	x	13	inches
%DMPAPER_QUARTO	=	15	Quarto	215	x	275	mm
%DMPAPER_10X14	=	16	10x14	10	x	14	inches
%DMPAPER_11X17	=	17	11x17	11	x	17	inches
%DMPAPER_NOTE	=	18	Note	8.5	x	11	inches
%DMPAPER_ENV_9	=	19	9 Envlp	3.875	x	8.875	inches
%DMPAPER_ENV_10	=	20	10 Envlp	4.125	x	9.5	inches

Other paper style codes may be defined by Windows or printer suppliers. If executed without a host printer attached, [error 57](#) is generated.

See also

[XPRINT ATTACH](#), [XPRINT GET PAPERS](#), [XPRINT SET PAPER](#)

XPRINT GET PIXEL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the color of a pixel on a host printer page.
Syntax	<code>XPRINT GET PIXEL (<i>x!</i>, <i>y!</i>) TO <i>rgbColor&</i></code>
Remarks	Not all printer drivers support the ability to retrieve the color of a pixel. If this feature is not supported, or if the coordinates are outside the printer client area, an invalid color value of -1 is returned. If no host printer is attached, error 57 is generated.
See also	Built In RGB Color Equates , XPRINT ATTACH , XPRINT COLOR , XPRINT SET PIXEL

XPRINT GET POS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the last point referenced (POS) by an XPRINT statement.
Syntax	<code>XPRINT GET POS TO <i>x!</i>, <i>y!</i></code>
Remarks	XPRINT GET POS allows you to retrieve the last point referenced (POS) by XPRINT statements. The coordinate points are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). If executed without a host printer attached, an error 57 is generated, and the values 0,0, are returned.
See also	XPRINT ATTACH , XPRINT SET POS

XPRINT GET PPI statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the resolution of the host printer page.
Syntax	<code>XPRINT GET PPI TO <i>x&</i>, <i>y&</i></code>
Remarks	<p>XPRINT GET PPI retrieves the resolution (points per inch) of the host printer page. The resolution is always specified in pixels, regardless of any XPRINT SCALE option. If executed without a host printer attached, error 57 is generated, and the values 0,0 are returned. This statement is particularly useful in drawing items such as rulers and graphs to a particular physical size. There are 25.4 millimeters per inch, so just divide by 25.4 to convert from pixels per inch to pixels per millimeter.</p>
See also	XPRINT ATTACH , XPRINT GET CLIENT , XPRINT GET MARGIN , XPRINT GET SIZE

XPRINT GET QUALITY statement

[Top](#) [Previous](#) [Next](#)

Purpose	Retrieve the print quality setting for the host printer .
Syntax	<code>XPRINT GET QUALITY TO <i>qual</i>&</code>
Remarks	XPRINT GET QUALITY retrieves the print quality setting for the host printer. The value 1 is draft mode, 2 is low resolution, 3 is medium resolution, and 4 is high resolution. If the printer does not support print quality settings, 0 is returned. If no host printer is attached, error 57 is generated.
See also	XPRINT ATTACH , XPRINT SET QUALITY

Purpose	Retrieve the current coordinate limits for the host printer page.
Syntax	<code>XPRINT GET SCALE TO <i>x1!</i>, <i>y1!</i>, <i>x2!</i>, <i>y2!</i></code>
Remarks	<p>XPRINT SCALE allows you to define your own world coordinate system for subsequent XPRINT statements. World coordinates may be floating point values, with the only requirement that <i>x1!</i> not equal <i>x2!</i>, and <i>y1!</i> not equal <i>y2!</i>.</p> <p>XPRINT GET SCALE retrieves the coordinate limits, which may be either custom world coordinates (if an XPRINT SCALE has been executed), or else default pixel coordinates. This allows you to save and restore a previous set of coordinates.</p>
See also	XPRINT SCALE , XPRINT SCALE PIXELS

Purpose	Retrieve the total size of the host printer page.
Syntax	<code>XPRINT GET SIZE TO <i>nWidth!</i>, <i>nHeight!</i></code>
Remarks	XPRINT GET SIZE allows you to retrieve the full size of the host printer page, including both the printable client area and unprintable margins. The sizes are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). If no host printer is attached, error 57 is generated, and the values 0,0 are returned.
See also	XPRINT ATTACH , XPRINT GET CLIENT , XPRINT GET MARGIN , XPRINT GET MIX , XPRINT GET PPI

Purpose Retrieve the [active printer tray](#).

Syntax XPRINT GET TRAY TO *papertray&*

Remarks XPRINT GET TRAY retrieves the paper tray which is active on the [host printer](#). A descriptive value is assigned to the long integer variable specified by *papertray&*. The following equates are predefined in the compiler, and represent the most common paper trays:

%DMBIN_UPPER	= 1
%DMBIN_LOWER	= 2
%DMBIN_MIDDLE	= 3
%DMBIN_MANUAL	= 4
%DMBIN_ENVELOPE	= 5
%DMBIN_ENVMANUAL	= 6
%DMBIN_AUTO	= 7
%DMBIN_TRACTOR	= 8
%DMBIN_SMALLFMT	= 9
%DMBIN_LARGEfmt	= 10
%DMBIN_LARGECAPACITY	= 11
%DMBIN_CASSETTE	= 14
%DMBIN_FORMSOURCE	= 15

Other tray codes may be defined by Windows or printer suppliers, so your program should be written to consider that possibility. You can use [XPRINT GET TRAYS](#) to obtain a list of all the paper trays supported by the [attached](#) host printer.

If the printer does not support the tray change requested, [error 5](#) is generated. If executed without a host printer attached, [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT GET TRAYS](#), [XPRINT SET TRAY](#)

Purpose	Retrieve a list of supported paper trays .																										
Syntax	<code>XPRINT GET TRAYS TO <i>trays\$</i></code>																										
Remarks	<p>XPRINT GET TRAYS retrieves a string which contains a list of all of the paper trays supported by the attached host printer. This string is assigned to the string variable specified by <code>trays\$</code>.</p> <p>The string contains a comma-delimited list of traytype, trayname... repeated as many times as necessary. For example:</p> <pre>"1,Upper,2,Lower,5,Envelope"</pre> <p>You can use PARSECOUNT to determine the number of delimited fields in the string, and PARSE\$() to easily extract the tray numbers and names. The following equates are predefined in the compiler, and represent the most common trays:</p> <table><tbody><tr><td><code>%DMBIN_UPPER</code></td><td><code>= 1</code></td></tr><tr><td><code>%DMBIN_LOWER</code></td><td><code>= 2</code></td></tr><tr><td><code>%DMBIN_MIDDLE</code></td><td><code>= 3</code></td></tr><tr><td><code>%DMBIN_MANUAL</code></td><td><code>= 4</code></td></tr><tr><td><code>%DMBIN_ENVELOPE</code></td><td><code>= 5</code></td></tr><tr><td><code>%DMBIN_ENVMANUAL</code></td><td><code>= 6</code></td></tr><tr><td><code>%DMBIN_AUTO</code></td><td><code>= 7</code></td></tr><tr><td><code>%DMBIN_TRACTOR</code></td><td><code>= 8</code></td></tr><tr><td><code>%DMBIN_SMALLFMT</code></td><td><code>= 9</code></td></tr><tr><td><code>%DMBIN_LARGEfmt</code></td><td><code>= 10</code></td></tr><tr><td><code>%DMBIN_LARGEcapacity</code></td><td><code>= 11</code></td></tr><tr><td><code>%DMBIN_CASSETTE</code></td><td><code>= 14</code></td></tr><tr><td><code>%DMBIN_FORMSOURCE</code></td><td><code>= 15</code></td></tr></tbody></table> <p>Other paper style codes may be defined by Windows or printer suppliers. If executed without a host printer attached, error 57 is generated.</p>	<code>%DMBIN_UPPER</code>	<code>= 1</code>	<code>%DMBIN_LOWER</code>	<code>= 2</code>	<code>%DMBIN_MIDDLE</code>	<code>= 3</code>	<code>%DMBIN_MANUAL</code>	<code>= 4</code>	<code>%DMBIN_ENVELOPE</code>	<code>= 5</code>	<code>%DMBIN_ENVMANUAL</code>	<code>= 6</code>	<code>%DMBIN_AUTO</code>	<code>= 7</code>	<code>%DMBIN_TRACTOR</code>	<code>= 8</code>	<code>%DMBIN_SMALLFMT</code>	<code>= 9</code>	<code>%DMBIN_LARGEfmt</code>	<code>= 10</code>	<code>%DMBIN_LARGEcapacity</code>	<code>= 11</code>	<code>%DMBIN_CASSETTE</code>	<code>= 14</code>	<code>%DMBIN_FORMSOURCE</code>	<code>= 15</code>
<code>%DMBIN_UPPER</code>	<code>= 1</code>																										
<code>%DMBIN_LOWER</code>	<code>= 2</code>																										
<code>%DMBIN_MIDDLE</code>	<code>= 3</code>																										
<code>%DMBIN_MANUAL</code>	<code>= 4</code>																										
<code>%DMBIN_ENVELOPE</code>	<code>= 5</code>																										
<code>%DMBIN_ENVMANUAL</code>	<code>= 6</code>																										
<code>%DMBIN_AUTO</code>	<code>= 7</code>																										
<code>%DMBIN_TRACTOR</code>	<code>= 8</code>																										
<code>%DMBIN_SMALLFMT</code>	<code>= 9</code>																										
<code>%DMBIN_LARGEfmt</code>	<code>= 10</code>																										
<code>%DMBIN_LARGEcapacity</code>	<code>= 11</code>																										
<code>%DMBIN_CASSETTE</code>	<code>= 14</code>																										
<code>%DMBIN_FORMSOURCE</code>	<code>= 15</code>																										
See also	XPRINT ATTACH , XPRINT GET TRAY , XPRINT SET TRAY																										

