CONSOLE TOOLS™

This is version 2.56 of the
Console Tools Help File
Build ID# 030703

Copyright and Trademark Information

CONSOLE TOOLS™

© 1998-2003 Perfect Sync, Inc.

TABLE OF CONTENTS

START HERE!

This Help File was written so that it could be read like a book. All of the pages are linked together in sequence, so you can simply click the **>>** button near the top of the screen to go from page to page. If you want to get the most out of it, you should start reading on this page and keep going at least until you've read the entire "User's Guide" section.

Yeah, right.

Ok, we know. You want to get started in a hurry. Typical programmer.

Fine.

PLEASE read the next few pages, at least. They contain the information that you'll *need* to get started. Particularly the pages called Three Critical Steps, Using Console Tools Functions, and Using the Predefined Equates.

The User's Guide is a narrative that walks you through the basics.

The Reference Guide is an alphabetical listing of all of the Console Tools functions.

The Appendices are narratives that describe different aspects of using the console window and console applications, plus several lists of things like Error Codes, Accelerator Key ID Numbers, Console Window Color Numbers, and so on.

Seriously, we hope you'll take this Help File seriously. A lot of effort went into creating it -- almost as much as the Console Tools DLL itself -- and it can provide you with a lot of information about using console windows, console applications, and, of course, Console Tools.

What is "Console Tools"?

Console Tools is a collection of functions that were designed for programmers that use the PowerBASIC Console Compiler (PB/CC). There are two versions of Console Tools currently available. For the features contained in each, see Console Tools Standard Features and Console Tools Pro Features. (And for information about Console Tools Plus Graphics, click here.)

PB/CC is a high-performance 32-bit BASIC compiler for Window 95, Windows 98, Windows ME, Windows NT, Windows 2000, and Windows XP, which, as a group, are called "Win32". PB/CC Version 2.00 was released in June of 1999. All versions of Console Tools are 100% compatible with both Version 1 and Version 2 of PB/CC.

PB/CC creates "Console Applications". Console Apps are different from most Windows programs because they don't normally have a Graphical User Interface (GUI). They have what looks like a DOS interface: an 80-column-by-25-row text screen

But they're different from DOS programs, too. They are true 32-bit Windows applications, so they can take advantage of Win32's fast 32-bit processing and the virtually unlimited memory that's available on most Windows machines. The programs that PB/CC produces are fully compiled, very small, and lightning fast. (For more information about PB/CC, you should visit http://powerbasic.com or contact PowerBASIC at (800) 780-7707 or (831) 659-8000.)

Console Apps are ideal for terminal emulators, command-line programs, data-entry-intensive programs, scientific applications, and many other types of programs that don't really require a GUI. And they're perfect for creating Windows versions of DOS programs, because they allow the same well-proven screens and menu layouts to be moved to Windows without the disruption of the user having to learn a 100% new program.

The only real disadvantages of Console Apps are that the console can be difficult to manage, and they can be a little dull. With no graphics and sound effects, or any of the other "goodies" that people are used to seeing on a modern computer screen, a Console App can look a little old-fashioned next to a flashy Windows program.

Console Tools lets you have the best of both worlds.

Console Tools allows you to add GUI elements like standard Windows Message Boxes, Input Boxes, Progress Boxes, "Splash" Boxes, List Boxes, and Pulldown/Popup Menus to your PB/CC Console Apps.

Console Tools also provides a wide range of features that have little or nothing to do with GUI elements. Console Tools functions provide a wealth of information about the state of the console window (size, location, maximized/minimized/fullscreen/etc.). Console Tools also provides you with functions to programmatically change the console screen's size, location, and state. It also provides screen-buffer POKE and PEEK functions, screen BLOAD and BSAVE functions that can work with disk files or Console Tools built-in Screen Buffers, and a wide variety of other console-related programming tools.

Console Tools Standard Features

- Console Tools Console Window Control features include functions that allow you to change your program's title bar text and icon, control your program's Window State (maximized, minimized, hidden, etc.), control the size and screen location of the console window, and control many other aspects of the console window.

- The Console Window Information functions can give your program useful information about its own Window State the size and screen location of the console window, useful facts like whether or not the console window has Scroll Bars, and on and on.

- Console Message Boxes look and act just like standard Windows Message Boxes, except that they "behave" when used in a console application. (Standard Windows Message Boxes can be very hard to handle in console apps -- *especially* Windows 95/98/ME apps with Maximized console windows. Console Tools handles all of those problems for you.)

- Console Input Boxes can be used to get simple text input from your user, or numeric input, or "password" input where the typed characters are displayed as stars.

- Console Tools Splash Boxes can be used for pop-up messages of almost any kind. Unlike Message Boxes (which require the user to click a button) Splash Boxes can be left on the screen while your program does other things. They're great for logo/copyright notices and other types of "non-intrusive" messages that don't require a response from the user.

- Console Tools Progress Boxes look like the familiar "Percent Done" displays that have a colored bar (usually blue) that gets longer as the job progresses.

- Console List Boxes provide your programs with an easy way to present a list of choices to the user, with a familiar Windows look and feel.

- You can use Console Tools to Save and Load Screens or even *portions* of screens on a row-by-row basis. Screens can be saved to disk, or saved in one of the eight Console Tools Screen Buffers.

- The ConsolePEEK and ConsolePOKE functions allow you to access the console screen buffer's characters and attributes.

- And lots of other miscellaneous tools like a DELAY function and a function that can tell you whether or not two copies of your program are running at the same time.

Also see Console Tools Pro.

Console Tools Pro Features

Console Tools Pro offers the same features as the Standard Console Tools DLL, plus several additional functions and some significant extensions/upgrades of the standard functions.

- A complete Pulldown Menu system that you can use in your programs, with professional details like accelerator keys, checkmarks, disabled items, and horizontal and vertical separator bars.
- Console Tools Pro can also be used to create non-pulldown "DOS-Style" menu systems.
- A complete PopUp Menu system, which allows you to create right-click "context" menus.
- Custom Fonts that you can design and use in Splash Boxes and Input Boxes.
- The Windows Handle of the Custom Font is also made available, so you can use it with Windows API functions.
- Console Tools Pro provides 256 Console Screen Buffers instead of just eight, so you can save *many* more screens (and partial screens) in memory, for use later.
- Two-field versions of the Console Input Box are included in the Pro DLL, for things like User Name / Password input.
- A Mini Word Processor Input Box that can be used to edit text (and files) up to 30,000 characters in length.
- A "read-only" version of the Mini Word Processor is provided, for viewing large amounts of text.
- The Console Tools Pro OnTimer function allows your programs to execute a pre-defined function at a specific interval, and the OnCtrlBreak and OnShutdown features allow you to detect and act upon events that normally shut down a console application without warning. The OnStateChange function can detect and act upon a change in your program's Window State.
- Console Tools Pro also allows your PB/CC applications to be minimized to the Windows System Tray (instead of the Task Bar).

Available License Terms

IMPORTANT NOTE: Your use of Console Tools is legally restricted by the terms of the Software License that you purchased from Perfect Sync, Inc. You are legally responsible for making sure that the terms of the License Agreement are followed.

The number of computers on which the Console Tools development package may be installed is limited by the Software License.

If you develop Shareware, Freeware, Public Domain, or Demo software that requires the use of Console Tools, you must either ***1)*** *distribute the CTDEMO.DLL file instead of the CONTOOLS.DLL file, or* ***2)*** *contact Perfect Sync (*support@perfectsync.com*) and request permission to distribute the CONTOOLS.DLL file with your not-for-profit program.*

Please read Licensing Terms and DLL Distribution Rights for complete information about your legal rights and responsibilities.

Software License Agreement and DLL Distribution Rights

PLEASE READ THIS ENTIRE SECTION. It describes your legal rights and obligations.

Console Tools Standard License
The Console Tools Standard License allows you to install the Console Tools development package (the contents of the Console Tools Installation File as originally received from Perfect Sync or from an Authorized Reseller) on a single development computer, to use the package for software development, and to distribute the Console Tools Standard Runtime File(s) with applications which you develop, which require the Runtime File(s) to operate properly, and which add significant functionality to the Runtime File(s).

Console Tools Pro License
The Console Tools Pro License allows you to install the Console Tools Pro development package (the contents of the Console Tools Installation File as originally received from Perfect Sync or from an Authorized Reseller) on up to four (4) development computers, to use the package for software development, and to distribute the Console Tools Pro Runtime File(s) with applications which you develop, which require the Runtime File(s) to operate properly, and which add significant functionality to the Runtime File(s).

IMPORTANT NOTE: Each Console Tools Runtime File is serialized. The unique Authorization Code that is embedded in each copy of the Runtime File(s) will allow Perfect Sync to attribute unauthorized or improper distribution to the original licensee. Attempting to change the embedded Authorization Code is a violation of U.S. and international law, and the Runtime File(s) will self-deactivate or malfunction if tampering is detected. Perfect Sync cannot be held responsible for damage to computer files that may be caused by a Console Tools Runtime File that has been intentionally altered. (See LIMITED WARRANTY below.)

If you have not purchased a Console Tools Software License from Perfect Sync or an authorized distributor then you are not legally entitled to use the Console Tools Runtime File(s) for software development or to distribute the Console Tools Runtime File(s) in any manner whatsoever. You may be violating the law and may be subject to prosecution if you distribute this product or use it for software development. Please refer to the U.S. Copyright Act (and other applicable U.S. and international laws and treaties) for information about your legal obligations regarding the use and distribution of copyrighted and other legally protected works.

Software License Agreement

By using the Console Tools DLL for software development you agree to the following terms:

This Software License is an agreement between the Licensee ("you") and the Licensor (Perfect Sync, Inc.). By installing Console Tools (the "software") on a computer system and/or by using the Console Tools machine-executable files (the "Runtime Files") for software development, you agree to the following terms:

LICENSE
The software and documentation is protected by United States copyright law and international treaties. It is licensed for use on a single computer system (Console Tools Standard License) or on four computer systems (Console Tools Pro License). If this software is installed on a computer network, you must obtain a separate license for each network workstation (or group of four workstations) where the software can be used for software development, regardless of whether or not the software is actually used concurrently on multiple workstations.

DISTRIBUTION

Only individuals or corporations that have purchased a Console Tools License from Perfect Sync or from an authorized distributor may reproduce and distribute the Console Tools Runtime File(s), and then only with application(s) that 1) are written by the licensee, 2) require the Runtime File(s) to operate, and 3) add significant functionality to the Runtime File(s). In that case, and provided that your application bears your complete and legal copyright notice or the following notice (in no less than a 10pt font)...

Portions © Copyright 2001 Perfect Sync, Inc

...you may distribute the Console Tools Runtime File(s) royalty free.

The Perfect Sync Authorization Code which is provided in human-readable form with the Console Tools installation package is also embedded in the Console Tools Runtime File(s) and is considered to be part of the Runtime File(s). The Authorization Code may be distributed as part of a machine-readable computer program that meets the requirements above, but it may not be distributed in human-readable form (including source code), disclosed physically, electronically, or verbally to any third party, or distributed in any other form. Disclosure or improper distribution of the Authorization Code would allow the unauthorized use of the Console Tools Runtime File(s) by others, and is legally equivalent to the unauthorized distribution of the Runtime File(s) themselves.

No other portion of the Console Tools package, including documentation, header files, and sample program code, may be distributed in any form except by Perfect Sync or an authorized distributor.

LIMITED WARRANTY

Perfect Sync, Inc. warrants that the physical disks (if any) and physical documentation (if any) are free of defects in workmanship and materials for a period of thirty (30) days from the date of purchase. If the disks or documentation are found to be defective within the warranty period, Perfect Sync, 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.

PERFECT SYNC, 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 Michigan, in the United States of America, and any action hereunder shall be brought only in Michigan. 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 Perfect Sync, Inc.

If you have any questions about your rights and responsibilities under this Software License, please contact Perfect Sync, Inc. 8121 Hendrie Blvd., Suite C, Huntington Woods, Michigan (USA) 48070.

You can reach us by electronic mail at support@perfectsync.com, or via fax at (248) 546-4888.

Console Tools Authorization Codes

This is a topic that all Console Tools programmers should read and understand thoroughly. If you have any questions about it, please contact support@perfectsync.com***.***

Unfortunately, not everybody is honest and not everybody obeys the law. That's the reason that our houses have locks on their doors.

We at Perfect Sync have every expectation that *you*, as a Console Tools licensee, intend to comply with the terms of the Console Tools License Agreement. But it would be very difficult for you to guarantee that *everybody who uses your program* will be equally honest, especially if your program is widely distributed or if it is available for download from the internet.

Like many programming tools, Console Tools contains certain security measures that make it more difficult for people to use it illegally. Notice that we said "more difficult", not "impossible". Frankly there is no such thing as 100% security when it comes to protecting a computer program from illegal use. If a "cracker" is determined enough, and has enough time, they can bypass virtually any security system. Just as a determined thief can break into your home, office, or car.

Every Console Tools Runtime File (i.e. each DLL) is serialized. That means that your copies of the Runtime Files contain a unique, embedded key number called an Authorization Code. Nobody else's Console Tools Runtime Files have the same Authorization Code as your copies of the Runtime Files. This allows Perfect Sync to identify a Console Tools Runtime File that is being used illegally (i.e. distributed in violation of the Console Tools License Agreement) and to determine the identity of the original licensee.

In order to use a Console Tools Runtime File, you must prove to the Runtime File that you know its correct Authorization Code by using the ConsoleToolsAuthorize function. This is done so that when you distribute the Console Tools Runtime Files *legally*, nobody else will be able to remove them from your program and use them *illegally*. They won't have the correct Authorization Number, and the Runtime Files will not function properly without it.

Protect Your Authorization Code!

Your Authorization Code must be treated as confidential information. If your Authorization Code becomes known to other people, it will allow them to use your copy of the Console Tools Runtime File(s) illegally. YOU are legally responsible for preventing that from happening!

Using the ConsoleToolsAuthorize Function

If you don't use the ConsoleToolsAuthorize function at all, the InitConsoleTools function will refuse to work, making it impossible for your program to use Console Tools in any way.

If you use the ConsoleToolsAuthorize function with the Authorization Code that matches your Runtime File -- using the exact Code that was provided *with* the Runtime File -- it will work normally.

But it's not quite *that* simple...

It would be relatively easy for somebody to write a program that used the ConsoleToolsAuthorize function to test all of the possible Authorization Codes one by one, until it found one that worked with your Runtime File. The ConsoleToolsAuthorize function returns `SUCCESS` when it accepts an Authorization Code, so all it would take would be a simple "loop" program that stopped when the correct Code was found.

So the ConsoleToolsAuthorize function also returns `SUCCESS` when certain other codes are used.

There are approximately 4.2 billion possible Authorization Codes. Of those, only one is the correct Code for your Runtime File, but about 64,000 "Dummy Codes" will also cause the ConsoleToolsAuthorize function to return `SUCCESS`. This makes it much more difficult to use the ConsoleToolsAuthorize function to determine the correct Authorization Code for a given Runtime File.

THIS IS A VERY IMPORTANT POINT: If one of the 64,000 Dummy Codes is used instead of the correct code, the ConsoleToolsAuthorize function will return `SUCCESS`, the InitConsoleTools function will work properly, and *all other Console Tools functions will appear to work properly*. But in reality, the Console Tools Runtime File will purposely malfunction. At random intervals, many different Console Tools functions will produce results that are completely or partially incorrect. For example, every so often a function like ConsoleWindow will fail. Or a ConsoleMessageBox will fail to appear. Or certain values might be set to zero. This will make the Console Tools Runtime Files seem to work properly *most* of the time, but they will be unreliable.

Don't worry, the Console Tools Runtime Files have been tested *extremely* thoroughly to make sure that no random errors will be produced when the *correct* Authorization Code is used.

And we have taken great care to make sure that a simple typo will not result in a Console Tools program that malfunctions unexpectedly. Among other things, the code numbers have been chosen so that accidentally mis-typing *any single digit* of a valid Authorization Code will *never* produce a Dummy Code that ConsoleToolsAuthorize will accept. If you mis-type two of the eight digits of a valid Code there is less than a one-in-10,000 chance that you will accidentally type a Dummy code that ConsoleToolsAuthorize will accept. If you mis-type three out of eight digits... well, you should probably take typing lessons before attempting to use Console Tools.

With just a little bit of care when you type the Authorization Code into your program, you can rest assured that Console Tools will work properly from that point forward.

IMPORTANT NOTE: Be sure to test the return value of the ConsoleToolsAuthorize function to make sure that it is `SUCCESS`. This will virtually guarantee that you typed the Authorization Code correctly, and that the Console Tools Runtime File will work properly.

Installing Console Tools on your Development Computer

IMPORTANT INFORMATION!
You must perform these steps before
using Console Tools for the first time!

Console Tools is provided as a Setup Program that unpacks all of the necessary disk files. Simply execute the program and it will walk you through several choices, such as the name of the directory where Console Tools will be installed. The default directory is `\CONTOOLS`, and the rest of this section will assume that you used the default.

Console Tools Standard uses a runtime file called `CT_Std.DLL`, and Console Tools Pro uses `CT_Pro.DLL`. We refer to these as the `CT_*.DLL` files. These files are used internally by all Console Tools programs.

In order for your programs to be able to *use* the appropriate Console Tools DLL, they will need to be able to *find* it. In most cases it will be necessary for you to place a second copy of the `CT_*.DLL` file somewhere on your computer's hard drive. (We recommend that you leave the original copy in the `\CONTOOLS` directory, to serve as a backup.)

The ideal location for the `CT_*.DLL` file is *the same directory as the executable program that you are developing*, but keep in mind that you may be developing console programs in more than one directory. If that is the case, you may choose to place the DLL in your Windows System Directory. On Windows NT/2000/XP systems, place a copy of the DLL in the `\WinNT\System32\` directory. On Windows 95/88/ME systems, place it in `\Windows\System\`. (It is important to note that, by default, Windows Explorer hides the System Directory from view. If you have not already done so, we recommend changing your Explorer settings to "Show hidden files and folders".)

If you write a program using ConsoleTools, and when you run it you see a Windows message box that says something like...

The dynamic link library CT_Pro.DLL could not be found in the specified path

...it means that Windows was unable to link Console Tools to your program's EXE. This almost *always* means that the `CT_STD.DLL` or `CT_PRO.DLL` file needs to be copied to a location where your program can find it.

Your Authorization Code

The first time you attempt to run a Console Tools sample program you will see an error message about a line of code that looks something like this:

```
        MY_CT_AUTHCODE = &h........
```

Type your Console Tools Authorization Code where you see the eight dots, and save the file. From then on you will be able to run the sample programs (and programs that you write) without re-entering your code.

That's all there is to it! Console Tools is now installed and ready for use.

We suggest that you use the Windows Explorer program to examine the files that were placed in the `\CONTOOLS` directory. A variety of icons, sample programs, and other files are provided. You should also look at your system's Start Menu, where you will find several Console Tools components listed under the heading "Perfect Sync Development Tools".

Now you're ready to begin using Console Tools. See Getting Started: Three Critical Steps for Every Program

Getting Started: Three CRITICAL Steps for Every Program

Actually, if you're starting a *new* programming project there's only one step. Paste the contents of the `\CONTOOLS\SKELETON.BAS` file into your program. It contains everything you'll need to get started on a project. (Or if you click here you will see a copy of the `SKELETON.BAS` program that you can cut-and-paste directly from this Help File.)

If you're adding Console Tools to an *existing* program...

You must follow these steps whenever you add Console Tools to a project. *Failure to do so will cause Console Tools functions to fail, and it will often cause Application Errors (very serious Windows errors, also known as General Protection Faults) when your program is run.*

STEP 1   Locate your program's WinMain or PBMain function. Every PB/CC program has one; it will look like *one* of the following four lines:

```
FUNCTION WinMain(several parameters)

FUNCTION Main(several parameters)

FUNCTION PBMain

FUNCTION PBMain AS LONG
```

Add one of the two following lines of code to the beginning of your program at some point *before* the beginning of the WinMain or PBMain function. This line must not be located inside a sub or function; it must be part of your program's "main" code. (You should change the `\CONTOOLS\` portion of the code if you chose to install Console Tools in a different directory.)

If you are using Console Tools **Pro**...

```
$INCLUDE "\CONTOOLS\CT_PRO.INC"
```

Or, if you are using the Console Tools **Standard**...

```
$INCLUDE "\CONTOOLS\CT_STD.INC"
```

IMPORTANT NOTE: If your program uses the WIN32API.INC file that is supplied with PB/CC, you should add the line above *after* the line that says `$INCLUDE "WIN32API.INC"`. This is because the two files contain duplicate equates. The Console Tools INC files automatically recognize that certain equates have already been defined, and they do not redefine them. If you include the Console Tools INC file *before* the WIN32API.INC file, you may get a PB/CC compiler message that says "**Duplicate Named Constant".** (Please note that the use of Console Tools does not *require* the use of the WIN32API.INC file.)

STEP 2   Add the following line immediately *after* the beginning of your WinMain or PBMain function. Ideally, it should be the *very first executable line* of code in your program.

```
ConsoleToolsAuthorize %MY_CT_AUTHCODE
```

STEP 3   Add the following line immediately *after* the ConsoleToolsAuthorize line...

```
InitConsoleTools 0, 0, 0, 0, 0, 0
```

NON-PB/CC USERS PLEASE NOTE: If your program is a *non-native* console application (i.e. a program which uses the Windows "AllocConsole" API function to create its own console window) please read the **Remarks** section of InitConsoleTools. Most programs -- including all PB/CC programs -- are *native* console applications and do not need to be concerned with the additional remarks.

That's it! After you've performed those three steps, your program can use *most* of the Console Tools functions that are described in this Help File. See Using Console Tools Functions.

IMPORTANT NOTE: If your program uses 1) the Console Tools Pulldown/Popup Menu feature or 2) the Console Tools ScreenSave/ScreenLoad feature, or 3) Graphics Tools, you will have to modify the InitConsoleTools line (just above) before you can use those features! Please refer to the User's Guide under Pulldown and Popup Menus and/or Saving and Loading Screens. If you are using Graphics Tools (Console Tools Plus Graphics) please see ConsoleGfx.

Using the Console Tools Functions

Once you have installed Console Tools and have performed the Three Critical Step for Every Program, you can use all of the Console Tools functions as if they were part of the PB/CC language.

We have divided all of the various Console Tools functions into several groups to help you learn about related functions more quickly.

- Getting Runtime Information about the Console Window

  ConsoleInfo, ConsoleState, ConsoleIsMinimized, ConsoleIsMaximized, ConsoleIsForeground, ConsoleIsFullScreen, ConsoleIsHidden, ConsoleHasScrollBars, ConsoleHasToolbar, ConsoleMetrics, MouseOverCol, MouseOverRow, MouseOverConsole, LocOfCol, LocOfRow, ColOfLoc, RowOfLoc

- Controlling the Console Window

  ConsoleTitle, ConsoleIcon, ConsoleSysTrayIcon, ConsoleWindow, ConsoleNormal, Console80x, ConsoleToolBar, ToggleConsoleToolbar, ToggleFullScreenMode, ConsoleToForeground, ConsoleToTop, ConsoleTopMost, ConsoleFocus, ConsoleMove, ConsoleKey, DeleteWindowMenuItem, DefaultWindowMenu, ConsoleControl, ConsolePropMenu

- Console Message Boxes

  ConsoleMessageBoxDefaults, ConsoleMessageBox

- Console Input Boxes

  ConsoleInputBoxDefaults, ConsoleInputBox, ConsoleInputBoxCancel

- Splash Boxes

  SplashBoxDefaults, SplashBoxShow, SplashBoxRefresh, SplashBoxHide

- Progress Boxes

  ProgressBoxDefaults, ProgressBoxShow, ProgressBoxCancel, ProgressBoxUpdate, ProgressBoxHide, ProgressBoxRefresh

- Console List Boxes

  ConsoleListBox, ConsoleListBoxDefaults

- The "On" Functions

  OnTimer, OnCtrlBreak, OnShutdown, OnStateChange

- Pulldown and Popup Menus

  MenuDefinition, MenuSystemCreate, PulldownMenu, PopUpSystemCreate, PopupMenu, MenuItemProperty, MenuSystemDestroy

- Saving and Loading Screens

  ConsoleScreenSave, ConsoleScreenLoad, ClearSavedScreen

- Peeking and Poking the Screen Buffer

  ConsolePOKE, ConsolePEEK

- Miscellaneous Functions

  InitConsoleTools, InitCTInternals, AlreadyRunning, ConsoleToolsVersion, ConsoleToolsSerialNumber, WindowsVersion, CustomFont, iString, Delay

- Low Level Functions

  hConsoleWindow, hConsoleWindowMenu, hCustomFont, VirtualKey

- Graphics Functions (Console Tools Plus Graphics ONLY)

  ConsoleGfx, GfxTextHole, ConsoleColor, MatchConsoleFont

Getting Runtime Information about the Console Window

These functions will provide most of the information that you'll need about the console window.

ConsoleState provides information about the overall Window State of the Console, i.e. whether it is Minimized, Maximized, Hidden, FullScreen, and so on.

ConsoleInfo provides information about the size and location of the console window and the computer screen. You can use ConsoleInfo to get data about the Console Screen Buffer, the console window's "Print Area" (the usually-black area where text is printed), the entire console window (including the title bar and borders), the entire computer screen, and the "Usable Desktop" (the computer screen minus the task bar).

ConsoleMetrics provides information about the individual elements of the console window, such as the console font, the title bar, and the border that surrounds the console window.

ConsoleIsForeground can tell your program whether or not it is currently in the "Windows Foreground", i.e. whether or not your program is "active" and has the keyboard focus.

ConsoleHasScrollBars is a function that can tell your program whether or not the current console window has Scroll Bars. Console windows with Scroll Bars are much more difficult to manage, so you'll probably want to avoid them.

ConsoleHasToolbar can tell your program whether or not the Windows 95/98/ME console toolbar is currently visible.

The MouseOverCol and MouseOverRow functions can tell your program which row/column the Windows mouse cursor is currently located over, and the MouseOverConsole function returns a True value if the mouse is currently located anywhere over the console window's "print area".

The LocOfCol and LocOfRow functions can be used to obtain the actual screen locations (in pixels) of console columns and rows, and the ColOfLoc and RowOfLoc functions can be used to obtain the column/row locations of pixel locations.

The rest of the console window information functions (ConsoleIsMinimized, ConsoleIsMaximized, ConsoleIsFullScreen, and ConsoleIsHidden) are convenience functions that duplicate or enhance some of the more commonly used features of ConsoleState.

Controlling the Console Window

Many different Console Tools functions fall into this category.

ConsoleTitle and ConsoleIcon can be used to change your program's Title Bar.

ConsoleSysTrayIcon can be used to place your program's icon in the Windows System Tray instead of the Task Bar.

DeleteWindowMenuItem and DefaultWindowMenu can be used to change the title bar's "Window Menu".

ConsoleMove can be used to change your program's location on the screen.

ConsoleWindow can be used to Maximize, Minimize, Un-Minimize, Restore, Hide, and Show your program (as well as several other variations on those options). It can also switch your program into the FullScreen or Windowed mode. (ToggleFullScreenMode can also be used for this purpose.)

ConsoleNormal is an easy-to-use function that's designed to put the console window into a "normal" state (not hidden, not fullscreen, etc.) regardless of its current settings.

Console80x is used to switch the console window to common 80-column configurations like 80x25, 80x43 and 80x50.

ConsoleToolBar and ToggleConsoleToolbar can be used to hide and show the Windows 95/98/ME Console Toolbar.

ConsolePropMenu can be used to display the console window's Properties Menu.

ConsoleToForeground, ConsoleToTop, ConsoleTopMost, and ConsoleFocus are used to affect your program's place in the Windows "Z-Order" and the Windows Keyboard Focus.

ConsoleControl provides low-level control over things like the console window's buffer size and the size (and scrolling) of the console window itself.

ConsoleKey can be used to send "virtual keystrokes" to your program.

Console Message Boxes

You'll only need to learn about two functions to use Console Message Boxes. Actually, you can get by with one...

ConsoleMessageBox is used to display a Message Box on the screen. The function returns a value that corresponds to the button that the user clicked to get rid of the Message Box. (You can also use the simplified MSGBOX syntax, but all that does is call the ConsoleMessageBox function with predefined parameters.)

The use of the ConsoleMessageBoxDefaults function is completely optional. If you want to, you can use it to establish default values for any or all of the Message Box settings, to simplify the use of the ConsoleMessageBox function.

Console Input Boxes

Using Console Input Boxes requires knowledge of only two basic functions. A third "convenience" function is also provided.

ConsoleInputBox is used to display an Input Box on the screen. The return value of the function is the string that the user types, or the "default" string if the user selects the Cancel button.

Since a function can return only one value, and since the ConsoleInputBox function returns the string that the user types, the value of the ConsoleInputBoxCancel function can be checked immediately after ConsoleInputBox is used, to determine whether or not the user clicked the Cancel button. (Your program may or may not need this information.)

The use of the ConsoleInputBoxDefaults function is completely optional. If you want to, you can use it to establish default values for any or all of the Input Box settings, to simplify the use of the ConsoleInputBox function.

Splash Boxes

You can use basic Splash Boxes after you've learned about two simple functions. More advanced Splash Boxes require a third function, and a fourth "convenience" function can also be used.

SplashBoxShow and SplashBoxHide are fairly self-explanatory. They Show and Hide Splash Boxes. The easiest way to use these functions is to **1)** use the SplashBoxShow "delay" parameter to tell Console Tools how long to display the Splash Box, and then **2)** use SplashBoxHide immediately after SplashBoxShow.

If you want a Splash Box to be displayed while your program is working on something else, you'll need to learn about SplashBoxRefresh.

The use of the SplashBoxDefaults function is completely optional. If you want to, you can use it to establish default values for any or all of the Splash Box settings, which will simplify the use of the SplashBoxShow function.

Progress Boxes

There are three basic Progress Box functions, and a fourth function is required if you want your Progress Box to have a Cancel Button. A "convenience" function is also provided.

ProgressBoxShow and ProgressBoxHide are the basic tools for displaying Progress Boxes. You will use ProgressBoxShow once to "set up" the Progress Box text, title bar, and screen location, and then your program will use it again, repeatedly, whenever it needs to change the display. Finally, when the task reaches 100%, your program will use the ProgressBoxHide function to get rid of the Progress Box display.

If you simply want to update the "percent done" display (without changing the Progress Box's text, title bar, or screen location) you can use the ProgressBoxUpdate function instead of repeatedly using ProgressBoxShow.

If your Progress Box has a Cancel Button, your program can periodically check the ProgressBoxCancel function to find out whether or not the user has clicked Cancel. If they have, your program can respond by either displaying an "Are You Sure?" message box, or by simply quitting.

The use of the ProgressBoxDefaults function is completely optional. If you want to, you can use it to establish default values for any or all of the Progress Box settings, which will simplify the use of the ProgressBoxShow function.

(The use of the ProgressBoxRefresh function, which was part of Console Tools version 1.00, is not necessary with Console Tools version 2.00 and above. Progress Boxes are now refreshed automatically.)

Console List Boxes

Creating Console List boxes only require the use of a single function, called ConsoleListBox. It is used to display a List Box and obtain the user's choice of the available items.

The use of the ConsoleListBoxDefaults function is completely optional. If you want to, you can use it to establish default values for most of the List Box settings, which will simplify the use of the ConsoleListBox function.

The "On" Functions

The "On" functions all tell Console Tools to perform certain operations when certain events occur.

The OnTimer function tells Console Tools to execute a particular function every time that a timer expires.

The OnCtrlBreak function tells Console Tools what to do when somebody presses Ctrl-Break. You can ignore it, sound a beep, pass it along to the PB/CC `INKEY$` function, or execute a particular function.

The OnShutdown function allows your program to intercept and handle Close, Windows Shutdown, and Windows Logoff events. (Please note that the Windows 95/98/ME console window does not support the detection of the Close event, so Console Tools is only able to support that event on Windows NT/2000/XP computers. The other events are supported on all versions of Windows.)

The OnStateChange function tells Console Tools to execute a particular function whenever your program's Window State changes to Maximized, Minimized, or Restored.

Pulldown and Popup Menus

Unlike most other Console Tools features, the use of Pulldown Menus and Popup Menus require some "programmatic preparation". An Input Box or a Message Box is a one-line function; but a Pulldown or Popup Menu requires several lines of code, at least.

This section of the Console Tools Help File is designed to walk you through the entire process for the first time. Once you get the idea, you can change the procedure.

STEP 1 Add an equate that is defined as `%MAXMENUBUFFER = 64` to the *very beginning* of your program. The arbitrary value "64" will allow you to use up to 64 Menu Buffers -- 64 different menus and submenus -- in your program. You can use a number as large as 1024 if you think you'll need that many.

STEP 2 Locate the InitConsoleTools line that you added to the beginning of your program (see Three Critical Steps For Every Program). Change the *second parameter* from zero (0) to `%MAXMENUBUFFER`. This tells the Console Tools DLL to prepare 64 Menu Buffers for your program. You probably won't need that many, at least not at first, but it's a convenient value to use during development.

STEP 3 Since Pulldown and Popup Menus can be created only by the Console Tools *Pro* DLL, if your program will be used on more than one computer it should check the ConsoleToolsVersion function for a *positive value* when it starts up. If a negative value is returned, your program won't be able to create menus later. An alternative method is to check the return value of the InitConsoleTools function in Step 2. If you use a valid value (from 1 to 1024) for the second parameter and the function returns `%ERROR_CT_INVALIDMENUNUMBER` then the Pro DLL is *not* available.

STEP 4 Use the MenuDefinition function to define all of your menus. You'll use it once for each top-level menu and once for each submenu. This is usually done during the "initialization code" of a program, but it can be done (and re-done) at any time. Don't worry about getting them right the first time... you can change them later. This Help File contains a complete description of the syntax for MenuDefinition. IMPORTANT NOTE: At least during testing, your program should check the return value of the MenuDefinition function, to warn you about errors in your menus *before* they cause serious problems. Failure to do this can cause an Application Error (a General Protection Fault) later.

STEP 5 Use either the MenuSystemCreate function *or* the PopupSystemCreate function to "build" the Menu System, depending on the type of menu that you want to create. During the development process you can use...

```
MenuSystemCreate 1,%MAXMENUBUFFER
```

or

```
PopUpSystemCreate 1,%MAXMENUBUFFER
```

`...`to tell Console Tools to build all 64 of the possible menus and submenus, even if you haven't used them all. IMPORTANT NOTE: At least during development, your program should check the return value of the "system create" functions for Error Codes. Failure to do this can cause an Application Error (General Protection Fault) later.

Note: It is possible to create both pulldown and popup menus in the same program. For example, you might use buffers 1-32 for pulldown menus, and 33-64 for popup menus. But any given buffer number cannot be used for both types of menus at the same time.

Attempting to use a popup menu system for a pulldown menu, or vice versa, will result in an Application Error.

STEP 6 Select either 6A or 6B:

> **STEP 6A (Pulldowns)** The most common way to use the PulldownMenu function -- the function that actually displays a pulldown menu system and returns the user's selection -- is to put it in a DO/LOOP structure. Sample code is provided in the PulldownMenu example. You can cut and paste the example code directly into your program, if you want to.
>
> **STEP 6B (Popups)** The most common way to use the PopupMenu function -- the function that actually displays a popup menu and returns the user's selection -- is to check the result of the PB/CC INKEY$ function and respond to a right-click on the console. Sample code is provided in the PopupMenu example.

STEP 7 Change the first parameter of the PulldownMenu or PopupMenu function to the menu buffer number of the top-level menu in your menu system. Most people use Menu #1, but any menu buffer can be used for a top-level menu. For example, if your top-level menu is Menu #1 and you are creating a Pulldown Menu, use `PulldownMenu 1, 0, 0`.

STEP 8 (Pulldown Menus ONLY) Add at least one "emergency exit" for your program, to allow an escape from the DO/LOOP structure. We suggest that you use an unusual keystroke like Ctrl-Q (for Quit) which causes the PulldownMenu function to return the value `%PULLDOWN_Q`, so that during testing you can *always* get your program to stop without using the Windows Task Manager.

STEP 9 Compile and run your program.

STEP 10 Fine-tune the menu system. Unless you did a *lot* of pre-planning, you're bound to have missed something.

STEP 11 Once you get your menu system on its feet, you'll probably want to investigate the MenuItemProperty function. It can be used to add "finishing touches" to your menu system, like checkmarks, disabled items, and pre-highlighted items. You might also want to revisit the MenuDefinition function. It contains a lot of features like horizontal and vertical separator bars that you might have skipped over on the first pass.

STEP 12 Check your menu system to make sure that as many items as possible have ampersands (&) which define *unique* hot-keys, to make your menu more keyboard-accessible. *No two items on a single menu or submenu should have the same hot-key*. You might also want to make sure that you haven't used any duplicate Item ID numbers or Accelerator Keys, unless you did so on purpose.

STEP 13 Review the following ideas...**1)** You're not limited to one menu *system*. You can have virtually as many top-level menus as you need. Some programs have a "Beginner's Menu", an "Intermediate Menu", and an "Expert's Menu", and they allow the user to switch to different levels by using the Pulldown Menus themselves. **2)** You could include foreign-language menus in your program. **3)** It's possible to load menu definition strings from a disk file. **4)** With some work, you could write a program that allowed your users to remove items from the menu strings and thereby modify their copy of your program. You could let them define their own Accelerator keys, too. **5)** It's possible to create a non-pulldown "DOS-Style" menu system with Console Tools. Check out the Console Tools Demo Program source code (CONSDEMO.BAS) for a simple example.

STEP 14 When you're truly done, change the value of `%MAXMENUBUFFER` from 64 to the highest-numbered menu buffer that your program actually uses. This will reduce the memory requirements for your program, and allow Console Tools to work slightly faster.

By the way, the source code for Console Tools Demo Program, which is provided with Console Tools, contains two complete Menu Systems: a pulldown menu system and a "DOS-Style" menu system.   We didn't point that out earlier because it's very important for you to understand the process of *developing* a menu system.

Saving and Loading Screens

The process of Saving and Loading Screens requires you to learn about two simple functions: ConsoleScreenSave and ConsoleScreenLoad. The ClearSavedScreen function is optional.

Console Tools can save and load screens in two basic ways: either to disk or to memory.

The disk method is very similar to the DOS BASIC commands BSAVE and BLOAD. In fact the files that Console Tools uses are 100% compatible with DOS BASIC.

The memory method uses the Console Tools Screen Buffers to save screens without using a disk file. This is useful when, for example, you want to display an error message on the screen and then restore the original screen when the message goes away. NOTE: All Console Tools functions that display screen elements (Message Boxes, Input Boxes, etc.) *automatically* restore the screen to its previous condition when the element is removed. The Screen Save/Load functions are provided for *your* programs that change the screen.

Both the Save and Load functions, whether they use disk or memory, allow you to optionally specify a *range* of screen row numbers to be saved or loaded. You could use this feature to load different parts of a screen from different files, or to save/restore just a portion of a screen, or to build Insert-Row and Delete-Row functions... just about anything you can imagine.

Because the PB/CC compiler does not allow programs to access the Windows Handles of the various screen pages, ConsoleScreenSave and ConsoleScreenLoad can only be used with PB/CC Page 1. (Please refer to the PowerBASIC PAGE function documentation for more information about "pages".)

Peeking and Poking the Screen Buffer

The ConsolePEEK function gives your programs the ability to "peek" at the screen , i.e. to read information *from* page 1 of the console window. This is similar to the PB/CC SCREEN and SCREENATTR functions, except that the ConsolePEEK function performs both jobs and is not limited to reading one byte at a time.

The ConsolePOKE function can be used to "poke" information onto the screen, i.e. to place characters or attributes (colors) onto page 1 of the console screen buffer. When used to display text, it is different from the PB/CC PRINT statement because it places characters onto the screen without affecting the screen colors and without affecting the location of the PB/CC cursor. When used to change the colors of an area on the screen, it is similar to the PB/CC COLOR statement when the optional third parameter is used, but it is much easier to use ConsolePOKE if you're making complex color changes. (It is also not necessary to change the PB/CC cursor location when using ConsolePOKE, as it is with COLOR.)

ConsolePEEK and ConsolePOKE are different from DOS PEEK and POKE functions because the DOS functions use strings that have alternating character/color bytes. ConsolePEEK and ConsolePOKE use different strings of bytes for characters and colors.

Because PB/CC does not allow programs to access the Windows Handle of the various screen pages, ConsolePEEK and ConsolePOKE can be used only with page 1 of the console screen buffer.

Note that ConsolePOKE and ConsolePEEK cannot be used to affect other (non-screen) areas of memory; you should use the PB/CC PEEK and POKE statements for that.

Miscellaneous Console Tools Functions

The AlreadyRunning function can be used to detect other programs that are currently running -- or other copies of the current program -- and to optionally send Window State commands (maximize/minimize/etc.) to other programs.

The WindowsVersion function returns basic information about which Windows Operating System (95/98/ME//NT/etc.) is being used at runtime.

The ConsoleToolsVersion function returns a number that can tell your program which version of the Console Tools DLL is installed on a computer.

The ConsoleToolsSerialNumber function returns the Serial Number that is embedded in the currently-loaded copy of the Console Tools DLL, in case you forget it.

The CustomFont function (Console Tools Pro only) can be used to create custom fonts for Splash Boxes and Input Boxes.

The CustomIcon function can be used to load an icon from a disk file, for use by other Console Tools functions.

The Delay function is an easy way to tell your program to pause for a period of time. It is similar to the PB/DOS DELAY statement.

The iString or "Interpreted String" function provides an easy way to include certain characters in your program's strings. For example, these two lines of code would produce identical results...

```
PRINT "Say " + CHR$(34) + "HELLO!" + CHR$(34) + _
      " to easy quotes in strings."

PRINT iString("Say \qHELLO!\q to easy quotes in strings.")
```

Screen results: **`Say "HELLO!" to easy quotes in strings.`**

The InitConsoleTools function initializes certain internal Console Tools variables. It is also used to tell the Console Tools DLL how many Menu Buffers and Screen Buffers your program intends to use. (If you are writing a non-native console application that uses the Windows API to destroy and re-create console windows, you may also need to use the InitCTInternals function.)

Low Level Console Functions

Console Tools provides four functions that will be useful if you need to use Windows API functions or perform similar "low-level" operations.

The hConsoleWindow function provides the Windows Handle of the console window that is owned by the current program.

The hConsoleWindowMenu function provides the Windows Handle of the Window Menu that is owned by the current program's console window.

The hCustomFont function provides the Windows Handle of the font that was created by a previous use of the CustomFont function.

The VirtualKey function can be used to send "virtual keystrokes" to any program for which you have a Windows Handle.

Graphics Functions

When Console Tools is combined with Perfect Sync's **Graphics Tools** DLL, you get something we call "Console Tools Plus Graphics".

Console Tools Plus Graphics has the ability to create a graphics window that is part of the console window. The graphics window can occupy any rectangular area, up to and including the entire console window. It is also possible to create several different "virtual graphics windows" with Graphics Tools.

Console Tools Plus Graphics provides over 80 different functions that allow you to draw lines, points, circles, ellipses, arcs, chords, pies, rectangles, Xagons (x-sided figures), parallelograms, rhombuses, user-defined polygons, Bezier curves, frames, bitmaps, icons (including animated icons), and much more. You can also use virtually unlimited fonts, with special effects like italics, underlining, shadows, and outlines.

All drawing takes place in a "world" that is always 1024x512 (or any dimensions that you specify) regardless of the current screen resolution or the size of the console window. If you create a graphics window that is the size of the entire console, you'll be drawing in a "square" world where circles look round and squares look square. If you create a graphics window that only uses (for example) the left half of the console, everything will be scaled (compressed or stretched) appropriately. Or you can change the scaling to get many different visual effects.

You can even designate rectangles of the graphics window to be "transparent" so that the console shows through "holes" in the graphics window, allowing you to use `PRINT`, `LOCATE`, `COLOR`, etc. Instead of a plain text-based console window, imagine being able to display a great-looking bitmap that doesn't look *anything* like a console application, but still being able to use normal text functions like `INKEY$` and `PRINT` for high-performance input and output.

Because they are only available to Console Tools Plus Graphics users, four different graphics-related functions are included with Console Tools, and they are described in the Reference Guide section of this document. *These four functions can only be used if both Console Tools and Graphics Tools are installed on your computer.*

See ConsoleGfx, GfxTextHole, MatchConsoleFont, and ConsoleColor.

For more information about Console Tools Plus Graphics, visit http://perfectsync.com.

Using the Predefined Equates

The Console Tools INC files (CT_Std.INC and CT_Pro.INC) contain a large number of predefined equates that make using Console Tools much easier.

An equate is an identifier that represents a numeric value. PB/CC recognizes equates by their percent sign (%) prefix. Please consult the PB/CC documentation for complete details.

IMPORTANT NOTE: You are free to change the *names* that are associated with the Console Tools predefined equates if they conflict with other equate names in your programs, but the *values* are a part of the Console Tools DLL and cannot be changed. For example, you could change `%SHOW = 5` to `%SHOWWINDOW = 5`, but changing it to `%SHOW = 100` would not work. It is the "5" that the Console Tools DLL recognizes, not "show" or "showwindow".

In most cases, equates will be used one at a time. For example, the equate `%SHOW` can be used with the ConsoleWindow function this way...

```
      ConsoleWindow  %SHOW
```

In many (but not all) cases you can add equates together to produce a desired effect. For example, the second parameter of the ConsoleMessageBox function might look like this...

```
      ConsoleMessageBox "Message", %OKONLY + %INFOBOX, "Title",...
```

... to tell the function to display a message box with just an "Ok" button *and* to use the "Info Box" icon and sound.

You could also use the "Logical OR" technique...

```
      ConsoleMessageBox "Message", %OKONLY OR %INFOBOX, "Title",...
```

...but some people find this method to be confusing. Using AND seems more appropriate than using OR, but AND doesn't work. To avoid such confusion, this help file uses the "plus" method for all examples.

Keep in mind that equates cannot *always* be added together to produce different results. See the Remarks section of ConsoleWindow for an example of equates that can't be combined.

Common Windows 95/98/ME Mouse Problems

Many different Console Tools users have encountered problems on Windows 95/98/ME computers when attempting to display GUI elements (Message Boxes, Input Boxes, Pulldown Menus, etc.) in response to a mouse click. These problems are due to an acknowledged bug in Windows 95/98/ME and do not affect Windows NT, Windows 2000, or Windows XP computers in any way.

Before we explain how you can solve this problem, we need to explain exactly what is going on.

The problems occur when a mouse-button-down event is used to trigger the display of a Console Tools GUI element. For example, you might want your program to display a Pulldown Menu whenever the top row of the console is clicked, or you might want it to display a Console List Box whenever a certain user-entry field is clicked. (The problem affects virtually all Console Tools GUI elements.)

Here is the sequence of events that causes problems for Windows 95, 98, and ME: Try to imagine it happening in slow motion...

1) The user clicks the console window, generating a mouse-button-down event.

2) PB/CC detects the event and passes the appropriate information to INSTAT, INKEY$, WAITKEY$, and the other PB/CC input functions.

3) Your program "sees" the mouse click and displays a GUI element.

4) The user releases the mouse button, generating a mouse-button-up event.

5) The GUI element sees the mouse-up event and gets "confused" because mouse-button-down events are usually detected *before* mouse-button-up events.

6) The GUI element refuses to recognize further mouse input, until the focus is given to another window and then returned to the GUI element. For example, you could click on the Windows desktop (or any application that happens to be running) and then click on your application to return the focus to your GUI element. At that point the mouse would begin working normally again. Another technique is to minimize and restore your application.

Fortunately, this problem is fairly easy to solve. Please read both of these solutions before deciding which one to use:

1) Instead of having your program detect mouse-button-*down* events, simply have it detect mouse-button-*up* events. Most users click down-and-up fairly quickly so they probably won't notice the difference, but the GUI element won't get "confused" because the first event that it will see will be the next mouse-button-down event that your user produces.

2) If that doesn't solve the problem, or if using up-events interferes with the mouse routines in other parts of your program, you can return to using mouse-button-down events and add a small routine that waits for the mouse-button-up event before it displays the GUI element. For example...

```
MOUSE 1,DOWN

IF INSTAT THEN
      'Get keyboard or mouse input...
      sInput$ = INKEY$
      IF sInput$ = CHR$(255,255,4,1) THEN
            'Mouse-button-down event detected.
            'Tell PB/CC to detect button-up events:
            MOUSE 1,UP
            WHILE NOT INSTAT
                  'Wait for up-event
            WEND
            'Clear the input buffer:
            INPUT FLUSH
            'Restore normal mouse operation:
            MOUSE 1,DOWN
            'And then finally...
            '* DISPLAY GUI ELEMENT HERE *
      END IF
END IF
```

Once again, this problem is caused by a known bug in Windows 95, 98, and ME, not a bug in Console Tools or PB/CC. It does not occur on Windows NT/2000/XP computers.

Suggested Reading

A lot of work went into creating this help file. In particular, an extensive set of Appendices, which cover a wide variety of topics, appears at the end of the "book".

Please take a quick look at this list so that if you have a problem you'll (hopefully) remember where to look...

Getting Technical Support

We worked very hard to make sure that this Help File contains everything that you'll need to know about using Console Tools. Before contacting Perfect Sync for Help ***please*** use this Help File's **Index** and **Find** features to search for words that might be related to your question. Almost every Console Tools topic is covered *twice* in this file: once in the User's Guide and once in the Reference Guide. And often once or twice in an Appendix, and again in a cross-reference...

If, after looking, you don't find an answer in this Help File, Perfect Sync provides *limited* free Technical Support to all developers who license Console Tools. Please send all pertinent technical questions and bug reports to support@perfectsync.com. Be sure to include your Console Tools Serial Number and an email address where we can send our response.

PLEASE NOTE: If you contact us and it turns out that the answer to your question is given in this Help File, that is probably the answer that you will receive: a polite "RTFM" and a topic name. If you feel that the Help File does not explain a topic well enough, please cut and past the unclear help text into your message, and ask a specific question.

If the answer to your question is not in this Help File but falls within the bounds that we have established, we'll do our best to **1)** answer your question quickly and completely via email and **2)** add the answers to the next edition of this Help File so that others can benefit.

If your question is outside the bounds that we have established, we reserve the right to decline to provide an answer. For example, Console Tools includes the hConsoleWindow function, which is useful when dealing with the Windows API. That is where our responsibility ends: providing a reliable function and an accurate explanation of the value that it returns. The function either works or it doesn't. If it doesn't work, we will endeavor to provide a bug fix for the DLL. If it *does* work, Perfect Sync is not responsible for investigating or explaining why a particular API function does not respond the way you expect when hConsoleWindow is used.

As the president of PowerBASIC is fond of saying, "*When you buy a hammer it doesn't come with instructions that tell you how to build a house*".

Perfect Sync reserves the right, at our sole discretion, to charge hourly fees for technical support that does not fall within the bounds of what we consider to be normal and reasonable. (No fees will be charged without the prior consent of the Console Tools Licensee.)

Questions about the licensing and distribution of the Console Tools DLL and other components may be directed to sales@perfectsync.com

For PowerBASIC PB/CC questions, please contact support@powerbasic.com. PowerBASIC also sponsors excellent "peer support forums" on their "Web BBS" site at www.powerbasic.com .

For help with Windows programming and the Windows API, a good place to start is the free MSDN site at microsoft.com.

There are also a large number of internet "newsgroup" sites that contain information about Windows and Console programming, most notably comp.lang.basic.powerbasic, alt.lang.powerbasic, alt.lang.basic, and a wide variety of sites that start with microsoft.public.

REFERENCE GUIDE

This section of the Console Tools Help File contains complete descriptions of all of the Console Tools functions, in alphabetical order.

If you're trying to figure out which function you need for a particular purpose, we suggest that you take a quick look at Using Console Tools Functions.

If you're looking for a predefined equate (a word that starts with % like `%SHOW` or `%DESKTOP_CENTER`, you'll need to look in the section called Predefined Equates, not in this section.

When the syntax for a Console Tools function is shown in this Reference Guide, you are not required to use the variable names that are shown. They are examples only. You can also choose to use a different PB/CC calling method if you prefer.

For example, the following generic syntax is provided for the ConsoleWindow function:

```
lResult& = ConsoleWindow(lState&)
```

In practice, you would use something like this in your program...

```
lResult& = ConsoleWindow(%SHOW)
```

PB/CC also allows the `TO` method of assigning a function's value to a variable...

```
ConsoleWindow %SHOW TO lResult&
```

And since (as the ConsoleWindow Help entry says) it is fairly safe to ignore the return value of this particular function, you could also use...

```
ConsoleWindow %SHOW
```

or

```
CALL ConsoleWindow(%SHOW)
```

Conventions Used In This Help File

Some programmers prefer to explicitly "type" their variable names. An example of this would be the addition of an ampersand (&) to the end of a variable name to indicate that a variable such as `Something&` is a PB/CC long integer. Other programmers, particularly those who program for 32-bit Windows, often use something called "Hungarian notation" where something is added to the beginning of the variable name. The Hungarian notation version of `Something&` would be `lSomething`, with the lower-case L standing for long. (Hungarian notations vary. For example, some use `i` for Integer, others use n.)

This Help File uses both prefixes and suffixes, so every variable you see will look like `lSomething&.` The following prefixes and suffixes are used in this Help File:

`lSomething&`	Long integer
`qSomething&&`	Quad integer
`sSomething$`	String
`lpzSomething`	ASCIIZ string (no suffix defined)
`fpSomething!`	Single precision floating point
`epSomething##`	Extended precision floating point

Predefined equates, whether they come from the Console Tools INC files or the WIN32API.INC file that is supplied with PB/CC, are shown in `THIS FONT` and are always prefixed by the % that is required by the PB/CC compiler. If you are using an equate from the WIN32API.INC file you should keep in mind that Microsoft does not use the leading percent sign, so their documentation will refer (for example) to `ERROR_FILE_NOT_FOUND` not `%ERROR_FILE_NOT_FOUND`.

Console Tools predefined equates, of course, are not found in Microsoft documentation.

AlreadyRunning

Purpose

Detects programs that are currently running by looking for their Title Line Text, and optionally sends "Change State" commands to them.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = AlreadyRunning(sTitle$, lAction&)
```

Parameters

sTitle$

The exact title bar text of the program to be detected.

lAction&

Can be zero (0) for "no action", or any one of these Window State commands: `%SHOWNORMAL, %SHOWMINIMIZED, %SHOWMAXIMIZED, %MAXIMIZE, %SHOWNOACTIVATE, %SHOW, %MINIMIZE, %SHOWMINNOACTIVE, %SHOWNA, %RESTORE, %SHOWDEFAULT.`

Return Value

lResult& will be `%FALSE` (zero) if the program with the title sTitle$ is not currently running, or Logical True (`-1`) if it is running.

Remarks

The AlreadyRunning function returns Logical True if a program with the title line *sTitle$* is detected, and False if it is not. If a program is found to be running and *lAction&* has a nonzero value, the AlreadyRunning function will send the appropriate command to the detected program, to tell it to perform the action specified by *lAction&*. It is, of course, up to the detected program to act upon the command.

AlreadyRunning can only locate a single copy of a program that is already running. If two or more instances of a program are running, it will find one, perform the specified action, and then exit without searching for additional instances. (This is a limitation of Windows.)

This function can be used to detect *any* program by its title line. Common uses would be **1)** detecting a previously-started copy of the current program, and **2)** detecting related programs and performing actions on them if they are not in the correct state, such as launching them if they are not already running.

Note that a program that uses the AlreadyRunning function *can* "detect itself", and this is not usually desirable. For example...

```
ConsoleTitle "MY PROGRAM"
IF AlreadyRunning("MY PROGRAM",%SHOW) THEN EXIT FUNCTION
```

...would set your program's title to "MY PROGRAM" and then the AlreadyRunning function would detect that there is a program with the title "MY PROGRAM" already running and send a `%SHOW` command to it. It would almost certainly be sending the command to itself. The correct method would be...

```
IF AlreadyRunning("MY PROGRAM",%SHOW) THEN EXIT FUNCTION
ConsoleTitle "MY PROGRAM"
```

Note that it is not possible to use AlreadyRunning to send a `%HIDE` command to a program. The numeric value of `%HIDE` is zero (False), so the AlreadyRunning function will not perform any action.

Note also that the AlreadyRunning function can only search for an *exact* window title. If the program that you need to detect does not always use the same title line, AlreadyRunning won't be able to find it. For example, many text-editor programs display the name of the currently-loaded file in their title line, and some programs display the current date and time.

Example

```
IF AlreadyRunning("MY PROGRAM",%SHOW) THEN EXIT FUNCTION
```

This example would look for an instance of a program with the Console Title Line text "MY PROGRAM", and, if it found it to be running, it would send a `%SHOW` command to the program. The function would then return with a Logical True value, so the example program would then perform an `EXIT FUNCTION` operation, presumably from the WinMain function, effectively ending the current program.

See Also

ConsoleTitle, ConsoleState, Appendix B: Console Window States, Appendix H: Logical True and False.

ClearSavedScreen

Purpose

Resets a Console Tools Screen Buffer to the "empty" condition, releasing memory.

Availability

Console Tools Standard and Pro

Warning

Console Tools Screen Buffers cannot be restored once they have been cleared. Make sure that your program no longer needs the contents of a buffer before you clear it.

Syntax

```
ClearSavedScreen lBuffer&
```

Parameters

lBuffer&

The number of the Console Tools Screen Buffer that you want to clear. If your program is using the Console Tools Standard DLL, lBuffer& must be an integer value between 0 and 7. If your program is using the Console Tools Pro DLL, lBuffer& must be an integer value between 0 and 255.

Return Value

ClearSavedScreen does not return a value. If you give it an invalid lBuffer& value, it performs no actions.

Remarks

This is strictly an optional "housekeeping" function that can be used **1)** if you want your program to be careful about releasing memory for other purposes, or **2)** if you want to make sure that your program can not accidentally restore a particular screen or partial screen.

This function has no effect on the active console screen buffer or any of the eight PB/CC Screen Pages. It only affects Console Tools Screen Buffers.

Example

```
ClearSavedScreen 1
```

This example would clear the Console Tools Screen Buffer #1.

See Also

ConsoleScreenSave, ConsoleScreenLoad.

ColOfLoc

Purpose

Returns the console column number that corresponds to a screen location.

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = ColOfLoc(lXPos&)
```

Parameters

lXPos&

The screen location, in pixels, based on 0,0 at the top-left of the screen, for which you want the console column number.

Return Value

If the x-axis (left-to-right) value that you specify for lXPos& corresponds to the x-axis location of a console column, this function will return the column number. (Keep in mind that a *range* of screen locations will correspond to each column. The size of the range will depend on the width of the console's columns.)

If the value that you specify for lXPos& is located either **1)** to the left of the left-most column or **2)** to the right of the right-most column, this function will return the theoretical column number. For example, if you specify a screen location that is located to the left of the left-most column of the console, this function will return a negative number that indicates the "imaginary" column number of the specified location.

Remarks

ColOfLoc stands for Column Of Location. Compare this function to the LocOfCol (Location of Column) function.

See Also

RowOfLoc

Console80x

Purpose

Changes the console screen and console buffer dimensions to 80 columns and a specified number of rows. (IMPORTANT TIP: See Microsoft Console Windows for a good reason for changing the row count.)

Availability

Console Tools Standard and Pro

Warning

Windows 95/98/ME can react strangely when this function is used if the Windows Console "Auto Font Size" feature is enabled. See ConsoleNormal for more details, and a way to avoid the strange effects.

Syntax

```
lResult& = Console80x(lRows&)
```

Parameters

lRows&

The number of rows that you want the console screen and console buffer to have, between 1 and a number that depends on your operating system and maximum screen resolution.

Return Value

lResult& will be `%SUCCESS` (zero) if the function is able to set the console screen and screen buffer to the desired setting.

lResult& will be `%ERROR_CT_INVALIDPARAMETER` if the value of lRows& is less than 1. See Appendix E: Console Tools Error Codes.

If the function fails, lResult& will be the Windows Error Number that caused the failure. The most common cause for failure is an attempt to create a console window that is too large to fit on the screen. Console80x will not create a console window that has scroll bars.

Remarks

This function is usually used to set the console window to 80x25 or 80x43. It can also be used to change the row count to "unusual" values like 24 or 26, to greatly increase the speed of the PB/CC PRINT statement on Windows 95/98/ME systems. (See Microsoft Console Windows for more information about this effect.)

The Console80x function can reliably be used to set the console to 80x50 if it is in the FullScreen Mode. In the Windowed Mode, however, Windows will often change the buffer size but refuse to allow the *window*-size to be changed to 80x50, resulting in a console window with a vertical scroll bar. The Console80x function automatically detects this condition and resets the console to the previous size, so that the scroll bar will not be displayed.

If your program requires a certain console window size, regardless of whether or not the scroll bar will be added, you can use the ConsoleControl function to create console window sizes that Console80x will reject.

Please note that Windows 95/98/ME sometimes scrolls all or part of an existing screen so that it is no longer visible after this function has been used. Ideally, unless the console screen is blank, your program should save the current screen, use Console80x to change the screen size, then restore the screen contents. (See ConsoleScreenSave.)

Example

```
IF Console80x(50) <> %SUCCESS THEN
      'display a warning that the
      'screen-sizing operation failed.
END IF
```

See Also

ConsoleControl, ConsoleInfo

ConsoleColor (Console Tools Plus Graphics ONLY)

Purpose

Returns the Windows Color value (between zero and `%MAXCOLOR`) which corresponds to a color number (between zero and fifteen) that you would normally use with the PB/CC `COLOR` statement.

Availability

Console Tools Standard and Pro

Syntax

```
lResult& = ConsoleColor(lColor&)
```

Parameters

lColor&

A PB/CC color value between zero (0) and fifteen (15). If you use a negative number for this parameter, the sign will be ignored. If you use a value larger than 15, the value will be normalized to the 0-15 range. (In other words, 16 will be treated as 1, 17 will be treated as 2, and so on.)

Return Value

This function will always return a Windows Color value.

Remarks

This function is provided as a convenient method of converting PB/CC color values to Windows TrueColor values. For example, the PB/CC `COLOR` statement recognizes the number 12 as "light red" (also known as "high red"). If you used 12 for the *lColor&* parameter of this function, the return value would be 255 (&h0000FF&), which is the TrueColor equivalent of "light red".

Examples

```
'create a graphics pen with the same color
'that PB/CC uses for "light red"...

PenColor  ConsoleColor(12)
```

See Also

Please refer to the Graphics Tools documentation for more information.

ConsoleControl

Purpose

Performs several different operations that affect the console window.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleControl(lParameter&, lValue&)
```

Parameters

lParameter&

The type of setting that is to be made: `%BUFFER_WIDTH`, `%BUFFER_HEIGHT`, `%CONSOLE_LEFT`, `%CONSOLE_TOP`, `%CONSOLE_RIGHT`, `%CONSOLE_BOTTOM`, `%WINDOW_LEFT`, or `%WINDOW_TOP`. See Remarks below.

lValue&

The value that is to be set. Legal values depend on the value of *lParameter&*.

Return Value

lResult& will be `%SUCCESS` (zero) if the function is able to change the specified setting to the specified value

lResult& will be `%ERROR_CT_INVALIDPARAMETER` if you specify a number for lParameter& that does not correspond to a predefined equate on the list above.

If the function fails, lResult& will *usually* be the Windows Error Number that caused the failure. The most common causes of failure are discussed under Remarks below.

In some cases, Windows will fail to perform the requested operation but not provide a Windows Error Number. In this case, lResult& will be `%ERROR_CT_UNKNOWNERROR`. The most common example of this is probably Windows 95/98/ME's refusal to change the width of the console screen buffer (`%BUFFER_WIDTH`) when the Auto Font Size feature is enabled. Since Windows 95/98/ME uses Auto Font Size by default, this is a common problem.

Remarks

`%BUFFER_WIDTH` and `%BUFFER_HEIGHT` can be used to change the size of the console screen buffer. (See Appendix A: Console Windows.) Windows will not allow the buffer to be changed to a size that cannot be displayed in the current console window, so when you are making the buffer smaller you must change the `%CONSOLE_` screen-size settings (just below) *before* you can change the buffer size. If you make the buffer larger than the console window, Windows will add scroll bars to the console window until you (optionally) resize the window by using the `%CONSOLE_` settings.

`%CONSOLE_LEFT`, `%CONSOLE_TOP`, `%CONSOLE_RIGHT`, and `%CONSOLE_BOTTOM` can be used to change the width and height of the console screen, and to scroll the window in different directions. Windows will not allow the console to be changed to a size that is larger than the current buffer, so when making the screen larger you must change the `%BUFFER_` settings *before* you can change the screen size. If you make the screen smaller than the buffer, Windows will add scroll bars to the console window until you (optionally) resize the buffer by using the `%BUFFER_` settings.

Please note that Windows 95/98/ME often scrolls all or part of an existing screen so that it is not visible after this function is used. Ideally, your program should save the current screen, use ConsoleControl to change the buffer or screen size, then restore the screen.

If you are using a screen/buffer size that is 80 columns wide and want to produce a console screen that does not have scroll bars, it's much easier to use the Console80x function than to use ConsoleControl.

`%WINDOW_LEFT` and `%WINDOW_TOP` can be used to change the left-right and up-down positioning of the console window on the screen. It's usually easier to set both positions at the same time by using the ConsoleMove function.

Example

```
IF ConsoleControl(%WINDOW_LEFT, 0) = %SUCCESS THEN
      'the console window has been moved so that
      'the left side of the screen is at pixel 0,
      'i.e. the left edge of the screen.
END IF
```

See Also

Console80x, ConsoleMove

ConsoleFocus

Purpose

Sets your program's keyboard focus to the console window. (This function can *not* "steal" the keyboard focus from another program and give it to your program. See Remarks below.)

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleFocus
```

Parameters

None.

Return Value

None.

Remarks

Windows does not allow a program to "steal" the keyboard focus from *another* program without also changing the "foreground window". If that is what you need to accomplish, use ConsoleToForeground instead of ConsoleFocus.

The ConsoleFocus function simply returns *your program's* keyboard focus to the console window if it has been directed to another element of your program. For example, if a user clicks on a Console Tools Progress Box and thereby gives it the keyboard focus, your program will not be able to use INSTAT (etc.) to detect keypresses until the focus is returned to the console window.

Example

```
PRINT "PLEASE PRESS ANY KEY TO CONTINUE...";
DO
      ConsoleFocus
      IF INSTAT THEN EXIT LOOP
      DELAY 0  'to avoid CPU loading (see Delay)
LOOP
```

See Also

ConsoleKey

ConsoleGfx (Console Tools Plus Graphics ONLY)

Purpose

Initializes the Console Graphics window, using the rows and columns that you specify. (Please note that this function does not make the graphics window *visible*. After using this function to create the graphics window, you must use the GfxWindow `%SHOW` function to make it visible. Please refer to the Graphics Tools documentation for more information about the GfxWindow function.)

Availability

Console Tools Standard and Pro

Warning

You must use an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function before you can use this function.

Syntax

```
lResult& = ConsoleGfx(lLeftCol&, _
                      lTopRow&, _
                      lRightCol&, _
                      lBottomRow&)
```

Parameters

Note: If you use zero (0) for all of these parameters, Console Tools will automatically create a graphics window which fills the entire console window, regardless of its current size. Also, if you use values that are too small (such as a one-row-high graphics window) Graphics Tools may refuse to create the requested window because it requires a minimum size of 32x32 pixels.

lLeftCol& and *lTopRow&*

The column and row that define the top-left corner of the graphics window.

lRightCol& and *lBottomRow&*

The column and row that define the bottom-right corner of the graphics window.

Return Value

This function will return `%SUCCESS` if the graphics window is created without errors, or

If you fail to specify an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function *before* you use this function, this function will return `%ERROR_CT_CANTBEDONE`. (The `_CT_` stands for Console Tools.)

`%ERROR_GT_UNKNOWNERROR` (the `_GT_` stands for Graphics Tools) will be reported if Windows reports that it was not able to create the requested window, or

`%ERROR_GT_INVALIDPARAMETER` if you attempt to use this function before using InitConsoleTools.

Remarks

This function creates (but does not make visible) a console graphics window. It will occupy the rectangular area of the console window that is defined by the rows and columns that you use for this function's parameters.

Please note that the parameters of this function are "backwards" from the usual row/column parameters of PB/CC functions such as `LOCATE`. Because this is a *graphics* function, it uses the Windows graphics convention of column/row not row/column.

You must not use this function until *after* the InitConsoleTools function has been used by your program, using an appropriate value for the *lGraphics&* parameter. If you attempt to use ConsoleGfx *before* InitConsoleTools, the graphics window will not be properly "attached" to the console window.

This function is very similar to the Graphics Tools InitGfx and OpenGfx functions, except that it automatically calculates the values of several of the necessary parameters for you. Specifically, it calculates the pixel values that correspond to the columns and rows that you specify, it automatically uses the console window as the parent window of the graphics window, and it automatically selects the console "toolset".

IMPORTANT NOTE: This function also performs other console-graphics-related initialization functions, so if you are using Console Tools Plus Graphics you *must* use this function *at least once* in your PB/CC program, even if you later use InitGfx or OpenGfx to create a graphics window.

For more information about creating a graphics window, see the InitGfx and OpenGfx entries in the Graphics Tools documentation.

Examples

```
FUNCTION PBMain PRIVATE AS LONG

      'Add your Authorization Code here...
      lResult& = ConsoleToolsAuthorize(%MY_CT_AUTHCODE)

      'Initialize Console Tools.  (The -1 tells
      'Console Tools that you are using Graphics Tools.)
      InitConsoleTools 0, 0, 0, -1, 0, 0

      '(You may be required to perform other steps
      'such as GraphicsToolsAuthorize here, depending on
      'the version of Graphics Tools that you are using.
      'See the Graphics Tools documentation for more
      'information.)

      'Initialize a full-console graphics window...
      ConsoleGfx 0,0,0,0

            ...or...

      'Initialize a graphics window that fills
      'columns 1-80, but just rows 1 through 10.
      ConsoleGfx 1, 1, 80, 10

      'Now make the graphics window visible...
      GfxWindow  %SHOW
```

See Also

Please refer to the Graphics Tools documentation for more information.

ConsoleHasScrollBars

Purpose

Reports the number of scroll bars that the console window currently has.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleHasScrollBars
```

Parameters

None.

Return Value

lResult& will be zero (`%FALSE`) if the console currently does not have scrollbars, and a nonzero (True) value if it does. Note that this function does *not* return Logical True, so you can *not* use things line `IF NOT ConsoleHasScrollBars THEN...`

lResult& will be the value one (1) if the console currently has only a vertical scroll bar (on the right side of the console window), or the value two (2) if it has only a horizontal scroll bar (along the bottom edge), or the value three (3) if it has both. (Tip: Remembering that the number 1 resembles a vertical line can help you remember the numbering system.)

Examples

```
IF ConsoleHasScrollBars THEN
      'console has at least one scroll bar
END IF

IF ConsoleHasScrollBars => 2 THEN
      'console has a horizontal scroll bar
      'and may have a vertical one.
END IF
```

See Also

Appendix A: Windows Consoles

ConsoleHasToolbar

Purpose

Returns a Logical True or False value to indicate whether or not the Windows 95/98/ME Console Toolbar is currently visible.

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = ConsoleHasToolbar
```

Parameters

None.

Return Value

This function will return a Logical True value if the Windows 95/98/ME console toolbar is currently visible, or False (zero) if it is not visible.

On Windows NT/2000/XP computers this function will always return False, because Windows NT, 2000, and XP do not support console toolbars.

Remarks

Unlike other toolbar-related functions, it is always safe to use ConsoleHasToolbar, regardless of whether or not the DeleteWindowMenuItem function has been used to remove the `%MENUITEM_TOOLBAR` item.

Example

```
IF ConsoleHasToolbar THEN
      'Hide the toolbar:
      ConsoleToolbar %OFF
END IF
```

See Also

ConsoleToolbar, ToggleConsoleToolbar

ConsoleIcon

Purpose

Changes the icon that is displayed in the console window's Title Bar and the Windows Task Bar.

Availability

Console Tools Standard and Pro

Warning

Using an invalid icon ID can, in rare circumstances, result in a Windows Application Error (General Protection Fault).

Syntax

```
lResult& = ConsoleIcon(lResourceID&)
```

Parameters

lResourceID&

Either **1)** the predefined equate `%IDI_CONSOLE` for the Console Tools Icon, or **2)** the equate `%IDI_CUSTOM` for an icon that was previously loaded with the CustomIcon function, or **3)** the ID number that was assigned to an icon by a Resource Editor, or **4)** one of the predefined equates `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, or `%IDI_WINLOGO` to specify a Windows Standard Icon. (It is *possible* to use Windows Standard Icons with this function, but it is rarely useful to do so.) See Using Icons for more details.

Return Value

lResult& will be `%SUCCESS` (zero) if the function is able to change the console window Icon.

If the function fails, lResult& will be the Windows Error Number that caused the failure.

Remarks

See Appendix D: Using Icons for complete instructions.

Example

```
IF ConsoleIcon(%IDI_ICON1) = %SUCCESS THEN
      'The console icon has been changed to
      'the icon that corresponds to %IDI_ICON1.
END IF
```

See Also

Appendix D: Using Icons

ConsoleInfo

Purpose

Provides the current values of a wide variety of console window parameters.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleInfo(lParameter&)
```

Parameters

lParameter&

An integer value corresponding to one of the following predefined equates:

```
%CONSOLE_LEFT, %CONSOLE_TOP, %CONSOLE_RIGHT,
%CONSOLE_BOTTOM, %CONSOLE_WIDTH, %CONSOLE_HEIGHT,
%CONSOLE_MAX_WIDTH, %CONSOLE_MAX_HEIGHT

%BUFFER_WIDTH, %BUFFER_HEIGHT

%PRINTAREA_LEFT, %PRINTAREA_TOP, %PRINTAREA_RIGHT,
%PRINTAREA_BOTTOM, %PRINTAREA_WIDTH, %PRINTAREA_HEIGHT,
%PRINTAREA_X_CENTER, %PRINTAREA_Y_CENTER

%WINDOW_LEFT, %WINDOW_TOP, %WINDOW_RIGHT, %WINDOW_BOTTOM,
%WINDOW_WIDTH, %WINDOW_HEIGHT, %WINDOW_X_CENTER,
%WINDOW_Y_CENTER

%DESKTOP_LEFT, %DESKTOP_TOP, %DESKTOP_RIGHT,
%DESKTOP_BOTTOM, %DESKTOP_WIDTH, %DESKTOP_HEIGHT,
%DESKTOP_X_CENTER, %DESKTOP_Y_CENTER

%SCREEN_LEFT, %SCREEN_TOP, %SCREEN_RIGHT, %SCREEN_BOTTOM,
%SCREEN_WIDTH, %SCREEN_HEIGHT, %SCREEN_X_CENTER,
%SCREEN_Y_CENTER
```

Return Value

If a value other than a number corresponding to one of the equates above is used, the ConsoleInfo return value will usually be `%ERROR_CT_INVALIDPARAMETER`. In some cases, however, the return value is undefined. (In other words, for certain values of lParameter&, unexpected values may be returned.)

If a valid lParameter& is used, the function will return the value of the requested parameter.

Remarks

The predefined equates that start with `%CONSOLE_` refer to the visible console window and return "character" values, i.e. numbers of rows or columns, based on 1,1 (Row 1, Column 1) at the top-left of the console.

The equates that start with `%BUFFER_` refer to the Console Screen Buffer and also return "character" values based on Row 1, Column 1 at the top-left of the buffer.

The equates that start with `%PRINTAREA_` refer to the portion of the console window where text is printed, and return Pixel values based on 0,0 at the top-left of the screen.

The equates that start with `%WINDOW_` refer to the entire console window, including the title bar and border, and return Pixel values based on 0,0 at the top-left of the screen.

The equates that start with `%DESKTOP_` refer to the "usable" area of the screen, i.e. everything except the task bar. They return Pixel values based on 0,0 at the top-left of the screen.

The equates that start with `%SCREEN_` refer to the entire screen, and return Pixel values based on 0,0 at the top-left of the screen.

The equates that end in `_X_CENTER` and `_Y_CENTER` can be used to get the x-axis and y-axis center points of the various screen areas.

The program `\CONTOOLS\CONSTATS.BAS`, which is included in the Console Tools Installation Program, can be used to display all of the "Console Statistics" that the ConsoleInfo function can provide.

Example

```
PRINT "The console window is located";
PRINT ConsoleInfo(%WINDOW_TOP);
PRINT " pixels from the top edge of the screen and.";
PRINT ConsoleInfo(%WINDOW_LEFT);
PRINT " pixels from the left edge of the screen."
```

Details

Unlike some Windows API functions (which often mix zero-based and one-based values), all of the ConsoleInfo row/column return values are one-based,. For example, when used on an 80-character-wide console window, ConsoleInfo will return a value from 1 to 80 for the `%CONSOLE_WIDTH` value instead of 0 to 79.

The pixel values, on the other hand, are all *zero*-based as Microsoft requires.

See Also

ConsoleControl

ConsoleInputBox

Purpose

Displays a Console Input Box and returns a string from the user.

Availability

Console Tools Standard and Pro
(Some features are limited to the Pro Version.)

Warnings

None.

Syntax

```
sResult$ = ConsoleInputBox(lType&, _
                           lXPos&, _
                           lYPos&, _
                           sPrompt$, _
                           sTitle$, _
                           sDefault$, _
                           lFlags&, _
                           lNormalize&)
```

(Also see Simplified Syntax below.)

Parameters

lType&

Use the value **1** for a one-line Text Input Box, or **2** for a one-line Numeric Input Box, or **3** for a Long Text Input Box, or **4** for a multi-line Text Input Box. Console Tools Pro users can also use **5** for a "Mini Word Processor" Text Input Box, or **6** for the Mini Word Processor in the "read only" mode, or **7** for a two-field input box (like you'd use for UserName/Password input), or **8** for a two-field input box that also allows the display of two different "prompt" labels.

Both Standard and Pro users can also optionally add the predefined equate `%TOPMOST` to produce an Input Box that will appear on top of all other windows, even if the console window is not currently visible. You can also optionally add one of the predefined equates `%BOLD` or `%MONOSPACE` to the lType& value to change the font to "System" or "Fixed Sys", respectively. Console Tools Pro users can add `%CUSTOMFONT` to specify the Console Tools Pro Custom Font. (The value `%DEFAULT` can also be used for the lType& parameter if you have previously defined a default by using the ConsoleInputBoxDefaults function.)

lXPos& and *lYPos&*

These parameters determine the screen location where the Input Box will be displayed. If you use zero (0) for both of these values the Input Box will be auto-located in the upper-left corner of the console window (not the upper-left corner of the screen). You can also use the predefined equate `%DESKTOP_CENTER` to auto-center the input box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the input box in the middle of the console, or you can specify a number to indicate a number of pixels from the top-left corner of the screen at 0,0. Also see LocOfCol and LocOfRow for a technique that allows Input Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Input Box "off the screen" and make it impossible for the user to see it or to select a button.* (The value `%DEFAULT` can also be used for these parameters if you have previously defined the defaults by using the ConsoleInputBoxDefaults function.)

sPrompt$

The text that will be shown in the "prompt" area of the Input Box. The location and size of the prompt area varies from lType& to lType&. You can use the various Shorthand strings like `\q` for Quotes in this parameter. (If you use an empty string for this parameter, and if you have defined a default prompt by using the ConsoleInputBoxDefaults function, the default prompt will be used.) If you are using Input Box type 8 (see *lType&* above) you must separate the strings for the two different prompts with CHR$(0). Example: "Prompt1" + CHR$(0) + "Prompt2".

sTitle$

The text that will be shown in the title bar of the Input Box and (optionally) the Input Box's buttons and "warning" message.

The length of the title bar varies based on the *lType&* parameter. You can use the various Shorthand strings like `\q` for Quotes in this parameter. If no title is specified, the fallback Input Box title "`Input:`" is used. To display an Input Box with nothing in the title line, use a single space like " ".

To change the text that is displayed in the Input Box's buttons, use a string like this for *sTitle$*:

```
"Title" + CHR$(0) + "Button1" + CHR$(0) + "Button2"
+ CHR(0) + "Optional Warning"
```

(The optional warning string will be displayed only if you use Input Box Type 5 and the `%WARNING` flag. See below for details.)

sDefault$

This is the default return value of the Input Box, and the text that is displayed when the Input Box is first shown. You can use the various Shorthand strings like `\q` for Quotes in this parameter, but you *must* use the `\e` (Enter) shorthand to create a Line Break. The usual `\n` (Newline) and `\r` (Return) shorthands are not understood by Windows when they are used in strings for this parameter. If you insert control characters into this string with CHR$ you will find that you have to use the sequence CHR$(32,13,10) to create a Line Break. If you use an empty string for this parameter, and if you have defined a "default default" by using the ConsoleInputBoxDefaults function, the default string will be used. (Also see note just below about using `%FIXEDLEN`.) If you are using Input Box type 7 or 8 (see *lType&* above) and you want to specify a default value for the second field, you must separate the two default values with CHR$(0). Example: "Default for field 1" + CHR$(0) + "Default for field 2".

lFlags&

This parameter can contain one or more of the following values. To use more than one lFlags& value at a time, add the values together (see Example below).

`%HOME` can be used to "home" the cursor when the input box is first shown, so that it appears at the beginning of the editable string instead of the end. Using this option also has the effect of un-highlighting the editable string. The Windows default for an Input Box is to highlight the editable string so that if the user simply begins typing the default value is immediately erased. This behavior is not always desirable.

`%NOEDIT` can be used when you just want a user to "Ok" or "Cancel" a string, but you don't want them to be able to change it.

`%PASSWORD` tells the Input Box to display a star (`*`) character for each character that the user types, instead of displaying the typed character. (The function returns the actual typed value, of course.) We recommend the use of the `%BOLD` option (see *lType&* above) when using `%PASSWORD`. If you are using Input Box type 7 or 8 (see *lType&* above) the `%PASSWORD` flag only affects the second input field.

`%FIXEDLEN` creates an Input Box that restricts the user to the length of the sDefault$ string. If you use this option, be sure to provide an appropriate-length sDefault$ string, or the user won't be able to type anything.

`%WARNING` When used with the Mini Word Processor (see *lType&* number 5 above) the use of this flag causes a warning message to be added next to the Cancel button: "WARNING: If you Cancel you will lose all of your changes!" This can be a helpful reminder if the user is expected to perform extensive editing and would be unhappy if they lost everything they'd done.

(The value `%DEFAULT` can also be used for the lFlags& parameter if you have previously defined a default by using the ConsoleInputBoxDefaults function.)

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays an Input Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Input Box, to make sure that the console screen is visible to the user. This can be important if the console screen contains information that the user needs to fill out the Input Box. (The value `%DEFAULT` can also be used for the lNormalize& parameter if you have previously defined a default by using the ConsoleInputBoxDefaults function.)

Return Value

If the user selects the Cancel button or presses the Escape Key, sResult$ will be equal to the sDefault$ string. If the user selects the Ok button (or, in the case of single line Input Boxes, if the user presses the Enter key) the value of sResult$ will be the *edited* value of sDefault$. This can be anything from an empty string to a string of 30,000 characters. The maximum length of sResult$ can be optionally limited by using the `%FIXEDLEN` flag (see above).

If you are using Input Box type 7 or 8 (see *lType&* above) the return values of the two input fields will be returned as a single string, with the two values separated by CHR$(0).

It is possible to tell which button the user selected by checking the ConsoleInputBoxCancel function immediately after the ConsoleInputBox function returns a value.

Remarks

See Input Boxes for a general discussion of Console Tools Input Box functions.

Example

```
sResult$ = ConsoleInputBox(1+%BOLD, _
                           %CONSOLE_CENTER, _
                           %CONSOLE_CENTER, _
                           "Password Required", _
                           "Please enter your Password:", _
                           SPACE$(16), _
                           %FIXEDLEN+%PASSWORD+%HOME, _
                           %DEFAULT)
```

This example would create a one-line Input Box with the editable text displayed in a Bold font. The Input Box would be centered vertically and horizontally on the Desktop, the title bar would say "Password Required", the Input Box Prompt would say "Please enter your Password:", the default return value would be 16 spaces, the user would not be able to type more than 16 characters, the star (*) character would be displayed instead of the characters that the user typed, and the cursor would start at the "home" position, .i.e. at the far-left of the edit field.

Simplified Syntax

The Console Tools Input Box has options that you'll probably never need, so we suggest that you design a simple "wrapper" function that eliminates the parameters that you won't usually use. For example, you could add this to your program...

```
FUNCTION INPUTBOX$(sPrompt$, sTtitle$, sDefault$)

      FUNCTION = ConsoleInputBox(1, _
                                 %CONSOLE_CENTER, _
                                 %CONSOLE_CENTER, _
                                 sPrompt$, _
                                 sTitle$, _
                                 sDefault$, _
                                 %HOME, _
                                 %TRUE)
END FUNCTION
```

You could then use a much easier INPUTBOX$ syntax for most input boxes. (Users of PowerBASIC's PB/DLL compiler will recognize the basic INPUTBOX$ syntax.)

```
sResult$ = INPUTBOX$("Please enter your name:", "", "")
```

See Also

ConsoleInputBoxDefaults, ConsoleInputBoxCancel

ConsoleInputBoxCancel

Purpose

Reports whether or not the Cancel button was selected (or the Escape key was pressed) when the ConsoleInputBox function was last used.

Availability

Console Tools Standard and Pro

Warning

This is a read-once function. See Remarks below.

Syntax

```
lResult& = ConsoleInputBoxCancel
```

Parameters

None,

Return Value

lResult& will be `%FALSE` (zero) if the Cancel button was not selected, or Logical True if it was selected. Note that pressing the Escape key has the same effect as clicking the Cancel button.

Remarks

If you need to know whether or not an Input Box's Cancel button was selected, you should check the value of this function immediately after returning from the ConsoleInputBox function (see Example below).

It is important to note that this is a Read-Once function. In other words, if the Cancel button was selected this function will return Logical True *the first time it is used*, but subsequent uses of ConsoleInputBoxCancel will always return `%FALSE` (zero). The return value of the function is automatically reset to `%FALSE` whenever the function is used.

Example

```
sResult$ = ConsoleInputBox(1,0,0,_
           "Password","Enter Password:", "" ,_
           %PASSWORD,0)

IF ConsoleInputBoxCancel THEN
      'user clicked Cancel
      EXIT FUNCTION
ELSE
      'authenticate the password in sResult$
END IF
```

See Also

ConsoleInputBox

ConsoleInputBoxDefaults

Purpose

Establishes default values for future Console Input Boxes.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleInputBoxDefaults lType&, _
                        lXPos&, _
                        lYPos&, _
                        sPrompt$, _
                        sTitle$, _
                        sDefault$, _
                        lFlags&, _
                        lNormalize&
```

Parameters

lType&

Use the value **1** for a one-line Text Input Box, or **2** for a one-line Numeric Input Box, or **3** for a Long Text Input Box, or **4** for a multi-line Text Input Box. Console Tools Pro users can also use **5** for a "Mini Word Processor" Text Input Box, or **6** for the Mini Word Processor in the "read only" mode, or **7** for a two-field input box (like you'd use for UserName/Password input), or **8** for a two-field input box that also allows the display of two different "prompt" labels.

Both Standard and Pro users can also optionally add the predefined equate `%TOPMOST` to produce an Input Box that will appear on top of all other windows, even if the console window is not currently visible. You can also optionally add one of the predefined equates `%BOLD` or `%MONOSPACE` to the lType& value to change the font to "System" or "Fixed Sys", respectively. Console Tools Pro users can add `%CUSTOMFONT` to specify the Console Tools Pro Custom Font.

lXPos& and *lYPos&*

These parameters determine the screen location where the Input Box will be displayed. If you use zero (0) for both of these values the Input Box will be auto-located in the upper-left corner of the console window (not the upper-left corner of the screen). You can also use the predefined equate `%DESKTOP_CENTER` to auto-center the input box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the input box in the middle of the console, or you can specify a number to indicate a number of pixels from the top-left corner of the screen at 0,0. Also see LocOfCol and LocOfRow for a technique that allows Input Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Input Box "off the screen" and make it impossible for the user to see it or to select a button.*

sPrompt$

The text that will be shown in the "prompt" area of the Input Box. The location and size of the prompt area varies from lType& to lType&. You can use the various Shorthand strings like `\q` for Quotes in this parameter. If you are using Input Box type 8 (see *lType&* above) you must separate the strings for the two different prompts with CHR$(0). Example: "Prompt1" + CHR$(0) + "Prompt2".

sTitle$

The text that will be shown in the title bar of the Input Box and (optionally) the Input Box's buttons and "warning" message.

The length of the title bar varies based on the *lType&* parameter. You can use the various Shorthand strings like `\q` for Quotes in this parameter. If no title is specified, the fallback Input Box title "`Input:`" is used. To display an Input Box with nothing in the title line, use a single space like " ".

To change the text that is displayed in the Input Box's buttons, use a string like this for *sTitle$*:

```
      "Title" + CHR$(0) + "Button1" + CHR$(0) + "Button2"
      + CHR(0) + "Optional Warning"
```

(The optional warning string will be displayed only if you use Input Box Type 5 and the `%WARNING` flag. See below for details.)

sDefault$

This is the default return value of the Input Box, and the text that is displayed when the Input Box is first shown. You can use the various Shorthand strings like `\q` for Quotes in this parameter, but you *must* use the `\e` (Enter) shorthand to create a Line Break. The usual `\n` (Newline) and `\r` (Return) shorthands are not understood by Windows when they are used in this parameter. If you insert control characters into this string with CHR$ you will find that you have to use the sequence CHR$(32,13,10) to create a Line Break. If you are using Input Box type 7 or 8 (see *lType&* above) and you want to specify a default value for the second field, you must separate the two default values with CHR$(0). Example: "Default for field 1" + CHR$(0) + "Default for field 2".

lFlags&

This parameter can contain one or more of the following values. To use more than one lFlags& value at a time, add the values together (see Example below).

`%HOME` can be used to "home" the cursor when the input box is first shown, so that it appears at the beginning of the editable string instead of the end. Using this option also has the effect of un-highlighting the editable string. The Windows default for an Input Box is to highlight the editable string so that if the user simply begins typing the default value is immediately erased. This behavior is not always desirable.

`%NOEDIT` can be used when you just want a user to "Ok" or "Cancel" a string, but you don't want them to be able to change it.

`%PASSWORD` tells the Input Box to display a star (`*`) character for each character that the user types, instead of displaying the typed character. (The function returns the typed value, of course.) We recommend the use of the

`%BOLD` option (see *lType&* above) when using `%PASSWORD`. If you are using Input Box type 7 or 8 (see *lType&* above) the `%PASSWORD` flag only affects the second input field.

`%FIXEDLEN` creates an Input Box that restricts the user to the length of the sDefault$ string. If you use this option, be sure to provide an appropriate-length sDefault$ string, or the user won't be able to type anything.

`%WARNING` When used with the Mini Word Processor (Type 5 above) the use of this flag causes a warning message to be added next to the Cancel button: "WARNING: If you Cancel you will lose all of your changes!" This can be a helpful reminder if the user is expected to perform extensive editing and would be unhappy if they lost everything they'd done.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays an Input Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Input Box, to make sure that the console screen is visible to the user. This can be important if the console screen contains information that the user needs to fill out the Input Box.

Return Value

None.

Remarks

ConsoleInputBoxDefaults can be used to set or change the *default* values for Console Input Box parameters. If you use this function to set one or more defaults, you can then use `%DEFAULT` (for numeric parameters) or an empty string (for string parameters) when calling the ConsoleInputBox function, and the predefined default(s) will be used. For example, if you wanted all (or most) of the Console Input Boxes in your program to use the same title bar text, you could use ConsoleInputBoxDefaults to set that text as the default title. Then any Console Input Box with "" as the sTitle$ parameter would automatically use the default title. (If you used ConsoleInputBox with something other than "" as the sTitle$ parameter, the default would be ignored for *that* Input Box.)

If you want to change Console Input Box defaults that you have already set, you can use ConsoleInputBoxDefaults more than once. If you want to change one default setting but not the others, use `%DEFAULT` or an empty string when using ConsoleInputBoxDefaults (see last Example).