Purpose Print an image from an [IMAGELIST](#)

Syntax `XPRINT IMAGELIST (x!,y!), hLst, index&, overlay&, style&`

Remarks One of the images stored in an IMAGELIST is printed on the [attached](#) host printer. The parameters *x!*,*y!* define the upper left corner of the position of the image. *hLst* is the handle of the IMAGELIST and *index*& is the selector of the image to be displayed (1=first, 2=second, etc.). If *overlay*& is non-zero, it specifies an overlay image to be added to the printed image from the image list. The parameter *style*& may be one of the following style bits:

`%ILD_NORMAL` Draws the image using the background color of the image list. If the background color is the default value `%CLR_NONE`, the image is drawn transparently.

`%ILD_TRANSPARENT` Draws the image transparently if there is a mask.

`%ILD_MASK` Draws the mask.

`%ILD_BLEND25` If there is a mask, the image is drawn blending 25% with the system highlight color.

`%ILD_BLEND50` If there is a mask, the image is drawn blending 50% with the system highlight color.

See also [XPRINT ATTACH](#), [IMAGELIST](#)

Purpose	Draw a line on a host printer page.
Syntax	<code>XPRINT LINE [STEP] [(x1!, y1!)] - [STEP] (x2!, y2!)[, rgbColor&]</code>
Remarks	The line is drawn from the first point, up to, but not including the second point. Coordinate points are specified in pixels, unless optional world coordinates have been defined with an XPRINT SCALE statement. Line width can be set using XPRINT WIDTH . If line width is set to 1 (the default), the line style can be set with XPRINT STYLE . If executed without a host printer attached, error 57 is generated.
<i>x1!, y1!</i>	Optional values which define the starting point of the line. If this optional first point is omitted, the line begins at the last point referenced (POS) in a preceding XPRINT statement. If the first STEP option is included, the x1! and y1! starting coordinates are relative to the last point referenced (POS) on the host printer page.
<i>x2!, y2!</i>	The ending point of the line. If the second STEP option is included, the x2! and y2! ending coordinates are relative to the starting coordinates.
<i>rgbColor&</i>	Optional RGB color value for the line. If <i>rgbColor&</i> is omitted (or -1), the line color defaults to the current foreground color for the host printer page.
See also	Built In RGB Color Equates , XPRINT ARC , XPRINT ATTACH , XPRINT BOX , XPRINT COLOR , XPRINT ELLIPSE , XPRINT PIE , XPRINT POLYGON , XPRINT POLYLINE , XPRINT SET MIX , XPRINT STYLE , XPRINT WIDTH
Example	<pre>' Draw a triangle. Note that, since LINE draws up to, ' but not including the second point, one extra point ' must be added when STEP is used. XPRINT LINE (10, 10) - (10, 100) ' left side XPRINT LINE STEP - (101, 100) ' base line XPRINT LINE STEP - (10, 10) ' back to top</pre>

Purpose	Draw a pie section on a host printer page.
Syntax	<code>XPRINT PIE (x1!, y1!) - (x2!, y2!), arcStart!, arcEnd! [, [rgbColor&] [, [fillcolor&] [, [fillstyle&]]]</code>
Remarks	<p>A pie section is an arc, with a line drawn from each end point to the center of the circle or ellipse. To specify a pie section, you would first define the full circle or ellipse of which it is a part, and then specify the points on the ellipse where the arc starts and stops.</p> <p>The full circle or ellipse is defined by its bounding rectangle, which is defined as the smallest rectangle which can be drawn around the circle or ellipse. For example, if the circle is centered at position (400,400), with a radius of 100 pixels, the upper left corner (x1,y1) of the bounding rectangle is (300,300), and the lower right corner (x2,y2) is (500,500).</p> <p>The start point and end point of the arc are specified by their angle, which must be given in radians. A complete circle or ellipse is 2*pi radians. On a 12-hour clock-face, the values 0 and 2*pi both refer to the position of 3 o'clock, while the value 1*pi refers to the position of 9 o'clock. Other positions are specified by a radian value relative to these. In Classic PowerBASIC, arcs are always drawn counter-clockwise from the starting point to the ending point.</p> <p>Prior to any XPRINT operations, a host printer must first be selected with XPRINT ATTACH. The coordinate points are specified in pixels (or world coordinates, if those were chosen with XPRINT SCALE). Line width can be set using XPRINT WIDTH. If line width is set to 1 (the default), the line style can be set with XPRINT STYLE. Because of the nature of a pie section, XPRINT PIE neither uses, nor updates, (last point referenced). If executed without a host printer attached, error 57 is generated.</p>
<i>x1!, y1!</i>	The upper left corner of the bounding rectangle of the full circle or ellipse.
<i>x2!, y2!</i>	The lower right corner of the bounding rectangle of the full circle or ellipse.
<i>ArcStart!</i>	The starting angle of the arc, in radians, from 0 to 2*pi.
<i>ArcEnd!</i>	<p>The ending angle of the arc, in radians, from 0 to 2*pi radians. Note that arcs are always drawn counter-clockwise from <i>arcStart!</i> to <i>arcEnd!</i>.</p> <p>Compared with a 12-hour clock-face, 0 or 2*pi radians is at 3 o'clock, and 1*pi radians is at 9 o'clock.</p>
<i>rgbColor&</i>	Optional RGB color of the pie edge. If omitted (or -1), the edge color defaults to the current foreground color for the host printer page.
<i>fillcolor&</i>	Optional RGB color of the pie interior. If <i>fillcolor&</i> is omitted (or -2), the

interior of the pie is not filled, allowing the background to show through. If *fillcolor*& is -1, the interior is painted with the same color as the edge. Otherwise, *fillcolor*& specifies the RGB color to be used.

fillstyle&

Optional fill style (pattern) to be used. If *fillstyle*& is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the *fillcolor*&, while the background is specified by the default background color for the host printer page. The optional *fillstyle*& may be:

0	Solid (default)
1	Horizontal Lines
2	Vertical Lines
3	Upward Diagonal Lines
4	Downward Diagonal Lines
5	Crossed Lines
6	Diagonal Crossed Lines

See also

[Built In RGB Color Equates](#), [XPRINT ARC](#), [XPRINT ATTACH](#), [XPRINT BOX](#), [XPRINT COLOR](#), [XPRINT ELLIPSE](#), [XPRINT LINE](#), [XPRINT STYLE](#), [XPRINT WIDTH](#)