Example

```
'set up defaults for all parameters...
ConsoleInputBoxDefaults  1+%BOLD, _
                         %DESKTOP_CENTER, _
                         %DESKTOP_CENTER, _
                         "Please Enter Data:", _
                         "My Program", _
                         "type here", _
                         %HOME, _
                         %TRUE

'use all of the defaults in an Input Box...
sResult$ = ConsoleInputBox(%DEFAULT,%DEFAULT,%DEFAULT,_
                           "", "", "", %DEFAULT, %DEFAULT)

'use some of the defaults in an Input Box...
sResult$ = ConsoleInputBox(2, %DEFAULT, %DEFAULT,_
                           "", "INPUT REQUIRED", "", 0, 0)

'change the default title bar text but nothing else...
ConsoleInputBoxDefaults  %DEFAULT, _
                         %DEFAULT, _
                         %DEFAULT, _
                         "", _
                         "NEW TITLE", _
                         "", _
                         %DEFAULT, _
                         %DEFAULT
```

See Also

ConsoleInputBox

ConsoleIsForeground

Purpose

Reports whether or not the console window is currently in the Windows Foreground and has the keyboard focus, i.e. whether or not your program is the currently-active program.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleIsForeground
```

Parameters

None.

Return Value

lResult& will be `%FALSE` (zero) if the console window is not currently the foreground window, or Logical True if it is.

Remarks

Under certain circumstances this function will return `%FALSE` even if your program *is* the currently-active program. For example, if your program displays a Console Tools Progress Box and your user gives it the keyboard focus by clicking on it, the ConsoleIsForeground function will return `%FALSE`. This function returns a True/False value depending on whether or not the console window *itself* is in the foreground.

Example

```
IF NOT ConsoleIsForeground THEN
      'If the user presses a key, your PB/CC program
      'will not be able to detect it.  Also, if
      'you use the VirtualKey function here, the
      'virtual keystrokes will be sent to whichever
      'program is currently in the foreground.
END IF
```

See Also

ConsoleToForeground

ConsoleIsFullScreen

Purpose

Reports whether or not the console window is in the FullScreen Mode.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleIsFullScreen
```

Parameters

None

Return Value

lResult& will be `%FALSE` (zero) if the console is not in the FullScreen Mode, or Logical True if it is.

Remarks

It is important to keep in mind that a console window can be "in two modes at once". For example, the ConsoleState function might report that the console window is in the `%MINIMIZED` mode and the ConsoleIsFullScreen function might return True. This means that either **1)** the version of Windows that you are using Minimizes the console window "in the background" when displaying a FullScreen console, or **2)** that the version of Windows that you are using will display a Minimized console window if the FullScreen Mode is switched off, or **3)** both of the above.

When your program needs to know the state of the console window, it is important to check both the ConsoleIsFullScreen and ConsoleState functions. If the ConsoleIsFullScreen function returns True, you can generally ignore the ConsoleState function (unless you're about to turn the FullScreen Mode off).

Example

```
IF ConsoleIsFullScreen THEN
      'display a text-only warning asking the user's
      'permission to switch to the Windowed Mode.
ELSE
      'display a ConsoleMessageBox
END IF
```

See Also

ConsoleState

ConsoleIsHidden

Purpose

Reports whether or not the console window is currently hidden.

Availability

Console Tools Standard and Pro

Warning

May report incorrect results if certain Windows API calls are used, or if the ConsoleMove function is used with large values. See Remarks.

Syntax

```
lResult& = ConsoleIsHidden
```

Parameters

None.

Return Value

lResult& will be `%FALSE` (zero) if the console window is not hidden, or Logical True if it is.

Remarks

Windows does not actually provide accurate information about whether or not a window is hidden, so Console Tools tracks your program's use of various functions that can affect the hidden/showing status of the console window. For this reason, if you want to use the ConsoleIsHidden function, it is important to *not* use Windows API calls that can affect the Window State. For example, if you were to use the ShowWindow API to hide or show the console window, Console Tools would not know about it and the internal tracking could become mis-aligned. If you stick to Console Tools functions and avoid using API calls that could possibly affect the Console Window State, this function will provide accurate results.

You should also note that it is possible to "hide" the console in another way. If the ConsoleMove function is used to move the console window to a point that is "off the screen", your user will not be able to see the console even though it is not technically "hidden".

Example

```
IF ConsoleIsHidden THEN
      'show the console window
      ConsoleWindow %SHOW
END IF
```

See Also

ConsoleState, ConsoleMove, Appendix B: Console Window States

ConsoleIsMaximized

Purpose

Reports whether or not the console window is currently Maximized.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleIsMaximized
```

Parameters

None.

Return Value

lResult& will be `%FALSE` (zero) if the console window is not currently Maximized, or Logical True if it is.

Remarks

Keep in mind that the console window can have "two states at once". See the Remarks section of ConsoleIsFullScreen for more details.

Example

```
IF ConsoleIsMaximized THEN
      'minimize the console
      ConsoleWindow %MINIMIZE
END IF
```

See Also

ConsoleState, ConsoleIsFullScreen, Appendix B: Console Window States

ConsoleIsMinimized

Purpose

Reports whether or not the console window is currently Minimized.

Warnings

None.

Availability

Console Tools Standard and Pro

Syntax

```
lResult& = ConsoleIsMinimized
```

Parameters

None.

Return Value

lResult& will be `%FALSE` (zero) if the console window is not currently Minimized, or Logical True if it is.

Remarks

Keep in mind that the console window can have "two states at once". See the Remarks section of ConsoleIsFullScreen for more details.

Example

```
IF ConsoleIsMinimized THEN
      'reset to previous window state...
      ConsoleWindow %UNMINIMIZE
END IF
```

See Also

ConsoleState, ConsoleIsFullScreen, Appendix B: Console Window States

ConsoleKey

Purpose

Sends "virtual keystrokes" to the console window.

Availability

Console Tools Standard and Pro

Warning

In order for a program to receive input from the keyboard -- even "virtual" input -- it must have the keyboard focus, so this function forces your program into the windows foreground by using the `ConsoleToForeground %HARD` function. See ConsoleToForeground for details.

Syntax

```
lResult& = ConsoleKey(sKeyDefinition$)
```

or

```
ConsoleKey sKeyDefinition$
```

Parameters

sKeyDefinition$

A string containing one "single keystroke" to be simulated. Multiple keystrokes require multiple uses of this function. (See Remarks.)

Return Value

If *sKeyDefinition$* is an empty string, `%ERROR_CT_INVALIDPARAMETER` will be returned. See Console Tools Error Codes. In all other cases, `%SUCCESS` will be returned. For this reason, the return value of this function can be safely ignored in most cases.

Remarks

The *sKeyDefinition$* parameter is usually created by using the PowerBASIC CHR$() function and one or more of the `%VIRTUAL_` equates that are listed in the Console Tools INC file. Since some of the equates correspond to ASCII values -- for example the ASCII value of "A" is 65 and %VIRTUAL_A = 65 -- it is also possible to use literal strings like "A" instead of `CHR$(%VIRTUAL_A)` in the *sKeyDefinition$* parameter. But not *all* key definitions correspond to ASCII values, so you should always consult the INC file before using a literal string.

It is very important to remember that this function can only send a "single keystroke" each time that it is used. A "single keystroke" can mean the letter A, or Ctrl-A, or Ctrl-Alt-A, or Shift-A, or Shift-Ctrl-A, and so on. It is *not* possible to send the letter A followed by the letter B without using this function twice. (To help you remember this, the function is named ConsoleKey, not ConsoleKeys.)

Examples

```
'send the letter A to the console keyboard
ConsoleKey "A"

'send Ctrl-A
ConsoleKey CHR$(%VIRTUAL_CTRL) + "A"

'send Ctrl-Alt-A
ConsoleKey CHR$(%VIRTUAL_CTRL, %VIRTUAL_ALT) + "A"

'send Alt-Spacebar (activates the Window Menu)
ConsoleKey CHR$(%VIRTUAL_ALT) + " "

'send Alt-F, then send X
ConsoleKey CHR$(%VIRTUAL_ALT) + "F"
ConsoleKey "X"

'send virtual keys as if a user had
'pressed Alt, F, and X all at the same time.
ConsoleKey CHR$(%VIRTUAL_ALT) + "F" + "X"
```

Details

This function directs the Windows Keyboard Focus to the console and then sends the "virtual keystrokes" via the VirtualKey function. It does *not* return the Keyboard Focus to the program that had it before the function was used.

This function cannot be used to send keystrokes to a Console Tools Pulldown Menu, or to a Console Message Box, or to any other graphical element. The keystrokes are sent to the console window itself, as if Console Tools was not in use. In most cases the keystrokes will be received by your PB/CC program's keyboard-input routines. In certain cases, such as System Hotkeys like Alt-Space, the console window itself will intercept the virtual keystrokes and your program will not be able to detect them.

See Also

Using Predefined Equates

ConsoleListBox

Purpose

Displays a Console List Box and returns the user's selection(s).

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
sResult$ = ConsoleListBox(lType&, _
                          lXPos&, _
                          lYPos&, _
                          sPrompt$, _
                          sTitle$, _
                          sItems$(), _
                          lDefault&, _
                          lFlags&, _
                          lNormalizeConsole&)
```

Parameters

lType&

Use **1** for a small List Box, or **2** for a wide List box, or **3** for a tall List Box, or **4** for a wide, tall List Box. The use of negative 1 through negative 4 will produce the same results as using 1 through 4, except that the List Box will be created without the "Ok" and "Cancel" buttons. You can also use a value that is greater than 50,000 (or less than negative 50,000) to tell Console Tools to create a certain sized List Box. The last three digits of the number define the height of the List Box, and any numbers that precede that define the width. For example, using `111222` would produce a List Box that was 111 dialog units wide and 222 units high. Using `-99050` would produce a List Box that was 99 units wide and 50 high, and since the number is negative the Ok and Cancel buttons would not be displayed. Please note that Console Tools will not allow you to create a List Box that is too small to display at least two items and the buttons, so the use of relatively small width or height values may not produce the expected results. The largest List Box that can be created is `999999`.

If you have used the ConsoleListBoxDefaults function to define a default List Box type, you can also use the predefined equate `%DEFAULT` for this parameter, and the default type will be used.

lXPos& and *lYPos&*

These parameters determine the screen location where the List Box will be displayed. If you use zero (0) for both of these values the List Box will be auto-located in the upper-left corner of the console window (not the upper-left corner of the screen). You can also use the predefined equate `%DESKTOP_CENTER` to auto-center the List Box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the List Box in the middle of the console, or you can specify a number to indicate a number of pixels from the top-left corner of the screen at 0,0. Also see LocOfCol and LocOfRow for a technique that allows List Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the List Box "off the screen" and make it impossible for the user to see it or to select an item.*

If you have used the ConsoleListBoxDefaults function to define a default lXPos& and lYPos&, you can also use `%DEFAULT` for these parameters.

sPrompt$

The text that is to be displayed in the "prompt area" of the List Box, between the title bar and the list of items that can be selected.

If you have used the ConsoleListBoxDefaults function to define a default prompt, you can also use an empty string for this parameter, and the default prompt will be used.

sTitle$

The text that will be shown in the title bar of the List Box and (optionally) the List Box's buttons.

To change the text that is displayed in the List Box's buttons, use a string like this for *sTitle$*:

```
    "Title" + CHR$(0) + "Button1" + CHR$(0) + "Button2"
```

If you have used the ConsoleListBoxDefaults function to define a default title, you can use an empty string for this parameter, and the default title and button labels will be used.

If both this parameter and the default title are empty strings, the title `SELECT:` will be used. If you want to create a List Box with no text in the title bar, use a single space (" ") for this parameter.

sItems$()

A string *array* that contains the items that are to be displayed in the List Box. The array is limited to 32,000 elements on Windows 95/98/ME computers. Use empty parentheses after the name of the array; do not type a number in the parentheses (see **Example** below). Please note that if any element of the array contains an empty string, Windows will stop at that point and will not display any more items. (This is the default Windows behavior and is beyond the control of Console Tools.) It is particularly important to make sure that the very first element of the array contains a non-empty string, or *nothing* will be displayed in the List Box. (Remember that many arrays have a "zero" element and, if your array does, it *would* be counted as the first element of the array.)

You can use Shortcuts (such as \q for a Quote symbol) in the items that make up the array.

Note that it is not possible to use ConsoleListBoxDefaults. to define a default item list for this parameter. You must always use an actual string array for this parameter.

lDefault&

If this parameter contains a non-zero value, and if the `%MULTIPLE` option (see *lFlags&* below) is not being used, the List Box will automatically highlight the entry that corresponds to the value. For example, using a value of two

(2) for this parameter would cause the second item to be highlighted.

If you have used the ConsoleListBoxDefaults function to define a "default default", you can also use the predefined equate `%DEFAULT` for this parameter, and the default value will be used.

lFlags&

Use zero (0) if **1)** only one item may be selected from the list, and **2)** you want the selected string to be returned. Or...

Use the predefined equate `%MULTIPLE` if you want the List Box to allow more than one item at a time to be selected. If this option is used, the *lDefault&* parameter will have no effect on the List Box, because Windows does not allow mutliple-selection listboxes to have default items.

Use `%RETURN_INDEX` if you want the return value of this function to be a string that represents the one-based *index* of the selected item, instead of the selected string itself. For example, if `%RETURN_INDEX` was used as the *lFlags&* parameter and the first item was selected from the listbox, the string "1" would be returned. If `%MULTIPLE + %RETURN_INDEX` was used and the second and fifth items were selected, the strings "2" and "5", separated by `CHR$(0)`, would be returned.

If you have used the ConsoleListBoxDefaults function to define a default lFlags& value, you can also use the predefined equate `%DEFAULT` for this parameter, and the default flags will be used.

lNormalizeConsole&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a List Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the List Box, to make sure that the console screen is visible to the user.

If you have used the ConsoleListBoxDefaults function to define a default setting, you can also use the predefined equate `%DEFAULT` for this parameter, and the default setting will be used.

Return Value

If the user presses the Escape key or Alt-C (the hot key for the Cancel button), or selects the Cancel button or Close (x) button, or uses another standard Windows Close technique (such as Alt-F4), this function will return an empty string.

If the user selects an item from the List Box by double-clicking it, or selects an item (by single-clicking or by using the arrow keys) and then clicks the Ok button or presses the Enter key or Alt-O (the hot key for the Ok button) this function will return the string that corresponds to the selected item.

If the `%MULTIPLE` option is used and the user selects two or more items and then clicks the Ok button or presses Enter or Alt-O, this function will return all of the selected items as a single string, with the individual items separated by `CHR$(0)`. We recommend the use of the PB/CC `PARSECOUNT` and `PARSE$` functions to count and access the individual items in the returned string.

If the `%RETURN_INDEX` option is used, instead of a string containing the selected item(s) this function will return a string that contains the one-based *index number(s)* of the selected item(s). If the first item is selected, the string will include "1", and so on. Please note that "1" will be returned for the first item regardless of the actual array element number to which it corresponds. In other words, if your *sItems$()* array uses zero (0) for its first element, the first element still corresponds to a return value of "1".

Remarks

If the user clicks the Ok button or presses Enter or Alt-O *while no item is selected*, the List Box will "beep" and will not allow the user to select "nothing". Windows normally requires "Cancel" to be selected if the user's choice is "nothing", and Console Tools follows this standard. If possible, your *sPrompt$* string and/or *sTitle$* string should make it clear that a selection or an explicit "Cancel" is required. We recommend that you use the *lDefault&* parameter to specify a default item, so that it will be less likely that a user will select "Ok" when no item is selected.

If the `%MULTIPLE` option is used, the List Box will recognize the standard Windows "extended" multi-selection techniques. For example, it is possible to hold down a Ctrl key and click on two or more items to select (or un-select) them, or to hold down a Shift key to select a range of items by using the mouse or the arrow keys.

Example

```
'Create a small List Box with 10 items.
'Center it on the screen, and allow
'multiple items to be selected...

DIM sMyItems$(1:10)

sMyItems$(1) = "One"
sMyItems$(2) = "Two"
sMyItems$(3) = "Three"
sMyItems$(4) = "Four"
sMyItems$(5) = "Five"
sMyItems$(6) = "Six"
sMyItems$(7) = "Seven"
sMyItems$(8) = "Eight"
sMyItems$(9) = "Nine"

PRINT ConsoleListBox(1, _
                     %CONSOLE_CENTER, _
                     %CONSOLE_CENTER, _
                     "Please select one or more Numbers...", _
                     "Pick a Number", _
                     sMyItems$(), _
                     0, _
                     %MULTIPLE, _
                     0)
```

See Also

ConsoleListBoxDefaults

ConsoleListBoxDefaults

Purpose

Establishes default values for future Console List Boxes.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleListBoxDefaults lType&, _
                       lXPos&, _
                       lYPos&, _
                       sPrompt$, _
                       sTitle$, _
                       sReserved$, _
                       lDefault&, _
                       lFlags&, _
                       lNormalizeConsole&
```

Parameters

lType&

Use **1** for a small List Box, or **2** for a wide List box, or **3** for a tall List Box, or **4** for a wide, tall List Box. The use of negative 1 through negative 4 will produce the same results as using 1 through 4, except that the List Box will be created without the "Ok" and "Cancel" buttons. You can also use a value that is greater than 50,000 (or less than negative 50,000) to tell Console Tools to create a certain sized List Box. The last three digits of the number define the height of the List Box, and any numbers that precede that define the width. For example, using `111222` would produce a List Box that was 111 dialog units wide and 222 units high. Using `-99075` would produce a List Box that was 99 units wide and 75 high, and since the number is negative the Ok and Cancel buttons would not be displayed. Please note that Console Tools will not allow you to create a List Box that is too small to display at least two items and the buttons, so the use of relatively small width or height values may not produce the expected results. The largest List Box that can be created is `999999`.

lXPos& and *lYPos&*

These parameters determine the screen location where the List Box will be displayed. If you use zero (0) for both of these values the List Box will be auto-located in the upper-left corner of the console window (not the upper-left corner of the screen). You can also use the predefined equate `%DESKTOP_CENTER` to auto-center the List Box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the List Box in the middle of the console, or you can specify a number to indicate a number of pixels from the top-left corner of the screen at 0,0. Also see LocOfCol and LocOfRow for a technique that allows List Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the List Box "off the screen" and make it impossible for the user to see it or to select an item.*

sPrompt$

The text that is to be displayed in the "prompt area" of the List Box, between the title bar and the list of items that can be selected.

sTitle$

The text that will be shown in the title bar of the List Box and (optionally) the List Box's buttons.

To change the text that is displayed in the List Box's buttons, use a string like this for *sTitle$*:

```
"Title" + CHR$(0) + "Button1" + CHR$(0) + "Button2"
```

sReserved$

It is not possible to use the ConsoleListBoxDefaults function to define a default item list. This parameter should always be an empty string.

lDefault&

If this parameter contains a non-zero value, and if the `%MULTIPLE` option (see *lFlags&* below) is not being used, the List Box will automatically highlight the entry that corresponds to the value. For example, using a value of two (2) for this parameter would cause the second item to be highlighted.

lFlags&

Use zero (0) if **1)** only one item may be selected from the list, and **2)** you want the selected string to be returned. Or...

Use the predefined equate `%MULTIPLE` if you want the List Box to allow more than one item at a time to be selected. If this option is used, the *lDefault&* parameter will have no effect on the List Box, because Windows does not allow mutliple-selection listboxes to have default items.

Use `%RETURN_INDEX` if you want the return value of this function to be a string that represents the one-based *index* of the selected item, instead of the selected string itself. For example, if `%RETURN_INDEX` was used as the *lFlags&* parameter and the first item was selected from the listbox, the string "`1`" would be returned. If `%MULTIPLE + %RETURN_INDEX` was used and the second and fifth items were selected, the strings "`2`" and "`5`", separated by `CHR$(0)`, would be returned.

lNormalizeConsole&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a List Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the List Box, to make sure that the console screen is visible to the user.

Return Value

None.

Remarks

The ConsoleListBoxDefaults function can be used to set or change the *default* values for ConsoleListBox parameters. If you use this function to set one or more defaults, you can then use `%DEFAULT` (for numeric parameters) or an empty string (for string parameters) when calling the ConsoleListBox function, and the predefined default(s) will be used. For example, if you wanted all (or most) of the Console List Boxes in your program to use the same title bar text, you could use ConsoleListBoxDefaults to set that text as the default title. Then any Console List Box with "" as the sTitle$ parameter would automatically use the default title. (If you used ConsoleListBox with something other than "" as the sTitle$ parameter, the default would be ignored for *that* List Box.)

If you want to change Console List Box defaults that you have already set, you can use ConsoleListBoxDefaults more than once. If you want to change one default setting but not the others, use `%DEFAULT` or an empty string when using ConsoleListBoxDefaults

Example

See ConsoleInputBoxDefaults for several examples that can be applied to the use of ConsoleListBoxDefaults.

See Also

ConsoleListBox

ConsoleMessageBox

Purpose

Displays a Console Message Box and returns the ID number of the button that was selected by the user.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleMessageBox(sText$, _
                             lStyle&, _
                             sTitle$, _
                             lIconID&,_
                             lNormalize&)
```

(Also see Simplified Syntax below.)

Parameters

sText$

The text that will be displayed in the message box. You can include the Shortcut `\n` (which stands for New line) to add carriage returns to the text, and `\q` can be used to insert quotation marks. The Tab Shortcut `\t` can also be used, with varying results. To use a string that contains backslashes *without* Console Tools processing them as shortcuts, see iString. You can also use an empty string for sText$ if you want the default text (from the ConsoleMessageBoxDefaults function) to be used. To create a Message Box with no text, use a single space character for sText$.

lStyle&

This parameter is made up of one or more predefined equates, added together. *You are required to use one parameter from the "Buttons" group.* All of the other groups are optional. You can also use the value `%DEFAULT` if you want a default style (previously defined with the ConsoleMessageBoxDefaults function) to be used.

Buttons Use one of the predefined equates `%OKONLY, %OKCANCEL, %YESNO, %YESNOCANCEL, %RETRYCANCEL,` or `%ABORTRETRYIGNORE` to specify which buttons you want the Message Box to have.

Default Button If you want to control which button is highlighted as the Default Button when the Message Box is first displayed, you can add one of the equates `%DEFBUTTON1`, `%DEFBUTTON2`, or `%DEFBUTTON3`.

Windows Icons/Sounds If you want to specify a standard Windows Icon/Sound combination you can add one of the equates `%HANDBOX, %QUESTIONBOX, %EXCLAMATIONBOX,` or `%INFOBOX.`

IMPORTANT NOTE: If you use both the `%HANDBOX` icon and the `%SYSTEMMODAL` flag (see Modality below), Windows will display a "safe mode" message box that can be displayed even if Windows is running low on memory. The "safe mode" message box is limited to three lines, and

Windows does not automatically break the lines of text to fit the message box, so you must include \n (newline) shorthands to specify line breaks.

If you want to use a *nonstandard* icon you must use the *lIconID&* parameter (below), not the *lStyle&* parameter. If you do use a nonstandard icon, the `%...BOX` equates shown above will only affect the Message Box *Sound*.

IMPORTANT NOTE: Standard Windows Event Sounds depend on the setup of your *user's* computer. In many cases the user's Event Sounds will be turned off, so no sounds will be heard.

<u>Modality</u> If you want a Message Box to be System Modal (meaning that it will continue to be displayed on top of *all* programs until the users selects a button) use `%SYSTEMMODAL.` (See note about using `%SYSTEMMODAL` with `%HANDBOX` above.)

The default Modality value is `%APPLMODAL` (Application Modal), which means that the user is required to select a button before your program will continue running. It is only necessary to add `%APPLMODAL` to the *lStyle&* parameter if you've used the ConsoleMessageBoxDefaults function to define `%SYSTEMMODAL` as the default style but you want a particular Message Box to be Application Modal.

The third option, `%TASKMODAL`, does exactly the same thing as `%APPLMODAL` when it is used in Console Applications.

<u>Special Effects</u> You can add the equates `%ICONMASK` , `%SETFOREGROUND`, `%NOFOCUS`, `%TOPMOST`, and `%RIGHT` to create special-purpose Message Boxes. Note that Console Tools automatically uses `%TOPMOST` if the lNormalize& option is used (see below). Note also that *Windows* is responsible for responding to these flags, not Console Tools. If a flag does not work, it means that your computer's version of Windows does not recognize the flag.

sTitle$

The text that will be displayed in the Message Box's title line. You can use the various Shorthand strings like `\q` for Quotes in this parameter. .An empty string can be used if you want the default title (from the ConsoleMessageBoxDefaults function) to be displayed. If an empty string is used and no default title has been set, the fallback title `"Message:"` is used. To display a Message Box with no title, use a single space character.

lIconID&

Either **1)** The value zero (0) to indicate that the icon which was specified by the `%...BOX` equate in the *lStyle&* parameter (above) should be used, or **2)** the value `%IDI_CONSOLE` to specify the Console Tools Icon, or **3)** one of the values `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, <u>or</u> `%IDI_WINLOGO` to specify a Windows Standard Icon, or **4)** The Resource ID Number of an icon in a resource file that was linked into your PB/CC program with the `$RESOURCE` metastatement. See Using Icons for complete information, or **5)** The value `%DEFAULT` can also be used for this parameter, if a default value has been set by the ConsoleMessageBoxDefaults function. IMPORTANT NOTE: If it is not zero, the *lIconID&* value overrides the icon that was specified by the `%...BOX` value in the *lStyle&* parameter (above) but it does not override the sound specified by that value. For example, if you used `%HANDBOX` in the *lStyle&* parameter and used `%IDI_CONSOLE` for the *lIconID&* parameter, the message box would be displayed with the Console Icon but you would still hear the sound that is associated with `%HANDBOX`.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Message Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Message Box, to make sure that the console screen is visible to the user. This can be important if the console screen contains information that the user needs to respond to the Message Box. Example: A message box that says "Are you sure that the displayed values are correct?" (The value `%DEFAULT` can also be used for the lNormalize& parameter if you've previously defined a default by using the ConsoleMessageBoxDefaults function.)

Return Value

lResult& will be one of the following values, depending on which button was selected by the user: `%OKBUTTON, %CANCELBUTTON, %ABORTBUTTON, %RETRYBUTTON, %IGNOREBUTTON, %YESBUTTON,` or `%NOBUTTON.` Note that pressing the Enter key or Space bar corresponds to selecting *the currently highlighted button*. (This allows the user to select a button with the arrow keys and then press Enter.). Pressing the Escape key corresponds to selecting the Cancel button of a message box that has a Cancel button, or the Ok button of an `%OKONLY` Message Box. The Escape Key has no effect on the other types of Message Boxes.

Remarks

If your program doesn't need to know which button was selected by the user (as is always the case with an `%OKONLY` Message Box) you can use this alternate syntax...

```
ConsoleMessageBox  sText$, _
                   lStyle&, _
                   sTitle$, _
                   lIconID&,_
                   lNormalize&
```

Example

```
lResult& = ConsoleMessageBox("Do you want to exit?", _
                             %YESNO+%HANDBOX+%DEFBUTTON2, _
                             "Quit Now?", _
                             0, _
                             %TRUE)

IF lResult& =  %YESBUTTON THEN EXIT FUNCTION
```

Details

Some Computers don't display the capital letter W correctly when it is used as the first letter of a line of text in a Message Box. (This is a Windows problem, not a Console Tools problem.) The left side of the W is sometimes cut off, resulting in a letter that looks (in extreme cases) more like an italic *N* than a W. We recommend that you add a space to the beginning of lines that start with W, to avoid this effect. It may be necessary to add a leading space every line, to achieve a consistent left margin.

Simplified Syntax

The Console Tools Message Box has options that you probably won't want to change every time you use the function, so we suggest that you **1)** use the function called ConsoleMessageBoxDefaults to establish default parameters, and **2)** design a simple "wrapper" function that eliminates the parameters that you won't usually change. For example, you could add this to your program...

```
FUNCTION MSGBOX(sText$, lStyle&, sTitle$) AS LONG

      FUNCTION = ConsoleMessageBox(sText$, _
                                   lStyle&, _
                                   sTitle$, _
                                   %DEFAULT
                                   %TRUE)
END FUNCTION
```

You could then use the easier MSGBOX syntax for most of your Message Boxes. (Users of PowerBASIC's PB/DLL compiler will recognize the MSGBOX syntax.)

```
lResult& = MSGBOX("Click OK", %OKONLY, "My Program")
```

See Also

ConsoleMessageBoxDefaults

ConsoleMessageBoxDefaults

Purpose

Establishes default values for future Console Message Boxes.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleMessageBoxDefaults  sText$, _
                           lStyle&, _
                           sTitle$, _
                           lIconID&,_
                           lNormalize&
```

Parameters

sText$

The text that will be displayed in the message box. You can include the Shortcut `\n` (which stands for New line) to add carriage returns to the text, and `\q` can be used to insert quotation marks. The Tab Shortcut `\t` can also be used, with varying results. To use a string that contains backslashes *without* Console Tools processing them as shortcuts, see iString.

lStyle&

This parameter is made up of one or more predefined equates, added together. *You are required to use one parameter from the "Buttons" group.* All of the other groups are optional.

Buttons Use one of the predefined equates `%OKONLY, %OKCANCEL, %YESNO, %YESNOCANCEL, %RETRYCANCEL,` or `%ABORTRETRYIGNORE` to specify which buttons you want the Message Box to have.

Default Button If you want to control which button is highlighted as the Default Button when the Message Box is first displayed, you can add one of the equates `%DEFBUTTON1`, `%DEFBUTTON2`, or `%DEFBUTTON3`.

Windows Icons/Sounds If you want to specify a standard Windows Icon/Sound combination you can add one of the equates `%HANDBOX, %QUESTIONBOX, %EXCLAMATIONBOX,` or `%INFOBOX.` See Using Icons for more information. If you want to use a *nonstandard* icon you must use the *lIconID&* parameter (below), not the *lStyle&* parameter. If you do use a nonstandard icon, the `%...BOX` equates shown above will only effect the Message Box *Sound*. IMPORTANT NOTE: Standard Windows Event Sounds depend on the setup of your *user's* computer. In many cases the user's Event Sounds will be turned off, so no sounds will be heard.

Modality The default value is `%APPLMODAL` (Application Modal), which means that the user is required to select a button before your program will continue running. If you want your Message Boxes to be System Modal (meaning that they will continue to be displayed on top of *all* programs until the users selects a button) use `%SYSTEMMODAL.` (The third option, `%TASKMODAL`, does exactly the same thing as `%APPLMODAL` when it is used in Console Applications.)

Special Effects   You can add the equates `%ICONMASK` , `%SETFOREGROUND`, `%NOFOCUS`, `%TOPMOST`, and `%RIGHT` to create special-purpose Message Boxes. Note that Console Tools automatically uses `%TOPMOST` if the lNormalize& option is used (see below). Note also that *Windows* is responsible for responding to these flags, not Console Tools. If a flag does not work, it means that your computer's version of Windows does not recognize the flag.

sTitle$

The text that will be displayed in the Message Box's title line. You can use the various Shorthand strings like `\q` for Quotes in this parameter. .An empty string can be used if you want the default title (from the ConsoleMessageBoxDefaults function) to be displayed. If an empty string is used and no default title has been set, the fallback title `"Message:"` is used. To display a Message Box with no title, use a single space character.

lIconID&

Either **1)** The value zero (0) to indicate that the icon which was specified by the `%...BOX` equate in the lStyle& parameter (above) should be used, or **2)** the value `%IDI_CONSOLE` to specify the Console Tools Icon, or **3)** one of the values `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, or `%IDI_WINLOGO` to specify a Windows Standard Icon, or **4)** The Resource ID Number of an icon in a resource file that was linked into your PB/CC program with the `$RESOURCE` metastatement. See Using Icons for complete information. IMPORTANT NOTE: If it is not zero, the *lIconID&* value overrides the icon that was specified by the `%...BOX` value in the *lStyle&* parameter (above) but it does not override the sound specified by that value. For example, if you used `%HANDBOX` in the *lStyle&* parameter and used `%IDI_CONSOLE` for the *lIconID&* parameter, the message box would be displayed with the Console Icon but you would still hear the sound that is associated with `%HANDBOX`.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Message Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Message Box, to make sure that the console screen is visible to the user. This can be important if the console screen contains information that the user needs to respond to the Message Box. Example: A message box that says "Are you sure that the displayed values are correct?"

Return Value

None.

Remarks

The ConsoleMessageBoxDefaults function can be used to set or change the *default* values for Console Message Boxes. If you use this function to set one or more defaults, you can then use `%DEFAULT` (for numeric parameters) or an empty string (for string parameters) when calling the ConsoleMessageBox function, and the predefined default(s) will be used. For example, if you wanted all (or most) of the Message Boxes in your program to use the same title bar text, you'd use ConsoleMessageBoxDefaults to set that text as the default title. Then any Console

Message Box with "" as the sTitle$ parameter would automatically use the default title. (If you used ConsoleMessageBox with something other than "" as the sTitle$ parameter, the default would be ignored for *that* Message Box.)

If you want to change Console Message Box defaults that you have already set, you can use ConsoleMessageBoxDefaults more than once. If you want to change one default setting but not the others, use `%DEFAULT` or an empty string when using ConsoleMessageBoxDefaults (see last Example).

Example

```
'set up defaults for all parameters...
ConsoleMessageBoxDefaults  "Please select Yes or No", _
                           %YESNO + %IDI_HAND, _
                           "My Program", _
                           %IDI_ICON1, _
                           %TRUE

'use all of the defaults in an Input Box...
lResult& = ConsoleMessageBox("" ,%DEFAULT,"" , _
                             %DEFAULT,%DEFAULT)

'use some of the defaults in a Message Box...
lResult& = ConsoleMessageBox("" ,%DEFAULT,"" ,0,%TRUE)

'change the default title bar text but nothing else...
ConsoleMessageBoxDefaults  "", _
                           %DEFAULT, _
                           "My Other Program", _
                           %DEFAULT, _
                           %DEFAULT
```

See Also

ConsoleMessageBox, Using Predefined Equates

ConsoleMetrics

Purpose

Provides several different console "metrics" or "element measurements".

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleMetrics(lType&)
```

Parameters

lType&

Specifies which metric (measurement) you are requesting. See **Remarks** below for a complete list of valid values.

Return Value

If lType& is a valid value, this function will return the requested console metric. (In some cases zero (0) is returned if the specified screen element is not currently visible.) All return values are in pixels.

If an invalid lType& value is used, `%ERROR_CT_INVALIDPARAMETER` will be returned.

Remarks

Many different measurements that relate to the *entire* console window (such as the size of the print area and the total number of rows/columns) can be obtained with the ConsoleInfo function. The ConsoleMetrics function, on the other hand, provides information about individual *elements* of the console window.

The following values can be obtained:

`%COL_WIDTH` and '`%ROW_HEIGHT`

The width or height of one column or row of the console's "print area". (Note that together, the `%COL_WIDTH` and `%ROW_HEIGHT` metrics can be used to determine the size of the font that the console window is currently using. See **Example** below.)

`%FRAME_WIDTH` and '`%FRAME_HEIGHT`

The width or height, in pixels, of the frame that surrounds the console window.

`%VSCROLLBAR_WIDTH` and `%HSCROLLBAR_HEIGHT`

The width of the console's vertical scroll bar, or the height of the console's horizontal scroll bar. If the specified scroll bar is not currently visible, zero (0) is returned.

`%TOOLBAR_HEIGHT`

The height of the Windows 95/98/ME console toolbar, in pixels. If the toolbar is not currently visible, zero (0) is returned.

`%PULLDOWN_HEIGHT`

The height of the pulldown menu bar that will be displayed if the PulldownMenu function is used. (Since the PulldownMenu function is modal -- i.e. since it is not possible to use other functions while the pulldown menu is actually being displayed -- the ConsoleMetrics function returns a value for this lType& value even if the menu is not currently being displayed. Otherwise it would not be possible to obtain this value.)

`%TITLEBAR_HEIGHT`

The height of the console title bar.

`%BORDER_SIZE`

The thickness of the border that Windows is using for the console window. This border is used in several places, such as the line between the title bar and the rest of the console.

Example

```
PRINT "This console is using a font that is";
PRINT ConsoleMetrics(%COL_WIDTH);
PRINT "pixels wide and";
PRINT ConsoleMetrics(%ROW_HEIGHT);
PRINT "pixels high."
```

See Also

ConsoleInfo

ConsoleMove

Purpose

Changes the location of the console window on the screen.

Availability

Console Tools Standard and Pro

Warning

It is possible to move the console window to a screen location that is not visible to the user. See Remarks.

Syntax

```
lResult& = ConsoleMove(lXPos&, lYPos&)
```

Parameters

lXPos& and *lYPos&*

Either **1)** the predefined equate `%DESKTOP_CENTER` to center the console window on the desktop, or **2)** the desired screen coordinates of the top-left corner of the console window, in Pixels, relative to the top-left corner of the screen at 0,0. *WARNING: It is possible to use values for lXPos& and lYPos& that will position the console window "off the screen" and make it impossible for the user to see it.*

Return Value

lResult& will be `%SUCCESS` (zero) if the ConsoleMove operation is successful.

If the operation does not succeed, lResult& will *usually* be the Windows Error Code that caused the failure.

Under certain circumstances, Windows will fail to perform the requested operation but it will not provide a Windows Error Number. In that case, lResult& will be `%ERROR_CT_UNKNOWNERROR`.

Remarks

If you use a negative value for lXPos& the ConsoleMove function will move the console to a point that is "to the left of left". In other words, if you use --10 the console will be moved to an imaginary point that is 10 pixels to the left of the left edge of the screen. Similarly, if you use a negative value for lYPos& the function will move the console to a point that is "above the top".

It is common for small negative numbers to be used. For example, when a console window with a four-pixel-wide border is Maximized, Windows will locate it at -4, -4 so that the top and left borders are off the screen.

Keep in mind that it is possible to move the console window completely "off the screen", i.e. to a screen location that is not visible to the user. If your console window disappears after ConsoleMove is used, you can use `ConsoleInfo(%WINDOW_TOP)` and `ConsoleWindow(%WINDOW_LEFT)` to find out where it is.

Under certain circumstances Windows will move the Console off the screen. Windows 95/98/ME will sometimes move the screen to 3000,3000 or -3000, -3000. Windows NT, 2000, and XP will sometimes move the screen to 32000, 32000 or -32000, -32000. (Your programs should assume that other large values are possible.)

Example

```
IF ConsoleMove(0,0) = %SUCCESS THEN
      PRINT "THE CONSOLE IS NOW LOCATED IN THE ";
      PRINT "TOP-LEFT CORNER OF THE SCREEN."
END IF
```

Details

You should note that it is possible to use ConsoleMove to "hide" the console window in a way that can't be detected by ConsoleIsHidden or ConsoleState.

See Also

ConsoleIsHidden, ConsoleState, Appendix B: Console States

ConsoleNormal

Purpose

Performs a variety of operations on the console to make sure that it is in a "normal" state.

Availability

Console Tools Standard and Pro

Warning

This function forces your program into the windows foreground by using the `ConsoleToForeground %HARD` function. See ConsoleToForeground for details.

Syntax

```
ConsoleNormal
```

Parameters

None.

Return Value

None.

Remarks

It is possible for the Windows Console to be displayed in a wide variety of states. Some of these states are very difficult to use, so Console Tools provides the ConsoleNormal function. It makes sure that...

- The FullScreen Mode is turned off.
- The console window is in the windows foreground and has the keyboard focus.
- The console window is Maximized (not Minimized or Restored).
- The console window is not Hidden.
- The console window is not located "off the screen".
- The console window is not being displayed in an odd size (usually far too small) as often happens when the Win95/98/ME Window Menu Font "Auto Size" option is used. See Microsoft Console Windows for more information about this effect.

The ConsoleNormal function is automatically used by the ConsoleMessageBox, ConsoleInputBox, SplashBoxShow, and ProgressBoxShow functions if you use their lNormalize& parameters.

Example

```
ConsoleNormal
PRINT "YOUR USER CAN DEFINITELY SEE THIS TEXT ";
PRINT "(unless the monitor is turned off ";
PRINT "or a screen-saver is running ";
PRINT "or they are not looking at the monitor)."
```

See Also

ConsoleState, Console Window States, Console Windows, ConsoleMessageBox, ConsoleInputBox, ProgressBoxShow, SplashBoxShow

ConsolePEEK

Purpose

Reads characters or screen attributes (colors) from the console screen buffer Page 1.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
sResult$ = ConsolePEEK(lType&, _
                       lRow&, _
                       lColumn&, _
                       lLength&)
```

Parameters

lType&

Use zero (0) or the predefined equate `%CHARS` to read characters from the console. Use a nonzero value or the predefined equate `%COLORS` to read the screen "attributes", i.e. color information.

lRow&

The console screen row where you want to start reading.

lColumn&

The console screen column where you want to start reading.

lLength&

The number of characters or color attribute bytes that you want to read.

Return Value

sResult$ will contain a string that corresponds to the requested screen area.

Remarks

This function simulates screen-buffer-PEEK operations in traditional DOS programs.

Unlike DOS PEEK operations, ConsolePEEK does not require the use of DEF SEG to specify the screen segment. In fact DEF SEG is not recognized by any of the PowerBASIC Windows compilers.

Unlike DOS PEEK operations, ConsolePEEK returns *either* a string of characters or a string of screen attributes (colors). DOS PEEK operations always return a single string of alternating characters and attributes. Microsoft Windows Console Screen Buffers are not constructed like DOS buffers, and since it is a common practice for a program to separate the alternating-character strings into two more useful strings, Console Tools performs that operation for you. If your circumstances are unusual and you need an alternating-character string, it would be relatively simple to construct a PB/CC routine to combine the two strings returned by ConsolePEEK. You would have to re-separate them, however, before the results could be used with ConsolePOKE.

Unlike DOS PEEK operations, ConsolePEEK always returns characters or attributes from screen Page 1, which may or may not be the currently visible page. Because the PB/CC compiler manages the eight different console window "display pages" internally -- it doesn’t allow programs to obtain the "handles" of the various pages -- it is currently only possible for ConsolePEEK to access Page 1. (See the PAGE statement in the PB/CC documentation.)

It *is* possible to read "across rows" with ConsolePEEK. For example, if you are using a console window with 80 columns and you tell ConsolePEEK to read more than 80 characters or attributes, you will receive a string that starts with the specified row/column and continues for the specified number of characters, regardless of the row-length. The Console Screen Buffer is treated as one long string with (rows times columns) characters and the same number of attribute bytes.

It is possible to use ConsolePEEK and ConsolePOKE to save and restore console screens or portions of screens. For an easier, row-oriented method see ConsoleScreenSave and ConsoleScreenLoad. If you need to save a portion of a row, or need to begin/end a screen save/load operation in the middle of a row, ConsolePEEK and ConsolePOKE are the appropriate tools.

Example

```
CLS
COLOR 14,1
LOCATE 10,10
PRINT "HELLO WORLD"
sChars$  = ConsolePEEK(%CHARS,  10, 16, 5)
sColors$ = ConsolePEEK(%COLORS, 10, 16, 5)
```

This example would return the string "WORLD" in sChars$, and sColors$ would contain a string that represents the colors in the letters of the word "WORLD".

Details

A Windows Console Screen Buffer is not really very similar to a DOS screen buffer. It is actually handled as two different buffers: one for characters and one for screen attributes. And each buffer uses two bytes per character or attribute instead of one, so that the console can support "Unicode" fonts. Console Tools hides most of those differences. Console Tools functions always return ASCII characters and attributes instead of Unicode strings.

See Also

Windows Consoles, ConsolePOKE, Appendix G: Console Window Screen Color Numbers

ConsolePOKE

Purpose

Places characters or attributes (colors) into the console window screen buffer Page 1.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsolePOKE lType&, _
            sString$, _
            lRow&, _
            lColumn&
```

Parameters

lType&

Use zero (0) or the predefined equate `%CHARS` to place characters into the screen buffer. Use a nonzero value or the predefined equate `%COLORS` to place screen attributes into the buffer.

sString$

The string of characters or screen-attribute-bytes to be placed into the buffer.

lRow&

The screen row where the first character or byte of sString$ should be placed.

lColumn&

The screen column where the first character or byte of sString$ should be placed.

Return Value

None.

Remarks

The ConsolePOKE function is used to "poke" information onto the screen, i.e. to place characters and attributes (colors) onto Page 1 of the console screen buffer.

When used to display text, it is different from the PB/CC PRINT statement because it places characters onto the screen *without* affecting the screen colors and without affecting the location of the PB/CC cursor.

When used to change the colors of an area on the screen, it is similar to the PB/CC COLOR statement when the optional third parameter is used, but it is much easier to use ConsolePOKE if you're making complex color changes. (It is also not necessary to change the PB/CC cursor location when using ConsolePOKE, as it is with COLOR.)

Unlike DOS POKE operations, ConsolePOKE does not require the use of DEF SEG to specify the screen segment. In fact DEF SEG is not recognized by any of the PowerBASIC Windows compilers.

Unlike DOS POKE operations, ConsolePOKE uses *either* a string of characters or a string of screen attributes (colors). DOS POKE operations always use a single string of alternating characters and attributes.

Unlike DOS POKE operations, ConsolePOKE always affects characters or attributes on screen Page 1, which may or may not be the currently visible page. Because the PB/CC compiler manages the eight different console window "display pages" internally -- it doesn’t allow programs to obtain the "handles" of the various pages -- it is currently only possible for ConsolePOKE to access Page 1. (See the PAGE statement in the PB/CC documentation.)

For more information about the Console Screen Buffer, see ConsolePEEK and Console Screen Buffers.

Example

```
'Place "HELLO WORLD..." onto Page 1 of the
'console screen buffer at row 13, column 26
ConsolePOKE %CHARS, "HELLO WORLD...", 13, 26

'Change the colors of the string above
sString$ = CHR$(1,2,3,4,5,6,7,8,9,10,11,12,13,14)
ConsolePOKE %COLORS, sString$, 13, 26
```

See Also

Windows Consoles, ConsolePEEK, Appendix G: Console Window Screen Color Numbers

ConsolePropMenu

Purpose

Activates the console window's Properties Menu (i.e. the Properties submenu of the console's Window Menu).