Example

```
' A full circle is 2*pi radians (100%).
' To show a 25% Pie, use the formula 0.25 * 2 * pi.
' The following divides a full circle into four 25% parts, each
' with its own colors, each slightly separated from the others.
' Note: 0 is at 3 o'clock, then it builds counter-clockwise.
LOCAL Pi2 AS DOUBLE
Pi2 = ATN(1)* 8 ' 2 * Pi can be useful here
XPRINT PIE (10, 9)-(110, 109), 0, Pi2 * 0.25, %BLUE, %LTGRAY, 3
XPRINT PIE (9, 9)-(109, 109), Pi2 * 0.25, Pi2 * 0.50, %RED, %LTGRAY, 4
XPRINT PIE (9, 10)-(109, 110), Pi2 * 0.5, Pi2 * 0.75, RGB(0,127,0),
%LTGRAY, 3
XPRINT PIE (10, 10)-(110, 110), Pi2 * 0.75, 0, %GRAY, %LTGRAY, 4
```

Purpose	Draw a polygon on a host printer page.
Syntax	<code>XPRINT POLYGON points [, [rgbColor&] [, [fillcolor&] [, [fillstyle&] [, fillmode&]]]</code>
Remarks	<p>The coordinate points are specified in pixels, unless optional world coordinates have been defined with an XPRINT SCALE statement. Line width can be set using XPRINT WIDTH. If line width is set to 1 (the default), the line style can be set with XPRINT STYLE. XPRINT POLYGON neither uses, nor updates, the last point referenced (POS). If executed without a host printer attached, error 57 is generated.</p>
<i>points</i>	<p>User-defined type that defines the number of vertices and the location of each. There must be at least two, and no more than 1024 vertices. The first member is a long integer point count, followed directly by the appropriate number of single precision floats to specify the actual coordinates. Floating point coordinates are required, because of the possibility of their use as world coordinates with XPRINT SCALE. You can use a type with a scalar list, like this:</p> <pre>TYPE PolyPoints count as long x1 as single y1 as single x2 as single y2 as single x3 as single y3 as single END TYPE</pre> <p>Or, you can create an array using point types, like this:</p> <pre>TYPE PolyPoint x as single y as single END TYPE</pre> <pre>TYPE PolyArray count as long xy(1 TO 3) as PolyPoint END TYPE</pre>
<i>rgbColor&</i>	Optional RGB color of the polygon edge. If omitted (or -1), the edge color defaults to the current foreground color for the host printer page.
<i>fillcolor&</i>	Optional RGB color of the polygon interior. If <i>fillcolor&</i> is omitted (or -2), the interior of the ellipse is not filled, allowing the background to show through. If <i>fillcolor&</i> is -1, the interior is painted with the same color as the edge. Otherwise, <i>fillcolor&</i> specifies the RGB color to be used.
<i>fillstyle&</i>	Optional fill style (pattern) to be used. If <i>fillstyle&</i> is omitted, the default fill style is solid (0). If a hatch pattern is chosen (1 to 6), the foreground color is specified by the <i>fillcolor&</i> , while the background is specified by the

default background color for the host printer page. The optional *fillstyle*& may be:

0	Solid (default)
1	Horizontal Lines
2	Vertical Lines
3	Upward Diagonal Lines
4	Downward Diagonal Lines
5	Crossed Lines
6	Diagonal Crossed Lines

fillmode& If *fillmode*& is missing (or zero), the winding mode is selected. This fills any region with a non-zero winding value. If *fillmode*& is non-zero, the alternate mode is selected. This fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, it fills the area between the first side and the second side, between the third side and fourth side, etc.

See also [Built In RGB Color Equates](#), [XPRINT ARC](#), [XPRINT ATTACH](#), [XPRINT BOX](#), [XPRINT COLOR](#), [XPRINT ELLIPSE](#), [XPRINT LINE](#), [XPRINT POLYLINE](#)

Purpose	Draw a series of connected lines on a host printer page.
Syntax	<code>XPRINT POLYLINE <i>points</i> [, <i>rgbColor&</i>]</code>
Remarks	<p>The coordinate points are specified in pixels, unless optional world coordinates have been defined with an XPRINT SCALE statement. Line width can be set using XPRINT WIDTH. If line width is set to 1 (the default), the line style can be set with XPRINT STYLE. XPRINT POLYLINE neither uses, nor updates, the last point referenced (POS). If executed without a host printer attached, error 57 is generated.</p>
<i>points</i>	<p>User-defined type that defines the number of vertices and the location of each. There must be at least two, and no more than 1024 vertices. The first member is a long integer point count, followed directly by the appropriate number of single precision floats to specify the actual coordinates. Floating point coordinates are required, because of the possibility of their use as world coordinates with SCALE. You can use a type with a scalar list, like this:</p> <pre>TYPE PolyPoints count as long x1 as single y1 as single x2 as single y2 as single x3 as single y3 as single END TYPE</pre> <p>Or, you can create an array using point types, like this:</p> <pre>TYPE PolyPoint x as single y as single END TYPE</pre> <pre>TYPE PolyArray count as long xy(1 TO 3) as PolyPoint END TYPE</pre>
<i>rgbColor&</i>	Optional RGB color of the polygon edge. If omitted (or -1), the edge color defaults to the current foreground color for the host printer page.
See also	Built In RGB Color Equates , XPRINT ARC , XPRINT ATTACH , XPRINT BOX , XPRINT COLOR , XPRINT ELLIPSE , XPRINT LINE , XPRINT POLYGON , XPRINT STYLE , XPRINT WIDTH

XPRINT RENDER statement

[Top](#) [Previous](#) [Next](#)

Purpose	Render an image on a host printer page.
Syntax	<code>XPRINT RENDER <i>BmpName\$</i>, (<i>x1!</i>, <i>y1!</i>) - (<i>x2!</i>, <i>y2!</i>)</code>
Remarks	<p>Renders an image, loaded from a resource or a disk file, on a host printer page. The parameter <i>BmpName\$</i> contains the name of an image to be loaded. If <i>BmpName\$</i> contains a period, it is presumed to be the name of a disk file. Otherwise, an attempt is made to load it from the program's resource data; if not found, it is then presumed to be a disk file. The parameters <i>x1!</i>, <i>y1!</i> define the upper left corner of the destination rectangle, while <i>x2!</i>, <i>y2!</i> define the lower right corner of that rectangle. If the destination rectangle is larger or smaller than the original, the image is stretched or shrunk to the requested size. If XPRINT RENDER is unsuccessful, an appropriate error is generated.</p> <p>The following code will retrieve the natural size of an image in a bitmap file, in pixels:</p> <pre>nFile& = FREEFILE OPEN "myimage.bmp" FOR BINARY AS nFile& GET #nFile&, 19, nWidth& GET #nFile&, 23, nHeight& CLOSE nFile&</pre>
See also	XPRINT ATTACH , XPRINT COPY , XPRINT STRETCH

Purpose	Define a custom world coordinate system for the client (printable) area of the host printer page.
Syntax	<pre>XPRINT SCALE (x1!, y1!)-(x2!, y2!) XPRINT SCALE PIXELS</pre>
Remarks	<p>XPRINT SCALE lets you define your own world coordinate system for all subsequent XPRINT statements. The custom coordinates remain until XPRINT SCALE is repeated, or the printer is detached. World coordinates may be floating point values, with the only requirement that x1! not equal x2!, and y1! not equal y2!. If either is equal, an error 5 is generated.</p> <p>If x2! is greater than x1!, coordinates grow larger as they move to the right. Otherwise, they grow larger as they move to the left.</p> <p>If y2! is greater than y1!, coordinates grow larger as they move downward. Otherwise, they grow larger as they move upward.</p> <p>XPRINT SCALE PIXELS resets the coordinate system to the original default pixel coordinates.</p>
See also	XPRINT ATTACH , XPRINT GET SCALE
Example	<pre>' Attach the default Windows printer XPRINT ATTACH DEFAULT ' Retrieve the client size (printable area) of the printer page XPRINT GET CLIENT TO ncWidth!, ncHeight! ' Retrieve the resolution (points per inch) of the attached printer XPRINT GET PPI TO x&, y& ' Width in inches of the printable area ncWidth! = ncWidth!/x& ' Height in inches of the printable area ncHeight! = ncHeight!/y& ' Set the scale to inches, for American letter-size paper ' in portrait mode. This is the equivalent to 8.5x11 minus the margins. XPRINT SCALE (0,0)-(ncWidth!,ncHeight!)</pre>

XPRINT SET COLLATE statement New! [Top](#) [Previous](#) [Next](#)

Purpose Change the XPRINT collate status.

Syntax XPRINT SET COLLATE *numrexp*

Remarks XPRINT allows you to set the collate status, if the printer driver supports both multiple copies and collate capability. XPRINT SET COLLATE enables or disables collating, depending upon the value of the *numrexp* (1=true 0=false). The following equates are predefined in the compiler to symbolically represent the possible status:

```
%DMCOLLATE_FALSE      = 0
%DMCOLLATE_TRUE       = 1
```

If the printer does not support collating, or other values are used, [error 5](#) is generated. If this statement is executed without a host printer [attached](#), [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT GET COLLATE](#)

XPRINT SET COLORMODE statement

[Top](#) [Previous](#)
[Next](#)

New!

Purpose	Change the XPRINT colormode status.
Syntax	<code>XPRINT SET COLORMODE <i>numexp</i></code>
Remarks	<p>XPRINT allows you to set the color or monochrome print mode if the printer driver supports it. XPRINT SET COLORMODE expects a numeric expression which evaluates to one of the following listed values. The following equates are predefined in the compiler to symbolically represent the possible status:</p> <pre>%DMCOLOR_MONOCHROME = 1 %DMCOLOR_COLOR = 2</pre> <p>If this statement is executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT GET COLORMODE

XPRINT SET COPIES statement **New!**

[Top](#) [Previous](#) [Next](#)

Purpose Change the XPRINT copy count.

Syntax `XPRINT SET COPIES numexp`

Remarks XPRINT SET COPIES allows you to set the number of copies to be automatically printed, if it is supported by the printer driver. The default value is one (1). If multiple copies are not supported by the printer driver, or the count requested is greater than that supported by the printer driver, [error 5](#) is generated. If this statement is executed without a host printer [attached](#), [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT GET COPIES](#)

XPRINT SET DUPLEX statement New!

[Top](#) [Previous](#) [Next](#)

Purpose Change the XPRINT duplex status.

Syntax `XPRINT SET DUPLEX numrexp`

Remarks XPRINT allows you to set the duplex status, if the printer supports printing on both sides of a page. XPRINT SET DUPLEX changes the mode to that specified by the *numrexp*. The following equates are predefined in the compiler to symbolically represent the possible duplex status:

<code>%DMDUP_SIMPLEX</code>	<code>= 1</code>	<code>(single sided printing)</code>
<code>%DMDUP_VERTICAL</code>	<code>= 2</code>	<code>(page flipped on the vertical edge)</code>
<code>%DMDUP_HORIZONTAL</code>	<code>= 3</code>	<code>(page flipped on the horizontal edge)</code>

If the printer does not support duplex printing, [error 5](#) is generated. If this statement is executed without a host printer [attached](#), [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT GET DUPLEX](#)

Purpose	Select a font for the XPRINT statement.
Syntax	<pre>XPRINT SET FONT <i>fonthndl&</i></pre>
<i>fonthndl&</i>	The numeric handle returned by the FONT NEW statement.
Remarks	<p>The font specified by <i>fonthndl&</i> is selected to be used by all of the following XPRINT statements. This is the most efficient way to change fonts and their general appearance (size, style, etc.).</p> <p>You can predefine virtually any number of fonts and attributes by executing FONT NEW statements for each of them. That makes them ready for immediate use when selected by XPRINT SET FONT.</p> <p>Prior to any XPRINT operations, a specific printer must first be selected with XPRINT ATTACH. If no specific font is selected, the default font is Courier New with no style attributes.</p>
See also	FONT NEW , XPRINT , XPRINT ATTACH , XPRINT CHR SIZE , XPRINT TEXT SIZE

Purpose Set the [color](#) mix mode for a [host printer](#) page.

Syntax `XPRINT SET MIX mode&`

Remarks Prior to any graphical operations, a host printer must first be selected with [XPRINT ATTACH](#). There are 16 mix modes available to use for mixing the drawing color with the color that already exists at the drawing location. The default mix mode is 13, %mix_CopySrc. The mix mode equates are predefined in Classic PowerBASIC.

%mix_Blackness Pixel is always 0 (black).

%mix_NotMergeSrc Pixel is the inverse of the MergeSrc color.

%mix_MaskNotSrc Pixel is a combination of the colors common to both the pixel and the inverse of the source.

%mix_NotCopySrc Pixel is the inverse of the pen color.

%mix_MaskSrcNot Pixel is a combination of the colors common to both the source and the inverse of the pixel.

%mix_Not Pixel is the inverse of the pixel color.

%mix_XorSrc Pixel is a combination of the colors in the source and in the pixel, but not in both.

%mix_NotMaskSrc Pixel is the inverse of the MaskSrc color.

%mix_MaskSrc Pixel is a combination of the colors common to both the source and the pixel.

%mix_NotXorSrc Pixel is the inverse of the XorSrc color.

%mix_Nop Pixel remains unchanged.

%mix_MergeNotSrc Pixel is a combination of the source color and the inverse of the pixel color.

%mix_CopySrc Pixel is the source color (default).

%mix_MergeSrcNot Pixel is a combination of the source color and the inverse of the pixel color.

%mix_MergeSrc Pixel is a combination of the source color and the pixel color.

%mix_Whiteness Pixel is always 1 (white).

See also [XPRINT ATTACH](#), [XPRINT GET MIX](#)

XPRINT SET ORIENTATION statement

[Top](#) [Previous](#)
[Next](#)

IMPROVED

Purpose	Set the paper orientation for a host printer page.
Syntax	<code>XPRINT SET ORIENTATION <i>orient</i>&</code>
Remarks	<p>XPRINT SET ORIENTATION sets the orientation of the paper in the host printer. The value 1 indicates portrait mode, while 2 indicates landscape mode. If a host printer is not attached, or does not support setting the orientation, error 57 is generated.</p> <p>Normally, XPRINT SET ORIENTATION should not be executed mid-page, as it is not possible to print one page in both portrait and landscape modes. It is best to execute this statement immediately after either XPRINT ATTACH or XPRINT FORMFEED. If executed mid-page, Classic PowerBASIC will automatically generate a new page, and apply the mode thereafter.</p> <p>If you find it necessary to print text in both horizontal and vertical format, this can be accomplished by creating an additional font with FONT NEW and XPRINT SET FONT, using the pitch and escapement attributes.</p>
See also	FONT NEW , XPRINT ATTACH , XPRINT GET ORIENTATION , XPRINT SET FONT

Purpose

Syntax

Remarks

Set a new [paper size/type](#).

XPRINT SET PAPER *papertype*&

XPRINT SET PAPER changes the paper style for the [host printer](#) to that designated by *papertype*&. The paper style is identified by an integral value given in the expression *papertype*&. The following equates are predefined in the compiler, and represent the most common paper styles:

%DMPAPER_LETTER	=	1	Letter	8.5	x	11	inches
%DMPAPER_TABLOID	=	3	Tabloid	11	x	17	inches
%DMPAPER_LEDGER	=	4	Ledger	17	x	11	inches
%DMPAPER_LEGAL	=	5	Legal	8.5	x	14	inches
%DMPAPER_STATEMENT	=	6	Statement	5.5	x	8.5	inches
%DMPAPER_EXECUTIVE	=	7	Executive	7.25	x	10.5	inches
%DMPAPER_A3	=	8	A3	297	x	420	mm
%DMPAPER_A4	=	9	A4	210	x	297	mm
%DMPAPER_A5	=	11	A5	148	x	210	mm
%DMPAPER_B4	=	12	B4	250	x	354	mm
%DMPAPER_B5	=	13	B5	182	x	257	mm
%DMPAPER_FOLIO	=	14	Folio	8.5	x	13	inches
%DMPAPER_QUARTO	=	15	Quarto	215	x	275	mm
%DMPAPER_10X14	=	16	10x14	10	x	14	inches
%DMPAPER_11X17	=	17	11x17	11	x	17	inches
%DMPAPER_NOTE	=	18	Note	8.5	x	11	inches
%DMPAPER_ENV_9	=	19	9 Envlp	3.875	x	8.875	inches
%DMPAPER_ENV_10	=	20	10 Envlp	4.125	x	9.5	inches

Other paper style codes may be defined by Windows or printer suppliers. You can use [XPRINT GET PAPERS](#) to obtain a list of all the paper styles supported by the attached host printer.

If the printer does not support the paper style specified, [error 5](#) is generated. If executed without a host printer attached, [error 57](#) is generated.