Availability

Console Tools Pro Only (see)

Warning

Attempting to use this function *after* you have used DeleteWindowMenuItem `%MENUITEM_PROPERTIES` can result in an Application Error (a Windows "General Protection Fault").

Syntax

```
ConsolePropMenu
```

Parameters

None.

Return Value

None.

Remarks

This function can be used to display the console window's Properties Menu, to allow your program's user to manually change certain console window settings.

Example

```
ConsolePropMenu
```

See Also

The Console Window Menu

ConsoleScreenLoad

Purpose

Loads a previously saved screen (or a portion of a screen) into the first PB/CC screen buffer (Page 1).

Availability

Console Tools Standard and Pro

Warning

If your program uses the ConsoleScreenLoad function with Screen Buffers (rather than disk files) it **must** also use the InitConsoleTools function to initialize the Console Tools Screen Buffers properly. See Remarks (below) for more details.

Syntax

```
lResult& = ConsoleScreenLoad(sLoadSpec$, _
                             lFirstRow&, _
                             lRows&, _
                             lColor&)
```

Parameters

sLoadSpec$

Either **1)** the name of a BLOAD-compatible disk file to be loaded, or **2)** a single character string indicating the number of the Console Tools Screen Buffer from which the screen should be loaded. For example, using `CHR$(5)` would load a screen or partial screen from buffer #5. Console Tools Standard DLL users can use `CHR$(0)` through `CHR$(7)`. Console Tools Pro users can use any of the 256 ASCII characters.

lFirstRow&

The screen row where the loading process should begin. Use 1 for full-screen loads. Numbers larger than 50 may not be used.

lRows&

The number of rows that should be loaded from the file or buffer. Use 0 to load *all* of the rows that are currently in the specified file or buffer. Numbers larger than 50 may not be used.

lColor&

Use zero (0) for a normal load. Use negative one (-1) for a load that automatically strips the blink-bit from incoming colors. (See Console Window Screen Colors for more information about this.) Use a number from 1 to 255 to force the screen to be loaded in a single color . (See Console Window Screen Colors for the color coding system.)

Return Value

lResult& will be one of the following values:

`%SUCCESS` (zero) if the requested screen is loaded properly, or

`%ERROR_CT_INVALIDSCREENNUMBER` if an invalid buffer number is specified (see the limitations of sLoadSpec$, above), or

`%ERROR_CT_INVALIDPARAMETER` if an invalid parameter is specified (such as row 51 or color 256), or

`%ERROR_CT_FILENOTFOUND` if the requested disk file does not exist, or if it exists and has zero length, or

`%ERROR_CT_INVALIDFILEFORMAT` if the requested file is less than seven bytes long (the minimum length for a BLOAD-compatible file), or

A number larger than `%ERROR_CT_FIRSTPBERROR` will be returned if a PowerBASIC PB/DLL error occurred during the loading of the requested disk file. (That's right: Console Tools was created with PB/DLL.) The actual error number that is returned by the function will be `%ERROR_CT_FIRSTPBERROR` plus the ERR value that is being reported. For example, a *Disk Media Error*, which is ERR number 72, would be reported as `%ERROR_CT_FIRSTPBERROR + 72,` which is equal to `999,001,072.` The original ERR number can be easily calculated by subtracting the predefined equate `%ERROR_CT_FIRSTPBERROR` from the return value of the function.

Remarks

VERY IMPORTANT NOTE: Every program that uses Console Tools must use the InitConsoleTools function to initialize the DLL. One of the parameters of the InitConsoleTools function tells the DLL the largest Console Screen Buffer Number that the program will use. If you use the ConsoleScreenLoad function to load screens from the Screen Buffers, you **must** change the third parameter of the InitConsoleTools function to reflect the largest buffer number that you will use. Using a too-large value for the third parameter of InitConsoleTools wastes memory. Using a number that is too small -- or forgetting to change the number from zero -- will result in failures of the ConsoleScreenLoad and ConsoleScreenSave functions.

Please also note that because the PB/CC compiler manages the eight different console window "display pages" internally -- it doesn't allow programs to obtain the "handles" of the various pages -- it is currently only possible for Console Tools to load screens onto Page 1. (See the PAGE statement in the PB/CC documentation.)

Examples

```
IF ConsoleScreenLoad("SCREEN1",1,0,0) = %SUCCESS THEN
      'the disk file SCREEN1 was loaded properly
ELSE
      CLS
      PRINT "SCREEN1 FAILED TO LOAD"
END IF

IF ConsoleScreenLoad("SCREEN2",10,1,0) = %SUCCESS THEN
      'the first line from the disk file SCREEN1 was
      'loaded into line 10 of screen page 1.
END IF

IF ConsoleScreenLoad("SCREEN3",1,25,-1) = %SUCCESS THEN
      'The disk file SCREEN3 was loaded into
      'console screen page 1.  If it was longer
      'than 25 lines, the remaining lines were
      'ignored.  Also, the blink-bit was stripped
      'from the colors in the file.
END IF

IF ConsoleScreenLoad(CHR$(1),1,0,0) = %SUCCESS THEN
      'the screen in buffer 1 was loaded properly
END IF
```

Details

The ConsoleScreenLoad function is, of course, compatible with files which have been created with the ConsoleScreenSave function.

ConsoleScreenLoad is also 100% compatible with files that have been created with the PB/DOS function BSAVE.

ConsoleScreenLoad and ConsoleScreenSave can easily be used to perform useful screen operations like Insert Row, Delete Row, Scroll Row Range, and many others.

See Also

ConsoleScreenSave, Console Screen Colors, Microsoft Console Windows, ConsolePEEK, ConsolePOKE

ConsoleScreenSave

Purpose

Saves the contents of the console window screen Page 1 (or a portion of the screen) for later loading with ConsoleScreenLoad.

Availability

Console Tools Standard and Pro

Warning

If your program uses the ConsoleScreenSave function with Screen Buffers (rather than disk files) it **must** also use the InitConsoleTools function to initialize the Console Tools Screen Buffers properly. See Remarks (below) for more details.

Syntax

```
lResult& = ConsoleScreenSave(sSaveSpec$, _
                             lFirstRow&, _
                             lRows&)
```

Parameters

sSaveSpec$

Either **1)** the name of the disk file where the screen should be saved, or **2)** a single-character string indicating the number of the Console Tools Screen Buffer where the screen should be saved. For example, using `CHR$(7)` would save a screen or partial screen in buffer #7. Console Tools Standard DLL users can use `CHR$(0)` through `CHR$(7)`. Console Tools Pro users can use any of the 256 ASCII characters.

lFirstRow&

The first row of the screen area to be saved. Use 1 for full-screen saves.

lRows&

The number of rows that should be saved. Use 0 to save *all* of the rows in the current screen buffer.

Return Value

lResult& will be one of the following values:

`%SUCCESS` if the screen was saved properly.

`%ERROR_CT_INVALIDPARAMETER` if an invalid lFirstRow& or lRows& value is used.

`%ERROR_CT_INVALIDSCREENNUMBER` if an invalid Screen Buffer number is used (see limitations of sSaveSpec$ above).

A number larger than `%ERROR_CT_FIRSTPBERROR` will be returned if a PowerBASIC PB/DLL error occurred during the saving of the requested disk file. (That's right: Console Tools was created with PB/DLL.) The actual error number that is returned by the function will be `%ERROR_CT_FIRSTPBERROR` plus the ERR value that is being reported. For example, a *Permission Denied* error, which is ERR number 70, would be reported as `%ERROR_CT_FIRSTPBERROR + 70,` which is equal to `999,001,070.` The original ERR number can be easily calculated by subtracting the predefined equate `%ERROR_CT_FIRSTPBERROR` from the return value of the function.

Remarks

VERY IMPORTANT NOTE: Every program that uses Console Tools must use the InitConsoleTools function to initialize the DLL. One of the parameters of the InitConsoleTools function tells the DLL the largest Console Screen Buffer Number that the program will use. If you use the ConsoleScreenSave function to save screens in the Screen Buffers, you **must** change the third parameter of the InitConsoleTools function to reflect the largest buffer number that you will use. Using a too-large value for the third parameter of InitConsoleTools wastes memory. Using a number that is too small -- or forgetting to change the number from zero -- will result in failures of the ConsoleScreenLoad and ConsoleScreenSave functions.

Please also note that because the PB/CC compiler manages the eight different console window "display pages" internally -- it doesn’t allow programs to obtain the "handles" of the various pages -- it is currently only possible for Console Tools to save screens from Page 1. (See the PAGE statement in the PB/CC documentation.)

Examples

```
IF ConsoleScreenSave("SCREEN1",1,0) = %SUCCESS THEN
      'The screen-save operation worked and
      'the file SCREEN1 has been created.  If
      'it already existed, it was overwritten.
END IF

If ConsoleScreenSave(CHR$(3),1,0) = %SUCCESS THEN
      'The entire Page 1 screen has been
      'saved to Console Screen Buffer #3
END IF

If ConsoleScreenSave(CHR$(7),3,2) = %SUCCESS THEN
      'Rows 3 and 4 of Page 1 have been
      'saved to Buffer #7.  (The 3,2
      'means "row 3 for 2 rows".)
END IF
```

Details

The files that the ConsoleScreenSave function creates are, of course, compatible with the ConsoleScreenLoad function.

The files that ConsoleScreenSave creates are also 100% compatible with the PB/DOS function BLOAD.

See Also

ConsoleScreenLoad, Console Window Screen Color Numbers, Microsoft Console Windows, ConsolePEEK, ConsolePOKE

ConsoleState

Purpose

Returns a value that corresponds to the current Window State of the console window.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleState
```

Parameters

None.

Return Value

lResult& will be one of the following predefined equates: `%RESTORED`, `%MINIMIZED`, `%MAXIMIZED`, or `%HIDE`. (See note about `%HIDE` vs. `%HIDDEN` in Remarks below.)

Remarks

The `%RESTORED`, `%MINIMIZED`, and `%MAXIMIZED` equates are defined in the Console Tools INC files to make the ConsoleState function easier to use. You could also use the equates `%SHOWNORMAL`, `%SHOWMINIMIZED`, and `%SHOWMAXIMIZED`, since they have exactly the same numeric values as `%RESTORED`, `%MINIMIZED`, and `%MAXIMIZED`, but the code...

```
    IF ConsoleState = %RESTORED THEN BEEP
```

...is more "natural" and easier to understand than...

```
    IF ConsoleState = %SHOWNORMAL THEN BEEP
```

...so the additional equates have been provided.

IMPORTANT NOTE: If you use the WIN32API.INC file in your programs, you must be careful to *not* use the equate `%HIDDEN` with the ConsoleState function. In the WIN32API.INC file, the equate `%HIDDEN` is defined as it relates to File Attributes, not Console States. A disk file that has the `%HIDDEN` attribute has the Microsoft-defined value of two (2), but a console window with the `%HIDE` attribute has the Microsoft-defined value of zero (0). The `%HIDE` and `%HIDDEN` equates are therefore not interchangeable, and you must be careful not to use...
`IF ConsoleState = %HIDDEN` because this is equivalent to
`IF ConsoleState = 2`, which is equivalent to
`IF ConsoleState = %MINIMIZED`, *not* `%HIDE.`

Keep in mind that console windows are *always* in two "basic" states at the same time. For example, a Console can be both Maximized and Showing, or Maximized and Hidden (meaning that if the Console were to be shown it would appear Maximized). So if the ConsoleState function returns `%HIDDEN`, your program *may* need to check the ConsoleIsMaximized and/or ConsoleIsMinimized functions to get "the rest of the story".

Also keep in mind that the ConsoleState function does not report the FullScreen status of the console window. It is possible for a console window to be in two states at the same time in a *different* way. For example, the ConsoleState function might return `%MINIMIZED` at the same time that the ConsoleIsFullScreen function returns True. This means that either **1)** the Windows operating system has minimized the console window "in the background" while the FullScreen display is being used, and/or **2)** if the FullScreen mode is later turned off with Alt-Enter, the console window will be revealed in the Minimized mode.

Example

```
IF ConsoleState = %MINIMIZED THEN
      'un-minimize the console window
      ConsoleWindow %UNMINIMIZE
END IF
```

See Also

Console Window States, ConsoleIsMinimized, ConsoleIsMaximized, ConsoleWindow

ConsoleStretch

Purpose

When used with other Console Tools functions, the ConsoleStretch function can create a console window that fills the entire computer screen. This includes covering the Windows Task Bar, which is usually not possible.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleStretch
```

Parameters

None.

Return Value

None.

Remarks

You must use source code that is very similar to sequence described here, or the ConsoleStretch function will not work properly. The `STRETCH.BAS` file that is supplied with Console Tools contains complete source code that follows this sequence, but we strongly recommend that you read this narrative.

STEP 1: By far the *best* way to use the ConsoleStretch function is to begin by using the CWC program to launch your PB/CC application. CWC is capable (via the `FONTSIZE: MAX` option) of creating a console window that fills the screen from left to right. The various Console Tools functions (including ConsoleStretch) can then be used to modify the console so that it fills the screen from top to bottom. Without CWC, the number of columns and rows of the final console will be highly unpredictable.

IMPORTANT NOTE: If you use CWC and ConsoleStretch together, you must not use CWC's `WINSTATE: MAXIMIZE` option. If you use `WINSTATE: MAX` or `MAXIMIZE` the ConsoleStretch function will not work properly. You should use `WINSTATE: SHOW` or, for even better results, `WINSTATE: HIDE`. Using `HIDE` will conceal all of the changes that we are going to make in the console window, until it is ready to display.

IMPORTANT NOTE: If you use CWC and ConsoleStretch together on a Windows 95, 98, or ME computer, you must not use CWC to disable the console toolbar. If you use `TOOL_BAR: NO` the ConsoleStretch function will not work properly on 95/98/ME computers. (We will be disabling the toolbar in a different way.)

After CWC has launched your program with the `FONTSIZE: MAX` option, the console will be 80 columns wide and it will fill the desktop from left to right, as much as possible. That is to say, CWC will choose the font size that does the best possible job of filling the screen from left to right, but it won't always be perfect. For example, if your computer is using the 800x600 screen resolution, CWC will choose a font that is 10 pixels wide, because 80 columns times 10 pixels equals the exact width of the screen (800 pixels). The screen will be filled from left to right. But if your computer is using the 1024x768 screen resolution, the best that CWC can do is use a font that is 12 pixels wide. Since 80 columns times 12 pixels equals 960 pixels, 1024 minus 960 (=64) pixels will be left over. The console will fill all but 64 pixels of the screen, from left to right. (If CWC used a font that was 13 pixels wide instead of 12, it would over-fill the screen because 80 times 13 equals 1040, which is obviously larger than 1024.)

STEP 2: Use the ConsoleMove function to position the console at the top-left corner of the screen. For best results, use the code that is supplied in the `STRETCH.BAS` file. It will position the console so that the top and left console borders are located "off the screen".

STEP 3: Use the ConsoleToolBar `%OFF` function to disable the Windows 95/98/ME toolbar.

STEP 4: We recommend that you finish configuring the console at this point. You may want to use the ConsoleIcon, ConsoleTitle, and other functions.

STEP 5: Because of the imperfections that are possible in step 1, your program should add a few columns to the console, if necessary, in order to perfectly fill the screen from left to right. You can use the ConsoleInfo(`%SCREEN_WIDTH`) function to find out how wide the screen is, and the ConsoleMetrics(`%COL_WIDTH`) function to find out how wide one column is. Dividing the first number by the second will give you the number of columns that are required to fill the screen.

If that number is not the default console size of 80 columns, you should use the ConsoleControl(`%BUFFER_WIDTH`) function to increase the column count. For example, if the screen width is 1024 and the column width is 12, the division equals 85.333. You would need to set the column count to 86 in order to *completely* fill the screen. (The PB/CC `CEIL` function can be used to round the number upward.)

STEP 6: Now that the console is capable of filling the screen from left to right, we need to add enough rows to fill the screen from top to bottom. Basically, you should repeat the procedure in step 5, but with a few modifications.

First, use the ConsoleInfo(`%SCREEN_`**`HEIGHT`**) function to determine the height of the screen. But you should then subtract the height of the console title bar and frame, as shown in the `STRETCH.BAS` file. Then you should divide the resulting number by the value of the ConsoleMetrics(`%`**`ROW_HEIGHT`**), function, and use the ConsoleControl(`%BUFFER_`**`HEIGHT`**) function to change the console height.

At this point, the console is *theoretically* large enough to fill the screen, but we have only changed the buffer size, not the "visible" size of the console. If we try to increase the visible size of the console by using the ConsoleControl(`%CONSOLE_`) functions, Windows will refuse to cover the Task Bar and will create a console that is smaller than we want, with scroll bars.

STEP 7: Use the ConsoleTopMost `%TRUE` function to give the console "topmost" status. This will allow the console to cover the task bar.

But we still can't use the ConsoleControl function to increase the visible size of the console. Windows won't create a console that is larger than the *desktop*.

STEP 8: Use the ConsoleStretch function. It will resize the console window so that it fills the entire screen, not just the desktop.

STEP 9: If you used `WINSTATE: HIDE` in step 1, you may or may not need to use the ConsoleState `%RESTORE` function to make the console visible, depending on which version of Windows you are using. Since doing that will not interfere with a program's operation, we recommend that you simply perform that step here.

That's it! The console should now fill the screen, including the area that is usually occupied by the Task Bar.

You should not change the order of the steps, but for the best possible results you may choose to modify a few things. For example, if you are using the 1024x768 screen that is described above, and therefore using a console that is 86 columns wide, you will find that two-thirds of the far-right column is not visible, and part of the title bar's Close (x) button will also be cut off. For this reason you may choose to use the ConsoleMove function to move the console very slightly to the left.

Examples

See the `STRETCH.BAS` file that is provided with Console Tools.

See Also

ConsoleTopMost

ConsoleSysTrayIcon

Purpose

Displays, changes, or removes a System Tray icon. (The System Tray is the portion of the Windows Task Bar where the clock is usually displayed). This function also instructs Console Tools **1)** to use the icon in place of a normal Windows Task Bar button whenever the application is minimized, **2)** which mouse action the icon should detect, and **3)** which window state to select (such as `%MAXIMIZE`) when that action is detected.

Availability

Console Tools Pro Only (see)

Warnings

None.

Syntax

```
lResult& = ConsoleSysTrayIcon(lIconID&, _
                              sTooltip$, _
                              lDetect&, _
                              lAction)
```

Parameters

lIconID&

Either **1)** The value `%IDI_CONSOLE` for the Console Tools Icon, or **2)** the value `%IDI_CUSTOM` for an icon that was previously loaded with the CustomIcon function, or **3)** one of the values `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, or `%IDI_WINLOGO` to specify a Windows Standard Icon, or **4)** the ID Number of an icon in a resource file that was linked into your PB/CC program with the `$RESOURCE` metastatement, or **5)** the value zero (0), which *removes* the application's icon (if any) from the System Tray. If you use an invalid value for this parameter, Console Tools will substitute `%IDI_CONSOLE`. See Using Icons for more information.

sToolTip$

If you use a string such as "Click Here for MyProgram", the string that you specify will be briefly displayed as a Windows ToolTip whenever the mouse cursor hovers over the application's System Tray Icon. The longest string that Windows can display is 64 characters. If you use an empty string for this parameter, no ToolTip will be displayed.

lDetect&

This parameter specifies the mouse button event that the icon will detect. Use a value of one (1) if you want the icon to detect left single clicks, or two (2) if you want it to detect left double clicks, or three (3) if you want it to detect right single-clicks. If you use an illegal value for this parameter, Console Tools will substitute 2.

lAction&

This parameter specifies the console window state that will be selected whenever the mouse event that is specified by *lDetect&* is detected. You may use the following window state values: `%UNMINIMIZE`, `%SHOWNORMAL`, `%MAXIMIZE`, or `%RESTORE`. You may also use `%SHOW`, which has the effect of displaying a normal Task Bar button without changing the console window's state. If you use an illegal value for this parameter, Console Tools will substitute `%UNMINIMIZE`.

Return Value

This function will always return `%SUCCESS` (zero).

Remarks

During the time that a System Tray icon is being displayed, if your program is minimized (by the user or programmatically) it will appear to minimize normally, but then the Task Bar button will disappear. Instead of clicking on a Task Bar button, your program's user will be required to click on the System Tray icon.

It is not possible for a user to press Alt-Tab (the Windows task-switching hot key) to select an application that has been minimized to the System Tray. Except for the System Tray icon and the Windows Task Manager, the application is completely hidden.

Minimizing an application to the System Tray has no effect on your program's ability to change its own window state by using the ConsoleWindow function.

If an error (or other important event) occurs in your program while it is minimized to the System Tray, a useful technique is to change the System Tray icon twice per second, to attract the user's attention. For example, you might embed the bright red `CONSOLEC.ICO` icon in your program, and alternate between that and the black `%IDI_CONSOLE` icon twice per second. This would result in a "blinking" icon in the System Tray.

Keep in mind that the console window's state will be affected *whenever* the specified mouse action is detected, even if the console is not currently minimized. This is useful if you want the System Tray icon to always `%MAXIMIZE` the console (for example), regardless of its current state.

If you use the Close (x) button or the Windows Task Manager to close or "end" a program while a System Tray icon is being displayed, the icon may not disappear right away. It will usually disappear the first time the mouse cursor is moved over the System Tray.

Examples

```
'Display the Console Tools icon in the System Tray,
'and %UNMINIMIZE the application whenever the icon
'is double-left-clicked...

ConsoleSysTrayIcon %IDI_CONSOLE, _
                   "Double Click Here", _
                    2, _
                    %UNMINIMIZE
```

See Also

ConsoleWindow, Console Window States

ConsoleTitle

Purpose

Changes the text that is displayed in the console window Title Line, or returns the text that is currently being displayed.

Availability

Console Tools Standard and Pro

Warnings

None.

<u>Setting the title</u>

Syntax

```
ConsoleTitle sNewTitle$
```

Parameters

sNewTitle$ is a string that contains the text that is to be displayed in the console window title bar.

Return Value

None. (If you check the return value, it will always be the string specified by sNewTitle$, even if the function was not successful.)

<u>Reading the current title</u>

Syntax

```
sCurrentTitle$ = ConsoleTitle("")
```

Parameters

None; must be an empty string.

Return Value

sCurrentTitle$ is the return value of the function, which will contain the text from the current console window title bar.

Remarks

Windows places certain limits on the text that can be displayed in a Window Title Bar.

If a string contains `CHR$(0)` Windows interprets that as an end-of-string marker.

Windows 95/98/ME is limited to title bar text that is a maximum of 127 characters long, and Windows NT/2000/XP is limited to 1023 characters. Strings that are longer than these limits will be truncated by the ConsoleTitle function. (If it didn't truncate them, Windows would reject the strings and the title would not be changed.)

Certain characters may be displayed "incorrectly", depending on the user's computer's System Font setting. For example, control characters (ASCII 1-31) and high-bit characters (ASCII 128-255) are displayed as small rectangles on most systems.

Examples

```
'set console title to "My Program"
ConsoleTitle "My Program"

'another way of doing it
sString$ = "My Program"
ConsoleTitle sString$

'print the current console title on the screen
PRINT ConsoleTitle("")

'Set the console title and make
'sure that Windows accepted it...
sString$ = "My Program"
ConsoleTitle sString$
'give Windows time to update the title bar...
DELAY 0.1
IF ConsoleTitle("") <> sString$ THEN
      'Windows rejected all or part of
      'the string for some reason.
      'Does it contain CHR$(0) or was it
      'longer than Windows allows?
END IF
```

Details

If you use the AlreadyRunning function, you should read the notes in the Remarks section of that function before deciding how to use ConsoleTitle.

See Also

AlreadyRunning, ConsoleIcon

ConsoleToForeground

Purpose

Places the console window in the Windows Foreground.

Availability

Console Tools Standard and Pro

Warning

This function's `%SOFT` option performs differently under Windows 95 and Windows NT than it does under Windows 98, Windows ME, Windows 2000, Windows XP, and later versions of Windows. See **Remarks** below.

Warning

It is considered "impolite" --- and possibly dangerous -- for a program to "steal" the foreground from another program if a user might be entering data into the other program. See **Remarks** below.

Syntax

```
lResult& = ConsoleToForeground(lType&)
```

Parameters

lType&

Use one of the predefined equates `%HARD` or `%SOFT` to specify the type of foreground-switching that you want the function to perform. See **Remarks** below.

Return Value

If the console window is *already* in the foreground when this function is used, then lResult& will be `%SUCCESS` (zero).

If the lType& parameter is `%HARD`, then lResult& will always be `%SUCCESS` (zero). If you use `%HARD` it is therefore safe to ignore the return value of the function, so you can use the alternate syntax...

```
ConsoleToForeground %HARD
```

If the lType& parameter is `%SOFT` then lResult& will be `%SUCCESS` (zero) if Windows allows the console window to be switched into the foreground. If Windows does not allow the switch, then lResult& will be the Windows Error Code that caused the failure. Under certain circumstances Windows will not provide an error code, so `%ERROR_CT_UNKNOWNERROR` can also be returned.

Remarks

This function tells Windows **1)** to bring the console window to the top of the "Z-Order" (i.e. to place it on top of all other normal windows), **2)** to direct all keyboard and mouse input (the "keyboard focus") to the console window, and **3)** to change various visual clues to indicate to the user that the console window is the active window (such as changing the color of the console title bar).

IMPORTANT NOTE: It is considered "impolite" -- and possibly dangerous -- for a program to "steal" the foreground and the keyboard focus from another program if a user might be entering data into the other program. Keystrokes and mouse-clicks that were intended for the other program might inadvertently be directed to your program. The unrestrained use of this function to interrupt the user's work --

especially when it isn't genuinely necessary -- can lead to high levels of user frustration. *Use this function with restraint*.

The ConsoleToForeground function can use two different methods to perform the foreground switch: `%SOFT` and `%HARD`.

The `%SOFT` method relies on a Windows function called "SetForegroundWindow". Under Windows 95 and Windows NT this is virtually 100% reliable, but *Microsoft intentionally changed the way that the SetForegoundWindow function works under Windows 98, Windows 2000, and all future versions of Windows*. Under Windows 98. ME, 2000, and XP, if you use the `%SOFT` (SetForegroundWindow) method when the console window does not *already* have the foreground, Windows will "blink" the program's task bar button to attract the user's attention, but *the console window will not be switched into the foreground*. (Microsoft felt that programmers were "abusing" the SetForegroundWindow function, so they effectively disabled it.) If the task bar is not visible for some reason (such as the "auto-hide mode", or a TOPMOST, fullscreen or game program that covers it up) the blinking will not be seen.

The `%HARD` method uses a proprietary technique to bypass the limitations of Windows NT/2000/XP. The ConsoleToForeground function is capable of *forcing* the console window into the foreground, if that is what your program requires.

Please note that certain Console Tools functions use the `%HARD` method internally. For example, if you want to use the ConsoleKey function to send a "virtual keystroke" to the console window, it would be impossible for Console Tools to perform that operation without first switching the console window into the foreground, thereby giving the console window the keyboard focus, so that it can receive the keystrokes. All Console Tools functions which use the `%HARD` method contain Warning messages in their Help File entries. See ConsoleKey, ToggleFullScreenMode, and the ConsoleWindow function when the `%FULLSCREEN` and `%WINDOW` options are used. The ConsoleNormal function also uses the `%HARD` method, and you should keep in mind that if you use the *lNormalize&* options that are available with the ConsoleMessageBox, ConsoleInputBox, SplashBoxShow, and ProgressBoxShow functions, they all use ConsoleNormal (and thereby use the `%HARD` method).

If you are concerned about such things, rest assured that the proprietary `%HARD` technique that the ConsoleToForeground function uses is very "clean". Unlike other techniques, it does not affect the Windows Registry, global Windows runtime settings, or anything else that could potentially affect other programs. It does not use any undocumented API functions, and it does not require console tools to "attach" itself to the program that currently has the foreground. *It will not affect any other program in any way*, other than taking the foreground from them.

Keep in mind that your program can *lose* the foreground and keyboard focus at any time -- even a split second after you have used the ConsoleToForeground function -- if somebody uses the mouse or a keyboard hot-key like Alt-Tab to give the focus to another program. If it is absolutely critical that your program take and *keep* the foreground, you should use the ConsoleToForeground function in a loop. (If you decide to do this, you must consider the possibility that another program might be trying to do the same thing at the same time, resulting in a deadlock where neither application can keep the foreground.)

One final note: the fact that the console window is in the foreground does not *necessarily* mean that it is visible to the user. See ConsoleTopMost for more details, and a solution to this (potential) problem.

See Also

ConsoleToTop, ConsoleTopMost, ConsoleIsForeground, ConsoleFocus

ConsoleToolBar

Purpose

Hides or shows the Windows 95/98/ME Console Toolbar.

Availability

Console Tools Standard and Pro

Warning

Attempting to use this function after the DeleteWindowMenuItem function has been used to remove the "Toolbar" menu item will produce unpredictable results, possibly including Windows Application Errors (General Protection Faults).

Syntax

```
lResult& = ConsoleToolbar(lOnOrOff&, _
                          lState&)
```

Parameters

lOnOrOff&

Use zero or the predefined equate `%OFF` to hide the Windows 95/98/ME Console Toolbar. Use a nonzero value or `%ON` to make it visible.

lState&

At the same time that it changes the status of the Windows 95/98/ME Console Toolbar, the ConsoleToolBar function can *optionally* hide or show the console window. If you do not want the ConsoleToolBar function to change the console window's state, use `%NO_CHANGE` for this parameter. Otherwise, the most common values for this parameter are `%SHOW` and `%HIDE` (see examples below) but it is possible to use any of the Window State equates that the ConsoleWindow function recognizes.

Return Value

lResult& will be one of the following values...

`%SUCCESS` if the Toolbar hide/show is successful, or if the toolbar was already in the desired state.

`%ERROR_CT_CANTBEDONE` if an attempt is made to show the toolbar on a Windows NT/2000/XP computer. Windows NT, 2000, and XP do not support Console Toolbars. (If you use the ConsoleToolbar function to *hide* the toolbar on an NT/2000/XP computer, the function will return `%SUCCESS` because the toolbar was already in the desired state.)

`%ERROR_CT_UNKNOWNERROR` if Console Tools attempts to change the toolbar state but is not successful. Possible errors include the previous removal of the Toolbar item from the Window Menu (see Warning above)

Remarks

We recommend that if your program uses this function to hide the Console Toolbar, that you then use the DeleteWindowMenuItem function to remove `%MENUITEM_TOOLBAR` from the Window Menu, so that your program's users can't manually re-activate the toolbar. But remember that you must hide the toolbar and *then* remove the item from the menu. Reversing those steps can result in Windows Application Errors.

Also keep in mind that Windows NT, 2000, and XP do not support the Console Toolbar, so if you want your programs to have a consistent look when they are run on different computers, you should probably use the ConsoleToolBar function to hide the Windows 95/98/ME toolbar.

Examples

```
'Turn the toolbar off without affecting the window state
IF ConsoleToolbar(%OFF, %NO_CHANGE) = %SUCCESS THEN
      'it worked
END IF

'Turn the toolbar on, and hide the resulting window
IF ConsoleToolbar(%ON, %HIDE) = %SUCCESS THEN
      'it worked
END IF

'Turn the toolbar on and display a message
'if it doesn't work...
IF ConsoleToolbar(%ON,%SHOW) = %SUCCESS THEN
      PRINT "THE TOOLBAR IS NOW VISIBLE."
ELSE
      IF WindowsVersion(%WIN_PLATFORM) = %PLATFORM_WIN_NT THEN
            PRINT "WINDOWS NT HAS NO TOOLBAR."
      ELSE
      PRINT "UNABLE TO ACTIVATE TOOLBAR."
      END IF
END IF
```

See Also

WindowsVersion, DeleteWindowMenuItem, ToggleConsoleToolbar

ConsoleToolsSerialNumber

Purpose

Returns a number that corresponds to the serial number that is embedded in the currently-loaded Console Tools DLL. (This is NOT the same as the Console Tools Authorization Code.)

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleToolsSerialNumber
```

Parameters

None.

Return Value

lResult& will be a positive Long Integer. Each copy of the Console Tools DLL that is provided to a developer has a unique serial number. In most cases, the serial number that is embedded in a DLL is based on the date that the DLL was originally produced for the developer.

Remarks

This use of this function should only be necessary if you lose your Console Tools Serial Number. (Perfect Sync Technical Support requires that a valid serial number be included with all requests for support.)

It is not usually practical to use this function in your programs (to make sure that the DLL that you supplied to a user is still in use, for example) because *your serial number may change in the future*. If Perfect Sync sends you an updated DLL it will have a different serial number.

Developers with special needs can request that a range of serial numbers be reserved for their use, and custom-serialized DLLs can be provided on request.

Please note that tampering with the serial number that is embedded in a Console Tools DLL violates the terms of your software license agreement and can cause unpredictable behavior.

Examples

```
PRINT ConsoleToolsSerialNumber
```

See Also

ConsoleToolsVersion

ConsoleToolsAuthorize

Summary

Tells Console Tools that it is authorized to begin operating.

Availability

Console Tools Standard and Pro

Warning

Every program that uses Console Tools *must* use this function *before any other Console Tools functions are used*, including `InitConsoleTools`.

The incorrect use of this function will cause your programs to malfunction. See **Remarks** below for more information.

Syntax

```
lResult& = ConsoleToolsAuthorize(%MY_CT_AUTHCODE)
```

(See **Example** below for recommended use.)

Parameters

`%MY_CT_AUTHCODE`

An equate that has been defined in the Console Tools INC file as the Authorization Code that was provided with your Console Tools Runtime Files. Every Console Tools licensee is provided with a unique Authorization Code in the form of an eight-digit hexadecimal number. See **Example** below.

Return Values

This function will return `%ERROR_CT_CANTBEDONE` if an invalid Authorization Code is used for `%MY_CT_AUTHCODE`.

A return value of `SUCCESS` (zero) indicates that *either* the correct Authorization Code *or* a "Dummy Code" was accepted by the function.

A return value of negative one (-1) indicates that your program has called the ConsoleToolsAuthorize function a second time. It is only necessary to call it once.

See **Remarks** below for more information.

Remarks

Every Console Tools Runtime File (each DLL) is "serialized". That means that it contains a unique, embedded key number called an Authorization Code. In order to use Console Tools, you must prove to the Runtime File that you know its correct Authorization Code.

If you don't use the `ConsoleToolsAuthorize` function at all, the `InitConsoleTools` function will refuse to work, making it impossible for your program to use Console Tools in any way.

If you use the `ConsoleToolsAuthorize` function with the Authorization Code that matches your Runtime File -- the exact number that was provided *with* your Runtime Files -- then Console Tools will work normally.

If you use the `ConsoleToolsAuthorize` function with a "Dummy Code", Console Tools will randomly, intentionally malfunction.

See Console Tools Authorization Codes for a complete description of how Authorization Codes and Dummy Codes are used.

Example

```
IF ConsoleToolsAuthorize(%MY_CT_AUTHCODE) <> %SUCCESS THEN
    PRINT "ERROR!  WRONG AUTH CODE!"
    EXIT FUNCTION
END IF
```

See Also

Authorization Codes

ConsoleToolsVersion

Purpose

Returns a number that indicates the version number of the currently-loaded Console Tools DLL.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = ConsoleToolsVersion
```

Parameters

None.

Return Value

lResult& will be a positive number if the Console Tools Pro DLL is in use, or a negative number if the Console Tools Standard DLL is in use. To get the actual version number, take the absolute value of the ConsoleToolsVersion return value. See **Example** below.

Remarks

If your program relies on features that are only available in the Console Tools Pro DLL, or if it relies on features (or bug fixes) in a certain revision level of the DLL, it should check the value of this function when the program is first initialized. If the proper version of the DLL is not installed on the user's computer, your program should display an error message and exit gracefully instead of attempting to use features and/or options that do not exist. We also recommend that your program check the WindowsVersion function at the same time, in case it requires Windows features (such as the Windows 95/98/ME Toolbar) that are not available on all computers.

Please note that versions 1.00 through 1.09 of Console Tools used Hex numbers (base 16) for their version numbers. Starting with version 1.10, this was changed to decimal numbers to reduce confusion and make the function easier to use. Since 100h (hex) equals 256 (decimal) it is relatively easy for programs to recognize version numbers below 1.10.

Examples

```
IF ConsoleToolsVersion < 0 THEN
      PRINT "Console Tools Standard DLL detected."
ELSE
      PRINT "Console Tools Pro DLL detected."
END IF

PRINT ConsoleToolsVersion
'The preceding line would print 110 for version 1.10,
'or 111 for version 1.11, and so on.

PRINT FORMAT$(ConsoleToolsVersion / 100, "#.##")
'The preceding line would print 1.10 for version 1.10.

IF ConsoleToolsVersion =>  256 OR _
   ConsoleToolsVersion <= -256 THEN
      'Console Tools versions 1.00 through 1.09 used
      'hex values for their version numbers.  The number
      '256 corresponds to 100h (hex), so a DLL with
      'a version number less than 1.10 has been detected.
      PRINT "Console Tools version less than 1.10 detected."
END IF
```

See Also

WindowsVersion

ConsoleTopMost

Purpose

Places the console window at the very top of the Windows Z-Order, making it the top-level window *of any kind*. Compare ConsoleToTop.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleTopMost lTopMost&
```

Parameters

lTopMost&

Use a nonzero (True) value to add the TOPMOST property to the console window, or `%FALSE` (zero) to remove it.

Return Value

None.

Remarks

This function changes the console window's TOPMOST status. TOPMOST is a special Windows property that forces a window to stay on top of other windows even if other windows become active. (When a window becomes active it normally covers up other windows.)

Keep in mind that the TOPMOST property only stays in effect until *another* window is given the TOPMOST property. For example, if you activate this Help File's Options/KeepHelpOnTop feature (thereby giving the Help File the TOPMOST property) it will not allow the console window to cover it. If you then use `ConsoleTopMost %TRUE` to give your program the TOPMOST property, it will be displayed on top of the Help File. The Help File will still have the TOPMOST property and will still block all normal windows, but since the console window was given the property *after* the Help File, the console will be able to cover it up. This also means that if you use the ConsoleTopMost function and *then* you activate the KeepHelpOnTop function, the console window will *not* be able to block it. For this reason, you may want to use the ConsoleTopMost function to periodically "refresh" the console's TOPMOST status.

Note also that this function only affects the *display* of the console window. It does not affect whether or not your program is the *active* program, i.e. whether or not it is in the Windows Foreground and has the keyboard focus. If you want to place your program on top of other programs *and* place it in the Windows Foreground, you will need to use the ConsoleToForeground function *in addition to* ConsoleTopMost or ConsoleToTop.

See Also

ConsoleFocus

ConsoleToTop

Purpose

Places the console window above all other *normal* windows in the Windows Z-Order, making it the top-level normal window. Compare ConsoleTopMost.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ConsoleToTop
```

Parameters

None.

Return Value

None.

Remarks

This function changes the Windows Z-Order so that your program's console window is displayed on top of all other *normal* windows, i.e. all other windows that do not have the special Windows TOPMOST property.

The ConsoleToTop function can not be used to place the console window on top of windows that have the TOPMOST property. (For an example of this, activate this Help File's "Options/KeepHelpOnTop" feature, and you'll see that it will not allow the console window to block it.) If you need to make sure that your program is the top-most window of *any* kind, see ConsoleTopMost.

Note also that this function only affects the *display* of the console window. It does not affect whether or not your program is the *active* program, i.e. whether or not it is in the Windows Foreground and has the keyboard focus. If you want to place your program on top of other programs *and* place it in the Windows Foreground, you will need to use the ConsoleToForeground function *in addition to* ConsoleToTop or ConsoleTopMost.

See Also

ConsoleFocus

ConsoleWindow

Purpose

Changes the Window State of the console window.

Availability

Console Tools Standard and Pro

Warning

If the `%FULLSCREEN` or `%WINDOW` option is used, and if the console window is not already in the requested state, in order to perform the screen-mode-switch the ConsoleWindow function will be required to force your program into the windows foreground by using the `ConsoleToForeground %HARD` function. See ConsoleToForeground for details.

Syntax

```
lResult& = ConsoleWindow(lState&)
```

Parameters

lState&

Use one (and only one) of the following predefined equates.

The first group of equates correspond to the `%SW_` equates in the WIN32API.INC file that is supplied with PB/CC, and those definitions correspond to the standard Window States. Unfortunately, the "official" definitions for these Window States have been copyrighted by Microsoft, so you'll need to check their documentation for detailed descriptions and subtle differences. Fortunately, they are mostly self-explanatory. Explanations of basic terms can be found in Appendix B: Console Window States.

```
%HIDE, %SHOWNORMAL, %SHOWMINIMIZED, %SHOWMAXIMIZED,
%MAXIMIZE, %SHOWNOACTIVATE, %SHOW, %MINIMIZE,
%SHOWMINNOACTIVE, %SHOWNA, %RESTORE, %SHOWDEFAULT
```

The equates in the second group represent Console Tools extensions of the standard Window States.

`%UNMINIMIZE` This option only affects console windows that are Minimized. It returns them to their previous state, either Maximized or Restored. (Note that `%RESTORE` and `%MAXIMIZE` force those states upon a window, without regard to its previous state.) See Console Window States for more information.

`%FULLSCREEN` This option puts the console window in the FullScreen mode, as if a user had pressed Alt-Enter while the console was in the Windowed mode. (See Warning above.)

`%WINDOW` This option is the opposite of `%FULLSCREEN`. It takes the console window out of the FullScreen mode. (See Warning above.)

Return Value

lResult& will be `%ERROR_CT_INVALIDPARAMETER` if a value other than one of the predefined equates (above) is used for lState&.

Otherwise, lResult& will be `%SUCCESS` (zero).

If you always use the predefined equates, it is safe to ignore the return value of this function. Note that the predefined ConsoleWindow equates *cannot* be added together (see Remarks below).

Remarks

Keep in mind that a console window can appear to have "two states at once" (see ConsoleState and ConsoleIsFullScreen for more information) and that you may need to perform two or more ConsoleWindow operations to make sure that the screen is in the desired state. It is not possible, however, for the ConsoleWindow function to perform more than one operation *at a time*.

For example, you might want to make sure that your program is not in the fullscreen mode *and* that it is maximized. But attempting to perform both steps at the same time by using...

```
ConsoleWindow %WINDOW + %MAXIMIZE
```

...will *not* work. In fact it, since `%WINDOW+%MAXIMIZE` produces a numeric value of zero (three plus negative three), it would do the same as using `ConsoleWindow %HIDE` because zero corresponds to `%HIDE`. (See the numeric values in the Console Tools INC files for more information.)

If you want to perform two different operations, you must use the ConsoleWindow function twice, like this:

```
ConsoleWindow %WINDOW
ConsoleWindow %MAXIMIZE
```

This may seem inconvenient, but if your program was allowed to use something like `%WINDOW+%MAXIMIZE`, **1)** the ConsoleWindow function would not know which operation to perform first (since the value is passed as a numeric sum) and **2)** if a problem occurred and the function returned an error value, your program would not be able to tell which operation failed.

Examples

```
ConsoleWindow %WINDOW
'The console is now in the window mode (i.e. not
'in the fullscreen mode).

ConsoleWindow %SHOW
'The console window is now visible, even if it was
'hidden by a previous process.

ConsoleWindow %MAXIMIZE
'The console window is now maximized.

ConsoleWindow %MINIMIZE
'The console window is now minimized.

ConsoleWindow %UNMINIMIZE
'The console window is now maximized again.

ConsoleWindow %RESTORE
'The console window is now in the "restored" state.

ConsoleWindow %MINIMIZE
'The console window has been minimized again.

ConsoleWindow %UNMINIMIZE
'This time the %UNMINIMIZE command has a different
'effect, because the window was Restored before it
'was Minimized.  The console window is now in the
'Restored state again.

ConsoleWindow %HIDE
'The console window is now invisible, and it does not
'appear on the task bar.  It can be selected with
'Alt-Tab but doing so has no effect.

ConsoleWindow %SHOW
'The console window is now visible again.

ConsoleWindow %FULLSCREEN
'The console is now in the full screen mode.

ConsoleWindow %MAXIMIZE
'The console is now maximized, but because it is
'still in the full screen mode, it probably looks
'the same as before.  (Results will depend on the
'version of Windows that is being used.)

ConsoleWindow %WINDOW
'The console now appears as a window, but it may or
'may not appear maximized.  (Results will depend on
'the version of Windows that is being used.)

ConsoleWindow %MAXIMIZE
'The console is now maximized.
```

Note: If a user presses Alt-Enter to toggle the FullScreen mode, or if they click on a title bar button like Minimize or Maximize while the example code is running, the console may or may not end up in the desired state. If a certain console state is *required* by your program, it should *periodically* check the console state and perform the necessary ConsoleWindow operations.

See Also

ConsoleState, Appendix B: Console Window States

CustomFont

Purpose

Creates a Console Tools Custom Font for use with Splash Boxes and Input Boxes.

Availability

Console Tools Pro Only (see)

Warning

This function is not available in the Console Tools Standard DLL.