See also

[XPRINT ATTACH](#), [XPRINT GET PAPER](#), [XPRINT GET PAPERS](#)

XPRINT SET PIXEL statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the color of a pixel on a host printer page.
Syntax	<code>XPRINT SET PIXEL [STEP] (<i>x!</i>, <i>y!</i>) [, <i>rgbColor&</i>]</code>
Remarks	<p>XPRINT SET PIXEL draws a single pixel on the host printer page. The optional color parameter is an RGB value; if not included, the color defaults to the current foreground color for the host printer. If the STEP option is included, the <i>x!</i> and <i>y!</i> coordinates are relative to the last point referenced (POS). The coordinate points are specified in pixels, unless world coordinates were set with an XPRINT SCALE statement. If no host printer is attached, error 57 is generated.</p>
See also	Built In RGB Color Equates , XPRINT ATTACH , XPRINT COLOR , XPRINT GET PIXEL

XPRINT SET POS statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the last point referenced (POS) by an XPRINT statement.
Syntax	<code>XPRINT SET POS [STEP] (<i>x!</i>, <i>y!</i>)</code>
Remarks	<p>XPRINT SET POS allows you to set the last point referenced (POS) by XPRINT statements. If the STEP option is included, the <i>x!</i> and <i>y!</i> coordinates are relative to the prior POS. The coordinate points are specified in pixels (or world coordinates, if those were defined with an XPRINT SCALE statement). If executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT GET POS

Purpose	Set the print quality for a host printer .
Syntax	<code>XPRINT SET QUALITY <i>qual</i>&</code>
Remarks	<p>XPRINT SET QUALITY sets the print quality setting for the host printer. The value 1 is draft mode, 2 is low resolution, 3 is medium resolution, and 4 is high resolution. It should be noted that some printers only allow higher resolutions to be set from the printer dialog (in XPRINT ATTACH CHOOSE). If no host printer is attached, or the printer does not support print quality settings, error 57 is generated.</p>
See also	XPRINT ATTACH , XPRINT GET QUALITY

Purpose Set a new active [printer tray](#).

Syntax XPRINT SET TRAY *numrexp*

Remarks XPRINT SET TRAY changes the active paper tray on the [host printer](#) to that specified by *numrexp*. The following equates are predefined in the compiler, and represent the most common paper trays:

%DMBIN_UPPER	= 1
%DMBIN_LOWER	= 2
%DMBIN_MIDDLE	= 3
%DMBIN_MANUAL	= 4
%DMBIN_ENVELOPE	= 5
%DMBIN_ENVMANUAL	= 6
%DMBIN_AUTO	= 7
%DMBIN_TRACTOR	= 8
%DMBIN_SMALLFMT	= 9
%DMBIN_LARGEfmt	= 10
%DMBIN_LARGECAPACITY	= 11
%DMBIN_CASSETTE	= 14
%DMBIN_FORMSOURCE	= 15

Other tray codes may be defined by Windows or printer suppliers, so your program should be written to consider that possibility. You can use [XPRINT GET TRAYS](#) to obtain a list of all the paper trays supported by the [attached](#) host printer.

If the printer does not support the tray change requested, [error 5](#) is generated. If executed without a host printer attached, [error 57](#) is generated.

See also [XPRINT ATTACH](#), [XPRINT GET TRAY](#), [XPRINT GET TRAYS](#)

Purpose	Copy and resize a bitmap to a host printer page.																
Syntax	<code>XPRINT STRETCH <i>hndl</i>, <i>id</i>, (<i>x1</i>,<i>y1</i>)-(<i>x2</i>,<i>y2</i>) TO (<i>x3</i>,<i>y3</i>)-(<i>x4</i>,<i>y4</i>) [<i>,mix</i>, <i>stretch</i>]</code>																
Remarks	<p>You can copy a complete bitmap, or a portion of it, to the host printer page, while resizing it to a larger or smaller size. The expression <i>hndl</i> specifies the handle of the source GRAPHIC BITMAP, GRAPHIC WINDOW, or dialog containing a GRAPHIC CONTROL. The expression <i>id</i>& is the unique control identifier in the range 1 to 65535, as assigned with the CONTROL ADD GRAPHIC statement. <i>id</i>& must be 0 for a GRAPHIC WINDOW or a . The destination of the copy operation is the host printer page. The bitmap is automatically resized to fit the destination parameters. You must use care that your parameters are valid for the specified bitmaps, or results of the operation are undefined.</p> <p>If the mix parameter is included, it is one of the values in the following table. If not included, a default of %MIX_COPYSRC is presumed. There are 8 mix modes available to use for mixing drawing colors with the colors which already exist at the at the drawing location. The mix mode equates are predefined in Classic PowerBASIC.</p> <table><tr><td>%MIX_NOTMERGESRC</td><td>Pixel is the inverse of the MergeSrc color.</td></tr><tr><td>%MIX_NOTCOPYSRC</td><td>Pixel is the inverse of the source color.</td></tr><tr><td>%MIX_MASKSRCNOT</td><td>Pixel is a combination of the colors common to both the source and the inverse of the pixel.</td></tr><tr><td>%MIX_XORSRC</td><td>Pixel is a combination of the colors in the source and the pixel, but not in both.</td></tr><tr><td>%MIX_MASKSRC</td><td>Pixel is a combination of the colors common to both the source and the pixel.</td></tr><tr><td>%MIX_MERGENOTSRC</td><td>Pixel is a combination of the pixel color and the and the inverse of the source color.</td></tr><tr><td>%MIX_COPYSRC</td><td>Pixel is the source color (default).</td></tr><tr><td>%MIX_MERGESRC</td><td>Pixel is a combination of the source color and the pixel color.</td></tr></table> <p>If the stretch parameter is included, it is one of the values in the following table. If not included, or it is the value zero (0), the stretch mode is unchanged. An appropriate choice of stretch mode can substantially enhance the quality of bitmaps which are changed in size. The stretch mode equates are predefined in Classic PowerBASIC.</p> <p>The 4 stretch modes are:</p>	%MIX_NOTMERGESRC	Pixel is the inverse of the MergeSrc color.	%MIX_NOTCOPYSRC	Pixel is the inverse of the source color.	%MIX_MASKSRCNOT	Pixel is a combination of the colors common to both the source and the inverse of the pixel.	%MIX_XORSRC	Pixel is a combination of the colors in the source and the pixel, but not in both.	%MIX_MASKSRC	Pixel is a combination of the colors common to both the source and the pixel.	%MIX_MERGENOTSRC	Pixel is a combination of the pixel color and the and the inverse of the source color.	%MIX_COPYSRC	Pixel is the source color (default).	%MIX_MERGESRC	Pixel is a combination of the source color and the pixel color.
%MIX_NOTMERGESRC	Pixel is the inverse of the MergeSrc color.																
%MIX_NOTCOPYSRC	Pixel is the inverse of the source color.																
%MIX_MASKSRCNOT	Pixel is a combination of the colors common to both the source and the inverse of the pixel.																
%MIX_XORSRC	Pixel is a combination of the colors in the source and the pixel, but not in both.																
%MIX_MASKSRC	Pixel is a combination of the colors common to both the source and the pixel.																
%MIX_MERGENOTSRC	Pixel is a combination of the pixel color and the and the inverse of the source color.																
%MIX_COPYSRC	Pixel is the source color (default).																
%MIX_MERGESRC	Pixel is a combination of the source color and the pixel color.																

%BLACKONWHITE	This is the default Windows stretch mode, and is most appropriate for monochrome bitmaps, or those with blocks of color. Performs a boolean OR of eliminated and existing pixels. It preserves black pixels at the expense of white pixels.
%WHITEONBLACK	Performs a boolean OR of eliminated and existing pixels. It preserves white pixels at the expense of black pixels.
%COLORONCOLOR	Deletes eliminated lines of pixels without trying to preserve their information.
%HALFTONE	This provides the highest quality for complex color bitmaps. The average color of the destination pixel block is kept approximately the same as the source pixel block.