Syntax

```
lResult& = CustomFont(sFontName$, _
                      lWeight&, _
                      lHeight&, _
                      lWidth&, _
                      lItalic&)
```

Parameters

sFontName$

A string containing the exact, registered, case-insensitive name of a font that exists on the computer at runtime. IMPORTANT NOTE: You will obtain the best results from CustomFont (by far!) if you specify a TrueType font. Safe values *usually* include "Arial", "Arial Black", "Courier New", and "Times New Roman". Standard non-TrueType fonts *usually* include "Courier", "FixedSys", "MS Sans Serif", "MS Serif", "Roman", and "System". These are all standard Windows Fonts, but we emphasize the word *usually* here because there is no guarantee that they have not been deleted from your user's system. If you want to make absolutely sure that a particular font is used, you should secure the proper copyright and distribution rights, and distribute the font with your application.

lWeight&

A number from 0 to 9 (or a multiple of 100 between100 and 900) that affects the **BOLDNESS** of the created font. NOTE: Windows only allows certain weights to be used for non-TrueType fonts. For example, when creating a font using the non-TrueType "MS Sans Serif" font you may find that using the weights from 0-5 produce one boldness, and 6-9 produce a somewhat bolder look.

lHeight&

If the value of lHeight& is zero, Windows uses the default height for the font. If the value is greater than zero, Windows transforms the value into "device units" and matches it against the cell height of the available fonts. If the value is less than zero, Windows transforms this value into "device units" and matches its absolute value against the character height of the available fonts. When you're trying to produce a certain-sized font, it's often helpful to try various positive and negative values. NOTE: The lHeight& value is not the "point size" of the font. A font with a height of 20 will not be the same size as a 20-point font.   NOTE: Windows only allows certain values for lHeight& when you use non-TrueType fonts. For example, if you use the non-TrueType font called "MS Sans Serif" you will find that the range of values from 1-12 produces the same height lettering.

lWidth&

If the value of lWidth& is zero, Windows uses the font's own "aspect ratio" and the font's height value to determine the width of the characters. If the lWidth& value is greater than zero, it specifies the width of the characters in "logical units". NOTE: Windows only allows certain lWidth& values for non-TrueType fonts. For example, you will find that certain ranges of width values have no effect on the appearance of the MS Sans Serif font.

lItalic&

Use `%FALSE` (zero) for a NONITALIC font or a nonzero value for an *ITALIC* one.

Return Value

lResult& will be `%SUCCESS` if the function creates the new font as requested.

lResult& will be `%ERROR_CT_FEATURENOTAVAILABLE` if you attempt to use this function when the Console Tools Pro DLL is not installed.

lResult& will be `%ERROR_CT_FONTINUSE` if the font cannot be created because a custom font is currently in use. For example, if you use the CustomFont function to create a font and then use it in a Splash Box, you will not be able to create a new custom font until you have stopped using the font, i.e. until you have used the SplashBoxHide function.

lResult& will be `%ERROR_CT_UNKNOWNERROR` if Windows refuses to create the new font for some reason. Windows does not provide an Error Code if this function fails, so no other information about the failure is available.

Remarks

Please note that the Custom Font can not be used to change the font of the console window itself. This is a limitation of the Microsoft Console Window.

Examples

```
CustomFont "Ariel", 9, 20, 0, 1
```

This example would produce an Arial font (a common TrueType font) that was very bold (9 on a scale of 0 to 9), and 20 units high, normal (auto) width, and italic. Here's a sample of what it would look like:

Console Tools Custom Font

See Also

SplashBoxShow, ConsoleInputBox, hCustomFont

CustomIcon

Purpose

Loads an icon from a disk file, for use with various Console Tools functions.

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = CustomIcon(sFileName$)
```

Parameters

sFileName$

The file name, with optional drive and path, of a valid icon (`*.ICO`) file.

Return Value

lResult& will be `%SUCCESS` if the function loads the requested icon file without errors.

lResult& will be `%ERROR_CT_UNKNOWNERROR` if Windows cannot load the icon but does not give a reason.

Otherwise, lResult& will be a Windows Error Code. See the `%ERROR_` equates in the Win32API.INC file that is supplied with PowerBASIC. The most common of these is probably `%ERROR_FILE_NOT_FOUND`.

Remarks

After an icon has been loaded with this function, *most* other Console Tools functions that can use icons can display the icon by using `%IDI_CUSTOM`. The most notable exception is the ConsoleMessageBox function which, because of a limitation of Microsoft Windows, cannot use file-based icons.

Examples

See ConsoleIcon.

See Also

Using Icons

DefaultWindowMenu

Purpose

Resets the console window's Window Menu to the default configuration.

Availability

Console Tools Standard and Pro

Warning

The default configuration of the Window Menu does not have the **Edit**, **Properties**, and (on Win95/98/ME only) **Toolbar** items that are normally associated with a console window's Window Menu. This function resets the Window Menu to the *Windows Default* Window Menu, not to the *Console Window* default configuration.

Syntax

```
DefaultWindowMenu
```

Parameters

None

Return Value

None.

Remarks

This function can be used **1)** to remove the Edit, Properties, and Toolbar items from a program's Window Menu in a single step, or **2)** to restore other items that may have been removed from the Window Menu by the DeleteWindowMenuItem function.

See Also

DeleteWindowMenuItem, The Console Window Menu

Delay

Purpose

Causes your program to pause for the specified number of seconds or fractions of seconds.

Availability

Console Tools Standard and Pro

Warning

Because of the way Windows performs task-sharing, and because of the wide variety of CPU speeds that your program may run on, the Delay function can only guarantee that it will pause for *at least* the specified number of seconds. It is usually fairly accurate, but it should not be used for critical timing operations.

Syntax

```
Delay epAmount##
```

Parameters

epAmount##

An Extended-Precision floating point number that specifies the length of time that the function should pause, in seconds. IMPORTANT NOTE: The fact that this is an extended-precision number is <u>not</u> meant to imply that the function is that accurate. The Console Tools Delay function uses extended-precision numbers so that it can be as accurate as possible, but Windows usually interferes with that precision.

Return Value

None.

Remarks

See the warnings above regarding the accuracy of the Delay function.

Note also that the PB/CC compiler does not allow fractional values to be used without a leading zero. If you use the following line in your program...

```
Delay .5
```

...PB/CC will generate a "**Period not allowed**" error when you try to compile your program. This does *not* mean that the Delay function does not accept fractional values, it simply means that the PB/CC compiler requires a leading zero on the number. To specify a fractional delay less than one second you must use one of the following techniques...

```
Delay 0.5
```

<u>or</u>

```
CALL Delay(.5)
```

Details

The Console Tools Delay function is based on the TIMER function, but it is "smart" about midnight. Some timing routines fail (usually exiting early or locking up) if the requested delay starts before midnight and ends after midnight, but the Console Tools Delay function compensates properly for the TIMER midnight-reset to zero.

Delay accepts values up to 86,399 seconds, which is the number of seconds in one day, minus one.

Using `Delay 0` (zero) is the same as using the Windows API function `SLEEP 1,` which is often used to "release time slices" during certain operations. For example, if your PB/CC program contains the code...

```
WHILE NOT INSTAT
WEND
```

...it will place a significant load on the computer's CPU. If you use this instead...

```
WHILE NOT INSTAT
      DELAY 0
WEND
```

...the CPU load will be greatly reduced, if not eliminated.

DeleteWindowMenuItem

Purpose

Removes items from the Console Window Menu (*not* from a Console Tools Pulldown Menu).

Availability

Console Tools Standard and Pro

Warning

Once the **Edit**, **Properties**, and (on Win95/98/ME only) **Toolbar** items have been removed, a complex operation using the Windows API and the hConsoleWindowMenu function is required to restore them (and to restore their submenus). *The Console Tools DLL does not directly support the restoration of these items once they have been deleted from the Window Menu.*

Syntax

```
lResult& = DeleteWindowMenuItem(lItemID&)
```

Parameters

lItemID&

One of the following predefined equates: `%MENUITEM_SIZE`, `%MENUITEM_MOVE`, `%MENUITEM_MINIMIZE`, `%MENUITEM_MAXIMIZE`, `%MENUITEM_CLOSE`, `%MENUITEM_RESTORE`, `%MENUITEM_TOOLBAR`, `%MENUITEM_PROPERTIES`, `%MENUITEM_EDIT`, `%MENUITEM_DEFAULTS`, or `%MENUITEM_SEPARATOR`. Most of these equates correspond to `%SC_` values in the WIN32API.INC file. (The other equates represent Console Tools extensions that rely on values that are not documented by Microsoft.)

Return Value

lResult& will be `%SUCCESS` if the removal of the menu item was successful.

If Windows fails to remove the menu item, lResult& will *usually* contain a Windows Error Number that explains the failure.

Under certain circumstances, Windows will fail to perform the requested operation but it will not provide a Windows Error Number. In that case, lResult& will be `%ERROR_CT_UNKNOWNERROR`.

Remarks

The removal of certain menu items also deactivates the corresponding Windows Hot Keys and other related functions. For instance, if you use the example code shown below to remove the `%MENUITEM_CLOSE` item, **1)** the Close item will be removed from the Window Menu, **2)** the Close button (**x**) will no longer work (although in some cases it will not be visually changed to the "inactive" state right away), **3)** the Alt-F4 hotkey will no longer close the program, and **4)** double-clicking the console icon will no longer close the program.

Similarly, if you use this function to remove the `%MENUITEM_TOOLBAR` item, your user's ability *and your program's ability* to change the state of the Windows 95/98/ME toolbar (via the ConsoleToolbar and ToggleConsoleToolbar functions) will be disabled. You must make sure that you have programmatically set the *permanent condition* of an item like the toolbar before removing the Window Menu item that controls it.

Removal of the Window State functions (Maximize/Minimize/Restore) from the Window Menu does *not* affect your program's ability to use the ConsoleState function.

If your program needs to remove one of the standard Window Menu items (Restore, Move, Size, Minimize, Maximize, or Close) and then re-activate it later, you can use the DefaultWindowMenu item to restore the Window Menu to the Windows Default configuration. Please note that the default configuration does not include the Edit, Properties, or Toolbar items.

The DefaultWindowMenu function can also be used to remove the Edit, Properties, and Toolbar items in a single step.

Example

```
DeleteWindowMenuItem %MENIITEM_CLOSE
```

See Also

Microsoft Console Windows, DefaultWindowMenu, The Console Window Menu

GfxTextHole (Console Tools Plus Graphics ONLY)

Purpose

Creates a transparent area (a "hole") in the Graphics Tools graphics window, through which the console window can be seen.

Availability

Console Tools Standard and Pro

Warning

You must use an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function before you can use this function.

Syntax

```
lResult& = GfxTextHole(lNumber&, _
                       lLeft&, _
                       lTop&, _
                       lRight&, _
                       lBottom&)
```

Parameters

lNumber&

The number of the hole that is to be created, from one (1) to thirty-two (32), or to the number that was specified with GfxOption `%GFX_MAX_HOLES`. See the Graphics Tools documentation for more information about the GfxOption function.

lLeft& and *lTop&*

The column and row that define the top-left corner of the rectangular hole that you want to create.

lRight& and *lBottom&*

The column and row that define the bottom-right corner of the hole.

Return Value

The value of `lResult&` will be `%SUCCESS` (zero) if the hole is created without errors, or...

If you fail to specify an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function *before* you use this function, this function will return `%ERROR_CT_CANTBEDONE`. (The `_CT_` stands for Console Tools.)

`%ERROR_GT_CANTBEDONE` (the `_GT_` stands for Graphics Tools) will be returned if hole number *lNumber&* has already been created. (Use the Graphics Tools FillHole function to free up the number before re-using it.) Or...

`%ERROR_GT_INVALIDPARAMETER` will be returned if **1)** an invalid *lNumber&* value is used, or **2)** a row or column value that is less than one (1) is used.

Remarks

This function creates a "hole" in the graphics window, through which the console window will be visible. Each hole is assigned an arbitrary number, so that it can be easily deleted ("filled") later, if necessary.

Please note that the parameters of this function are "backwards" from the usual row/column parameters of PB/CC functions such as `LOCATE`. Because this is a *graphics* function, it uses the Windows graphics convention of column/row not row/column.

This function is virtually identical to the Graphics Tools DrawHole function, except that it allows the use of column/row values instead of pixel values. Please refer to the Graphics Tools documentation for more information about creating holes in the graphics window.

Examples

```
'Create Hole #1 as a two-row, 12 column hole
'with its top-left corner at row 20, column 3
'(i.e. rows 20-21 and columns 3-14)...

GfxTextHole  1, 3, 20, 14, 21
```

See Also

Please refer to the Graphics Tools documentation for more information.

hConsoleWindow

Purpose

Returns the value of the Windows Handle of the console window.

Availability

Console Tools Standard and Pro

Warning

The incorrect use of the value that this function returns can cause serious problems, including Application Errors (General Protection Faults). Use it with caution.

Syntax

```
lResult& = hConsoleWindow
```

Parameters

None.

Return Value

lResult& will be a Long Integer containing the Windows Handle of the console window that is owned by the current program.

Remarks

You'll only need this function if you want to use the Windows API to perform functions that Console Tools does not provide. (Console Tools provides only *some* of the functions that require the use of the Console Window Handle.)

Note that the Console Window Handle is *not* the same as the GETSTDIN, GETSTDOUT, or GETSTDERR handle values that are provided by PB/CC. Those Windows Handles are used for other purposes.

A PB/CC Long Integer is usually used for this value so that the standard Windows value for `%INVALID_HANDLE_VALUE` (-1) can be used. A DWORD variable can also be used (with the appropriate bit conversion) but DWORDs cannot hold negative values.

See Also

hConsoleWindowMenu

hConsoleWindowMenu

Purpose

Returns the Windows Handle of the Window Menu (formerly known as the System Menu) that is owned by the console window that is owned by the current program.

Availability

Console Tools Standard and Pro

Warning

The incorrect use of the value that this function returns can cause serious problems, including Application Errors (General Protection Faults). Use it with caution.

Syntax

```
lResult& = hConsoleWindowMenu
```

Parameters

None.

Return Value

lResult& will be a Long Integer containing the Windows Handle to the Window Menu that is owned by the current program's console window.

Remarks

You will only need this function if you want to use the Windows API to perform functions that Console Tools does not provide. (Console Tools provides *most* of the functions that require the use of the Console Window Menu Handle.)

Note that the Console Window Menu Handle is *not* the same as the GETSTDIN, GETSTDOUT, or GETSTDERR handle values that are provided by PB/CC. Those Windows Handles are used for other purposes.

A PB/CC Long Integer is usually used for this value so that the standard Windows value for `%INVALID_HANDLE_VALUE` (-1) can be used. A DWORD variable can also be used (with the appropriate bit conversion) but DWORDs cannot hold negative values.

See The Console Window Menu for more information.

See Also

hConsoleWindow

hCustomFont

Purpose

Returns the Windows Handle of the Custom Font that was created by the CustomFont function.

Availability

Console Tools Pro Only (see)

Warning

The incorrect use of the value that this function returns can cause serious problems, including Application Errors (General Protection Faults). Use it with caution.

Syntax

```
lResult& = hCustomFont
```

Parameters

None.

Return Value

lResult& will be a Long Integer containing the Windows Handle to the Custom Font that was created by a previous call to the CustomFont function. If the CustomFont function has not yet been used, or if it failed when it was used, a handle to the fallback font, which is based on the standard Windows "Fixed Sys" font, will be returned.

Remarks

The CustomFont function creates fonts that are perfectly normal Windows fonts, and they can be used by virtually any window *graphics* function. (Unfortunately they can't be used by *text* functions like the console window itself.) If you are an experienced Windows programmer, the Font Handle can be very useful.

You will only need this function if you want to use the Windows API to perform functions that Console Tools does not provide.

A PB/CC Long Integer is usually used for this value so that the standard Windows value for `%INVALID_HANDLE_VALUE` (-1) can be used. A DWORD variable can also be used (with the appropriate bit conversion) but DWORDs cannot hold negative values.

See Also

CustomFont

InitConsoleTools

Purpose

Initializes the Console Tools DLL.

Availability

Console Tools Standard and Pro

Warning

Every program that uses the Console Tools DLL *must* call this function once, and only once, at the very beginning of the program, immediately after ConsoleToolsAuthorize. Failure to do so will cause most Console Tools functions to fail, and Application Errors are possible. See Three Critical Steps for Every Program.

Warning

Failure to use *correct* values for the `lMenuBuffers&` and `lScreenBuffers&` parameters will usually result in serious program malfunctions.

Syntax

```
lResult& = InitConsoleTools(hExeInstance&, _
                            lMenuBuffers&, _
                            lScreenBuffers&, _
                            lGraphics&, _
                            lReserved2&, _
                            lReserved3&)
```

Parameters

hExeInstance&

This value should normally be zero. (If you want Console Tools to use resources -- icons, bitmaps, etc. -- that are embedded in a module *other than* your PB/CC program, such as a DLL, you can use the instance handle of that module for this parameter.)

lMenuBuffers&

If your program uses the Pulldown Menu functions (which are only available in the Console Tools Pro DLL) you must tell Console Tools how many Menu Buffers to create. Using a number that is too small (or zero) will result in Menu Buffers that do not work. Using a number that is too large will result in a wasted memory. Console Tools Standard DLL must use zero (0) for this parameter. Console Tools Pro DLL users may use values from 0 to 1024. (If each unused Menu Buffer requires a theoretical average of 16 bytes, and if your program uses 1024 for this parameter but does not actually use any Menu Buffers, your program will waste 16k of memory.)

lScreenBuffers&

If your program uses Console Tools Screen Buffers to save and restore screens, you must tell the Console Tools DLL how many Screen Buffers to create. Using a number that is too small (or zero) will result in Screen Buffers that do not work. Using a number that is too large will result in a small amount of wasted memory (about 8 bytes per buffer). Console Tools Standard DLL users may use values between 0 and 7, Console Tools Pro DLL users may use 0 to 255.

lGraphics&

Tells Console Tools whether or not you are using Perfect Sync's Graphics

Tools package at the same time as Console Tools. If you are NOT using Graphics Tools, use zero (0) for this parameter. If you are using Graphics Tools Version 1 (GfxTools.DLL) use one (1) for this parameter. If you are using Graphics Tools Version 2 STANDARD (GfxT_Std.DLL) use two (2) for this parameter. If you are using Graphics Tools Version 2 PRO (GfxT_Pro.DLL) use three (3) for this parameter. You can also use the value negative one (-1) for this parameter, to tell Console Tools to *search* for Graphics Tools, but this option is slightly slower than using a specific number. The example programs in this document use negative one so that they can be used with any version of Graphics Tools, but we recommend that you use a hard-coded numeric value between 1 and 3.

lReserved_&

These three variables are reserved for possible future additions to the Console Tools DLL. They are provided so that every program that uses Console Tools will include the same number of InitConsoleTools parameters, regardless of when it was written. Then, if future Console Tools DLLs require the use of additional initialization parameters, old programs will pass the default value of zero (0) to the new functions.

Return Value

lResult& will be one of the following values...

`%ERROR_CT_INVALIDMENUNUMBER` if an invalid lMenuBuffers& value is used.

`%ERROR_CT_INVALIDSCREENNUMBER` if an invalid lScreenBuffers& value is used.

`%ERROR_CT_CANTBEDONE` if the InitConsoleTools function is used more than once in a program. It must be used once and only once.

`%ERROR_CT_NOCONSOLE` if the InitConsoleTools function is used by a program that does not have a console window. (See InitCTInternal for more information.)

`%ERROR_CT_UNKNOWNERROR` if the InitConsoleTools function fails internally, usually due to a Windows memory problem.

`%SUCCESS` if none of the errors above were detected.

Remarks

The InitConsoleTools function must be used *once and only once* in every program that uses the Console Tools DLL. In most programs -- including all *native* console applications such as those produced by PB/CC -- the InitConsoleTools function should be used at the very beginning of the WinMain function. Please read Three Critical Steps for Every Program for more information.

If your program is a *non-native* console application (i.e. a program that uses the Windows "AllocConsole" API function to create its own console window) then you should use the InitConsoleTools function immediately *after* the console window has been created. (If you are writing non-native console applications, also see InitCTInternals.)

Example

This example would initialize Console Tools with 128 Menu Buffers (numbered 1-128) and 33 Screen Buffers (numbered 0-32).

Note that these example values will cause the InitConsoleTools function to fail if the Console Tools **Standard** DLL is being used. You must use values that are compatible with the version of Console Tools that you are using.

```
'This line should already be in your program...
FUNCTION PBMain PRIVATEAS LONG

'Then, at the very start of the PBMain or WinMain function...
'Add your Authorization Code here...
ConsoleToolsAuthorize %MY_CT_AUTHCODE
InitConsoleTools 0, 128, 32, 0, 0, 0
```

Note that the first parameter of WinMain and the first parameter of InitConsoleTools must have the same variable name.

See Also

Three Critical Steps for Every Program

InitCTInternals

Purpose

Re-initializes certain internal functions when Console Tools is used with *non-native* console applications, i.e. programs that create their own consoles by using the Windows API. This function is never used with *native* console applications, such as those produced by PB/CC.

Availability

Console Tools Standard and Pro

Warning

Do not attempt to use this function unless your *non-native* console application uses the Windows API "AllocConsole" function to create its own console window. (Most console applications are *native* console applications, meaning that Windows recognizes certain internal flags in the executable file and automatically creates a console window.) See Remarks below for more details.

Syntax

```
InitCTInternals
```

Parameters

None

Return Value

`%ERROR_CT_NOCONSOLE` if the InitCTInternals function is used before a console window has been created.

`%ERROR_CT_UNKNOWNERROR` if the InitCTInternals function fails internally, usually due to a Windows memory problem.

`%SUCCESS` if none of the errors above are detected.

Remarks

If your program is a non-native console application which creates a console window by using the Windows "AllocConsole" API function, you should use the InitConsoleTools function (*not* InitCTInternals) immediately after the console has been created for the first time. If your program then destroys and re-creates a console window, it would normally be necessary to use InitConsoleTools again, but that function is limited to a single use per program. The InitCTInternals function can be used to re-initialize certain Console Tools internal functions after a console window has been destroyed and re-created.

iString (String "Shorthands")

Purpose

This function interprets "Shorthand" sequences in a string and inserts the corresponding control characters (and other hard-to-type characters) into the string. Shorthands are used internally by several Console Tools functions that use strings.

Availability

Console Tools Standard and Pro

Warnings

None. (If you are upgrading from Console Tools version 1 to 2, see **Parameters** below for an important warning.)

Syntax

```
sResult$ = iString(sOriginal$)
```

Parameters

sOriginal$

If this string contains one or more of the following "Shorthand" sequences, the iString routine will "interpret" the Shorthand strings and produce a modified string for the return value. (iString stands for "Interpreted String".) If no shorthand sequences are found, the original string is returned. The sOriginal$ string is *never* changed.

`\r`	Return	CHR$(13)	
`\n`	Newline	CHR$(10)	
`\q`	Quote	CHR$(34)	Double Quotation Mark (")
`\e`	Enter	CHR$(32,13,10)	Required for Input Boxes.
`\t`	Tab	CHR$(9)	

IMPORTANT NOTE: The iString function only recognizes *lower-case* letters. If you use `\N` (for example) it will not be interpreted as a shorthand string.

Console Tools uses the iString function internally when processing strings for Message Boxes, Input Boxes, Progress Boxes, and Splash Boxes, so you can always use the Shorthands when typing strings for these functions. The iString function can also be used by your program, if you have a use for it.

If you need to use a string that contains one of the strings above but you do not want it to be interpreted by iString, use a double-backslash to "escape" the shorthand. For example, the string `"\mydir\newprog"` contains the shorthand `\n` but you probably wouldn't want it to be replaced with a Line Feed Character. You have two choices: **1)** use the PB/CC UCASE$ function to convert the string to upper case, so that the lower-case shorthand string will not be detected, or **2)** use `"\mydir\\newprog"`. The double backslash is a signal to the iString function that you mean a "literal backslash" and that it is not a shorthand. The iString function would then remove the double backslash and the return value of the function would be `"\mydir\newprog"` with a single backslash.

In C++ and some other programming languages, Shorthand sequences are known as "Escape Sequences". We have avoided this term because it often causes confusion about the Escape character `CHR$(27)`, which has nothing to do with Shorthands.

WARNING: Please note that in Console Tools Version 1, the `\r` and `\n` shorthands were accidentally reversed. The `\r` shorthand produced CHR$(10) and `\n` produced CHR$(13), instead of vice versa. This error was fixed in Console Tools Version 2.00. If you are upgrading from version 1 to version 2, you should double-check your use of those two shorthands, to make sure that the new version of Console Tools produces the desired results.

Return Value

sResult$ will be the sOriginal$ string with the Shorthand strings converted into the corresponding characters.

Example

```
sString1$ = "MY \qSTRING\q"
sString2$ = "MY " + CHR$(34) + "STRING" + CHR$(34)
PRINT iString(sString1$)
PRINT sString2$
```

See Also

SplashBoxShow, ConsoleInputBox, ConsoleMessageBox

LocOfCol *and* LocOfRow

Purpose

These functions return the actual screen locations (in pixels) of console columns and rows.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = LocOfCol(lNumber&)
lResult& = LocOfRow(lNumber&)
```

Parameters

lNumber&

The number of the column or row for which you want the current screen location.

Return Value

If the LocOfCol (Location of Column) function is used, it returns a pixel value that corresponds to the x-axis (left-to-right) screen location of the left edge of the specified column.

If the LocOfRow (Location of Row) function is used, it returns a pixel value that corresponds to the y-axis (top-to-bottom) screen location of the top edge of the specified row.

If a valid column or row number is provided, these functions return an actual screen location (in pixels) based on 0,0 at the top-left of the screen.

If an invalid column or row number is provided (such as column "negative two" or row number 2000) these functions return the theoretical location of the column or row. In other words, they return the location where the column or row would be located, if it existed.

Remarks

The primary purpose of these functions is to allow the positioning of GUI elements such as Input Boxes, List Boxes, Splash Boxes, and Progress Boxes at particular row/column locations. For example, using...

```
ProgressBoxShow %CANCEL_BUTTON, _
                0, _
                100, _
                100, _
                "Text", _
                "Title", _
                0
```

...would cause a Progress Box to be displayed at screen coordinates 100,100. (The third and fourth parameters of the ProgressBoxShow function control the location of the Progress Box.) But using this code instead...

```
ProgressBoxShow %CANCEL_BUTTON, _
                0, _
                LocOfCol(10), _
                LocOfRow(10), _
                "Text", _
                "Title", _
                0
```

...would produce a Progress Box that was located at column 10, row 10 of the console window, regardless of the current location of the console.

Example

See **Remarks** above.

See Also

ColOfLoc, RowOfLoc

MatchConsoleFont (Console Tools Plus Graphics ONLY)

Purpose

Changes the width and/or height of the Graphics Tools font to correspond to the size of the font that is currently being used to display text in the console window.

Availability

Console Tools Standard and Pro

Warning

You must use an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function before you can use this function.

Syntax

```
MatchConsoleFont  lOption&
```

Parameters

lOption&

See **Remarks** below.

Return Value

If you fail to specify an appropriate value for the *lGraphics&* parameter of the InitConsoleTools function *before* you use this function, this function will return `%ERROR_CT_CANTBEDONE`.

Otherwise the return value of this function will always be `%SUCCESS`, so it is generally safe to ignore the return value.

Remarks

This function calculates the width and height of the font that is currently being used to display text in the console window, and adjusts the width and/or height of the Graphics Tools font accordingly.

Please note that all Graphics Tools font size adjustments must be "approved" by Windows, and Windows will not always create exactly the font size that you request. If this function does not provide the results that you expect, try using a different (preferably TrueType) Graphics Tools font. You can also try setting the font size to *approximately* the right width and height before using this function to set the final size. See the Graphics Tools documentation GfxFont entry for more information about fonts.

Please also note that the use of this function does not guarantee that the Graphics Tools font that is created will *appear* to be the same size as the console font. For example, it is very common for TrueType fonts to use more "empty space" than console fonts, above and below each character. This can result in a font that takes up the same amount of vertical space as the console font, but appears to be somewhat smaller.

To change the height of the Graphics Tools font to match the console font, use an *lOption&* value of `%HEIGHT_SAME`. To change the height so that it is one-and-one-half times the height of the console font, use `%HEIGHT_150_PERCENT`. You can also use `%HEIGHT_DOUBLE`. and `%HEIGHT_TRIPLE`.

To change the Graphics Tools font's width in a similar manner, use `%WIDTH_SAME`, `%WIDTH_150_PERCENT`, `%WIDTH_DOUBLE`, or `%WIDTH_TRIPLE`.

You can also use this function to adjust the height and width at the same time, by adding those values together. For example, using...

```
MatchConsoleFont %WIDTH_SAME + %HEIGHT_DOUBLE
```

...would adjust both dimensions of the Graphics Tools font simultaneously.

Examples

```
MatchConsoleFont %WIDTH_SAME + %HEIGHT_SAME
```

See Also

Please refer to the Graphics Tools documentation for more information.

MenuDefinition

Purpose

Processes a Menu Definition String for future use in a Pulldown or Popup Menu.

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Warning

If is very important to check the return value of this function during program development.

Syntax

```
lResult& = MenuDefinition(lMenuNumber&,
                          sMenuDefinition$)
```

Parameters

lMenuNumber&

The number of the Console Tools Menu Buffer into which the Menu Definition String is to be inserted. This parameter can have a value from 1 to the number that was used for lMenuBuffers& in the InitConsoleTools function.

sMenuDefinition$

A literal string or a string variable containing a valid Menu Definition, according to the syntax described in **Remarks** below.

Return Value

lResult& will be one of the following values:

`%SUCCESS` (zero) if no errors are found.

`%ERROR_CT_FEATURENOTAVAILABLE` if this function is used when the Console Tools Pro DLL is not installed.

`%ERROR_CT_INVALIDMENUNUMBER` if the value that is used for lMenuNumber& is not between 1 and the number specified for lMenuBuffers& in the InitConsoleTools function.

`%ERROR_CT_INVALIDITEMNUMBER` if a string with more than 128 items is used for sMenuDefinition$.

IMPORTANT NOTE: The following error codes all include the *Item Number* of the item in the string that caused the string to be rejected. To extract the item number from the Error Code, subtract the `%ERROR_CT_` value shown. Example: If `%ERROR_CT_INVALIDITEM` is detected it will be reported as the predefined value `%ERROR_CT_INVALIDITEM` *plus the item number of the offending item*. Since `%ERROR_CT_INVALIDITEM` is equal to 999,002,000 (see Error Codes), if the error was detected in the first item of the string it would be reported as error code 999,002,001. Please note also that this is not the item ID number. The first item is item number 1, the second is item number 2, etc. Ignore the Item ID number, which is located after the **=** or **>** symbol.

`%ERROR_CT_INVALIDITEM + ItemNumber` if the string contains an invalid item. Subtract the value `%ERROR_CT_INVALIDITEM` from the return value of the function to get the item number of the offending item, then check the item to make sure that its syntax is correct. See **Remarks** below for the proper syntax.

`%ERROR_CT_INVALIDSUBMENU + ItemNumber` if the string contains a reference to a submenu number that is not between 1 and the number that was used for lMenuBuffers& in the InitConsoleTools function. Subtract the value `%ERROR_CT_INVALIDSUBMENU` from the return value of the function to get the item number of the offending item.

`%ERROR_CT_SELFREFERENCE + lItem` if a menu item refers to its own menu number as a submenu. For example, Menu Buffer 1 will not accept a string that contains an item which refers to Menu 1 as a submenu. Subtract the value `%ERROR_CT_SELFREFERENCE` from the return value of the function to get the item number of the offending item.

`%ERROR_CT_INVALIDITEMNUMBER + lItem` if a menu item is assigned an ID number outside the range of 1 to 32750. Subtract the value `%ERROR_CT_INVALIDITEMNUMBER` from the return value of the function to get the item number of the offending item.

Remarks

The example...

```
MenuDefinition 1, "File=101 | Edit=102 | Help=103"
```

...would define a three-item menu (or submenu) called "Menu Number 1", It would contain three items: File, Edit, and Help. At runtime, the PulldownMenu or PopupMenu function would return the item ID numbers 101, 102, and 103, respectively, when those items were selected. You should normally use item ID numbers between 100 and 32,750. (More about the reserved numbers 1-99 later.)

These examples use a convention of numbering the items in menu #1 as 101, 102, 103, etc. and items in menu #2 as 201, 202, 203. We have found this to be helpful in designing and maintaining menu strings, but you may use almost any numbering system that you like as long as you use numbers between 100 and 32,750.

Please note that the "delimiter" character between the menu items is the "pipe" symbol, ASCII value 124 (&h7c) which appears on most keyboards as a vertical line with a small break. On many computer screens, in many fonts, it appears as an unbroken line.

Adding ampersands (&) to the three items like this...

```
MenuDefinition 1, "&File=101 | &Edit=102 | &Help=104"
```

...would cause the letters F, E, and H to be underlined at runtime, like File, Edit, and Help. If this was the definition for a top-level pulldown menu, pressing the Alt-F, Alt-E, or Alt-H keys would then pull down the corresponding submenu. If this was the definition for a submenu or a popup menu, pressing the letters F, E, and H would have the same effect as clicking on the menu item. (This is standard Windows behavior.)

To define a submenu for the File item in the example, first use another MenuDefinition statement to define the items for the submenu.

```
MenuDefinition 2, "&Open=201 | &Close=202 | E&xit=203"
```

This would create a menu with the items Open, Close, and Exit. Note that the underlined character does not have to be the first character.

When adding menu numbers, remember to change the number of menu buffers that are declared in the InitConsoleTools statement at the beginning of your program. *You do not have to use all of the buffers that you declare*, so you may choose to use a large number like 1024 in the InitConsoleTools declaration during development and testing, and then, when the program is complete and you want to fine-tune it, reduce the declaration to the actual value. Final programs can use the number 1024, but it wastes memory and slows down menu processing somewhat.

To make the menu #1's File item the "parent" of menu #2, change the first equal sign in the menu #1 definition to a *greater-than* symbol, and change the ID number to the submenu number that you want to activate when the item is clicked. The greater-than symbol looks like a small arrow that points to the submenu number, like this...

```
MenuDefinition 1, "&File > 2 | &Edit=102 | &Help=104"
MenuDefinition 2, "&Open=201 | &Close=202 | E&xit=203"
```

(To be clear, you would not have three MenuDefinition lines in your program at this point. You would only have two. The definition for menu #1 would have been *changed* during the coding process.)

Now instead of the File item of menu #1 returning a value when a user clicks on it, it will cause menu #2 to be displayed. You can repeat this process as many times as you need to. Submenus can be nested to virtually any level.

Let's jump ahead a little and look at the example after a few changes have been made.

```
MenuDefinition 1, "&File > 2 | &Edit > 3 | &Help > 4"
MenuDefinition 2, "&Open > 5 | &Close=202 | E&xit=203"
MenuDefinition 3, "Cu&t=301 | &Paste=302 | &Copy=303"
MenuDefinition 4, "&Index=401 | &About=402"
MenuDefinition 5, "&New=501 | &Existing=502 | &Archive=503"
```

With these strings, you would have a top-level menu with File, Edit, and Help. Each of those menus would pull down a submenu. The pulldown for File would have a submenu that contained the items Close and Exit with the return values 202 and 203, but clicking File/Open would pull down a sub-sub-menu (menu number 5) with the items New, Existing, and Archive.

A few rules...

You can't have any "circular" references. For example, menu #1 can't pull down #2, which then pulls down #3, which then pulls down #1 or #2 again.

You can't use non-existent references. For example you can't use >8 (go to 8) if menu #8 is not going to be defined. Don't worry about the *order* in which you define your menus... you do *not* have to define your submenus first and work up, to avoid referencing a menu that hasn't been defined yet. Just make sure that all of the references that you use are defined *somewhere*, or the MenuSystemCreate or PopupSystemCreate function will refuse to build your menus.

You *can* use the same ID numbers twice. There are circumstances when you might want two different menu items to perform the same action, so you might want to assign the same ID. This means that you must be careful not to *accidentally* assign the same value to two different items. That's part of the reason we suggest the numbering scheme that is described above.

The MenuDefinition function proofreads each string as it is submitted to the buffer. If the string is valid, the MenuDefinition function returns `%SUCCESS` (zero) to indicate zero errors. During menu testing, it's a good idea to use a structure like this...

```
IF MenuDefinition(1, "File>2|Edit>3") <> %SUCCESS THEN
      PRINT "ERROR IN MENU 1"
END IF

IF MenuDefinition(2, "Open>5|Close=202") <> %SUCCESS THEN
      PRINT "ERROR IN MENU 2"
END IF
```

... and so on, so that you will be alerted if you make a mistake. You could also put your menu strings into an array called sMenuArray$(1:5) and use something like...

```
FOR lMenu& = 1 TO 5
    lResult& = MenuDefinition(lMenu&,sMenuArray$(lMenu&))
    IF lResult& <> %SUCCESS THEN
       BEEP
       PRINT "Error #"+STR$(lResult&);
       PRINT " in Menu "+STR$(lMenu&)
    END IF
NEXT
```

Then, when your menu system has been perfected, you can remove the extra code to make your program smaller and faster. You can also use the function that is provided in the CTERROR.BAS file to display a Message Box when an error is found.

Now let's add some final details to the example menu. If you include an item that is just a hyphen (ASCII 45) in a menu string, like this...

```
MenuDefinition 2, "&Open > 5 | &Close=202 | - | E&xit=203"
```

... the Console Tools Menu System will insert a horizontal separator bar at that point in the menu.

If you include an item that is just a "caret" symbol (ASCII 94) like this...

```
MenuDefinition 2, "&Open > 5 | &Close=202 | ^ | E&xit=203"
```

... that tells the menu system to add a *vertical* separator bar and start a new column with that item. The caret, which looks like an up-arrow, is intended to mean "go up to the top of the next column".

If you add a Tab character by using `CHR$(9)` in a menu string or by including the `\t` tab shortcut like this...

`MenuDefinition 3, "Cu&t=301 | &Paste=302 | &Copy`**`\t`**`XXX=303"`

...the menu system will add several spaces after the word "Copy" and, with the example string, then print "XXX". All of the similar-length items in a submenu will be tabbed-over by the same amount. Here's a more useful example of using a tab...

`MenuDefinition 3, "Cu&t`**`\t`**`Ctrl-X=301 | &Paste`**`\t`**`Ctrl-C=302` (etc)

This would produce items with "Accelerator Key labels" after the main item label. "Cut" would have "Ctrl-X" out to the right, and "Paste" would have "Ctrl-C" out to the right by the same distance.

Then, to activate the Accelerator Keys, you must assign the special "reserved ID numbers" from 1-99 that were mentioned above. According to Appendix F: Accelerator Key Codes, the `%Pulldown_X` value (for Ctrl-X) is 88 and the `%Pulldown_C` value (for Ctrl-C) is 67, so you would change the line to...

`MenuDefinition 3, "Cu&t\tCtrl-X=`**`88`**`| &Paste\tCtrl-C=`**`67`** (etc.)

...and then, when your user presses Ctrl-X or selects the Cut item from your menu, the ID number 88 will be returned by the Pulldown Menu. (The `%Pulldown_` ID numbers are not assigned automatically when you use Ctrl-something in a menu, because some people prefer Ctrl**+**something or another convention.) You can use the keys from F1 to F12, from Ctrl-A to Ctrl-Z, from Ctrl-0 to Ctrl-9, and Alt-Enter as Accelerator Keys. As Microsoft suggests, Alt-keys are not allowed to be Accelerator Keys. They are reserved for using the keyboard to pull down top-level menus (see "&" above).

The next major step in the normal menu creation process is the use of either the MenuSystemCreate or PopupSystemCreate function. We suggest that you read those sections of the Help File next.

If you want to add Checkmarks, Disabled Items, and Pre-Highlighted items to your menus at runtime, see the MenuItemProperty function.

Example

See Using Pulldown and Popup Menus.

See Also

MenuSystemCreate, PulldownMenu,
PopUpSystemCreate, PopupMenu
MenuItemProperty, MenuSystemDestroy

MenuItemProperty

Purpose

Changes one property of a Console Tools Pulldown or Popup Menu Item (*after* it has been created with the MenuSystemCreate or PopUpSystemCreate function).

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Syntax

```
lResult& = MenuItemProperty(lMenuNumber&, _
                            lItemIDNumber&, _
                            lProperty&)
```

Parameters

lMenuNumber&

The number of the Console Tools Menu Buffer where the item is located, from 1 to the number that was used for lMenuBuffers& in the InitConsoleTools function..

lItemIDNumber&

The item ID number of the menu item to be changed (see **Example** below).

lProperty&

One of the following predefined equates:

`%CHECKON` or `%CHECKOFF` can be used to activate/deactivate a checkmark next to a menu item

`%DISABLE` or `%ENABLE` can be used to change a menu item so that it can no longer be selected, or to re-activate it.

`%HILITEON` or `%HILITEOFF` can be used to pre-highlight an item, or to remove the highlight.

Return Value

lResult& will be one of the following values:

`%ERROR_CT_MENUDOESNOTEXIST` will be returned if you attempt to use a Menu Buffer number that has not yet been "created" with the MenuSystemCreate or PopUpSystemCreate function.

`%ERROR_CT_INVALIDPARAMETER` will be returned if you use an invalid value for lProperty&.

If Windows fails to change the requested menu item property, a Microsoft Error Number will *usually* be returned.

Under certain circumstances, Windows will fail to make the requested change but it will not provide a Windows Error Number. In that case, lResult& will be `%ERROR_CT_UNKNOWNERROR`.

`%SUCCESS` will be returned if no are detected.

Remarks

Before you can use this function you must first **1)** use the MenuDefinition function to define a menu, and **2)** use either the MenuSystemCreate or PopUpSystemCreate function to build the menu system. All of those functions must return `%SUCCESS` or the MenuItemProperty function will not work.

Menu Item Properties cannot be applied to top-level menu items, or to any menu items that have submenus. To disable an item that has a submenu you can **1)** disable all of the items on the submenu so that they cannot be selected, or **2)** use the MenuDefinition function to re-define the menu so that the item with the submenu is no longer included, then use MenuSystemCreate or PopUpSystemCreate to rebuild the menu system, or **3)** use the MenuDefinition function to redefine the menu item that has a submenu so that it has an item number instead of a "go to" reference, then use MenuSystemCreate or PopUpSystemCreate to rebuild the menu system, then disable the item.

All Menu Item Properties are reset to "off" when a menu system is destroyed, or when it is re-created with the MenuSystemCreate or PopUpSystemCreate function.

Example

```
lResult& = MenuItemProperty(9, 901, %CHECKON)
```

This example would add a checkmark next to item 901 of the Menu String in Buffer #9.

See Also

Using Pulldown and Popup Menus, MenuDefintion, MenuSystemCreate, PulldownMenu,
PopUpSystemCreate, PopupMenu,
MenuSystemDestroy

MenuSystemCreate

Purpose

Builds a complete, interconnected Pulldown Menu System from strings that were submitted to the MenuDefinition function. The menu system that is created by this function can then be displayed with the PulldownMenu function. (To create a Popup Menu instead of a Pulldown Menu, use PopUpSystemCreate instead of MenuSystemCreate.)

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Syntax

```
lResult& = MenuSystemCreate(lFirstBuffer&, _
                            lLastBuffer&)
```

Parameters

lFirstBuffer& and *lLastBuffer&*

The range of Menu Buffers that is to be processed into a Pulldown Menu System. The lowest legal number for either parameter is 1, and the largest is the value that was used for the lMenuBuffers& parameter of the InitConsoleTools function. If lLastBuffer& is less than or equal to lFirstBuffer&, only lFirstBuffer& will be processed.

Return Value

lResult& will be one of the following values...

`%ERROR_CT_INVALIDMENUNUMBER` if the Menu System was not initialized properly with InitConsoleTools or if the lFirstBuffer& or lLastBuffer& parameter is not between 1 and the number of Menu Buffers that were specified with InitConsoleTools.

`%ERROR_CT_INVALIDMENUSYSTEM` if the MenuSystemCreate function was unable to create the Menu System because of **1)** a "circular reference" such as Menu 1 having Menu 2 as a submenu, which then has Menu 1 as a submenu, or **2)** an error in a Menu Definition String. The proper use of the MenuDefinition function -- especially checking the return value -- will avoid most problems.

`%ERROR_CT_UNKNOWNERROR` if Windows refused to allow the creation of a new (blank) menu. Windows does not provide information about why it refused, but we assume that it relates to insufficient memory. (This should be a very rare problem, considering the relatively small amount of memory that blank menus require.)

`%SUCCESS` (zero) if none of the problems above were detected.

Remarks

The first step in building a Pulldown Menu System involves the MenuDefinition function. The MenuSystemCreate function can only be used after the error-free use of MenuDefinition to fill at least one Menu Buffer.

See Using Pulldown and Popup Menus for a complete description of this function.

Example

```
lResult& = MenuSystemCreate(16,32)
```

This example would build a Pulldown Menu System using Menu Buffers 16 through 32.

See Also

MenuDefintion, MenuItemProperty, PulldownMenu, MenuSystemDestroy

MenuSystemDestroy

Purpose

Destroys a Pulldown or Popup Menu System that was previously created with MenuSystemCreate or PopUpSystemCreate.

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Warning

The incorrect use of this function can cause Application Errors (General Protection Faults). See Remarks below.

Syntax

```
lResult& = MenuSystemDestroy(lFirstBuffer&, _
                             lLastBuffer&)
```

Parameters

lFirstBuffer& and *lLastBuffer&*

The range of Menu Buffers that should be destroyed. If lLastBuffer& is less than or equal to lFirstBuffer&, only lFirstBuffer& will be destroyed.

Return Value

lResult& will be one of the following values:

`%ERROR_CT_INVALIDMENUNUMBER` if the lFirstBuffer& and/or lLastBuffer& parameter is less than zero (0) or greater than the number of Menu Buffer numbers that was defined by the lBufferNumber& parameter of the InitConsoleTools function.

`%SUCCESS` if valid Menu Buffer numbers are specified.

Remarks

This is strictly a "housekeeping" function that your program can use to free up memory for other uses. The MenuSystemCreate and PopUpSystemCreate functions *automatically* use this function before they create a new Menu System with a range of buffer numbers, so it is not necessary to perform this step yourself.

It is not necessary to use MenuSystemDestroy at the end of a program that uses Pulldown or Popup Menus. All menu systems are destroyed automatically.

If you use MenuSystemDestroy to destroy the top-level menu of a menu system, the PulldownMenu and PopupMenu functions will fail when the menu is used.

If you use MenuSystemDestroy to destroy a submenu and then your user tries to access a menu that *uses* that submenu, the program will usually trigger an Application Error (a General Protection Fault).

See Using Pulldown and Popup Menus for more information.

Example

```
lResult& = MenuSystemDestroy(1, 10)
```

This example would destroy the menus and submenus in Buffers 1 through 10.

See Also

MenuSystemCreate, MenuDefintion, MenuItemProperty, PulldownMenu

MouseOverCol *and* MouseOverRow

Purpose

These functions return the current location of the Windows mouse cursor, in terms of console window "row and column" numbers.

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
lResult1 = MouseOverCol
lResult2 = MouseOverRow
```

Parameters

None.

Return Value

These functions will return negative one (-1) if **1)** the mouse cursor is not currently located over the "print area" of the console window, or **2)** if the console window is in the Fullscreen Mode.

If the console window is in the Window Mode and the mouse cursor is located over the "print area" of the console window, these functions will return Column and Row numbers from 1 to the Width/Height of the console window. For example, if the console window currently has 80 columns and 25 rows, the MouseOverCol function will return a value from 1 to 80 and the MouseOverRow function will return a value from 1 to 25.

Remarks

Keep in mind that these functions will return Column and Row information *even if the console is not currently visible, and even if the console does not currently have the Windows focus.* For example, if your program's console window is partially or completely covered up by another application (including the Windows Task Bar), and if the user moves the mouse cursor over the area of the screen that the console window occupies, MouseOverCol and MouseOverRow *will* return values. If that is the behavior that you need, you can use the MouseOverCol and MouseOverRow functions by themselves. If you want your program to respond to MouseOverCol and MouseOverRow values only when the console is the foreground window, use the ConsoleIsForeground function in conjunction with MouseOverCol and MouseOverRow.

One common use for the MouseOverCol and MouseOverRow functions is determining where the mouse cursor was located when a user clicked on the console window while a Pulldown Menu was on the screen. See the Details section of PulldownMenu for more information.

Another common use for the MouseOver functions is the display of "tip" information. For example, your program could reserve row 25 of the console window for the display of text that describes the values that are legal in the user-entry field that the mouse is currently over. Whenever your program detects that the mouse has been moved to a new screen location, it could update row 25 with a new tip.

Unlike the PB/CC MOUSEX and MOUSEY functions, the MouseOverCol and MouseOverRow function do not rely on the PB/CC MOUSE, INSTAT, INKEY$, or WAITKEY$ functions. MouseOverCol and MouseOverRow can be used at any time, regardless of your program's use of the PB/CC mouse functions.

Example

```
CLS
CURSOR OFF
PRINT "The mouse is currently over Row .. Column ..
DO
        LOCATE 1,32
        PRINT MouseOverRow;
        LOCATE 1,42
        PRINT MouseOverCol;
        IF INSTAT THEN EXIT LOOP
LOOP
```

See Also

MouseOverConsole
PulldownMenu

MouseOverConsole

Purpose

Returns a Logical True value if the Windows mouse cursor is currently located over the console window's "print area", or False if it is not.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
lResult& = MouseOverConsole
```

Parameters

None.

Return Value

If the Windows mouse cursor is currently located over the "print area" of the console window, this function returns a Logical True value. Otherwise it returns False (zero).

Remarks

Please note that, like the MouseOverCol and MouseOverRow functions, this function returns a value regardless of whether or not the console is covered by another window, and whether or not the console window is the current foreground window. See MouseOverCol for more information about this.

Unlike the PB/CC MOUSEX and MOUSEY functions, the MouseOverConsole function does not rely on the PB/CC MOUSE, INSTAT, INKEY$, or WAITKEY$ functions. It can be used at any time, regardless of your program's use of the PB/CC mouse functions.

Example

```
LOCATE 1,1
IF MouseOverConsole THEN
      PRINT "MOUSE IS OVER CONSOLE"
ELSE
      PRINT "Mouse is NOT over console"
END IF
```

See Also

MouseOverCol and MouseOverRow

OnCtrlBreak

Purpose

Tells Console Tools what action it should take when your program's user presses Ctrl-Break.

Availability

Console Tools Pro Only (see)

Warning

If you use this function's `CODEPTR` option, see notes about potential problems in **Remarks** below.

Syntax

```
OnCtrlBreak  dwAction???
```

or

```
OnCtrlBreak  CODEPTR(FuncName)
```

Parameters

dwAction???

Specifies the action that will be taken when Ctrl-Break is pressed. This parameter must be either **1)** the value one (1) to ignore Ctrl-Break keypresses, or **2)** the value two (2) to sound a `BEEP`, or **3)** the value three (3) to send `CHR$(0,0)` (which is not produced by any other process) to the PB/CC keyboard input functions (`INKEY$`, etc.), or **4)** the value zero (0) to disable Ctrl-Break detection and allow Windows to close your program, or **5)** a PB/CC `CODEPTR` value that corresponds to the address of a function that should be executed. If you use the `CODEPTR` option, your function *must* conform to the guidelines shown in **Remarks** below.

or FuncName

The name of the function that should be executed when Ctrl-Break is pressed. If you use the `CODEPTR(FuncName)` option, your function *must* conform to the guidelines shown in **Remarks** below.

Return Value

None.

Remarks

If a user presses Ctrl-Break, the default Windows behavior is to instantly close a console application. The OnCtrlBreak function can be used to override this default behavior and provide a more orderly handling of the Ctrl-Break keypress.

The easiest way to use this function is to specify a numeric value for dwAction??? that is between zero (0) and three (3). The actions that these numbers produce are described under **Parameters** above.

It is also possible to use a `CODEPTR` value for dwAction??? but this technique requires certain precautions. It is not *difficult* to do, but we strongly recommend that you read all of the following sections before using `CODEPTR` with OnCtrlBreak.

Using CODEPTR with OnCtrlBreak

If you use `CODEPTR` to give OnCtrlBreak the address of a function that will be executed when Ctrl-Break is pressed, the function *must* conform to the following prototype.

```
FUNCTION MyFunction(BYVAL lPlaceHolder AS LONG) AS LONG
      'your code goes here
      FUNCTION = 1
END FUNCTION
```

If you prefer to use type identifiers instead of the `AS` syntax, you may use this alternate prototype:

```
FUNCTION MyFunction&(BYVAL lPlaceHolder&)
      'your code goes here
      FUNCTION = 1
END FUNCTION
```

The name of the function and the name of the "placeholder" variable are completely optional. You are not required to use "MyFunction" and "lPlaceHolder".

It will not usually be used by your function, but if you do not include the placeholder variable as shown, Windows will generate an Application Error (a General Protection Fault) when Ctrl-Break is pressed. (If you examine its value, the placeholder variable will always have a value of one. Compare the OnCtrlBreak function to the OnShutdown function for more information about this value.)

If you do not include the `FUNCTION = 1` line, Windows will assume that your function did not handle the Ctrl-Break keypress and will close your program as soon as your OnCtrlBreak function is finished executing. The `FUNCTION = 1` line effectively tells Windows "we handled the keypress, so you don't need to do anything". If, on the other hand, you *want* Windows to close your program after your OnCtrlBreak function has executed, you can use `FUNCTION = 0`.

Suggested Uses of OnCtrlBreak with CODEPTR

Using `CODEPTR` with OnCtrlBreak can provide an extremely fast, efficient method of checking for an interrupt signal from a user. Instead of periodically checking `INSTAT` (which is relatively slow) to find out whether or not a key like Escape has been pressed, your program can...

1) Create a GLOBAL variable called (for example) `lUserCancel&`.

2) Create a simple function called (for example) `UserInterrupt` which simply sets the value of `lUserCancel&` to a nonzero value whenever the `UserInterrupt` function is executed. It would look something like this...

```
FUNCTION UserInterupt(BYVAL lPlaceHolder AS LONG) AS LONG
      DIM lUserCancel AS GLOBAL LONG
      lUserCancel = 1
      FUNCTION = 1
END FUNCTION
```

3) In your main program, add a periodic check of the value of the `lUserCancel` variable (instead of checking `INSTAT`), and add code that causes your program (or simply the current task) to exit gracefully when the value of `lUserCancel&` is nonzero.

4) Add this code...

```
    OnCtrlBreak  CODEPTR(UserInterrupt)
```

...to the beginning of your program. (Unless you use the PB/CC `DECLARE` statement, the PowerBASIC compiler will require you to define your UserInterrupt function *before* the `OnCtrlBreak CODEPTR` line. You must always declare or define a function before it can be used with the `CODEPTR` function.)

If you perform all of those steps correctly, when a user presses Ctrl-Break your `UserInterrupt` function will be executed by Console Tools, which will set the value of your `lUserCancel` variable to a nonzero value, which will cause your main program to take some action. This process is usually *much* faster than using `INSTAT` or `INKEY$` to check for keyboard input, because it involves only the periodic checking of a super-efficient integer variable. But remember that it is limited to Ctrl-Break. No other keypress can be detected with OnCtrlBreak.

You could improve this technique by adding a ConsoleMessageBox to the `UserInterrupt` function, to ask "Are you sure?" Then, if the user clicks the Yes button, the `lUserCancel` value would be set.

<u>Possible Problems Caused By Multi-Threaded Operation</u>

If you use a simple numeric value from 0 to 3 for the dwAction??? parameter of this function you do *not* need to concern yourself with this warning.

Many different uses of OnCtrlBreak are possible, but *if you use the CODEPTR option to cause Ctrl-Break to execute a function* you must keep in mind that Windows is a "multi-tasking" operating system. When Ctrl-Break is pressed and Console Tools executes your function (such as the `UserInterrupt` example above), your program actually splits into two parts called "threads" and *both* parts execute at the same time. *Your "main" program does not stop running while your OnCtrlBreak function executes.* If you keep your OnCtrlBreak function very simple -- such as setting the value of a variable or displaying a message box -- you shouldn't have any problems. But if you try to do too much in your OnCtrlBreak function you can get into trouble very easily.

For example, let's say that your program is a complex file-processing program, and that instead of stopping the program you want a CtrlBreak keypress to create a disk file that contains a "snapshot" of the current job's status. If your OnCtrlBreak function uses the PB/CC `FREEFILE` function to obtain an unused file number (for use with `OPEN`), it would be possible for both "halves" of your program -- both threads -- to use `FREEFILE` at exactly the same time, obtain the same "free" file number, and then interfere with each other by trying to use the same number in an `OPEN` statement. This is just one of *thousands* of possible examples. Unless you're familiar with writing multi-threaded Windows applications, you should avoid doing *anything* in your OnCtrlBreak function that could possibly conflict with something that your main program is doing. To be completely safe, your OnCtrlBreak function should simply set a certain variable to a certain value so that the main program will see the change and act upon it.

Tip: If you want to, the OnCtrlBreak and OnShutdown functions can both use the same "handler" function. Since OnCtrlBreak always passes a parameter value of one (1) to its function, and OnShutdown passes a value of two, five, or six, the common handler function can examine the value of the parameter and decide what to do.

Example

```
OnCtrlBreak  CODEPTR(MyCtrlBreakHandler)
```

See Also

OnShutdown

OnShutdown

Purpose

Changes the way Windows handles Close, Shutdown, and LogOff events, and tells Console Tools which function to execute when one of those events is detected.

Availability

Console Tools Pro Only (see)

Warning

Microsoft Windows 95, Windows 98, and Windows ME do not provide **Close** event notification of any kind to console applications, so it is not possible for Console Tools to detect Close events on 95/98/ME systems. The use of the OnShutdown function to detect Close events will be effective only when your programs are run on Windows NT/2000/XP systems. We therefore recommend that you disable the Close event by using the DeleteWindowMenuItem function.

Warning

We strongly recommend that you read the entire **Remarks** section (below) before using this function.

Syntax

```
OnShutdown dwAction???
```

or

```
OnShutdown CODEPTR(FuncName)
```

Parameters

dwAction???

Either **1)** the value zero (0) to disable the detection of shutdown events and allow Windows to handle them normally, or **2)** a PB/CC `CODEPTR` value that contains the address of a function that will be executed whenever a shutdown event is detected. See **Remarks** below for important additional information.

or FuncName

The name of the function (in your program) that should be executed when a shutdown event is detected. See **Remarks** below for important additional information.

Return Value

None.

Remarks

A Windows **Shutdown** event is generated when Windows is shut down by the user (via the Start Menu's 'Shut Down" item) or by a program (via the Windows API).

A Windows **Logoff** event is very similar to a Shutdown event, except that Windows is not completely shut down and the user is given the opportunity to Log On again.

A Windows **Close** event is generated when a user clicks your application's Close (**x**) button, presses Alt-F4, uses the Window Menu to select Close, or double-clicks your application's title bar icon. IMPORTANT NOTE: Windows Close event notification is *not supported* by the Windows 95/98/ME console window, so Console Tools cannot detect Close events on 95/98/ME computers. We therefore recommend that you use the DeleteWindowMenuItem function to disable the Close event if your program will be run on Windows 95/98/ME computers.

Please note that it *is* possible for your program to be shut down without any of these events being generated. For example, Windows does not currently support a "somebody just spilled a Coke on the keyboard" event or a "somebody tripped over the power cord" event.

Normally, when a Windows Close, Shutdown, or Logoff event takes place, all console applications are closed immediately. Unlike other Windows applications, console apps are not normally given the opportunity to close gracefully, or to reject the event and stop the Close/Shutdown/Logoff process.

The OnShutdown function gives your PB/CC programs the same *limited* ability that other Windows apps have, to detect and control those events.

IMPORTANT WARNING: Once the Logoff or Shutdown process has been started, it can't be completely stopped *by any program of any type*. For example, if a Shutdown is initiated via the Start Menu and one or more programs are closed, and then a certain program refuses to close, the other programs will *not* be automatically re-started. This may result in the closing of programs such as the Sound subsystem, driver programs for things like Zip drives, certain types of network drivers, email notification programs, *any* program in your Windows StartUp directory, and many other "infrastructure" programs that may be *very* important to the applications that are still running. You should therefore not attempt to use the Console Tools OnShutdown function to completely stop the Shutdown or Logoff process except in an emergency. *Windows provides these events primarily to allow applications to close gracefully, not to refuse to close.*

VERY IMPORTANT WARNING: Once the Close/Logoff/Shutdown process has been started, Windows may actually disable certain internal console functions *before* it alerts your program that the event is in progress. Those internal console functions, which are part of Windows (not Console Tools or PB/CC) allow console applications to display information and receive keyboard and mouse input, so it will often be impossible for your program to continue running even if it intercepts the Close/Shutdown/Logoff event. This is another very important reason that your programs should react to the Close/Shutdown/Logoff events by exiting gracefully, as quickly as possible. *It is important to note that this warning includes the Close event.* You must always avoid relying on console input and output during your program's shutdown. For example, if you want to display a "Do you want to Save before exiting?" message, you should always use a ConsoleMessageBox instead of using `PRINT` and `INKEY$`, because those low-level console functions simply may not work.

With all of those warnings, you may be wondering what benefits the OnShutdown function actually provides. The answer is "a few seconds of warning". Without the OnShutdown function, Windows will close your PB/CC program *instantly*, without giving it the opportunity to save its current data or do any other "cleanup" work, such as closing a network connection or a database connection.

The Close Event

Perfect Sync recommends that under normal circumstances you simply disable the Windows Close event by using the DeleteWindowMenuItem function. If you do that, you won't need to use the OnShutdown function to detect Close events because they will never be generated. But if you choose not to do that...

When a user clicks the Close (**x**) button or uses any other standard Windows Close technique, Windows will notify the OnShutdown function that your program is being closed. (Keep in mind that *from that moment forward* your program's console *may* be unable to display text or receive keyboard/mouse input.) The OnShutdown function will then execute the function that you have specified with the dwAction??? parameter. (More about that function later.) If your program does not close within five (5) seconds, Windows will display a message that says "This Windows program cannot respond to the End Task request". It will then give the user the following choices:

1) Wait five seconds. If the user selects this option Windows will wait five seconds and then, if the application has not shut down, re-display the same three choices.

2) End Task. This *instantly* shuts down your program, regardless of what it is doing.

3) Cancel. This option cancels the Close event and your program will be able to continue running, but it will (probably) be unable to perform console input or output.

If your program closes while those choices are still on the screen, the message box will automatically disappear.

So once again, the bottom line is that your program should always exit as quickly and gracefully as possible when a Windows Close event is detected.

The Shutdown and Logoff Events

Shutdown and Logoff events are very similar to Close events (see above), except that...

1) They are initiated by the Start Menu or by a program that uses a Windows API function such as ExitWindowsEx.

2) They cannot be disabled by using the DeleteWindowMenuItem function.

3) The Windows timeout is 20 seconds instead of 5.

Your OnShutdown Function

Your OnShutdown function (i.e. the function that you use with `CODEPTR` for the dwAction??? parameter) *must* conform to the following prototype.

```
FUNCTION MyFunction(BYVAL lType AS LONG) AS LONG
      'your code goes here
      FUNCTION = 1
END FUNCTION
```

If you prefer to use type identifiers instead of the `AS` syntax, you may use this alternate prototype:

```
FUNCTION MyFunction&(BYVAL lType&)
      'your code goes here
      FUNCTION = 1
END FUNCTION
```

The name of the function and the name of the "type" variable are completely optional. You are not required to use "MyFunction" and "lType". But you *must* include a single `BYVAL` Long Integer parameter, or an Application Error will be generated.

(Unless you use the PB/CC `DECLARE` statement, the PowerBASIC compiler will require you to define your OnShutdown function *before* using `OnShutdown CODEPTR`. You must always declare or define a function before it can be used with the `CODEPTR` function.)

If you do not include the `FUNCTION = 1` line, Windows will assume that your program did not handle the event and will *immediately* shut it down. Setting the return value to one (1) effectively tells Windows "I promise to shut down". *You must use the value positive one (1). No other value will work properly.*

When a shutdown event is detected and your function is executed, the *lType* parameter will contain a number that indicates the type of shutdown event that is taking place. The number two (2) indicates a Close event, five (5) indicates a Logoff event, and six (6) indicates a Shutdown event. (Windows defines these numbers, not Console Tools. If you're curious, zero corresponds to a Ctrl-C event, which is handled by PB/CC, one corresponds to a Ctrl-Break event, which is handled by the OnCtrlBreak function, and three and four are not currently used by Windows.)

The "your code goes here" portion of the function should be as simple as possible. Ideally, it should simply assign a value to a `GLOBAL` variable so that other parts of your program can see the value and exit accordingly.

When designing your OnShutdown functon you must keep in mind that Windows is a "multi-tasking" operating system. When a shutdown begins and Console Tools executes your function, your program actually splits into two parts called "threads" and *both* parts execute at the same time. *Your "main" program does not stop running while your OnShutdown function executes.* If you keep your OnShutdown function very simple -- such as setting the value of a global variable or displaying a "Do you want to Save before exiting?" message box -- you shouldn't have any problems. But if you try to do too much in your OnShutdown function you can get into trouble very easily.

For example, let's say that you want your OnShutdown function to save a file before your program exits. If your OnShutdown function uses the PB/CC `FREEFILE` function to obtain an unused file number (for use with `OPEN`), it would be possible for both "halves" of your program -- both threads -- to use `FREEFILE` at exactly the same time, obtain the same "free" file number, and then interfere with each other by trying to use the same file number in an `OPEN` statement. This is just one of *thousands* of possible examples. Unless you're familiar with writing multi-threaded Windows applications, you should avoid doing *anything* in your OnShutdown function that could possibly conflict with something that your main program is doing. To be completely safe, your OnShutdown function should simply set a global variable to a certain value so that the main program will see the change and act upon it.

You should also keep in mind that your main program can interfere with your OnShutdown function in other ways. For example, if the main program above was already in the process of saving a file and exiting when the shutdown event was detected, it would be possible for the OnShutdown function to be *partially* executed when the main program closed. And when the main program closes, all threads are immediately closed, so the OnShutdown functon would never finish running. Again, unless you keep your OnShutdown function very simple, things can become very complex, very quickly.

Tip: If you want to, the OnShutdown and OnCtrlBreak functions can both use the same "handler" function. Since OnCtrlBreak always passes a parameter value of one (1) to its function, and OnShutdown passes a value of two, five, or six, the common handler function can examine the value of the parameter and decide what to do.

Example

```
OnShutdown CODEPTR(MyShutdownFunction)
```

See Also

OnCtrlBreak

OnStateChange

Purpose

Executes a function whenever your PB/CC program's Window State changes to Restored, Maximized, or Minimized.

Availability

Console Tools Pro Only (see)

Warning

See notes regarding Multi-Threading below.

Syntax

```
lResult& = OnStateChange(CODEPTR(dwAction???))
```

...or...

```
OnStateChange CODEPTR(dwAction???)
```

Parameters

dwAction???

The action that should be taken (i.e. the function that should be executed) when a Window State change is detected. This parameter must be a PB/CC CODEPTR value which points to a properly-formatted PB/CC function. To disable a previously-specified function, use zero (0) for this parameter. See **Remarks** below for details.

Return Value

This function is available only in the Console Tools Pro DLL, so lResult& will be `%ERROR_CT_FEATURENOTAVAILABLE` if you attempt to use this function when the Console Tools Standard DLL is installed.

Otherwise, this function will always return `%SUCCESS,` so it is usually safe to ignore the return value of this function, as shown in the second **Syntax** entry above.

Remarks

This function is similar to the other "On" functions -- OnTimer, OnCtrlBreak, and OnShutdown.

The function that you use with OnStateChange (i.e. the function that is used with `CODEPTR`) must conform to the following prototype:

```
FUNCTION MyFunction(BYVAL lState AS LONG) AS LONG
      'your code goes here
END FUNCTION
```

If you prefer to use type identifiers instead of the `AS` syntax, you may use this alternate prototype:

```
FUNCTION MyFunction&(BYVAL lState&)
      'your code goes here
END FUNCTION
```

The name of the function and the name of the "state" variable are completely optional. You are not required to use "MyFunction" and "lState&".

Unless you use the PB/CC `DECLARE` statement, the PowerBASIC compiler will require you to define your OnStateChange function *before* using `OnStateChange CODEPTR`. You must always declare or define a function before it can be used with the PB/CC `CODEPTR` function.

Whenever your program's Window State changes, your function will be executed and the lState& parameter will contain either `%RESTORE`, `%MAXIMIZE`, or `%MINIMIZE` to indicate the new Window State.

Important Warnings About Multi-Threading

Because the OnStateChange executes in a second "thread of execution" and your main program is never interrupted, you must be very careful to make sure that the two threads do not interfere with each other. For example, if your OnStateChange function happens to use the PB/CC `FREEFILE` function at exactly the same time as your main program, both functions could receive the same "free" file number and then interfere with each other when they both tried to use `OPEN` with the same file number. This is just one of *thousands* of possible examples. Unless you're familiar with writing multi-threaded Windows applications, you should avoid doing *anything* in your OnStateChange function that could possibly conflict with something that your main program is doing. To be completely safe, your OnStateChange function should simply set a global variable to a certain value so that your main program will see the change and act upon it.

Example

```
'Beep whenever the console is minimized...

OnStateChange CODEPTR(StateHasChanged&)

FUNCTION StateHasChanged&(BYVAL lState&)
      IF lState& = %MINIMIZE THEN BEEP
END FUNCTION
```

See Also

OnTimer, OnCtrlBreak, OnShutdown

OnTimer

Purpose

Executes a function every *x* seconds or milliseconds, or suspends or disables a timer that is already running.

Availability

Console Tools Pro Only (see)

Warnings

See **Remarks** below for several important precautions.

Syntax

```
OnTimer lHowOften&, CODEPTR(FuncName)
```

or

```
OnTimer lHowOften&, dwAction???
```

Parameters

lHowOften&

Use a positive number to specify a number of seconds, or a negative number to specify a number of milliseconds, or zero (0) to disable a timer that is already running.

FuncName

The name of the function which will be executed when the timer expires. See **Remarks** below for more information.

or dwAction???

Either **1)** a PB/CC `CODEPTR` value which specifies the address of the function that is to be executed each time the timer expires, or **2)** zero (0) if no action is to be taken. See **Remarks** below for more information.

Return Value

If Windows is unable to create the requested timer, this function will return `%ERROR_CT_UNKNOWNERROR`. Otherwise this function will return `%SUCCESS` (zero).

Remarks

The function that you use with OnTimer (i.e. the function that is used with `CODEPTR`) must conform to the following prototype:

```
FUNCTION MyFunction(BYVAL lTime AS LONG) AS LONG
      'your code goes here
END FUNCTION
```

If you prefer to use type identifiers instead of the `AS` syntax, you may use this alternate prototype:

```
FUNCTION MyFunction&(BYVAL lTime&)
      'your code goes here
END FUNCTION
```

The name of the function and the name of the "time" variable are completely optional.

You are not required to use "MyFunction" and "lTime".

Unless you use the PB/CC `DECLARE` statement, the PowerBASIC compiler will require you to define your OnTimer function *before* using `OnTimer CODEPTR`. You must always declare or define a function before it can be used with the PB/CC `CODEPTR` function.

When the timer expires and your function is executed, the lTime& parameter will contain a value that represents the current number of milliseconds since midnight. Since there are 86,400 seconds in a day, this value will always be a value between zero (0) and 86,400,000. (You can ignore this value if you want to, but it can be very useful for performing timing operations.)

The Console Tools OnTimer function differs from the DOS BASIC `ON TIMER` function in several very important ways. Even if you're not familiar with DOS BASIC, the following points will describe exactly how OnTimer works.

1) You must always keep in mind that Windows is a "multi-tasking" operating system. While DOS BASIC checks the timer between every line of code that is executed, the Console Tools OnTimer function checks the timer *continuously*. (Also see warnings below regarding multi-threading.)

2) While a DOS BASIC "main program" is suspended each time the timer expires (so that the `ON TIMER` function can execute) your OnTimer function will be executed without interrupting your main program. It does this by splitting your application into two different parts, called "threads", and both parts will execute at the same time.

3) DOS BASIC's `ON TIMER` only supports seconds, but the Console Tools OnTimer function supports milliseconds. For example, by specifying a *lHowOften&* value of `--250`, you could create a timer which expires every 250 milliseconds (every one-quarter of a second). The actual timer resolution that your program will produce will depend on how fast your computer is, and how busy Windows is, performing other tasks. If you use a very small timer value (under 50 milliseconds or so) the timer's accuracy will be greatly reduced.

4) DOS BASIC's `ON TIMER` must be turned on with the `TIMER ON` function, but the Console Tools OnTimer function begins "ticking" as soon as it is executed.

5) DOS BASIC's `ON TIMER` must be turned off with the `TIMER OFF` function, but the Console Tools OnTimer function is turned off by using a *lHowOften&* value of zero (0).

6) DOS BASIC's `ON TIMER` can be "suspended" with the `TIMER STOP` function, but the OnTimer function must be suspended by using a *dwAction???* value of zero. This causes the timer to continue ticking, but no action will be taken until the *dwAction???* value is changed to a `CODEPTR` value.

7) While DOS BASIC's `ON TIMER` function resets its internal counter when it returns to the main program, OnTimer runs continuously. For example, a DOS BASIC program with a one-second timer would normally execute its function once per second. But if the function takes one-half second to execute, the timer is effectively slowed down to `1.5` seconds because the counter does not start ticking until the function is finished executing. The Console Tools OnTimer function, on the other hand, behaves like a normal Windows timer, not a DOS timer. If the timer is set for one second and the function takes one-half second to execute, it will still be executed once per second. The only exception to this rule occurs when the function takes longer than the *lHowLong&* value to execute. For example, if the timer is set for one second and the function takes two seconds to execute, the timer will expire again while the function is executing. So when the function finishes, OnTimer will see that the timer has already expired and will immediately execute the function again. (OnTimer will not run your function twice, in different threads, if the function "overlaps" the timer expiration in this way.)

Important Warnings About Multi-Threading

Because, as was noted above, the OnTimer function executes in a second "thread of execution" and your main program is never interrupted, you must be very careful to make sure that the two threads do not interfere with each other. For example, if your OnTimer function happens to use the PB/CC `FREEFILE` function at exactly the same time as your main program, both functions could receive the same "free" file number and then interfere with each other when they both tried to use `OPEN` with the same file number. This is just one of *thousands* of possible examples. Unless you're familiar with writing multi-threaded Windows applications, you should avoid doing *anything* in your OnTimer function that could possibly conflict with something that your main program is doing. To be completely safe, your OnTimer function should simply set a global variable to a certain value so that your main program will see the change and act upon it.

Creating Multiple Timers

Console Tools only supports one timer per program (as is recommended by Microsoft), but it is possible to simulate multiple timers.

For example, if your program needs one timer that expires once per second and another timer that expires once per minute, you could create a single timer that expires once per second. Your function could then contain a STATIC counter variable which would be incremented every time the function was executed. When the count reached sixty, that would mean that sixty seconds had elapsed, and the function could perform the once-per-minute operation. (It would, of course, then need to reset the STATIC counter to zero.)

Example

```
'Display a ticking clock in the title bar:

OnTimer 1, CODEPTR(TitleBarTime)

FUNCTION TitleBarTime(lNotUsed&) AS LONG
      ConsoleTitle "Current Time: " + TIME$
END FUNCTION
```

See Also

OnCtrlBreak, OnShutdown

PopupMenu

Purpose

Displays a Popup Menu that was previously created with the MenuDefinition and PopUpSystemCreate functions, and returns a value to indicate the user's menu selection. For more information, see Pulldown and Popup Menus.

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Warning

The use of a Menu System that was not created properly, or that was damaged by the improper use of the MenuSystemDestroy function, can result in an Application Error (a General Protection Fault).

Syntax

```
lResult& = PopupMenu(lMenuNumber&, _
                     lCellAlignment&, _
                     lWhichSide&)
```

Parameters

lMenuNumber&

The Menu Buffer number of the top-level menu to be displayed, from 1 to the value that was used for lMenuBuffers& in the InitConsoleTools function.

lCellAlignment&

If you use zero (0) for this parameter, the menu will appear on the screen with one corner of the popup positioned at the current mouse cursor location. If you use `%ALIGNDOWN` it will appear *near* the mouse cursor, but moved very slightly downward so that the entire row of text that was clicked will be visible. If you use `%ALIGNLEFT` it will be moved very slightly to the left, so that it is aligned with the left edge of the column that was clicked. If you use `%ALIGNRIGHT` it will be aligned with the right edge of the column that was clicked. You can also use combinations like `%ALIGNDOWN+%ALIGNLEFT`.

lWhichSide&

If you use zero (0) for this parameter, Windows will usually display the menu so that the top-left corner of the popup is positioned at the current mouse cursor location. If you use `%LEFTSIDE` it will be displayed so that the top-right corner of the popup is positioned at the mouse cursor location, i.e. the menu will usually be displayed to the left of the mouse cursor. (See **Remarks** below for more information.)

Return Value

lResult& will be one of the following values:

A value from 100 to 32750 which corresponds to the selection of a menu item. Menu item numbers are assigned by your program, using values in strings that are submitted to the MenuDefinition function.

`%POPUP_NONE` (value 29) if the user clicks on something other than the popup menu, or presses the Escape key, or presses the Enter key when no menu item is highlighted, or performs any other action (such as pressing an Alt key) which causes the popup menu to disappear before an item is selected.

`%ERROR_CT_INVALIDMENUNUMBER` if a value for lMenuNumber& is used that is outside the range of Menu Buffer numbers specified in the InitConsoleTools function.

`%ERROR_CT_MENUDOESNOTEXIST` if you use the number of a Menu Buffer that has not been created (without errors) by the PopUpSystemCreate function.

`%ERROR_CT_UNKNOWNERROR` if Windows refused to allow the display of the menu on the screen. This is usually related to extremely low memory.

Remarks

If you use an ampersand (&) in a Menu Definition string to underline a letter, the popup menu will respond to the corresponding key. For example, if you use &File=100 in a Menu Definition string, the resulting menu item will be **File**. Pressing the F key would then have the same effect as selecting the File item with the mouse.

Unlike Pulldown Menus, Popup Menus do not respond to Accelerator Keys. It is not possible to select an item from a popup menu or submenu by using a Ctrl-key.

The lWhichSide& parameter (see above) can be used to specify which side you *prefer* the popup menu to favor, but Windows not always comply with that request. For example, if you use the default mode (zero) so that popups are usually displayed to the right of the mouse cursor, but you click a screen location that is too close to the right edge of the desktop, Windows will ignore the lWhichSide& parameter and display the popup on the left side of the mouse cursor.

Example

```
'This example assumes that Menu Buffer #1
'contains a properly-created top-level menu.

IF INKEY$ = CHR$(255,255,8,2) THEN

    'The user right-clicked the console.
    'Display top-level menu #1 as a popup...

    lUserSelection& = PopupMenu(1, 0, 0)

    IF lUserSelection& = %Pulldown_None THEN
        'user did not make a selection
    ELSE
        'user selected a menu item that was
        'assigned the number lUserSelection&.
    END IF

END IF
```

See Also

Pulldown and Popup Menus, MenuDefinition, PopupSystemCreate

PopUpSystemCreate

Purpose

Builds a complete, interconnected Popup Menu System from strings that were submitted to the MenuDefinition function. The menu system that is created by this function can then be displayed with the PopupMenu function. (To create a Pulldown Menu instead of a Popup Menu, use MenuSystemCreate instead of PopUpSystemCreate.)

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Syntax

```
lResult& = PopUpSystemCreate(lFirstBuffer&, _
                             lLastBuffer&)
```

Parameters

lFirstBuffer& and *lLastBuffer&*

The range of Menu Buffers that is to be processed into a Popup Menu System. The lowest legal number for either parameter is 1, and the largest is the value that was used for the lMenuBuffers& parameter of the InitConsoleTools function. If lLastBuffer& is less than or equal to lFirstBuffer&, only lFirstBuffer& will be processed.

Return Value

lResult& will be one of the following values...

`%ERROR_CT_INVALIDMENUNUMBER` if the Console Tools Menu System was not initialized properly with InitConsoleTools or if the lFirstBuffer& or lLastBuffer& parameter is not between 1 and the number of Menu Buffers that were specified with InitConsoleTools.

`%ERROR_CT_INVALIDMENUSYSTEM` if the PopUpSystemCreate function was unable to create the Menu System because of **1)** a "circular reference" such as Menu 1 having Menu 2 as a submenu, which then has Menu 1 as a submenu, or **2)** an error in a Menu Definition String. The proper use of the MenuDefinition function -- especially checking the return value -- will avoid most problems.

`%ERROR_CT_UNKNOWNERROR` if Windows refused to allow the creation of a new (blank) menu. Windows does not provide information about why it refused, but we assume that it relates to insufficient memory. (This should be a very rare problem, considering the relatively small amount of memory that blank menus require.)

`%SUCCESS` (zero) if none of the problems above were detected.

Remarks

The first step in building a Popup Menu System involves the MenuDefinition function. The PopUpSystemCreate function can only be used after the error-free use of MenuDefinition to fill at least one Menu Buffer.

See Using Pulldown and Popup Menus for a complete description of this function.

Example

```
lResult& = PopUpSystemCreate(16,32)
```

This example would build a PopUp Menu System using Menu Buffers 16 through 32.

See Also

MenuDefintion, MenuItemProperty

ProgressBoxCancel

Purpose

Returns True/False to indicate whether or not a Progress Box's Cancel button has been selected.

Availability

Console Tools Standard and Pro

Warning

This is a "read-once" function. See Remarks below.

Syntax

```
lResult& = ProgressBoxCancel
```

Parameters

None.

Return Value

lResult& will be `%FALSE` (0) if the current Progress Box's Cancel button has not been selected, or Logical True if it has.

Remarks

"Selected" means that **1)** the Cancel button was clicked or that **2)** when the Progress Box had the Keyboard Focus, the user pressed the Enter key, the Space Bar, or Alt-C, the Cancel button's hot key.

IMPORTANT NOTE: The ProgressBoxCancel function automatically resets itself to `%FALSE` each time the function is used, so it will return Logical True the *first* time it is checked after the Cancel button has been selected, and `%FALSE` in subsequent checks. This is to allow a program like the example below to use the ProgressBoxCancel function repeatedly, if the user selects the No Button on the ConsoleMessageBox.

Example

```
'See ProgressBoxShow for an
'explanation of that function...
ProgressBoxShow %CANCELBUTTON, _
                0, _
                %CONSOLE_CENTER, _
                %CONSOLE_CENTER, _
                "Click Cancel to Interrupt...", _
                "Progress of Job", 0

FOR lCount& = 0 TO 100
    ProgressBoxUpdate lCount&
    DELAY 0.2   'slow down this demo a little
    IF ProgressBoxCancel THEN
       'user selected Cancel
       lResult& = ConsoleMessageBox("Cancel?",_
                                    %YESNO, _
                                    "Cancel?", _
                                    %DEFAULT, _
                                    0)
       IF lResult& = %YESBUTTON THEN
              EXIT FOR
       END IF
    END IF
NEXT

'now get rid of the Progress Box...
ProgressBoxHide
```

Note that this routine works properly, in part, because the ProgressBoxCancel resets itself to %FALSE every time the function is used. If it didn't, the user would not be able to select the No button and continue with the "job".

See Also

ProgressBoxDefaults, ProgressBoxUpdate, ProgressBoxShow, ProgressBoxHide

ProgressBoxDefaults

Purpose

Establishes default values for future Console Tools Progress Boxes.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ProgressBoxDefaults lType&, _
                    lPercent&, _
                    lXPos&, _
                    lYPos&, _
                    sText$, _
                    sTitle$, _
                    lNormalize&
```

Parameters

lType&

Use the predefined equate `%CANCELBUTTON` for a Progress Box with a Cancel Button, or `%NOCANCEL` for one without. You can also optionally add the predefined equate `%TOPMOST` to produce a Progress Box that will appear on top of all other windows, even if the console window is not currently visible.

lPercent&

This parameter is ignored. It is not useful to set a default value for lPercent& because this is the parameter that changes most often.

lXPos& and *lYPos&*

The screen location where the upper-left corner of the Progress Box will be displayed. Use zero (0) to leave the position of an existing Progress Box alone, or use a number other than zero to specify a screen location based on 0,0 at the top-left of the screen. You can also use `%DESKTOP_CENTER` to auto-center the Progress Box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the Progress Box in the middle of the console. Also see LocOfCol and LocOfRow for a technique that allows Progress Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Progress Box "off the screen" and make it impossible for the user to see it or to click the Cancel button.*

sText$

The text that will be displayed just above the Progress Box's Progress Bar. You can use Shorthands in this string.

sTitle$

The text that will be shown in the title bar of the Progress Box and (optionally) the Progress Box's button.

To change the text that is displayed in the Progress Box's button, use a string like this for *sTitle$*:

```
                "Title" + CHR$(0) + "Button1"
```

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Progress Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Progress Box, to make sure that the console screen is visible to the user.

Return Value

None.

Remarks

The ProgressBoxDefaults function can be used to set or change the *default* values for Console Tools Progress Boxes. If you use this function to set one or more defaults, you can then use `%DEFAULT` (for numeric parameters) or an empty string (for string parameters) when calling the ProgressBoxShow function, and the predefined default(s) will be used. For example, if you wanted all (or most) of the Progress Boxes in your program to use the same title bar text, you'd use ProgressBoxDefaults to set that text as the default title. Then any Progress Box with "" as the sTitle$ parameter would automatically use the default title. (If you used ProgressBoxShow with something other than "" as the sTitle$ parameter, the default would be ignored for *that* Progress Box.)

If you want to change Progress Box defaults that you have already set, you can use ProgressBoxDefaults more than once. If you want to change one default setting but not the others, use `%DEFAULT` or an empty string when using ProgressBoxDefaults (see last Example).

Example

```
'set up defaults for all parameters...
ProgressBoxDefaults %CANCELBUTTON, _
                   0, _
                   %CONSOLE_CENTER, _
                   %CONSOLE_CENTER, _
                   "Progress of Job...", _
                   "WORKING", _
                   %FALSE

'use all of the defaults in a Progress Box...
ProgressBoxShow %DEFAULT, _
                %DEFAULT, _
                %DEFAULT, _
                %DEFAULT, _
                "" , _
                "" , _
                %DEFAULT

'use some of the defaults in a Progress Box...
ProgressBoxShow %NOCANCEL, _
                %DEFAULT, _
                %DEFAULT, _
                %DEFAULT, _
                "" , _
                "PLEASE WAIT!", _
                %DEFAULT

'change the default title bar text but nothing else...
ProgressBoxDefaults %DEFAULT, _
                   %DEFAULT, _
                   %DEFAULT, _
                   %DEFAULT, _
                   "", _
                   "WORKING VERY HARD", _
                   %DEFAULT
```

See Also

ProgressBoxCancel, ProgressBoxUpdate, ProgressBoxShow, ProgressBoxHide

ProgressBoxHide

Purpose

Removes a Progress Box from the screen.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ProgressBoxHide
```

Parameters

None.

Return Value

None.

Remarks

Progress Boxes are "non-modal". That means that you can put one on the screen and then your program can go do something else. (A Message Box, on the other hand, is a good example of "modal". Your program stops until the user selects a button, then the Message Box goes away and your program continues.)

ProgressBoxHide is used to remove a Progress Box from the screen when your program is done with it.

Example

See ProgressBoxShow for an example that demonstrates all of the Progress Box functions.

See Also

ProgressBoxCancel, ProgressBoxUpdate, ProgressBoxDefaults, ProgressBoxShow

ProgressBoxRefresh

Purpose

In Console Tools version 1.00, this function was used to periodically "refresh" a Progress Box display. Console Tools 2.00 provides Progress Boxes that refresh themselves automatically, so the use of this function is no longer required. Console Tools 2.00 contains a "do nothing" ProgressBoxRefresh function, for backward compatability.

Availability

Console Tools Standard and Pro

Syntax

```
ProgressBoxRefresh
```

See Also

ProgressBoxCancel, ProgressBoxUpdate, ProgressBoxDefaults, ProgressBoxShow, ProgressBoxHide

ProgressBoxShow

Purpose

Displays a Progress Box, or updates an existing Progress Box.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ProgressBoxShow lType&, _
                lPercent&, _
                lXPos&, _
                lYPos&, _
                sText$, _
                sTitle$, _
                lNormalize&
```

(Also see Simplified Syntax below.)

Parameters

lType&

Use the predefined equate `%CANCELBUTTON` for a Progress Box with a Cancel Button, or `%NOCANCEL` for one without. You can also optionally add the predefined equate `%TOPMOST` to produce a Progress Box that will appear on top of all other windows, even if the console window is not currently visible. If you have previously used the ProgressBoxDefaults function to establish a default type, you can also use `%DEFAULT` for this parameter.

lPercent&

A number from 0 to 100, indicating the Percentage that should be displayed. (Actually, the Progress Box's *text* indicator can display overflow values from 101-999. The *graphical* indicator is strictly limited to 0-100.) Note that this parameter is unusual because you can *not* use `%DEFAULT`. It would not be useful to do so, because the lPercent& parameter is the one that changes most often.

lXPos& and *lYPos&*

The screen location where the upper-left corner of the Progress Box will be displayed. Use zero (0) to leave the position of an existing Progress Box alone, or use a number other than zero to specify a screen location based on 0,0 at the top-left of the screen. You can also use `%DESKTOP_CENTER` to auto-center the Progress Box in the middle of the desktop, or `%CONSOLE_CENTER` to auto-center the Progress Box in the middle of the console. Also see LocOfCol and LocOfRow for a technique that allows Progress Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Progress Box "off the screen" and make it impossible for the user to see it or to click the Cancel button.* If you have previously used the ProgressBoxDefaults function to establish a default position, you can also use `%DEFAULT` for these parameters.

sText$

The text that will be displayed just above the Progress Box's Progress Bar. You can use Shorthands in this string. If you have previously used the ProgressBoxDefaults function to establish a default text string, you can also use an empty string for this parameter, and the default text will be used.

sTitle$

The text that will be shown in the title bar of the Progress Box and (optionally) the Progress Box's button.

To change the text that is displayed in the Progress Box's button, use a string like this for *sTitle$*:

```
"Title" + CHR$(0) + "Button1"
```

If you have previously used the ProgressBoxDefaults function to establish a default title, you can also use an empty string this parameter, and the default title and button label will be used.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Progress Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Progress Box, to make sure that the console screen is visible to the user. IMPORTANT NOTE: Remember that Progress Boxes are usually displayed over and over as a job progresses -- at least 100 times -- so using the lNormalize& option will force the constant re-display of the console window. If you want your user to be able to use other programs while the Progress Box is being updated, you should use `%FALSE` for this parameter. (If you have previously used the ProgressBoxDefaults function to establish a default value, you can also use `%DEFAULT` for this parameter.)

Return Value

None.

Remarks

The ProgressBoxShow function is always used to *create* a Progress Box. After it appears on the screen, however, there are two different ways to change the way a Progress Box looks...

1) If all you want to do is change the "percent done" part of the display, we recommend that you use the simple ProgressBoxUpdate function.

2) If you want to change the text, title bar, or screen location of a Progress Box, you should use the ProgressBoxShow function more than once.

If it is used repeatedly to change a Progress Box, the ProgressBoxShow function will execute more quickly if you do not change certain parameters.

For example, if you specify a screen location the first time a Progress Box is displayed (like `%CONSOLE_CENTER` in the example below) and then use zero (0) for the location parameters in the rest of the program (also as shown in the example), the ProgressBoxShow function will know that it does not need to move the Progress Box every time it is displayed. This also allows users to drag the Progress Box out of their way, if they don't like the location you chose. Note also that `%DESKTOP_CENTER` cannot be used repeatedly. If you want to force your Progress Box to be located in the middle of the screen, and not allow it to be moved, you will need to calculate a screen location (using the ConsoleInfo function) and give the ProgressBoxShow function a nonzero lXPos& and lYPos& value every time the function is used.

The same "leave it alone" strategy is valid for *all* of the ProgressBoxShow parameters *except for lPercent&*. If a parameter is simply "the same as last time" you should use zero or an empty string so the ProgressBoxShow function will execute faster.

There is an additional benefit to not constantly setting certain Progress Box parameters. For example, if you specify a value for the title bar (sTitle$) every time the ProgressBoxShow function is used, the title bar will be blanked out and redisplayed over and over, resulting in a slight flicker. (That's just the way Windows works; Console Tools does not actually *tell* the title bar to clear before it is redisplayed.)

See Progress Boxes for a general discussion of this function.

Example

```
ProgressBoxShow %CANCELBUTTON, _
                0, _
                %CONSOLE_CENTER, _
                %CONSOLE_CENTER, _
                "Click Cancel to Interrupt...", _
                "Progress of Job", _
                %TRUE

FOR lCount& = 0 TO 100
    ProgressBoxUpdate lCount&
    DELAY 0.2   'slow down this demo a little
    IF ProgressBoxCancel THEN
       EXIT FOR
    END IF
NEXT


'This next loop will execute somewhat more slowly because
'every parameter is specified every time ProgressBoxShow
'is used...