See also

[XPRINT ATTACH](#), [XPRINT COPY](#), [XPRINT RENDER](#)

Purpose	Set the line style to be used by various XPRINT statements.										
Syntax	<code>XPRINT STYLE <i>linestyle</i>&</code>										
Remarks	<p>XPRINT STYLE determines the line style which will be used when drawing various graphical objects, while the width value is set to 1. When the width value is greater than one, Windows always interprets the style as 0 (solid).</p> <p>Available line styles are:</p> <table><tr><td>0</td><td>Solid (default)</td></tr><tr><td>1</td><td>Dash</td></tr><tr><td>2</td><td>Dot</td></tr><tr><td>3</td><td>DashDot</td></tr><tr><td>4</td><td>DashDotDot</td></tr></table>	0	Solid (default)	1	Dash	2	Dot	3	DashDot	4	DashDotDot
0	Solid (default)										
1	Dash										
2	Dot										
3	DashDot										
4	DashDotDot										
See also	XPRINT ARC , XPRINT ATTACH , XPRINT BOX , XPRINT ELLIPSE , XPRINT LINE , XPRINT PIE , XPRINT WIDTH										
Example	<pre>' Draw a square box with blue, dotted lines XPRINT WIDTH 1 XPRINT STYLE 2 XPRINT BOX (10, 10) - (110, 110), 0, %BLUE</pre>										

XPRINT TEXT SIZE statement

[Top](#) [Previous](#) [Next](#)

Purpose	Calculate the size of text to be printed on a host printer .
Syntax	<code>XPRINT TEXT SIZE <i>txt\$</i> TO <i>nWidth!</i>, <i>nHeight!</i></code>
Remarks	<p>This statement calculates the total size of the printed text, based upon the current font for the host printer. The sizes returned are specified in pixels, unless optional world coordinates have been defined with an XPRINT SCALE statement.</p> <p>This allows you to easily calculate the appropriate print position, particularly when using a proportional font. If this statement is executed without a host printer attached, error 57 is generated.</p>
See also	XPRINT , XPRINT ATTACH , XPRINT CHR SIZE , XPRINT SET FONT
Example	<pre>FUNCTION PBMAIN ' The following example draws the text both horizontally ' and vertically centered on the host printer page LOCAL x, y, w, h, w2, h2 AS LONG LOCAL sText AS STRING sText = "Classic PowerBASIC" XPRINT ATTACH "Lexmark C750" XPRINT COLOR %BLUE, -2 ' blue text, clear background XPRINT FONT "Times New Roman", 18, 3 ' 18p, bold, italic XPRINT GET CLIENT TO w, h ' get client size XPRINT TEXT SIZE sText TO w2, h2 ' get text size x = (w-w2) / 2 ' centered x-pos y = (h-h2) / 2 ' centered y-pos XPRINT SET POS (x, y) ' set position XPRINT sText ' draw the text XPRINT CLOSE END FUNCTION</pre>

XPRINT WIDTH statement

[Top](#) [Previous](#) [Next](#)

Purpose	Set the graphic line width to be used by various XPRINT statements..
Syntax	<code>XPRINT WIDTH <i>ncPixels</i>&</code>
Remarks	XPRINT WIDTH determines the line width which will be used when drawing various graphical objects. The default width is 1 pixel. The width is always specified in pixels, regardless of any XPRINT SCALE option. When the width is set to a value greater than 1, XPRINT STYLE parameters are always interpreted as 0 (solid).
See also	XPRINT ARC , XPRINT ATTACH , XPRINT BOX , XPRINT ELLIPSE , XPRINT LINE , XPRINT PIE , XPRINT STYLE
Example	<pre>FUNCTION PBMAIN XPRINT ATTACH "Lexmark C750" ' Draw a square box with thick, red lines XPRINT WIDTH 10 XPRINT BOX (10, 10) - (110, 110), 0, %RED XPRINT CLOSE END FUNCTION</pre>

Register Now!

[Top](#) [Previous](#) [Next](#)

When you register with PowerBASIC, you will receive early notification of product upgrades, new product announcements, special offers, a free subscription to the PowerBASIC Gazette newsletter and access to PowerBASIC's award winning technical support. Choose one of these easy options:

On-line: Use your favorite Internet browser to visit our World Wide Web site at: <http://www.PowerBASIC.com/> where you can quickly and easily register all of your PowerBASIC products using our [on-line registration form](#).

By Fax: Fill out the supplied registration card and fax it to +1 (941) 681-3100.

By Mail: Fill out the supplied registration card and mail it to:

PowerBASIC, Inc.
2061 Englewood Road
Englewood, FL. 34223
USA

Don't forget to include your e-mail address! Upgrade notices and special offers are also sent out via electronic mail. Don't be the last to know about a new product release.

Free Support

PowerBASIC One-on-One Free Technical Support is designed to answer questions about PowerBASIC products. Our assistance includes:

- The features of our products
- How these features can be used
- How to troubleshoot unexpected behavior

Other Free Technical Support Resources are available on our web site:

- [PowerBASIC Peer Support Forums](#)
- [TechNotes](#), [downloadable updates](#), [tutorials](#), sample files, sample applications, articles.

PowerBASIC One-on-One Free Technical Support is done via email. Please submit one question or issue at a time to support@PowerBASIC.com. This allows our Support Engineers to provide the best possible service. You must include your serial number. We can not accept lists of questions because they slow our response time to all of our customers.

Paid Technical Support

Custom programming, debugging your program, and issues involving third party products fall outside the bounds of Free Technical Support. Requests for these kinds of services require research and testing beyond the scope of compiler support.

Paid Technical Support is \$49 per incident. A single support incident is an issue that focuses on one aspect of the product - e.g. use of a specific feature of the product or assistance with a specific problem or error message. A single support incident pays for 30 minutes. Issues that exceed 30 minutes must be classed and billed as multiple incidents.

For Paid Support, phone PowerBASIC Sales at +1 (941) 473-7300, Monday through Friday between 9:00AM and 5:00PM Eastern Time (except for holidays).

Final Note

We strongly encourage you to take advantage of our Free Technical Support, and all of the Support resources available to you (see <http://www.PowerBASIC.com/support/> before considering Paid Support.

See Also

[Register](#)

Add-on Libraries, Utilities and Tools

[Top](#) [Previous](#) [Next](#)

Third Party libraries, utilities, and tools can be found on the [Add-on's](#) section on the PowerBASIC web site.

See Also

[Additional Source Code and Examples](#)

[WIN32API.INC Updates](#)

Additional Source Code and Examples

[Top](#) [Previous](#)
[Next](#)

Additional source code, example files, and utilities can be found on the PowerBASIC web site. Click the following links to go to those web pages:

- The [FILES](#) section on the PowerBASIC web site
- The Source Code forum in the free [PowerBASIC Peer Support Forums](#) on the PowerBASIC web site

See Also

[Add-on Libraries, Utilities and Tools](#)

[WIN32API.INC Updates](#)

WIN32API.INC Updates

[Top](#) [Previous](#) [Next](#)

The WIN32API.INC (and the related .INC files) are frequently updated by PowerBASIC, and these updated files are provided as a courtesy service to PowerBASIC programmers. The latest edition of the [WIN32API.INC](#) files can be downloaded from the FILES section of the PowerBASIC web site.

Appendix D - Year 2000 compliance

[Top](#) [Previous](#) [Next](#)

This Classic PowerBASIC Compiler is Year 2000 compliant.

The BASIC programming language, as defined by PowerBASIC, currently supports just one date related keyword: DATE\$.

The [DATE\\$](#) system variable allows a program to retrieve the operating systems current date or change the system date. The system date is the value maintained by your computer's hardware.

Using the syntax:

```
A$ = DATE$
```

a program can retrieve the current system date. The date is always returned using a 4-digit year, such as "01-01-2004".

Using the syntax:

```
DATE$ = "01-01-2004"
```

a program can change the system date. Most BASIC dialects, including Classic PowerBASIC, also allow a two-digit year to be used when changing the system date. When used in this fashion, most BASICs assume the 20th century (1900 - 1999). In Classic PowerBASIC, all two-digit years less than 80 are assumed to be in the 21st century (2000 - 2099).

License

This software is protected by United States copyright law and international treaties. It is licensed for use by one specific person, whose name will be registered with PowerBASIC, Inc., on one computer at a time. It may be moved from one computer to another as long as there is no possibility of it being used on more than one computer at the same time. If this software is used on a network, one licensed copy of the software is required for each person who uses the software. If the licensed product is a compiler, you may distribute the programs you create royalty free. You may not distribute the licensed compiler. If the licensed product includes one or more runtime modules, you may reproduce and distribute them royalty free, provided they are distributed only in conjunction with, and as part of your software program, and provided that they bear your copyright notice or the copyright notice which appears on the PowerBASIC, Inc. disk label. The runtime modules are those files that are required to execute your software program, and which are specifically designated as "runtime modules" in the accompanying PowerBASIC, Inc. documentation. Your use of any of the demonstration or sample programs provided with this product are governed by the notices and restrictions of the respective author or copyright holder. Except as stated above, you may not resell, transfer ownership, barter, donate, rent, lease, lend, or share the licensed software to/with another person or entity. By written request to PowerBASIC, Inc., you may specify a change of licensed user if the replacement user is your employee or family member.