FOR lCount& = 0 TO 100
    ProgressBoxShow %CANCELBUTTON, _
                    lCount&, _
                    1, _
                    1, _
                    "Click Cancel to Interrupt...", _
                    "Progress of Job", _
                    %TRUE
    DELAY 0.2   'same delay as previous example
    IF ProgressBoxCancel THEN
       EXIT FOR
    END IF
NEXT

'now get rid of the Progress Box...
```

```
ProgressBoxHide
```

Details

See Progress Boxes for a general discussion of Progress Box functions.

Simplified Syntax

The Console Tools Progress Box has options that you probably won't need to change very often, so we suggest that you design one or more simple "wrapper" functions that eliminate the parameters that you won't usually use. For example, you could add this to your program...

```
SUB PROGRESSBOX(lPercent&)

      ProgressBoxShow %NOCANCEL, _
                      lPercent&, _
                      %CONSOLE_CENTER, _
                      %CONSOLE_CENTER, _
                      "Progress Of Job", _
                      "WORKING", _
                      %FALSE
END SUB
```

You could then use a much easier PROGRESSBOX syntax for most Progress Boxes.

```
'show progress box at 50%
PROGRESSBOX 50
```

See Also

ProgressBoxCancel, ProgressBoxUpdate, ProgressBoxDefaults, ProgressBoxHide

ProgressBoxUpdate

Purpose

Changes the "percent done" display of a Progress Box that was created with the ProgressBoxShow function.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
ProgressBoxUpdate lPercent&
```

Parameters

lPercent&

The percentage that should be displayed in the Progress Box. For complete details, see ProgressBoxShow.

Return Value

If this function is used before a Progress Box has been created with ProgressBoxShow, the value `%ERROR_CT_CANTBEDONE` will be returned. Otherwise, `%SUCCESS` (zero) will be returned. It is therefore usually safe to ignore the return value of this function.

Remarks

For a complete discussion of Progress Boxes, see ProgressBoxShow.

Example

See ProgressBoxShow for an example that demonstrates all of the Progress Box functions.

See Also

ProgressBoxCancel, ProgressBoxDefaults, ProgressBoxShow

PulldownMenu

Purpose

Displays a Pulldown Menu that was previously created with the MenuDefinition and MenuSystemCreate functions, and returns a value to indicate the user's menu selection. For more information, see Pulldown and Popup Menus.

Availability

Console Tools Pro Only (see)

Warning

Your program must use an appropriate `lMenuBuffers&` value in the InitConsoleTools function before it can use this function.

Warning

The use of a Menu System that was not created properly, or that was damaged by the improper use of the MenuSystemDestroy function, can result in an Application Error (a General Protection Fault).

Syntax

```
lResult& = PulldownMenu(lMenuNumber&, _
                        lOption&, _
                        lHighlight&)
```

Parameters

lMenuNumber&

The Menu Buffer number of the top-level menu to be displayed, from 1 to the value that was used for lMenuBuffers& in the InitConsoleTools function.

lOption&

Unless you are using a nonstandard console size, you should normally use zero (0) for this parameter. See **Remarks** below.

lHighlight&

Use `%OFF` for a menu that does not automatically highlight its first item, or use `%ON` for a menu that does. The normal value for this parameter is `%OFF` unless you are creating a DOS-style menu. (To see the highlight turn on and off, tap the Alt key when a pulldown menu is showing.) Whenever the highlight is on, Accelerator keys will not work until the user turns the highlight off by pressing Escape once or by tapping the Alt key.

Return Value

lResult& will be one of the following values:

A value from 31 to 90 which corresponds to the use of an Accelerator Key *or* to the selection of a menu item that was set up to accept that Accelerator Key. Accelerator Keys can include the function keys F1 through F12, Ctrl-A through Ctrl-Z, and Ctrl-0 through Ctrl-9. Example: If the PulldownMenu function returns a value of 31 (which corresponds to the `%PULLDOWN_F1` equate) that means that a user pressed the F1 key *or* selected a menu item that was assigned the value 31.

A value from 100 to 32750 which corresponds to the selection of a menu item. Menu item numbers are assigned by your program, using values in strings that are submitted to the MenuDefinition function.

`%PULLDOWN_CLOSE` (value 30) if the user clicks the Close (**X**) button, double-clicks the console icon, selects Close from the Console's Window Menu or presses Alt-F4.

`%PULLDOWN_ENTER` (value 13) if the user presses the Enter key.

`%PULLDOWN_ESCAPE` (value 27) if the user presses the Escape key

`%PULLDOWN_PGUP` (value 97) or `%PULLDOWN_PGDN` (value 98) if the user presses the PgUp (Page Up) or PgDn (Page Down) key.

`%PULLDOWN_ALTENTER` (value 99) if the user presses Alt-Enter.

`%PULLDOWN_NONE` (value 29) if the user clicks on the console window "print area". If you want your program to be able to detect which row and column was clicked, see **Details** below.

`%ERROR_CT_INVALIDMENUNUMBER` if a value for lMenuNumber& is used that is outside the range of Menu Buffer numbers specified in the InitConsoleTools function.

`%ERROR_CT_MENUDOESNOTEXIST` if you use the number of a Menu Buffer that has not been created (without errors) by the MenuSystemCreate function.

`%ERROR_CT_UNKNOWNERROR` if Windows refused to allow the display of the menu on the screen. This is usually related to extremely low memory.

Remarks

If your program will be run on Windows 95/98/ME computers, you will probably want to use the ConsoleToolbar function to hide the Console Toolbar before you use the PulldownMenu function. If you don't, the Pulldown Menu will *partially* cover up the Windows 95/98/ME Toolbar, and on most computers the end result is not very "clean". It's also a good idea to use the DeleteWindowMenuItem function to remove `%MENUITEM_TOOLBAR` from the console Window Menu, so that the user can't re-activate the Toolbar manually. (Even if your program uses the Pulldown Menu full-time, thereby blocking the normal methods of accessing the Window Menu, it's possible for a user to right-click on your program's task bar button and re-activate the toolbar. Removing the menu item makes that impossible.)

If your program uses the DeleteWindowMenuItem function to remove the Close item from the console window's Window Menu, you might be surprised to see the Close item reappear when a Console Tools Pulldown Menu is on the screen. The Pulldown Menu function automatically re-activates the Close item *temporarily*, whenever the function is used. Because the Pulldown Menu can intercept the Close command (Alt-F4 etc.), the command does *not* have the power to close your program in the middle of an important operation, as it normally does. Your program can look for the value `%PULLDOWN_CLOSE` and decide whether or not to allow the program to Close, then exit gracefully. See Console Windows for more information.

You might also be surprised that the Alt-Enter Windows Hotkey (which toggles the console window between the FullScreen and Windowed Mode) is intercepted. If you want to allow your program to switch modes you should use the ToggleFullScreen function when the value `%PULLDOWN_ALTENTER` is returned. IMPORTANT NOTE: Pulldown Menus can not be used when the console window is in the FullScreen mode, so your program should probably display a ConsoleMessageBox informing the user that the FullScreen Mode is not available.

If your program's console window is in the FullScreen mode when the PulldownMenu function is used, the function will automatically switch the screen back to the Windowed Mode. Windows does not allow the use of pulldown menus (other than purely text-based menus) in the FullScreen Mode.

If your program is already in the Windowed Mode (i.e. not FullScreen) the PulldownMenu function will perform a ConsoleWindow `%SHOW` operation before displaying the menu, to make sure that the console is not hidden.

It is your responsibility to make sure that your program is otherwise visible, i.e. not minimized, not located off the edge of visible screen, and so on. We suggest the routine use of the ConsoleNormal function before the PulldownMenu function is used for the first time.

The lOption& Parameter

Unfortunately, when operations described in the previous three paragraphs are performed, they can, under certain circumstances, have an undesirable effect on the console window. In particular, if you are using nonstandard console row and/or column counts (such as a 100x30 console instead of a standard size like 80x25) on a Windows 95, 98, or ME computer, the console's size may change unexpectedly.

To disable the PulldownMenu function's Window State checking, use one (1) for the *lOption&* parameter. This will usually result in a stable console on Windows 95/98/ME computers when nonstandard console sizes are used.

Example

```
'This example assumes that Menu Buffer #1
'contains a properly-created top-level menu.

ConsoleToolbar %OFF,%SHOW

DO

    'display top-level menu #1...

    lUserSelection& = PulldownMenu(1, 0, %OFF)

    IF lUserSelection& = %Pulldown_Escape THEN
           'user pressed Escape

    ELSEIF lUserSelection& = %Pulldown_Enter THEN
           'user pressed Enter

    ELSEIF lUserSelection& = %Pulldown_AltEnter THEN
           'user selected Alt-Enter.

    ELSEIF lUserSelection& = %Pulldown_Close THEN
           'user used a standard Windows Close function

    ELSEIF lUserSelection& = %Pulldown_None THEN
           'user clicked on console window "print area"
           'See Details below for more information.

    ELSEIF lUserSelection& = %Pulldown_Q THEN
           'THIS IS THE "EMERGENCY EXIT"
```

```
               'REMOVE THIS AFTER TESTING!
               'user pressed Ctrl-Q
                EXIT FUNCTION  'exit COMPLETELY during testing

        ELSEIF lUserSelection& => %Pulldown_A AND _
               lUserSelection& <= %Pulldown_Z THEN
               'user pressed an Accelerator key from
               'Ctrl-A to Ctrl-Z or selected a menu
               'item with the same value.

        ELSEIF lUserSelection& => %Pulldown_0 AND _
               lUserSelection& <= %Pulldown_9 THEN
               'user pressed an Accelerator key from
               'Ctrl-0 to Ctrl-9 or selected a menu
               'item with the same value.

        ELSEIF lUserSelection& => %Pulldown_F1 AND _
               lUserSelection& <= %Pulldown_F12 THEN
               'user pressed an Accelerator key from
               'F1 to F12 (without using the Ctrl key)
               'or selected a menu item with the same value.

        ELSEIF lUserSelection& => %ERROR_CT_FIRSTERROR AND _
               lUserSelection& <= %ERROR_CT_LASTERROR THEN
               'lUserSelection& contains an error number

        ELSE
               'user selected a menu item that was
               'assigned the number lUserSelection&.

        END IF

    LOOP
```

Details

For general information about the PulldownMenu function, see Using Pulldown Menus.

If you want your users to be able to click on the console window, and for your program to be able to detect which row/column was clicked, you'll need to use the Console Tools MouseOverRow and MouseOverCol function instead of the PB/CC MOUSEY and MOUSEX functions. Here's why:

When a Pulldown Menu is on the screen and a user clicks on the console window, the click is detected and handled by Console Tools. In the process, the click is "cleared", so it is no longer available to be detected by your program. That means that the PB/CC INKEY$ and WAITKEY$ functions can't detect the click, and the MOUSEX and MOUSEY functions can't tell you which row/column was clicked.

You can, however, use the MouseOverRow and MouseOverCol functions to determine where the mouse cursor is currently located. Locate the part of your program that handles the `%PULLDOWN_NONE` return value (see example above) and have your program check the values of MouseOverRow and MouseOverCol at that point. See MouseOverCol and MouseOverRow for more information.

Please note that it is not possible to use this technique to distinguish between single and double clicks, or to detect which mouse button (left, right, or middle) was clicked.

See Also

MenuSystemCreate, MenuDefintion, MenuItemProperty, MenuSystemDestroy

RowOfLoc

Purpose

Returns the console row number that corresponds to a screen location.

Availability

Console Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = RowOfLoc(lYPos&)
```

Parameters

lYPos&

The screen location, in pixels, based on 0,0 at the top-left of the screen, for which you want the console row number.

Return Value

If the y-axis (top-to-bottom) value that you specify for lYPos& corresponds to the y-axis location of a console row, this function will return the row number. (Keep in mind that a *range* of screen locations will correspond to each row. The size of the range will depend on the height of the console's rows.)

If the value that you specify for lYPos& is located either **1)** above the top of the console or **2)** below the bottom of the console, this function will return the theoretical row number. For example, if you specify a screen location that is located above the top of the console, this function will return a negative number that indicates the "imaginary" row number of the specified location.

Remarks

RowOfLoc stands for Row Of Location. Compare this function to the LocOfRow (Location of Row) function.

See Also

ColOfLoc

SplashBoxDefaults

Purpose

Establishes default settings for future Splash Boxes.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
SplashBoxDefaults lType&, _
                  lDelay&, _
                  lXPos&, _
                  lYPos&, _
                  sLine1$, _
                  sLine2$, _
                  sLine3$, _
                  lIconID&, _
                  lNormalize&
```

Parameters

lType&

Use 1 for a one-line Splash Box, 2 for a two-line Splash Box, 3 for a three-line Splash Box, or 4 for a one-line Splash Box that automatically word-wraps to display up to four lines. The numbers 5 through 8 can also be used, to produce half-width versions of types 1 through 4. You can also optionally add one of the predefined equates `%BOLD` or `%MONOSPACE` to the lType& value to change the font to "System" or "Fixed Sys". respectively. Console Tools Pro users can add `%CUSTOMFONT` to specify the Console Tools Pro Custom Font.

lDelay&

The number of seconds (from 0 to 60) or milliseconds (from 61 to 60,000) that you want the Splash Box to be "guaranteed" (see SplashBoxShow Remarks).

lXPos& and *lYPos&*

The screen location where the top-left corner of the Splash Box should be located. Use zero (0) to leave the position of an existing Splash Box alone, or use a nonzero number to specify a screen location relative to the top-left corner of the screen at 0,0. You can also use `%DESKTOP_CENTER` to auto-center the Splash Box on the desktop, or `%CONSOLE_CENTER` to auto-center the Splash Box in the middle of the console. Also see LocOfCol and LocOfRow for a technique that allows Splash Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Splash Box "off the screen" and make it impossible for the user to see it.*

sLine1$*, *sLine2$, and *sLine3$*

The text that will be displayed in the Splash Box. If you specify text for a line that does not exist (such as using sLine2$ with a one-line Splash Box) it will be ignored. Keep in mind that Type 4 (see above) is a one-line Splash Box, even though it appears to display multiple lines. You can use the Quote Shortcut (\q) in these strings but the other shortcuts (\t, \n etc.) will be ignored.

lIconID&

Either **1)** The value `%IDI_CONSOLE` for the Console Tools Icon, or **2)** the value `%IDI_CUSTOM` for an icon that was previously loaded with the CustomIcon function, or **3)** one of the values `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, or `%IDI_WINLOGO` to specify a Windows Standard Icon or **4)** the ID Number of an icon in a resource file that was linked into your PB/CC program with the `$RESOURCE` metastatement. See Using Icons for complete information.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Splash Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Splash Box, to make sure that the console screen is visible to the user.

Return Value

None.

Remarks

The SplashBoxDefaults function can be used to set or change the *default* values for Console Tools Splash Boxes. If you use this function to set one or more defaults, you can then use `%DEFAULT` (for numeric parameters) or an empty string (for string parameters) when calling the SplashBoxShow function, and the predefined default(s) will be used. For example, if you wanted all (or most) of the Splash Boxes in your program to be one-line Splash Boxes that use the same bold font, you'd use SplashBoxDefaults to set the default type to `1+%BOLD`. Then any SplashBoxShow with `%DEFAULT` as the lType& parameter would automatically use `1+%BOLD`. (If you used SplashBoxShow with something other than `%DEFAULT` as the lType& parameter, the default would be ignored for *that* Splash Box.)

If you want to change Splash Box defaults that you have already set, you can use SplashBoxDefaults more than once. If you want to change one default setting but not the others, use `%DEFAULT` or an empty string when using SplashBoxDefaults (see last Example).

Examples

```
SplashBoxDefaults 3+%BOLD, _
                  10, _
                  1, _
                  1, _
                  "Console Tools", _
                  "Splash Box", _
                  "Defaults", _
                  0, _
                  %TRUE

'display a Splash Box using
'all of the default values...
SplashBoxShow  %DEFAULT, _
               %DEFAULT, _
               %DEFAULT, _
               %DEFAULT, _
               "" , _
               "" , _
               "" , _
               %DEFAULT, _
               %DEFAULT
SplashBoxHide

'display a Splash Box using some
'of the defaults...
SplashBoxShow  %DEFAULT, _
               %CONSOLE_CENTER, _
               %CONSOLE_CENTER, _
               %DEFAULT, _
               "" , _
               "is a great", _
               "developer's tool", _
               %DEFAULT, _
               %DEFAULT
SplashBoxHide

'change the default font but leave
'the other defaults unchanged...
SplashBoxDefaults 3+%MONOSPACE, _
                  %DEFAULT, _
                  %DEFAULT, _
                  %DEFAULT, _
                  "" , _
                  "" , _
                  "" , _
                  %DEFAULT, _
                  %DEFAULT
```

See Also

SplashBoxShow, SplashBoxHide, SplashBoxRefresh

SplashBoxHide

Purpose

Removes a Splash Box from the screen.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
SplashBoxHide
```

Parameters

None.

Return Value

None.

Remarks

Splash Boxes are "non-modal". That means that you can put one on the screen and then your program can go do something else. (A Message Box, on the other hand, is a good example of "modal". Your program stops until the user selects a button, then the Message Box goes away and your program continues.)

SplashBoxHide is used to remove a Splash Box from the screen when it is no longer needed.

Example

See SplashBoxShow for an example that uses SplashBoxHide.

See Also

SplashBoxShow, SplashBoxDefaults. SplashBoxRefresh

SplashBoxRefresh

Purpose

"Refreshes" the display of a Splash Box that was created with SplashBoxShow.

Availability

Console Tools Standard and Pro

Warnings

None.

Syntax

```
SplashBoxRefresh
```

Parameters

None.

Return Value

None.

Remarks

Splash Boxes, after their "guaranteed" delay period (see SplashBoxShow) are "non-modal". That means that you can put one on the screen and then your program can go do something else. (A Message Box, on the other hand, is a good example of "modal". Your program stops until the user selects a button, then the Message Box goes away and your program continues.)

Normally, Windows programs automatically "refresh" their own displays. For example, normal Windows programs "know" when another application is covering them up, and when they have been revealed again, and they refresh their displays automatically. If they didn't, all the user would see would be group of blank rectangles on the screen when a program was uncovered.

Unfortunately, Console Applications are different. Because Microsoft never really intended for Console Applications to use graphical elements like Splash Boxes, there is no auto-refresh system.

This is not usually a problem. It only becomes an issue if your program does not use the Splash Box "guarantee" feature (see SplashBoxShow), or if your program "goes away for a long time" while a Splash Box is on the screen.

In the example below, the program "guarantees" the Splash Box display for thirty seconds, then BEEPs, and then simulates being busy for thirty more seconds, during which time the Splash Box is not refreshed. If the program were to be covered up by another application and then revealed during the second thirty seconds, the Splash Box would appear as a gray rectangle. (Try the example and see for yourself. Run the example, then click-and-drag another program's window back and forth over the Splash Box. IMPORTANT NOTE: Because Console Tools Splash Boxes are given the TOPMOST property, most programs cannot cover them up. Certain types of programs, however (such as the PB/CC debugger, or this Help File with the Options/KeepHelpOnTop feature turned on) *can* cover up a TOPMOST window because they too have the TOPMOST property.

To get around the fact that Windows does not automatically refresh Console Applications, Console Tools provides the SplashBoxRefresh function. It executes much more quickly than repeated uses of SplashBoxShow (which would accomplish the same thing). Remove the word `REM` and run the example again, and you'll see a much cleaner display when the Splash Box is hidden and revealed by other TOPMOST programs during the second thirty seconds.

Example

Please note that this example takes 60 seconds to run. For the first 30 seconds, the Splash Box "guarantee" delay is in effect. (That's the 30 in the second parameter of SplashBoxShow.) Then the program will BEEP and for the next 30 seconds, if the REM has not been removed, it will be possible for another TOPMOST window to "erase" the Splash Box. With the REM removed, however, the Splash Box is constantly refreshed and never appears blank for more than 0.5 seconds.

```
SplashBoxShow  3+%BOLD, _
               30, _
               1, _
               1, _
               "Console Tools", _
               "Splash Box", _
               "\qGUARANTEE\q", _
               0, _
               %FALSE
BEEP
CALL BusyForThirtySeconds
SplashBoxHide

'----------------------

SUB BusyForThirtySeconds
    DIM X AS LOCAL LONG
    FOR X = 1 TO 60
        DELAY 0.5
        REM SplashBoxRefresh
    NEXT
END SUB
```

See Also

SplashBoxShow, SplashBoxHide, SplashBoxDefaults

SplashBoxShow

Purpose

Displays a Console Tools Splash Box, or updates an existing Splash Box.

Availability

Console Tools Standard and Pro
(Some features are limited to the Pro Version.)

Warnings

None.

Syntax

```
SplashBoxShow lType&, _
              lDelay&, _
              lXPos&, _
              lYPos&, _
              sLine1$, _
              sLine2$, _
              sLine3$, _
              lIconID&, _
              lNormalize&
```

(Also see Simplified Syntax below.)

Parameters

lType&

Use 1 for a one-line Splash Box, 2 for a two-line Splash Box, 3 for a three-line Splash Box, or 4 for a one-line Splash Box that automatically word-wraps to display up to four lines. The numbers 5 through 8 can also be used, to produce half-width versions of types 1 through 4. You can also optionally add one of the predefined equates `%BOLD` or `%MONOSPACE` to the lType& value to change the font to "System" or "Fixed Sys". respectively. Console Tools Pro users can add `%CUSTOMFONT` to specify the Console Tools Pro Custom Font.

lDelay&

The number of seconds (from 0 to 60) or milliseconds (from 61 to 60,000) that you want the Splash Box to be "guaranteed" (see Remarks below).

lXPos& and *lYPos&*

The screen location where the top-left corner of the Splash Box should be located. Use zero (0) to leave the position of an existing Splash Box alone, or use a nonzero number to specify a screen location relative to the top-left corner of the screen at 0,0. You can also use `%DESKTOP_CENTER` to auto-center the Splash Box on the desktop, or `%CONSOLE_CENTER` to auto-center the Splash Box in the middle of the console. Also see LocOfCol and LocOfRow for a technique that allows Splash Boxes to be positioned at specific row/column locations. *WARNING: It is possible to use numeric values for lXPos& and lYPos& that will position the Splash Box "off the screen" and make it impossible for the user to see it.*

sLine1$*, *sLine2$, and *sLine3$*

The text that will be displayed in the Splash Box. If you specify text for a line that does not exist (such as using sLine2$ with a one-line Splash Box) it will be ignored. Keep in mind that Type 4 (see above) is a one-line Splash Box, even though it appears to display multiple lines. You can use Shortcuts in these strings, to achieve different effects.

lIconID&

Either **1)** The value `%IDI_CONSOLE` for the Console Tools Icon, or **2)** the value `%IDI_CUSTOM` for an icon that was previously loaded with the CustomIcon function, or **3)** one of the values `%IDI_APPLICATION`, `%IDI_HAND`, `%IDI_QUESTION`, `%IDI_EXCLAMATION`, `%IDI_ASTERISK`, `%IDI_STOPSIGN`, `%IDI_BIGQUESTION`, `%IDI_COMPUTER`, or `%IDI_WINLOGO` to specify a Windows Standard Icon or **4)** the ID Number of an icon in a resource file that was linked into your PB/CC program with the `$RESOURCE` metastatement. See Using Icons for complete information.

lNormalize&

If you use `%FALSE` (zero) for this parameter, Console Tools will not check the state of the console window before it displays a Splash Box. If you use a nonzero value, Console Tools will perform a ConsoleNormal operation before displaying the Splash Box, to make sure that the console screen is visible to the user.

Return Value

None.

Remarks

Console Tools Splash Boxes all have the TOPMOST property, which tells Windows that they should not be covered up by other programs. This is useful when you want to improve the chances that a program's logo/copyright Splash Box (for example) will not be obscured by other programs. It is possible, however, for another window with the TOPMOST property to cover up a Splash Box. Examples of this would include the PB/CC debugger window, and this Help File if the Options/KeepHelpOnTop feature is turned on. See ConsoleTopMost for more details about the TOPMOST property.

The lDelay& parameter can be used to make a Splash Box "modal" for a period of time. That means that your program's execution pauses for that time, and the SplashBoxShow function concentrates on constantly "refreshing" the display. (A Message Box is another good example of "modal". Your program stops until the user selects a button, then the Message Box goes away and your program continues.) More about this shortly.

After their "guaranteed" delay period (if any) Splash Boxes are "non-modal". That means that you can put one on the screen and then your program can go do something else.

Normally, Windows programs automatically "refresh" their own displays. For example, normal Windows programs "know" when another application is covering them up, and when they have been revealed again, and they refresh their displays automatically. If they didn't, all the user would see would be group of blank rectangles on the screen when a program was uncovered.

Unfortunately, Console Applications are different. Because Microsoft never really intended for Console Applications to use graphical elements like Splash Boxes, there is no auto-refresh system.

This is not usually a problem. It only becomes an issue if your program does not use the Splash Box "guarantee" delay feature, or if your program "goes away for a long time" while a Splash Box is on the screen.

If you use non-modal Splash Boxes (i.e. Splash Boxes that stay on the screen after their delay period has expired) you should read SplashBoxRefresh.

Example

```
SplashBoxShow 2+%BOLD, _
              10, _
              %CONSOLE_CENTER, _
              %CONSOLE_CENTER, _
              "Console Tools", _
              "Splash Box", _
              "" , _
              %IDI_WINLOGO, _
              %TRUE
SplashBoxHide
```

Simplified Syntax

The Console Tools Splash Box has options that you'll probably never need, so we suggest that you design a simple "wrapper" function that eliminates the parameters that you won't usually use. For example, you could add this to your program...

```
SUB SPLASHBOX(sText$)

      SplashBoxShow  1+%BOLD, _
                     0, _
                     %CONSOLE_CENTER, _
                     %CONSOLE_CENTER, _
                     sText$, _
                     "" , _
                     "" , _
                     0, _
                     %TRUE
END SUB
```

You could then use a much easier SPLASHBOX syntax for most input boxes.

```
SPLASHBOX "Please Wait"
```

See Also

SplashBoxHide, SplashBoxDefaults, SplashBoxRefresh

ToggleConsoleToolbar

Purpose

Toggles the visible/hidden state of the Windows 95/98/ME Console Toolbar

Availability

Console Tools Standard and Pro

Warning

Attempting to use this function after the DeleteWindowMenuItem function has been used to remove the "Toolbar" menu item will produce unpredictable results, possibly including an Application Error (a General Protection Fault).

Syntax

```
lResult& = ToggleConsoleToolbar
```

Parameters

None.

Return Value

lResult& will be `%SUCCESS` on all Windows 95/98/ME computers, or `%ERROR_CT_CANTBEDONE` if this function is used on a Windows NT, Windows 2000, or Windows XP computer.

Remarks

This function does not perform any error checking, other than the detection of Window NT, 2000, and XP, which do not support console toolbars. If the DeleteWindowMenuItem function has been used to remove `%MENUITEM_TOOLBAR,` this function will not work properly but the return value will still be `%SUCCESS`.

See ConsoleToolbar for a more flexible function which also performs error checking.

See Also

Microsoft Console Windows

ToggleFullScreenMode

Purpose

Toggles the FullScreen/Windowed state of the console window.

Availability

Console Tools Standard and Pro

Warning

This function forces your program into the windows foreground by using the `ConsoleToForeground %HARD` function. See ConsoleToForeground for details.

Warning

Console Tools Message Boxes, Input Boxes, Splash Boxes, Progress Boxes, and Pulldown Menus cannot be used when the console is in the FullScreen mode.

Syntax

```
ToggleFullScreenMode
```

Parameters

None.

Return Value

None.

Remarks

See ConsoleWindow `%FULLSCREEN` and `%WINDOW` for a more flexible function.

See Also

Console Window States

VirtualKey

Purpose

Sends a "virtual keystroke" to the window that currently has the Keyboard Focus.

Availability

Console Tools Standard and Pro

Warning

This function should only be used to send keystrokes to other programs. Use ConsoleKey to send virtual keystrokes to your program's console window.

Syntax

```
lResult& = VirtualKey(sKeyDefinition$)
```

Parameters, Return Value, Remarks

See ConsoleKey for a complete description of this function. They are precisely the same, except that this function does not direct the Keyboard Focus to the console window.

WindowsVersion

Purpose

Returns information about which Windows operating system is in use at runtime.

Availability

Console Tools Standard and Pro

Syntax

```
lResult& = WindowsVersion(lType&)
```

Parameters

lType&

Use one of the following predefined equates: `%WIN_PLATFORM`, `%WIN_MAJORVERSION`, `%WIN_MINORVERSION`, or `%WIN_BUILDNUMBER`.

Return Value

For `%WIN_PLATFORM`, the return value will be one of the following predefined equates: `%PLATFORM_WIN_32`, `%PLATFORM_WIN_32s`, or `%PLATFORM_WIN_NT.` For most purposes, checking for `%PLATFORM_WIN_NT` is usually sufficient. Windows 95, Windows 98, and Windows ME return `%PLATFORM_WIN_32`. (Win32s refers to a 32-bit Windows system running on a 16-bit Windows 3.1 machine.)

For `%WIN_MAJORVERSION` the return value will be the major version number of the operating system, as defined by Microsoft. For example, on a Windows NT version 4.0 computer, the major version number would be reported as 4.

For `%WIN_MINORVERSION` the return value will be the minor version number of the operating system, as defined by Microsoft. For example, on a Windows NT version 4.0 computer the minor version number would be reported as 0.

On Windows NT/2000/XP computers, using `%WIN_BUILDNUMBER` will return a value that identifies the build number of the operating system. On Windows 95/98/ME computers the function returns a value that identifies the build number of the operating system *in the low-order word* of the value (The high-order word contains the major and minor version numbers.)

lResult& will be `%ERROR_CT_INVALIDPARAMETER` if an invalid lType& value is used.

Remarks

The return values of this function are defined by Microsoft. They are not modified or verified by Console Tools in any way.

Example

```
IF WindowsVersion(%WIN_PLATFORM) = %PLATFORM_WIN_NT THEN
      'the program is running on an NT computer
ELSE
      'the program is not running on an NT computer
END IF
```

See Also

ConsoleToolsVersion

Appendix A: Microsoft Console Windows

General Information About Console Windows
on Windows NT/2000/XP and Windows 95/98/ME Computers

Win95/98/ME Console Window PRINT Speed

Windows 95, Windows 98, and WIndows ME all have a well-known problem with the speed of the console display. On nearly all 95/98/ME systems, the PB/CC PRINT statement is much, much slower than it is on Windows NT, Windows 2000, and Windows XP systems. This is *not* a defect in PB/CC or Console Tools, it is a well-documented problem with Windows 95/98/ME which affects all console-mode and DOS applications, regardless of the programming language. (Some people have suggested that it is an intentional flaw, designed by Microsoft to discourage the use of DOS programs on Windows 95/98/ME systems.)

Fortunately, Perfect Sync has discovered that the Windows 95/98/ME console is only slow when it is in one of its "native" modes of 80x25, 80x43, or 80x50. If you change the number of rows to an unusual value like 80x24 or 80x26, the console window PRINT speed will typically increase by a factor of 10 to 250 times. Some users have reported speed improvements of over 500 times!

The Windows 95/98/ME console window's Window Menu Properties dialog only allows you to set the row count to 25, 43, or 50, but it is possible to *programmatically* change the size of the console window to other values. If your PB/CC program displays a lot of information on the screen, we suggest that you use the Console80x function to change the size of the console window to some value other than 25/43/50 rows. The ConsoleControl function can also be used to change the size of the console screen buffer.

Unfortunately, Windows 95, 98, and ME will not allow you to switch to the FullScreen mode when the console window is *not* 80x25, 80x43, or 80x50. If you use the Properties menu to perform the switch, an 80x25 screen will appear but several lines of text will be missing. If you press Alt-Enter or attempt to use the `ConsoleWindow %FULLSCREEN` function (see ConsoleWindow) Windows will display a message box that says:

This program must run in a window, not in a full screen.

Your display does not support the screen size currently set by the program.

So it may be necessary for you to choose between fast printing and the ability to use the fullscreen mode.

If both of those features are important to you, it is possible to use a different method for toggling the fullscreen mode. Instead of relying on the built-in Windows hotkey Alt-Enter, your program could watch for *Shift*-Enter (for example). When it detects that a user has pressed Shift-Enter, it could use the Console80x function to switch to a standard 25/43/50 screen size and then use `ConsoleWindow %FULLSCREEN` to programmatically activate the fullscreen mode. Keep in mind that most video cards only support two or three fullscreen modes (see next paragraph) so if your program uses 80x26 in the Window mode it may be necessary to jump all the way up to 80x50 in the fullscreen mode. The best solution for many programs would be to use an 80x24 console in the Window mode, and switch to 80x25 when a fullscreen display is required.

Windows NT, 2000, and XP do not have any of the problems noted above. Console printing is always fast, and NT/2000/XP allow you to switch to the fullscreen mode regardless of the number of screen rows. If you switch to the fullscreen mode when the row count is not

25/43/50, Windows NT, 2000, and XP will use the next-highest row count that your video card supports. All video cards support 80x25 and either 80x43 or 80x50, and many cards support all three. Some cards also support an 80x28 mode, and other "unusual" row counts may be available, depending on your hardware.

The Windows 2000/XP Console Window

The Windows 95, 98, ME, and NT console windows all use a default size of 80 columns by 25 rows. Windows 2000 and XP, on the other hand, uses a default size of 80 columns by **300** rows, which virtually always results in a console window that has scroll bars. We recommend, therefore, that you always use the Console80x function in your Console Tools programs, so that they will look and work the same on all versions of Windows.

The Console Window "Buffer"

When Windows creates a console screen for a console application, it first creates a "buffer" in memory, to hold the characters and attributes (colors) that make up the screen.

The buffer can be larger than the console window itself, but not smaller. Think of the visible console window as a "viewport" into the Console Buffer. If the Buffer is larger than the console window, Windows will automatically add scroll bars to the console window so that it is possible to see the whole thing. If you attempt to change the buffer size so that it is smaller than the window, or change the window size so that it is larger than the buffer, Windows will either refuse to make the change or it will automatically change the other parameter to match.

The term "Buffer" is also used to refer to screen "pages" that exist only in memory and can't be seen until they are made "active". PB/CC supports a total of 8 pages, and Console Tools provides a number of internal Screen Buffers for ConsoleScreenSave and ConsoleScreenLoad operations.

The FullScreen Console Mode

Unless the hotkey has been disabled or intercepted, pressing Alt-Enter when the console has the keyboard focus will cause the window to be toggled between the FullScreen Mode and the Windowed Mode.

Some Windows programs refer to a "Full Screen Mode" that means something else. It usually refers to a game or other graphics-oriented program that uses the entire screen area and does not allow Windows enough room to display a Task Bar, a title bar, borders, or any other normal Windows screen elements. In the case of a Console Application, the FullScreen Mode refers to a *non-graphics* mode where *only text* can be displayed. See ConsoleWindow and ConsoleIsFullScreen for more information.

The Window Menu Close Function

The Windows Close function, which can be invoked via the Alt-F4 hotkey or via the console window's Window Menu *has ability to close your program without any notification whatsoever*. Your program will simply shut down in the middle of whatever it's doing and "never know what hit it". On Windows 95/98/ME the Close functions brings up an "Are You Sure?" message box, and it allows the user to decide whether or not they really want to close the program. On Windows NT/2000/XP the Close function's operation is virtually instantaneous. See DeleteWindowMenuItem for a technique that can be used to deactivate the Windows Close function. Also see the OnShutdown function. Note also that the Console Tools Pulldown Menu has the ability to intercept the Close command.

The Windows Task Manager's "End Task" Function

The Task Manager program that is included with Windows NT/2000/XP and 95/98/ME includes the ability to perform an "End Task" operation on any program that is currently running. This ability cannot be bypassed, and the command cannot be intercepted. We suggest that, if possible, you write your PB/CC programs in such a way that disk files will not be damaged by an unexpected End Task. For example, you might consider writing to a temporary file and then performing a virtually instantaneous file-rename operation to make the changes take effect. (This is good advice for *any* program. You never know when the power is going to fail.)

The Windows 95/98/ME Console Toolbar

Unlike Windows NT, Windows 2000, and Windows XP, the Windows 95, 98, and ME operating systems provide a Console Toolbar just below the console Title Bar. Basically, the buttons on the toolbar provide shortcuts to functions that are available in the Window Menu (the Alt-Space "System Menu"). If your program is used on both Windows 95/98/ME and Windows NT/2000/XP we suggest that you use Console Tools' ConsoleToolBar function to hide the toolbar when your program initializes, so that you will have a consistent interface.

The Windows 95/98/ME ConAgent Program

Windows 95, 98, and ME use a "helper" program called CONAGENT.EXE whenever a DOS or Console Application is run. This "Console Agent" program provides a link between a program and a console's text window.

On Windows 95/98/ME computers, you can use the Windows Explorer program to locate the CONAGENT.EXE file (usually in the \WINDOWS\SYSTEM directory) and right-click on it to access a Properties Menu. If you change the ConAgent properties (font, screen location, etc.), then all future DOS and Console Applications will use those default settings when they start up.

The default font for any Windows 95/98/ME program that does not specify a font is "Courier New", and that's the font that the console window will use if you've never changed the ConAgent Properties. After you've changed the ConAgent Properties, the console window font will be determined by the settings that you've selected.

If you use a program's Window Menu to change the console properties while a program is running, Windows 95/98/ME does not provide a way to save the settings. They will be used during the current session only, and the next time the program is run it will use the default console settings.

The Windows 95/98/ME console window also provides an "auto-size" feature for setting font size. Theoretically, it allows Windows 95/98/ME to figure out the optimum size for a particular console window, and use it. Unfortunately, the auto-size feature is well know to be very unstable, and it often causes console windows to appear so small that the text is not readable. We strongly suggest that, unless you *never* adjust the ConAgent Properties, that you explicitly set the font size when using a Windows 95/98/ME console window, and *avoid the auto-size feature*. (For more details, see the ConsoleNormal function.)

The Windows NT/2000/XP Console Window Defaults

There is no ConAgent program (see just above) on Windows NT/2000/XP computers.

When you run a console program for the first time, Windows NT/2000/XP will use the computer's default settings for console programs. You can adjust a computer's default console settings by using the Windows Control Panel's "Console" program, which is the same thing as running the CONSOLE.CPL program. The default settings are stored in the Windows Registry Database (see bottom of page) under the key name `HKEY_CURRENT_USER\Console`.

If you change a console program's setting while it is running (by using the Window Menu), Windows NT/2000/XP will ask you whether or not the settings should be applied to the current session only, or saved for future sessions. If you choose to save the settings, Windows will save them in the Registry, keyed under the text in the console window title bar. Then, if any console program with that *exact* title is run, Windows NT/2000/XP will recognize it and use those settings. For example, if you save the settings for a program with "My Program" in the title bar, the settings will be stored under the key `HKEY_CURRENT_USER\Console\My Program`.

There is one minor glitch in the Windows NT/2000/XP save-settings-by-console-title system. Let's say that you have a console program called MYPROG.EXE, and you run it under Windows NT, 2000, or XP. And let's say that the first thing your program does is to use the ConsoleTitle function to change the console title bar to say "My Program". If you use the Window Properties Menu to change the font size (for example) and tell NT to save the settings, it will save it under "My Program" in the registry. The problem is that the next time your program is run it will have the console title MYPROG or MYPROG.EXE for a split second, before your program can change it. And that's the split second during which Windows looks at the console title and checks the registry. (If you think about it, Windows has no choice. It has to create the console window before your program actually starts running, and Windows NT/2000/XP consoles use a default title bar that contains the actual name of the program.) So... the settings that you saved under "My Program" will be ignored, and the default settings will be used.

A solution for this problem would be to either **1)** create an installation program that creates the necessary Registry entries, or **2)** use the Windows RegEdit program to rename the Registry Key after it has been created, so that it uses the program name (like MYPROG.EXE) that is displayed in the startup title bar.

Another glitch in the system is that if you use a Windows Shortcut to launch a console program, Windows NT, 2000, and XP use the shortcut name for the default console title. That means that you can configure everything perfectly, and your program can run perfectly for months or years, but if your user changes, copies, or renames the *shortcut* that is used to start your program, the console settings will be lost.

If you need to make sure that your program runs correctly after it has been installed on a new computer, without requiring the user to adjust and save the console settings, it would be relatively simple to use the Win32 API to create registry entries for a program. Then, whenever a console application with the specified title bar text is run, Windows NT, 2000, and XP will use the setup that is specified in the registry.

The *Windows Registry* is an operating system database that Windows maintains. It can be used to store program configuration settings of virtually any kind, and Windows makes extensive internal use of the registry.

Appendix B: Console Window States

Every Window, and that includes a console window, has a "Window State".

A Console Application's Window State can be changed by a user's mouse-click or keypress, by an external program that sends the appropriate signal to the application, or by the application itself using the Windows API or the Console Tools ConsoleWindow function to change its state.

There are three basic Window States that console windows can have...

Normal or "**Restored**", which refers to a free-floating console window. Most programs start up in this state, unless they are designed to automatically use the Window State from the last time they were run. (It is also possible to change a program's Windows Shortcut to force it to start in a mode other than Restored.)

"Minimized", which refers to a Console Application that currently only appears on the Task Bar and Alt-Tab popup. No console window is actually visible.

"Maximized", which refers to a console window that is located so that it touches the top-left corner of the screen, hiding the top and left window borders by placing them "off the screen". Maximized *non*-console windows almost always fill the desktop -- the entire screen except for the Task Bar -- but console windows often don't. (Windows 95/98/ME console windows that use the Courier New font are one exception: they do fill the desktop.)

Beyond those three basic states, a console window can be either "Hidden" or "Showing". **"Hidden"** refers to a Console Application that is "running" but is not currently visible on the screen or the Task Bar. **"Showing"** is the normal "visible" state.

You should think of a console window as having two states at the same time. For example, it could be both Hidden and Maximized. That would mean that the console window is not currently visible, but if the ConsoleWindow function was used to Show it, it would appear Maximized.

All of the Window States described above can be produced with various combinations of ConsoleWindow commands, such as...

```
ConsoleWindow %MINIMIZE
ConsoleWindow %MAXIMIZE
ConsoleWindow %RESTORE
ConsoleWindow %HIDE
ConsoleWindow %SHOW
```

The Microsoft-given name "Restore" can be very confusing. It does not mean "put it back the way it was" it means "put it back into the Normal condition". For example, if a window starts out as Maximized, and you send it a `%MINIMIZE` command and then a `%RESTORE` command, you would probably expect the window to end up Maximized again. But it won't: it will end up in the "Restored State", which is not the same thing as Maximized.

For this reason, Console Tools has extended the ConsoleWindow function to include a new command that we call...

```
ConsoleWindow %UNMINIMIZE
```

This command effectively reverses the last Minimize command and returns the window to the state that it was in before it was Minimized. (This accomplishes the same thing as clicking on the Task Bar button of a Minimized Console Application: it returns the window to its *previous* state.)

Windows also supports several other Window State commands. If you don't understand the subtle difference between some of these commands, we apologize in advance. The descriptions here are paraphrased from the "official" Microsoft documentation, which can be very ambiguous.

> `ConsoleWindow %SHOWDEFAULT`, which (at least theoretically) sets the console window to the state that was used when the program was started. We have found that the results from this command vary from computer to computer, probably because Console Applications do not track their startup parameters in the same way that other programs do. If you use this command, your results may vary.
>
> `ConsoleWindow %SHOWNORMAL`. Microsoft says that this command *"activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position." As with* `%SHOWDEFAULT`, we have found that this Windows API function does not perform the same way on all computers.
>
> ConsoleWindow `%SHOWMAXIMIZED`. is 100% identical to using `%MAXIMIZE`, because Microsoft defines their numeric values as the same number.
>
> `ConsoleWindow %SHOWMINIMIZED` activates the window and displays it as a Minimized window.
>
> `ConsoleWindow %SHOWMINNOACTIVE` shows a console window as Minimized but does not activate it. The currently active window remains active.
>
> `ConsoleWindow %SHOWNA` displays the console window in its current state but does not activate it. The currently active window remains active.
>
> `ConsoleWindow %SHOWNOACTIVATE` displays the console window in its most recent state and position, but does not activate it. The currently active window remains active.

And finally, Console Tools provides enhanced control over one more aspect of the console window's State...

> `ConsoleWindow %FULLSCREEN` can be used to switch the console window into a DOS-style FullScreen Mode. This mode does not use any graphical elements at all. There are no borders, title bar, or 95/98/ME toolbar. Console Tools GUI elements such as Message Boxes, Input Boxes, Progress Boxes, and Splash Boxes cannot be used in this mode *because Windows does not support graphics in the FullScreen Mode*. This is a Windows limitation, not a Console Tools limitation.

To make sure that the console window is *not* in the FullScreen Mode, use...