Restrictions

You may use the licensed software to create and maintain any form of target computer program for your own use. However, if you publish any target computer program, freeware or commercial, which is a tool for programmers (programming language, compiler, interpreter, programmer's library, etc.), you may not export a wrapper for any Classic PowerBASIC command which allows that command to be used in other programming languages. For example, you may write and publish your own target program to sort data in an array. But you may not publish a target program which exports the Classic PowerBASIC ARRAY SORT Command for use with a programming language other than Classic PowerBASIC.

Limited Warranty

PowerBASIC, Inc. warrants that the physical disks and physical documentation are free of defects in workmanship and materials for a period of sixty days from the date of purchase. If the disks or documentation are found to be defective within the warranty period, PowerBASIC, Inc. will replace the defective items at no cost to you. The entire liability of this warranty is limited to replacement and shall not, under any circumstances, encompass any other damages. PowerBASIC, Inc. specifically disclaims all other warranties, expressed or implied, including, but not limited to, any implied warranty of merchantability

or fitness for a particular purpose.

Governing Law

This license and limited warranty shall be construed, interpreted, and governed by the laws of the State of Florida, USA, and any action hereunder shall be brought only in Florida. If any provision is found invalid or unenforceable, the balance of this license and limited warranty shall remain valid and enforceable. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this product shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software. All rights not specifically granted herein are reserved by PowerBASIC, Inc.

See Also

[Year 2000 compliance](#)

A special checkbox control that can have 3 states - checked (set), unchecked (unset or cleared) or indeterminate (grayed).

Indeterminate

is defined by your application and usually means that the state is both checked and unchecked.

Also

see

[CONTROL ADD CHECK3STATE](#)

A keystroke or combination of keystrokes that acts as a hot-key or shortcut to perform some programmer-defined operation. When an accelerator is used, it usually results in a %WM_COMMAND or %WM_SYSCOMMAND message being sent to the dialog Callback Function. There are two forms of accelerator: command accelerators (such as indicated by an underscored character on control labels and menu items, for example "

File"

), and a keyboard accelerators,

which are separately configurable through an accelerator table, such as the

[ACCEL ATTACH](#) statement.

An encoded graphical image which may be stored in memory or a disk file. Typically uses the .BMP file extension. See [Graphics](#) for more information.

A special user-written function that receives messages from Windows when an event occurs, such as the user clicking on a button, typing on the keyboard, or drawing the dialog background "surface". Also see [FUNCTION/END FUNCTION](#) and [Callbacks](#).

A rectangle at the top of a window or dialog. It provides space for the title or application name. A caption may be used to move the window with a mouse or other pointing device. In DDT, a dialog caption can be specified with the %WS_CAPTION style. See [DIALOG NEW](#) and [DIALOG SET TEXT](#).

Insertion point indicating where the next character will be placed when a key is pressed. Typically, this is a flashing vertical bar.

The status of the "check box" portion of a [CHECK3STATE](#), [CHECKBOX](#), or [OPTION](#) button. When checked (or set), a CHECK3STATE and CHECKBOX contain an "X" in the box, and an OPTION button contains a solid dot in a circle. When unchecked (unset or cleared), the box/circle is empty. A CHECK3STATE control also offers an "indeterminate" state, where the box is shown grayed (indeterminate is defined by the application and usually represents both checked and unchecked).

A special control consisting of a label portion and a check box button. The check box portion indicates the Check State of the control, and is empty when the control is not selected (unchecked) and is set (checked or an "X" in the box) when selected. Also see [CONTROL ADD CHECKBOX](#).

A window or dialog that is confined to the client area of the parent window. If the parent window is the desktop, the child window can be placed anywhere on the screen.

Use the mouse to 'press' a button or control pointed at by the mouse pointer. When you press the left-mouse button a `%BM_CLICK` message is sent if the control is a button; otherwise, a `%WM_LBUTTONDOWN` message is sent to the dialog or window callback procedure.

Area inside a dialog box or window where controls are placed. Also see [DIALOG GET CLIENT](#). Controls also have a client area, see [CONTROL GET CLIENT](#).

Clipping is where output (text and/or graphics) is restricted to a specific region or area of the output device. For example, if the word "Hello" is 100 pixels wide, and is drawn within a region with 90 pixels of width, the right half of the last letter ("o") is not drawn.

A combination of a text box and a drop down list box. It allows the user to select a pre-defined item in the list box, or optionally type a new value. See [CONTROL ADD COMBOBOX](#).

A dialog that is commonly used in many applications. Such as the File Open dialog, Print dialog, Font selection, and Color selection dialog.

A specialized window for obtaining user input, such as a button, a text box, a combo box, etc.

Windows requires the programmer to assign a numeric identifier to each control to distinguish it from other controls in the dialog.

The location and size of a dialog or control . The 'x' coordinate refers to the horizontal location, where zero (0) is the left-most position on the screen. The 'y' coordinate refers to the vertical location, where zero (0) refers to the top-most position on the screen. On a screen that is 640 pixels wide by 480 pixels deep, the right-most position is 639 (0 to 639 is 640 total pixels) and the bottom-most position is 479.

Also known as the mouse pointer. Typically, it is shaped like an arrow and allows the user to "point" to an item on the screen.

A control that has been customized. For example, an oval button, a grid control, or an owner-draw control (such as a check-list box)

A button that receives a `%BM_CLICK` message when the user presses the ENTER key while a dialog has the focus. This usually results in a `%BN_CLICKED` message being sent to the parent window. Adding the `%BN_DEFAULT` style to a button is the usual way to specify the default button for a dialog.

The desktop represents the user's primary work area; it fills the screen and forms the visual background for all operations. All visual output, including dialogs, is drawn on top of the desktop. The desktop region excludes the Taskbar. The desktop window handle is often referenced with `%HWND_DESKTOP`.

A device context permits your application to draw or write text within a specified region of a window or dialog. All GDI drawing functions require a device context handle. Drawing on a device context should only occur during a `%WM_PAINT` event.

A window that contains child windows, called controls, that allow the user to enter text, choose options and so forth. Dialogs offer additional features over standard windows, for example, by automatically handling TAB, ENTER and ESCAPE key actions.

Dialog box and control dimensions and coordinates are device-independent. Because a dialog box may be displayed on system displays that have widely varying pixel resolutions, dialog box dimensions are specified in system-character widths and heights instead of pixels.

This helps to ensure the best possible appearance of characters in the dialog, and enables dialog boxes to appear with similar proportions and appearance on the screen, even when the resolution and aspect ratio can significantly vary.

The dialog box base units are computed from the height and width of the system font. As the display resolution is changed, Windows changes the system font size accordingly, thereby adjusting the dialog unit size proportionally to the resolution.

By default, the actual font used in PowerBASIC's Dynamic Dialog Tools (DDT) dialogs is 8 point "MS Sans Serif". When designing your dialogs, it is a good idea to make controls slightly wider and taller than the text label size. This additional tolerance helps to ensure that the text is not clipped if the default dialog font is slightly larger in some resolutions. You can also use the [DIALOG FONT](#) statement to specify a custom default DDT font.

You can convert between Dialog Units and Pixels with the built in [DIALOG PIXELS](#) and [DIALOG UNITS](#) statements.

Quickly pressing a mouse button twice. For example, to activate an icon on your desktop, such as "My Computer", you should "double-click" while the mouse cursor is over the icon. A Text Box is often referred to as an Edit control. See [CONTROL ADD TEXTBOX](#).

A dialog box, window, or control that is "enabled" can process messages from the keyboard and mouse. A dialog box, window, or control that is "disabled" cannot. See [CONTROL ENABLE](#) and [CONTROL DISABLE](#).

A term used to describe an action, such as a button click. With PowerBASIC's DDT, events are processed inside of Callback Functions. Also see [CB.MSG](#).

An event-driven application typically sits idle, waiting for the user to interact with it (i.e., press a key on the keyboard or click the mouse) which triggers an "event" inside of the application. See [Callbacks](#).

At any given moment, only one window, dialog, or control can receive mouse-clicks and keystrokes. It control is then described as having "the focus". Controls are usually notified when they gain or lose focus; however, it should be noted that Windows does not explicitly guarantee that the control losing focus will receive its notification before the control that gains focus does. See [CONTROL SET FOCUS](#).

A set of instructions for describing how Windows should draw text on video displays and other output devices. In Windows, a font is a collection of characters and symbols that share a common design.

A dialog window with child control on it.

An empty box, typically drawn around a group of controls in a dialog to show the user that they are somehow related. See [CONTROL ADD FRAME](#).

A graphic control or window may be used to draw graphics and text. See [CONTROL ADD GRAPHIC](#) and [GRAPHIC WINDOW](#).

The term group is also used describe the relationship of child controls in a dialog, such as

with a "group" of radio buttons. A group of radio buttons is formed by the first control in the group having the `%WS_GROUP` style, and the first control after the group also having the `%WS_GROUP` style.

A 32-bit Long-integer (LONG) or Double-word (DWORD) variable containing a unique value for identifying a Windows object such as a window, control, bitmap, cursor, file, etc. See [CONTROL HANDLE](#).

A common prefix used to describe a 32-bit handle to a control. See [CB.CTL](#).

A common prefix used to describe a 32-bit handle to a dialog. `hDlg` may be a Long-integer or Double-word variable (i.e., `hDlg&` or `hDlg???`), but a Double-word variable is recommended. See [CB.HNDL](#).

A common prefix used to describe a 32-bit handle to a Window.

A small picture used in Windows to represent an item such as a program or a minimized window. See [DIALOG SET ICON](#).

Bitmaps and icons are often referred to as images.

An `ImageList` is a structure which contains any number of graphical images, either bitmaps or icons, but not a mixture.

A special window (control) that displays static text. Unlike buttons and other controls, the user cannot interact with a Label control. See [CONTROL ADD LABEL](#).

A special window (control) that displays multiple text items. The user can choose one or more from the list using the mouse or keyboard. See [CONTROL ADD LISTBOX](#).

A list of commands from which the user can select using the mouse or keyboard. When the user selects an item, Windows sends a message to the Callback Function to indicate the selection. See [MENU NEW BAR](#) and [Menus](#).

Windows is a message-based operating system. Whenever an event occurs which may require a reaction from the application, a message is sent to the appropriate Callback Function for notification. It is up to the programmer to identify and respond to messages. A Message is often considered by programmers to be an Event. Also see [CB.MSG](#) and [callbacks](#).

A message pump is a loop of code that retrieves messages from the message queue and sends them to the intended dialog or window callback. A message pump is mandatory for modeless dialog and modeless DDT applications. See [DIALOG DOEVENTS](#).

A modal dialog box requires the user to respond to a request before the application continues. Typically, a modal dialog box is used when a chosen command needs additional information before it can proceed. The parent window is disabled while a modal dialog is displayed. See [DIALOG SHOW MODAL](#).

A modeless dialog box allows the user to supply information to the dialog box and return to the previous task without canceling or removing the dialog box. This allows the user to work with more than one dialog box simultaneously. See [DIALOG SHOW MODELESS](#).

A mouse pointer is often referred to as a Cursor. See [MOUSEPTR](#).

A special button that is typically part of a group. Just like the push buttons on an automobile radio, only one maybe selected at a time. When one option button is checked (set) all other option buttons in the group are unchecked (unset or cleared). See [CONTROL ADD OPTION](#).

An overlay is an image that is drawn transparently over another image.

A window that owns one or more child windows. A child control is always displayed within the confines of its parent window. A child window is not constrained in the same manner. The smallest point or dot that can be displayed on the screen. A common screen resolution is 1024 pixels wide by 768 pixels high.

An application, including any DLLs loaded with the application. In Windows, each process is assigned its own address space, which cannot be directly read or written to by any other process.

A Push Button is commonly referred to as a Command Button or Button.

A radio button is commonly referred to as an option button. See [CONTROL ADD OPTION](#).

A central storage location that contains current information about the computer hardware and software configuration.

A collection of icons, menus, dialog boxes and other items embedded into a DLL or EXE for use by an application. See [#RESOURCE](#) and [resource files](#).

An element of the Windows user interface that converts mouse or keyboard input into values that a window procedure can use to shift the contents of a window's client area horizontally or vertically. Also used to display current location in a window relative to the size of the object displayed, by setting the size and position of the 'Thumb'. Scroll bar controls send %WM_HSCROLL and %WM_VSCROLL messages. See [CONTROL ADD SCROLLBAR](#).

A static control is often referred to as a Label. See [CONTROL ADD LABEL](#).

A bit-mask describing how Windows should draw a window, dialog box, or control.

A window or set of windows that belong to the same window class, and whose messages are intercepted and processed by another window procedure before being passed to the class window procedure. This allows the programmer to change the default behavior of a window or control (such as drawing a button as round instead of a rectangle).

A box at the left end of a caption activates a pop-up menu that contains the system commands (such as Close, Minimize, Maximize, Move, etc.).

The order in which controls are activated (receive the focus) when the TAB key is pressed.

A special window (child control) that allows the user to type in text from the keyboard. See [CONTROL ADD TEXTBOX](#).

A bar containing a row of buttons or other controls. They act as short-cuts for common operations, and usually appear just below the menu bar at the top of an window or dialog.

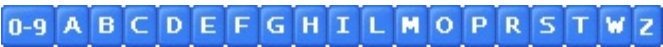
Win32 API The 32-bit Windows Application Programming Interface. A set of special DLLs included with Windows 95/98/ME and NT/2000/XP that contain pre-defined Subs and Functions your program can access. These Subs and Functions make up Windows itself and allow both the operating system and applications to draw on the screen, access devices like keyboards and a mouse, print to printers, provide networking and more.

A basic "on-screen" box used to contain and display information. It is the basis for all programs in the Windows operating system. Dialog boxes and the Child controls contained within are all windows in their own right.

A set of attributes that defines how a window looks and behaves. Before an application can create and use a window, a window class must have first been created and registered

for that window. Windows includes a number of built-in window classes, such as "BUTTON", "STATIC", "EDIT", and others. A custom control will have its own unique window class name.

The order that Windows draws each window and dialog on the screen. A window or dialog at the bottom of the z-order is drawn first. The next window is drawn on top, and the next on top of that, etc., until all windows and dialogs have been drawn on the screen. A window at the top of the z-order is displayed above all other windows.



0-9

[3-State](#)

A

[Accelerator](#)

B

[Bitmap](#)

C

[Callback](#)

[Caption](#)

[Caret](#)

[Check State](#)

[Checkbox](#)

[Child Window](#)

[Click](#)

[Client Area](#)

[Clipping](#)

[Combo box](#)

[Common Dialog](#)

[Control](#)

[Control ID](#)

[Coordinates](#)

[Cursor](#)

[Custom Control](#)

D

[Default Button](#)

[Desktop](#)

[Device Context](#)

[Dialog](#)

[Dialog Units](#)

[Double-Click](#)

E

[Edit Control](#)

[Enable/Disable](#)

[Event](#)

[Event-Driven](#)

F

[Focus](#)

[Font](#)

[Form](#)

[Frame](#)

G

[Graphic](#)

[Group](#)

H

[Handle](#)

[hCtl](#)

[hDlg](#)

[hWnd](#)

I

[Icon](#)

[Image](#)

[ImageList](#)

L

[Label](#)

[List Box](#)

M

[Menu](#)

[Message](#)

[Message Pump](#)

[Modal](#)

[Modeless](#)

[Mouse Pointer](#)

O

[Option Button](#)

[Overlay](#)

P

[Parent](#)

[Pixel](#)

[Process](#)

[Push Button](#)

[Radio Button](#)

R

[Registry](#)

[Resource](#)

S

[Scroll bar](#)

[Static control](#)

[Style](#)

[Subclass](#)

[System Menu](#)

T

[Tab order](#)

[Text box](#)

[Toolbar](#)

W

[Win32 API](#)

[Window](#)

[Window Class](#)

Z

[Z-Order](#)

Style

A bit-mask describing how Windows should draw a window, dialog box, or control.