```
ConsoleWindow %WINDOW
```

It is also possible to manually toggle a console window between the FullScreen and Window mode by pressing the Windows Alt-Enter hotkey when the console has the keyboard focus, or by using the ToggleFullScreenMode function.

The ConsoleIsFullScreen function can be used to determine whether or not the console is in the FullScreen mode. It is important to note that the console can appear to be in two conflicting modes when the FullScreen Mode is active. The return values from the ConsoleState, ConsoleIsMinimized and ConsoleIsMaximized functions will vary, depending on several factors. If the ConsoleIsFullScreen function returns Logical True, the results of the other functions indicate (at most) the state that the window will assume when the FullScreen Mode is turned off.

Appendix C: The Console "Window Menu"

The Console Window's Window Menu (formerly known as the "System Menu") can be accessed in three different ways. While the console program is running and the console is visible...

1) Press Alt-Space

2) Left-click on the icon that's on the left-hand side of the console title bar, or

3) Right-click anywhere on the console title bar.

Any of these three actions will cause a pulldown menu to appear. The items that normally appear on the Window menu are:

Restore
Move
Size
Minimize
Maximize
Close
Edit >
Properties

On Windows 95/98/ME systems, the menu will also include "Toolbar" between Edit and Properties. If the console Toolbar is currently visible, there will be a checkmark next to the menu item.

The right-pointing arrow next to the Edit item indicates that a submenu exists. It usually contains the items Mark, Copy, Paste, and Scroll.

Depending on the current state of the console window, various menu items may be "disabled" and will therefore be displayed in gray. For example, if the console window is currently Maximized, the "Maximize" item will be disabled because it would have no effect if it was used.

Items can be removed from the Window Menu with the Console Tools DeleteWindowMenuItem function. This allows you to make sure, for example, that your user cannot Close your application by using the Close item on the Window Menu.

Window Menu items often interact with other (related) Windows functions. For example, if you use the DeleteWindowMenuItem function to remove the Close item from the Window Menu, **1)** the menu item will disappear, **2)** the **"X"** button on the right-hand side of the title bar will be disabled, **3)** the Alt-F4 hotkey will be disabled, and **4)** the window will no longer Close if a user double-clicks the title bar icon.

Appendix D: Using Icons

Perfect Sync, Inc. hereby disclaims any responsibility for copyright/trademark infringement that may result from the use of Console Tools to display icons.

All of the advice in this Help File regarding copyright/trademark law is included as a courtesy to Console Tools users. It is not intended to replace the advice of an attorney regarding these matters.

Console Tools Message Boxes and Splash Boxes can display many different icons, and the ConsoleIcon function allows your programs to specify which icon will be displayed in the console window title bar.

Modern icons are actually three icons in one. The icon file (`*.ico`) usually contains a Small icon (16x16 pixels) *and* a Large icon (32x32 pixels) and some contain the new 48x48-pixel icon format. Windows uses the different-sized icons for different things, like desktop icons and taskbar icons. If an icon file does not contain the image size that Windows needs for a particular purpose, it does the best it can and "scales" another size to fit. The results are usually not very good, so you should try to use icon files that contain all three sizes

You should keep in mind that Windows is not 100% consistent when it comes to using icons. For example, if you embed an icon in a PB/CC program (see below), Windows NT/2000/XP will display that icon in the Windows Explorer next to the file name. The Windows 95/98/ME Explorer, on the other hand, will display the standard icon for a console application and ignore the embedded icon.

Windows uses the standard prefix `IDI_` to refer to an icon, and Console Tools uses the same terminology. For example, `IDI_WINLOGO` refers to a Windows Logo icon that is standard on all Windows computers.

The Console Tools Icon

Using the value `%IDI_CONSOLE` with Console Tools functions will produce an icon that looks like a small console window with a black screen. This icon is part of the Console Tools DLL and you may use it in your programs without permission from Perfect Sync. (All three icon sizes are included in the icon that is embedded in the DLL and in the sixteen `.ICO` files that are provided with Console Tools.)

Standard Windows Icons

Microsoft provides six standard icons on all Windows computers, and Console Tools makes three additonal Windows icons available.

`%IDI_APPLICATION` is the name given to the "application" icon. The default picture is a small, gray computer screen. Pretty dull.

`%IDI_HAND` used to be called `IDI_STOP` and `IDI_ERROR`. The current default picture is a red circle with a big white X in the middle. (As far as we know, a Hand has never been the default picture for this icon, but that's what Microsoft calls it.)

`%IDI_QUESTION` is an icon that looks like a white "cartoon bubble" with a blue Question Mark inside.

`%IDI_EXCLAMATION` used to be called `IDI_WARNING`. It looks like a yellow triangle, pointing up, with a black Exclamation Point inside.

`%IDI_ASTERISK` used to be called `IDI_INFORMATION`. It looks like a white "cartoon bubble" with a blue lower-case letter `i` inside, presumably meaning "Information".

`%IDI_WINLOGO` is the familiar "waving" Windows Logo. *You must receive permission from Microsoft before you can legally use their logo in your programs.*

`%IDI_STOPSIGN` is a red Stop Sign with white lettering.

`%IDI_BIGQUESTION` is a large Question Mark *without* a surrounding "bubble".

`%IDI_COMPUTER` is a picture of a computer.

Other Icons

Basically, Console Tools can use *any* standard-format icon.

IMPORTANT NOTE: Many icons are protected by copyright laws and cannot be used in your programs without permission from the owner. You are free to use the sixteen `CONSOLE*.ICO` icons that are provided with Console Tools, but if you use icons that were not supplied to you by Perfect Sync then you are responsible for obtaining the legally-required permission. Many different public domain icons can be found on the internet. Just make sure that somebody else didn't post a copyrighted icon and illegally label it "public domain" or "free".

You can always design your own icons to avoid copyright problems. Perfect Sync uses an icon editor called MicroAngelo, which is available from impactsoft.com. We did have a few problems with their Demo version, however. It was not able to create icons that could be used by PB/CC programs. Microsoft Visual Studio 97 also includes an icon editor, and the Microsoft Image Editor that is provided with PB/CC can also be used to create certain types of icons.

IMPORTANT NOTE: Most corporate logos and many other symbols are copyrighted, and you may not use images that resemble them without permission, even if you create the icon yourself.

Using Icon Files

See the CustomIcon function for instructions for loading icons from disk files at runtime.

Embedded Icons

It is possible to avoid the use of external icon files by "embedding" an icon in your EXE file.

In addition to an Icon Editor, you will need a program called a Resource Editor. The process of adding icons to a PB/CC program -- whether or not you use Console Tools -- involves several steps, and those steps vary quite a bit, depending on the Resource Editor that you use. The following steps will walk you through the process of using the Microsoft Visual Studio 97 Resource Editor. If you're using a different Resource Editor, these steps will illustrate the basic technique.

STEP 1 Choose a "base name" for your resource file. It is common practice the use the base name of the program that will *use* the resource. This example will assume that your program is called *`PROGNAME`*`.BAS`. You should insert your program's base name (the file name minus the dot and extension) wherever you see *`PROGNAME`* below.

STEP 2 Locate the directory where `*PROGNAME*.BAS` is stored, which we will call *PROGDIR*, and create a subdirectory of *PROGDIR* called `RESOURCE`.

STEP 3 Decide which icon(s) you want to use. Console Tools provides sixteen icons called `CONSOLE*.ICO`, and if you search your hard drive for files with the `.ICO` extension you will probably find several others. (Please note the copyright warnings above.)

STEP 4 Start Microsoft's Visual Studio. Click on File/New, select the Files *tab*, and double-click on Resource Script. Notice the word "Script1" that appears near the top of the screen.

STEP 5 Click on View/ResourceIncludes, and change the Symbol Header File name to *PROGNAME*.**H**. Then click the Ok button.

STEP 6 Click on File/Save, then navigate to the `RESOURCE` directory that you created above.

STEP 7 Enter *PROGNAME*.**RC** in the File Name field, and click the Save button.

STEP 8 Instead of Script1 you should now see *PROGNAME*.RC near the top of the screen. (Optional Step for Microsoft Visual Studio users only... Right-click on *PROGNAME*.RC and use the Properties Menu to turn off the "MFC Features".) Right-click on *PROGNAME*.RC. If you want to create a new icon, select Insert from the pop-up menu. If you want to use an existing icon (from an `ICO` file) select Import. In either case, you should then double-click on the word "Icon".

STEP 9 Either use the built-in icon editor to draw an icon, or use the "Import Resource" dialog to find the icon file that you want to use.

STEP 10 When you're done adding the icon, you will notice that the word `IDI_ICON1` has been added below *PROGNAME*.RC. If you want to change the name to something like `IDI_MYICON`, you can right-click on `IDI_ICON1`, then select Properties and change the name. This example will assume that you will not change the name.

STEP 11 Use View/ResourceSymbols to find out which ID Number was automatically assigned to the icon. Write it down; you'll need this number later. (It's possible to assign a specific ID number, but we won't get into that here. It can get complicated.)

STEP 12 Use File/Save to save the resource file, then exit from Visual Studio.

STEP 13 If you don't already know where it is, use Windows Explorer to find your PB/CC "BIN" directory It is usually called `\PBCCxx\BIN` where `xx` is the PB/CC version number, but the exact drive and path that you will use will depend on where you installed PB/CC. This example calls this directory *BINDIR*. You should insert the appropriate path wherever you see *BINDIR* below.

STEP 14 Use the Windows Start Menu to open an MS-DOS prompt window, and use the CD command to navigate to your `\`*PROGDIR*`\RESOURCE` directory.

STEP 15 Type the following line and press the Enter key:

```
BINDIR\RC.EXE    PROGNAME
```

STEP 16 If you did everything correctly up to this point, the RC program should have created a file called *PROGNAME*.RES. Use the DIR command to make sure that it was created. If it wasn't, stop now and re-check the results of the previous steps. In some cases it will be necessary to add files like `afxres.h` and other `.h` files to your `RESOURCE` directory before RC will work properly. If you have problems, the RC program should tell you the names of the missing files. Locate them (in the Visual Studio directories) and copy them to the `RESOURCE` directory, then repeat Step 15.

STEP 17 Once the `.RES` file has been created, type the following line and press the Enter key:

```
BINDIR\PBRES.EXE    PROGNAME
```

STEP 18 If you did everything correctly, the PBRES program will have created a file called *PROGNAME*`.PBR`. Use DIR to confirm that it was created.

STEP 19 Type EXIT and press the Enter key to close the MS-DOS session.

STEP 20 Start the editor that you use to write PB/CC programs. Add the following line to the beginning of your program...

```
$RESOURCE "PROGDIR\RESOURCE\PROGNAME.PBR".
```

STEP 21 Add the following line after the `$RESOURCE` line...

```
%IDI_ICON1  = 101
```

If you used a different name for the icon, you should use it instead of `%IDI_ICON1`. Don't forget the leading percent sign (%) that PB/CC requires for equates. If the number that you wrote down for the icon's resource ID was not 101, type that number instead.

STEP 22 If you haven't already done so, add the Console Tools DLL to your program. See Three Critical Steps For Every Program for more details.

STEP 23 To test whether or not this whole process was performed correctly, add the following line to your program, right after the InitConsoleTools line in the WinMain function...

```
ConsoleIcon  %IDI_ICON1
```

Again, if you used a different name for the icon, type that instead.

When you compile and run the program, the ConsoleIcon function should change the console window's title bar icon to be the icon from the resource file. You can also use the icon with Console Tools Message Boxes and Splash Boxes.

You can repeat this process to add as many icons as you want to the Resource File. (You could also write a batch file to automate some of the steps.) Note that if you use more than one icon, they should all be in the *same* `$RESOURCE` file. And if you use PB/DLL in addition to PB/CC, keep in mind that Console Tools cannot use icons that are embedded in DLLs.

One final note... You don't have to include any extra files (`ICO/RC/RES/PBR/`etc) when you distribute your program. The resource file containing the icon has been embedded directly into your program's EXE file.

Appendix E: Console Tools Error Codes

Suggestion
Use the function that's in the CTERROR.BAS file to turn Console Tools Error Codes into plain English. You can $INCLUDE the file in your program during development to help you translate the codes and display the results in a Console Message Box.

Frequently Asked Question:
Whoa! Why are these error numbers so ***LARGE?***

The Answer:
Microsoft made us do it.

Well, they didn't actually write us a letter or anything. They just made rules for 32-bit Windows programs that require the use of certain number ranges. Basically, Microsoft has reserved all of the "reasonable" numbers for itself, so that Windows can report a wide variety of error numbers when it has problems.

There are well over 4,000,000,000 (4 *billion*) possible Error Codes. Microsoft has reserved about half a billion of those for non-Microsoft use. They've reserved this range of numbers...

536,870,912 to **1,073,721,824**

...for "Application-Defined Error Codes". That means that everybody but Microsoft is supposed to use that range of numbers. Console Tools and *your* programs are supposed to be limited to this range when they create new Error Codes. (If you're curious about the unusual numbers, it's the range of numbers with Bit 29 set.)

Console Tools could have easily used the numbers that start with 1,000,000,000 so that they'd be easy to read, but we figured that *you'd* rather use that range for your programs, since it's the "best" range of number that the Microsoft rules have to offer.

So we chose the range 999,000,000 to 999,999,999. All Console Tools Error Codes -- in fact the Error Codes from all Perfect Sync software development products -- fall into that range. The predefined equates `%ERROR_CT_FIRSTERROR` and `%ERROR_CT_LASTERROR` correspond to those numbers, and everything else falls in between.

If a Console Tools function reports an Error Code that is not in that range, you can count on the fact that Windows reported a Windows Error, and Console Tools is simply passing the number along.

`%ERROR_CT_FIRSTERROR  (999,000,000)`

The lowest-numbered Console Tools Error

`%ERROR_CT_LASTERROR   (999.,999,999)`

The highest-numbered Console Tools Error.

Use these two values to see if a number falls into the range of Console Tools Error Codes. Example:

```
IF lValue& => %ERROR_CT_FIRSTERROR AND _
   lValue& <= %ERROR_CT_LASTERROR THEN
        'VALUE CORRESPONDS TO A
        'CONSOLE TOOLS ERROR CODE.
END IF
```

Console Tools Error Codes

in numeric order

`ZERO (0)   %SUCCESS`

No Error. (Same as Microsoft's `ERROR_SUCCESS`, which we feel is oxymoronic.)

```
999,000,000   %ERROR_DLL_NOT_AUTHORIZED
              %ERROR_CT_DLL_NOT_AUTHORIZED (same value)
```

The InitConsoleTools or InitCTInternals function was used before the ConsoleToolsAuthorize function. See Authorization Codes.

`999,000,001   %ERROR_CT_INVALIDPARAMETER`

An illegal value was passed to a Console Tools function. Many different functions report this error.

`999,000,002   %ERROR_CT_CANTBEDONE`

An example of this error would be attempting to activate the Console Toolbar on a Windows 2000 computer. Windows NT, 2000, and XP do not support Toolbars. The exact meaning of this error depends on the function that reported it.

`999,000,010   %ERROR_CT_INVALIDMENUNUMBER`

Your program used a Menu Buffer number that is not between 1 and the number that you specified for `lMenuBuffers&` in the InitConsoleTools function, or it tried to specify an illegal value for `lMenuBuffers&` in the InitConsoleTools function.

`999,000,011   %ERROR_CT_INVALIDSCREENNUMBER`

Your program used a Screen Buffer number that is not between 0 and the number that you specified for `lScreenBuffers&` in the InitConsoleTools function, or it tried to specify an illegal value for `lScreenBuffers&` in the InitConsoleTools function.

`999,000,020   %ERROR_CT_FILENOTFOUND`

The requested file does not exist in the location that was specified, or your program cannot currently "see" the drive/path, possibly due to an incorrect login or network failure.

`999,000,021   %ERROR_CT_INVALIDFILEFORMAT`

The specified file does not have the format that is required for the requested operation.

`999,000,040   %ERROR_CT_INVALIDMENUSYSTEM`

Your program attempted to create a menu system that was invalid. This occurs most commonly when a "circular reference" is accidentally used, or when errors that were reported by the MenuDefinition function were not corrected. Also see MenuSystemCreate.

`999,000,041   %ERROR_CT_MENUDOESNOTEXIST`

Your program used an invalid Menu Buffer number with the PulldownMenu function. Either that Menu Buffer is empty, or the menu has not been created (without errors) with MenuSystemCreate.

`999,000,090   %ERROR_CT_FONTINUSE`

Your program tried to use the CustomFont function to define a font, but the Custom Font is currently being displayed and cannot be changed. Make sure that your program uses the appropriate Hide command (like SplashBoxHide) for the function that is *using* the custom font before attempting to redefine the custom font.

`999,000,099   %ERROR_CT_FEATURENOTAVAILABLE`

You attempted to use a feature that is not available in your Console Tools DLL. This usually results from an attempt to use a Console Tools Pro feature when the Console Tools Standard DLL is installed. It can also result from using undocumented parameters with certain functions.

`999,000,999   %ERROR_CT_NOCONSOLE`

This error is only returned by the InitConsoleTools and InitCTInternals functions when Console Tools is used in *non-native* console applications that create their own consoles by using the Windows API. This error means that Console Tools was unable to locate a console window that was owned by the current application. This can only happen in non-native console applications which attempt to use InitConsoleTools or InitCTInternals before a console window has been created.

`999,001,000` to
`999,999,998`

See *Error Ranges*, just below, for numbers that fall in this range.

`999,999,999   %ERROR_CT_UNKNOWNERROR`

A Console Tools function detected that an error had taken place but was unable to determine the cause. This error will also be reported if Console Tools uses a Windows API function which fails but does not provide any information about the reason for the failure.

Error Ranges

The following equates represent *ranges* of Error Codes. For example, a PowerBASIC ERR value from 1 to 999 would be reported as `%ERROR_CT_FIRSTPBERROR` plus the ERR value. Since `%ERROR_CT_FIRSTPBERROR` has a value of 999,001,000 this would result in Error Numbers from 999,001,001 to 999,001,999 being reported. You can subtract the value of `%ERROR_CT_FIRSTPBERROR` from the reported value to get the original ERR value.

`999,001,000 to 999,001,999     %ERROR_CT_FIRSTPBERROR`

The Console Tools DLL was created using the PowerBASIC PB/DLL compiler. If a PowerBASIC "ERR" error is detected, Console Tools adds the value `%ERROR_CT_FIRSTPBERROR` to the ERR value to make it clear that it is a PowerBASIC error, not a Windows Error Code with the same value. (Windows Error Codes are reported without adding a predefined equate.) Consult the PB/CC Help File for more information about the ERR value that is reported. PB/CC and PB/DLL use the same Error Code numbers.

`999,002,000 to 999,002,999     %ERROR_CT_INVALIDITEM`

Console Tools detected an invalid item in a menu string that was submitted to the MenuDefinition function. The last three digits of the error number represent the item number in the string. Example: If the third menu item in a string contained an error, the number `999,002,003` would be reported.

`999,003,000 to 999,003,999     %ERROR_CT_INVALIDSUBMENU`

The MenuDefinition function found a reference to a submenu number that had a number less than 1 or greater than the value that was specified for `lMenuBuffers&` in the InitConsoleTools function. The last three digits of the error number represent the item number of the invalid reference. For example, if item 16 contained a reference to an invalid submenu number, the value `999,003,016` would be reported.

`999,004,000 to 999,004,999     %ERROR_CT_SELFREFERENCE`

The MenuDefinition function detected a menu "self-reference". An example of this would be Menu #2 containing an item that pointed to Menu #2 as a submenu. The last three digits of the error number represent the item number of the invalid reference. For example, if item 7 contained a reference to an invalid submenu number, the value `999,004,007` would be reported.

`999,005,000 to 999,005,999     %ERROR_CT_INVALIDITEMNUMBER`

This value can be reported two different ways by the MenuDefinition function. First, it will be reported as 999,005,000 if a menu string contains more than 128 items. Second, if your program uses an item number that is greater than 32750, Console Tools report 999,005,000 plus the offending item's number. For example, if item 3 in a menu string was assigned the value 88000, the number 999,005,003 would be reported. Note that the item's number (3) is added, not the illegal item's value (88,000).

`999,999,999    %ERROR_CT_UNKNOWNERROR`

See above.

Appendix F: Accelerator Key Codes

Items marked with [V] are different from the corresponding Virtual Key Codes (see the Console Tools INC files).

```
%Pulldown_Enter       = 13
%Pulldown_Escape      = 27
%Pulldown_None        = 29
%Pulldown_Close       = 30
%Pulldown_F1          = 31        '[V]
%Pulldown_F2          = 32        '[V]
%Pulldown_F3          = 33        '[V]
%Pulldown_F4          = 34        '[V]
%Pulldown_F5          = 35        '[V]
%Pulldown_F6          = 36        '[V]
%Pulldown_F7          = 37        '[V]
%Pulldown_F8          = 38        '[V]
%Pulldown_F9          = 39        '[V]
%Pulldown_F10         = 40        '[V]
%Pulldown_F11         = 41        '[V]
%Pulldown_F12         = 42        '[V]
%Pulldown_0           = 48
%Pulldown_1           = 49
%Pulldown_2           = 50
%Pulldown_3           = 51
%Pulldown_4           = 52
%Pulldown_5           = 53
%Pulldown_6           = 54
%Pulldown_7           = 55
%Pulldown_8           = 56
%Pulldown_9           = 57
%Pulldown_A           = 65
%Pulldown_B           = 66
%Pulldown_C           = 67
%Pulldown_D           = 68
%Pulldown_E           = 69
%Pulldown_F           = 70
%Pulldown_G           = 71
%Pulldown_H           = 72
%Pulldown_I           = 73
%Pulldown_J           = 74
%Pulldown_K           = 75
%Pulldown_L           = 76
%Pulldown_M           = 77
%Pulldown_N           = 78
%Pulldown_O           = 79
%Pulldown_P           = 80
%Pulldown_Q           = 81
%Pulldown_R           = 82
%Pulldown_S           = 83
%Pulldown_T           = 84
%Pulldown_U           = 85
%Pulldown_V           = 86
%Pulldown_W           = 87
%Pulldown_X           = 88
%Pulldown_Y           = 89
```

```
%Pulldown_Z           = 90
%Pulldown_PgUp        = 97
%Pulldown_PgDn        = 98
%Pulldown_AltEnter    = 99
```

Appendix G: Console Window Screen Color Numbers

Microsoft console windows use a DOS-like numbering system for colors. A single "color byte", with a possible value of 0 to 255, is used to describe both the foreground and background colors.

Basically, four bits of the eight-bit byte are used for the foreground and four are used for the background. Three of the four bits correspond to Red, Green, and Blue, and the fourth bit is the "intensity" bit.

Examples: If only the Red bit was set, a dark red color would be displayed.   If the Red and Intensity bits were set, a bright red color would be displayed, If the Red and Blue bits were set, then a dark magenta (purple) color would be displayed. Adding the intensity bit would change it to bright magenta. If all four of the bits are set, white is produced. If none of the bits are set, black is produced.

Sixteen different colors (0-15) can be represented by four bits. With 16 foreground colors and 16 background colors, that's 256 different combinations. To convert a foreground and background color into a color byte value, you can use this formula:

```
      ColorByte = (BackColor * 16) + ForeColor
```

To convert a color byte into foreground and background colors, use this code:

```
      BackColor = ColorByte \ 16    'note use of Integer Division
      ForeColor = ColorByte MOD 16
```

Rather than list all of the possible color combinations here, Console Tools comes with a file called `COLORS.256`. You can use the ConsoleScreenLoad function like this...

```
      ConsoleScreenLoad "\CONTOOLS\COLORS.256", 1, 25, 0
```

...to display the screen. All of the 256 possible color combinations will be displayed.

It is important to note that the FullScreen Mode changes the way Windows interprets one of the color bits. The "Background Intensity bit" becomes the "Foreground Blink bit" when the console is in the FullScreen Mode. That means that, like a DOS screen, a FullScreen console window cannot display high-intensity background colors. If you specify a high-intensity background color for a FullScreen display, the low-intensity version of the background color will be shown and the foreground color will blink. If you use Alt-Enter to turn the FullScreen Mode off, the blinking will stop and the background colors will be high-intensity.

To help you with this situation, the ConsoleScreenLoad function can optionally strip the "blink bit" from incoming screens.   To see this effect, change the last parameter of the previous example to --1...

```
      ConsoleScreenLoad "\CONTOOLS\COLORS.256", 1, 25, -1
```

Minus one is the ConsoleScreenLoad switch for "strip blink bit". (You can also experiment with the numbers 1-255 for this parameter.)

It would be fairly simple to use code like this...

```
      ConsoleScreenLoad "SCREEN.FIL",1,25, NOT ConsoleIsFullScreen
```

...to automatically strip the blink bit if the console window is not in the FullScreen Mode. Since ConsoleIsFullScreen returns a Logical True/False value, if the console *is* in the FullScreen Mode it will return True. And since NOT True equals False, the False value (i.e. zero) would be passed to the ConsoleScreenLoad function when the console was in the FullScreen Mode, and the blink bit would *not* be stripped. If the console was *not* in the FullScreen mode the ConsoleIsFullScreen function would return False, and NOT False equals True (-1), so the blink bit *would* be stripped. (It's convoluted, but it works perfectly.)

Appendix H: Logical True and False

VERY IMPORTANT NOTE: All Console Tools functions that return True/False values return *Logical True* and *Logical False* according to the descriptions in this Appendix. If you use Console Tools True/False functions with DWORD variables (which cannot accept the Logical True value of negative one) then the functions will *appear* to malfunction. Please read the following section of the Help File for a complete description of Logical True and False.

There are two different ways to look at the values of True and False.

The technical definition of False is **Zero**, and the technical definition of True is **Nonzero**. In other words, all Microsoft API functions and virtually all programming languages recognize zero as False and *everything else* as True.

Since computers use binary numbers -- ones and zeros -- it is fairly common to use zero for False and one for True. This works fairly well when all you're trying to do is specify a simple True/False value. For example, consider the following code...

```
%False  = 0
%True   = 1
DO
      INCR lCount&
      IF lCount& = 100 THEN lComplete& = %True
LOOP UNTIL lComplete& = %True
```

This code is very straightforward. You could also use this code...

```
%False  = 0
%True   = 1
DO
      INCR lCount&
      IF lCount& = 100 THEN lComplete& = %True
LOOP UNTIL lComplete&
```

...to accomplish exactly the same thing, because the simple expression "lComplete&" would be evaluated by PB/CC and when the value was True (nonzero) the program would exit from the loop.   And you could even do this...

```
%False  = 0
%True   = 1
DO
      INCR lCount&
      IF lCount& = 100 THEN lComplete& = %True
LOOP WHILE lComplete& = %False
```

... and it would work fine. But there is a significant problem with a True/False system that uses one (1) for the %True value. The following code will not perform the way you might expect...

```
'"broken" code...
%False  = 0
%True   = 1
DO
      INCR lCount&
      IF lCount& = 100 THEN lComplete& = %True
LOOP WHILE NOT lComplete&
```

This code looks like it should work, but there's a serious problem. Like all high-performance compilers, PB/CC uses binary ("bitwise") operations for logical operators like AND, OR and NOT. The value one (1) is evaluated as True -- remember that *all* nonzero values evaluate as True -- but here's the problem: if you do this...

```
%True = 1
PRINT NOT %True
```

... the screen will display negative two (-2) instead of zero, the value that you probably expected. Here's the reason. If you write out the value of one (1) in binary ones and zeros, you get this...

```
0000000000000001       'fifteen zeros and a one
```

The NOT operator reverses all of the bits (see the PB/CC documentation), so `NOT 1` yields this...

```
1111111111111110       'fifteen ones and a zero
```

...which evaluates to --2. **So, if you use one (1) for your %True value, "NOT %True" evaluates to --2, which is nonzero and therefore *also* evaluates as True.**

In the "broken" example above, while `lComplete&` is zero, `NOT lComplete&` evaluates to a nonzero value so the loop continues running. But if `lComplete&` is set to one (1) then `NOT lComplete&` *still* evaluates to a nonzero value and the loop *still* continues running.

It is possible, fortunately, to use a value for %True that works "logically" in virtually all cases. Consider this...

The binary representation of `%False` (*always* defined as zero) is sixteen zeros..

```
0000000000000000
```

If you do this...

```
%False = 0
PRINT NOT %False
```

...you will see that the value negative one (-1) is displayed. This is because the binary value `1111111111111111`, which is the same as `NOT 0000000000000000`, evaluates to negative one. (The reason that this bit pattern evaluates to --1 is pretty complicated, but if you're interested you can read about it in the PB/CC documentation. Take our word for it: `1111111111111111` evaluates to negative one in *all* computer languages that use "Signed Integers", as PB/CC does.)

So if you modified the "broken" example code above to use negative one for %True instead of one...

```
%False  = 0
%True   = -1
DO
      INCR lCount&
      PRINT lCount&
      IF lCount& = 100 THEN lComplete& = %True
LOOP WHILE NOT lComplete&
```

...it would work "logically", and the program would exit from the loop when 100 was reached.

The bottom line is that the use of --1 for %True can make your code easier to write *and* read.

There's one final "glitch" that you have to keep in mind, however. Negative one is (of course) a negative number, and *not all variable types can be used to store negative numbers*. The 32-bit PB/CC variable type "Long Integer" -- which is the fastest and most efficient PB/CC variable type -- *can* use negative numbers, so this is not usually a problem. But some programs and some Windows API functions use 32-bit DWORD variables, which are unsigned variables and won't accept negative values, so you'll be forced to use +1 and avoid NOT and other "bitwise" operations.

Suggested Reading
Look at the ISFALSE and ISTRUE functions in the PB/CC documentation, as well as the NOT operator and the IF/THEN statement.

Appendix I: Microsoft Error Numbers

If you need to know what a particular Microsoft Error Number means, look at the predefined equates that start with `%ERROR_` in the WIN32API.INC file that comes with PB/CC, and find the number that corresponds to the return value of the Console Tools function that reported the error.

The name of the equate (like `%ERROR_FILE_NOT_FOUND`) should tell you something about the error.

If the number is too large to be a Microsoft Error Number -- perhaps a number between 999,000,000 and 999,999,999 -- it is probably a Console Tools Error Number.

A discussion of Microsoft Error Numbers is beyond the scope of this Help File. (Not only that, but the "official" Microsoft error descriptions are protected by Microsoft's copyright, so at best we could only provide less-accurate summaries. We chose not to do that.)

We suggest that you use the free MSDN Online internet service to research Microsoft Error Codes, at microsoft.com. (Note that Microsoft error definitions do not use the leading percent sign. If you type a leading percent sign into the MSDN Online "search" field, you won't get any results.)

Another good source of information is the Microsoft WIN32.HLP file, which is supplied with most Microsoft programming languages (except Visual Basic). If you are a registered PB/CC owner you can download the file from PowerBASIC's web site. Older versions of WIN32.HLP are supplied with some other languages (like Borland's Resource Workshop) but the information is less complete and less up-to-date. The WIN32.HLP file does *not* contain descriptions of the individual Microsoft Error Numbers, but it does explain when various functions return the error. (Hint: Use the WIN32.HLP "Find" tab, not "Index" or "Contents".)

Appendix J: Tips for Developing and Debugging Console Applications

It wouldn't be possible to include a *complete* guide to writing and debugging console applications in this Help File. That topic could easily fill a large book, all by itself! What we've attempted to do here is give you a few pointers, to get you started.

- Be aware of the differences between Windows NT/2000/XP and Windows 95/98/ME. Whenever possible, test your programs on both operating systems *frequently* during development. (Windows 95, 98, and ME are similar enough that separate testing is usually not necessary, but NT is very different from 95/98/ME.) Many different screen operations like "maximize" can have very different effects on computers running different operating systems. And if you're using the Windows API, dual testing is absolutely *critical*. There are significant differences between the Windows NT/2000/XP and 95/98/ME API functions.

- Be aware that different computers will often have different Default Console Configurations (different fonts, etc.). A command like Maximize that produces one effect on your development computer may produce very different effects on other computers, even if they're using the same operating system. See Microsoft Console Windows for more information.

- Learn to use the Windows Task Manager. If you press Ctrl-Alt-Del, Windows will display the Task Manager and show you a list of the programs that are currently running on your computer. (On Windows NT/2000/XP computers you'll have to select the Task Manager button after you press Ctrl-Alt-Del.) The most important use of the Windows Task Manager will be shutting down programs that are not working correctly. This is sometimes called "doing an *End Task*". For example, if your program enters an endless loop you may not be able to stop it any other way. The Close button (**X**) may not work. Even more importantly, if you use `ConsoleWindow %HIDE` to remove your program from the screen and Task Bar, using the Task Manager may be the only way to shut it down, short of re-booting your computer. Keep in mind that using End Task stops your program no matter what it happens to be doing at the moment. If you use End Task while your program is writing to a disk file, for example, you will almost certainly end up with a corrupt file. If you are using Windows NT, 2000, or XP you can also use the Task Manager to perform an *End Process* operation, which will work even when a program does not respond to an End Task command. (For that reason alone, you should consider updating your development computer from Windows 95/98/ME to Windows NT/2000/XP.)

- On NT computers, leave the Task Manager running at all times. This will provide you with a "CPU Load" indicator in the System Tray (the lower-right corner of the screen where the time is usually displayed). This will help you know when your program is running away, or when it is using so much CPU time that other applications will be adversely affected. (The Console Tools `Delay 0` function can be used to fix this problem.)

- On Windows 95/98/ME computers, install and use the System Resource Meter that is included on the operating system installation disks. It will provide information similar to the NT CPU Load indicator (see just above).

- Be aware of the differences that your program will have to deal with if it is run on computers that have a different Screen Resolution (800x600, 640x480, 1024x768, etc.) Graphical elements will appear to be larger or smaller, depending on the screen settings. In some cases, two computers with the same screen resolution will produce different results because of the different Video Display Drivers that may be installed. Windows 95 supports "virtual desktops" that can confuse your program into thinking that a different display resolution is being used.

Appendix K: The WIN32API.INC File

The WIN32API.INC file is a large (750k+) "include file" that is provided with the PB/CC compiler. It contains predefined equates, user-defined-type structures, and function declarations for large portions of the basic Windows 32-bit Application Program Interface (the "Win32 API").

There are some advantages to using `$INCLUDE "WIN32API.INC"` in your PB/CC programs. Probably the greatest advantage is the convenience of being able to use the Win32 API without adding anything else to your code. Without the WIN32API.INC file, you'd have to cut-and-paste the equates, UDTs and declarations for the API functions that you want to use directly into your program each time you use an API function for the first time.

The *disadvantage* of using the WIN32API.INC file is compilation speed. If your program's source code files total 75k -- a fairly average program -- then adding WIN32API.INC would increase the total source-code volume by a factor of ten. In other words, instead of loading, reading, and parsing 75k of text, the PB/CC compiler would have to wade through 825k of text. And *most* programs never use 99.99% of the Win32 API, so all of that extra reading and parsing is usually wasted. And the time is wasted over and over again, every time you compile the program.

The WIN32API.INC file is not required by Console Tools. The Console Tools INC file, which is only 35k, contains duplicate equates for everything that Console Tools needs.

We strongly recommend, unless your program uses a large number of API functions, that instead of using $INCLUDE "WIN32API.INC" you consider the cut-and-paste method. If you use WIN32API.INC as a template, and *include only the parts of the WIN32API.INC file that your program actually needs*, your PB/CC compilation times will be cut dramatically.

Appendix L: The SKELETON.BAS Program

Rather than following the process described in Three Critical Steps For Every Program, you can use the `\CONTOOLS\SKELETON.BAS` program as a starting point for a *new* programming project.

We have duplicated the `SKELETON.BAS` file here, to make it easy to cut-and-paste directly into your new program...

```
'THIS IS THE SOURCE CODE FOR A "SKELETON" PROGRAM
'THAT YOU CAN USE AS A TEMPLATE WHEN BEGINNING A
'PB/CC PROJECT WHICH INCLUDES CONSOLE TOOLS.

'If your program will need the WIN32API.INC file,
'include it here, *before* the Console Tools INC file...
'$INCLUDE "WIN32API.INC"

'You'll need to edit this directory if you
'installed Console Tools someplace else.
'You must use ONE of the following lines,
'depending on whether you are using the
'Standard or Pro version of Console Tools.
$INCLUDE "\CONTOOLS\CT_STD.INC"
$INCLUDE "\CONTOOLS\CT_PRO.INC"

FUNCTION PBMain PRIVATE AS LONG

    LOCAL lResult&

    'PLEASE NOTE: Whenever the variable name lResult& is
    'used below, your program should check lResult& for
    'errors, at least during development.  If no errors
    'are found, the value %SUCCESS (zero) will be returned.
    'See the Help File for more details.

    'Add your Authorization Code here...
    lResult& = ConsoleToolsAuthorize(%MY_CT_AUTHCODE)

    lResult& = InitConsoleTools(0, 0, 0, 0, 0, 0)

    'Make sure the user can't use Alt-F4 to shut
    'down this program unexpectedly...
    DeleteWindowMenuItem %MENUITEM_CLOSE

    ConsoleTitle "My Program's Title"

    ConsoleIcon  %IDI_CONSOLE

    'You might want to hide the Windows 95/98/ME Toolbar
    'here, especially if your program will use Pulldown
    'Menus OR if it will be run on both NT and 95/98/ME...
    ConsoleToolBar %OFF, %SHOW

    'ADD YOUR CODE HERE...
```

```
        'Don't forget to add a WAITKEY$ or DELAY at the end
        'of your program if you need it to pause so that
        'you can see the screen when it ends...

    END FUNCTION

'[end of file]
```

Appendix M: Using CTDEMO.DLL for Shareware, Freeware, Public Domain, Demo, and other Not-For-Profit Software

This topic is no longer pertinent to Console Tools.

Starting with Console Tools Version 2.50, it is no longer necessary to use a "Demo DLL" with Shareware, Freeware, Public Domain, or Demo software.

Appendix N: The HIDECONS.EXE Program

PLEASE NOTE: The HideCons program was first distributed with Console Tools Version 1.00. When Console Tools Version 2.00 was released, it included a new program called CWC which can perform the same "Hide Console" function as HideCons, but it does much, much more. The HideCons program is still included with Console Tools so that programmers that are using it will be able to continue, but we do suggest that you use CWC instead of HideCons.

Some programmers prefer that their console applications run full-time as Hidden windows (see Console Window States) and only appear on the screen when an error message needs to be displayed. If you use a Windows Shortcut to launch a program, you can specify whether it should start out Normal, Maximized, or Minimized, but you can't specify Hidden, so the console window always appears at least briefly when the application is launched. (This is a function of the way Windows works. It is not a defect in PB/CC or Console Tools.)

Some other programmers prefer to use Shortcuts to launch their applications as minimized windows, so that (for example) the Console Icon and Console Title can be set before the window becomes visible. Unfortunately, certain console parameters such as the window size and location cannot be adjusted while the console window is minimized. To produce a clean display during program initialization, it is usually necessary to Hide the application with the ConsoleWindow function, make all of the necessary adjustments, and then Show it again. But since Windows Shortcuts do not allow you to launch applications as Hidden, the final result is usually less than ideal because the console window "flashes" on the screen when the program is started, before it can be hidden. (Another problem with this method is the difficulty associated with programmatically creating Shortcuts on your user's machine during the installation process.)

For those and other reasons, Perfect Sync has developed the `HIDECONS.EXE` program.

"HideCons" stands for Hide Console, and it is a program that can be used to launch a console application in the Hidden mode so that it does not appear on the screen immediately after it is started. For example, if you open a Command Prompt window and type the following...

```
      HIDECONS  MYPROG.EXE
```

...Windows will run the HideCons program, and HideCons, in turn, will run the program called `MYPROG.EXE` in a *hidden* console window. MyProg will remain hidden until it (optionally) uses the ConsoleWindow function to make itself visible. (See "Making a Hidden Console Visible" below.)

The HideCons program does not remain in memory. As soon as it has launched the requested application, it terminates. The application that is launched is not dependent upon HideCons once it starts running. It does not run "inside" HideCons.

Additional Command Line Parameters

The HideCons program supports the passing of command line information to your console applications. For example...

```
      HIDECONS  MYPROG.EXE  TESTING 1 2 3
```

...would run MyProg in a hidden window, and the string "TESTING 1 2 3" would be passed to MyProg's COMMAND$ function. (See the PB/CC documentation for help with COMMAND$.)

Using the HideCons Program

There are three basic ways to use HideCons.

Command Line Method: As shown above, you can use a command line string to pass the name of a program (and optional parameters) to HideCons. This can be done from a Command Prompt, from a Batch File, or from a Windows Shortcut.

HIDECONS.CFG Method: If the command line that is passed to HideCons is blank (as it would be, for example, if you simply double-clicked on `HIDECONS.EXE` in the Windows Explorer program), HideCons will look for a file called `HIDECONS.CFG`. If it finds it in the current directory, HideCons will open the file and read a single line of text, which it will then process as if was a command line. For example, placing this string in the `HIDECONS.CFG` file...

```
MYPROG.EXE  TESTING 1 2 3
```

...and running HideCons *without* a command line would have exactly the same effect as the command line example shown above. Note that the word `HIDECONS` does *not* appear *in* the CFG file. If it did, HideCons would launch HideCons, and an endless loop would begin.

IMPORTANT NOTE: The `HIDECONS.CFG` file must contain a line of text which ends with a Carriage Return/Line Feed. Most text editors can be used to produce files of this type, but you must make sure that you press the Enter key at the end of the line of text, before saving the file. The CFG files can also, of course, be created programmatically.

Combination Method: If you pass the name of a CFG File on the command line, the HideCons program will read that file instead of `HIDECONS.CFG`. For example, using...

```
HIDECONS  MYPROG.CFG
```

...would tell HideCons to look for a file called `MYPROG.CFG` and read the first line of text from that file. You can use any file name, but it must have the extension CFG. Please note that the name `MYPROG.CFG` in this example does not necessarily mean that the program being run is called `MYPROG.EXE`. The `MYPROG.CFG` file could just as well contain the name of a program like `MP.EXE`, or anything else.

If you only use HideCons for one program, you can probably use any of the methods described above. If you use it for two or more programs, we recommend the Combination Method because it allows a single HideCons program to run many different programs, via different Shortcuts.

Making a Hidden Console Visible

Because of the way Windows works, it is not possible to simply use `ConsoleWindow %SHOW` to un-hide an application that has been hidden with HideCons. It will work sometimes, but it is not reliable. (The reasons are obscure, complicated, and unimportant.)

We recommend that you use `ConsoleWindow %RESTORE` or `ConsoleWindow %MAXIMIZE` when you want an application that has been hidden by HideCons to become visible. It is also possible to use `ConsoleWindow %SHOW` twice.

You should also note that, since the user will click on HideCons to launch your program, your program will not have the keyboard focus when it becomes visible. If you want your program to become the active window (the one that has a highlighted title bar and that receives keyboard input) you will need to use the ConsoleToForeground function immediately after using ConsoleWindow to un-hide the program.

Changing the HideCons Program's Name

Since it may not be desirable for you to distribute a program called HideCons with your applications -- after all, you probably want your users to double-click on something like `MyProg.EXE` instead of `HIDECONS.EXE` -- we have made it possible to rename HideCons. And if you do rename it, it will automatically look for a different CFG file name.

For example, let's say that your PB/CC program is called MyProg.EXE. You have two basic alternatives:

1) You could put the string `MYPROG.EXE` in a text file called `HIDECONS.CFG`, and whenever HideCons was run without a command line it would automatically read the `HIDECONS.CFG` file and launch MyProg. Or...

2) If you wanted to disguise the HideCons program by giving it a different name, you could rename your PB/CC program to something like `MP.EXE`, rename the `HIDECONS.EXE` program to `MYPROG.EXE`, and create a text file called `MYPROG.CFG` containing the string `MP.EXE`. Then when the file `MyProg.EXE` (which was actually a copy of HideCons) was run without a command line, the program would automatically detect that it had been renamed to MyProg, read the `MYPROG.CFG` file, and launch `MP.EXE`.

Appendix O: The CWC (Console Window Configuration) Program

By using the Window Properties/Font Menu, the user of a console application can change the font type and/or font size while a program is running, but Windows does not provide a technique that allows applications to change those settings programmatically. According to Microsoft, it is simply not possible for a Windows console application to change its own console window's font type or font size while it is running. And while it is possible to manually pre-configure a computer so that a certain font type and size will be used for a console application, this technique has several limitations, especially on Windows 95/98/ME computers.

So Perfect Sync created the CWC Program. CWC stands for Console Window Configuration. As the name implies, CWC allows you to reliably pre-configure a console window and then launch a PB/CC console application so that it runs inside the console window.

You can pre-select either the TrueType (Lucida) font or the Non-TrueType (Bitmap/Raster) font, and you can specify any font size that is valid for a console window. (The Window Property Menu does not allow the manual selection of all of the valid font sizes, but CWC allows you to use them all.)

You can optionally specify a font size of "MAX" and CWC will use the largest font that will allow an 80x25 console window to fit on the desktop. This option works regardless of the screen resolution and the size/location of the task bar, and CWC completely bypasses the buggy Windows 95/98/ME Auto Font Size feature. Options are also provided to make it easier to use other console sizes, like 80x43.

CWC can also be used to launch a console application in the fullscreen mode, pass a command line to an application, pre-set the console title, and turn off the Win95/98/ME console toolbar

CWC can also launch a console app with the Window State that you specify, such as Maximized, Minimized, or Hidden. (Launching a PB/CC app in a hidden window allows your program to change the console title, console icon, console size, and many other parameters before the console window appears on the screen, resulting in a very clean and professional-looking startup.)

The CWC Program

CWC consists of two files. The program itself is called `CWC.EXE`, and the program's configuration file is called `CWC.CWC`. Both files can be found in the directory where you installed Console Tools (usually `\CONTOOLS`).

To make CWC easier to use, the files can be renamed. For example, if you want to use CWC to configure a console window for a program called `MyProg.EXE` you could rename `CWC.EXE` to something like `RunMP.EXE`. Then, whenever it was run, the CWC program would automatically detect its new name and look for a configuration file called `RunMP.CWC`. (This technique also allows the use of CWC with several different programs in one directory, each with its own configuration file.)

For clarity, the rest of this discussion will assume that you have not renamed the CWC files.

The first step in using CWC is to place `CWC.EXE` and `CWC.CWC` in the same directory as the program that you want to configure. (It is not strictly necessary to use the same directory, but it makes things easier.)

The second step is to use a text editor to modify the `CWC.CWC` file. When you first install CWC it will look something like this:

```
------- Console Window Configuration (CWC) File -------
PROGNAME:
CMD_LINE:
TITLEBAR:
FONTTYPE: Bitmap
FONTSIZE: Max
TOOL_BAR: No
WINSTATE: Maximized
FULLSCRN: No
QUICK_ED: No
COMMENTS:
-------------------------------------------------------
```

On the line that says PROGNAME, type the exact name of the program that you want to configure. Save the file, and run the `CWC.EXE` program. It will read the `CWC.CWC` file, configure the console window (more about this in a moment), and run your program. The CWC program itself will never appear on the screen. It is virtually undetectable.

Now we'll examine all of the lines in the `CWC.CWC` file, and describe what they do.

PROGNAME: The name of the program that you want to run in a pre-configured console window. If you placed the CWC files in the same directory as the target program, you do not have to specify the drive and directory. If you do not specify a file extension, `.EXE` will be assumed. Please note that on Windows 95/98/ME computers, the executable program must be located on **1)** a local drive or **2)** a network drive that has been mapped to a drive letter. Windows 95, 98, and ME do not allow CWC to run programs from a network "UNC" directory (i.e. from a location that starts with two backslashes).

CMD_LINE: (Optional): If you enter something on this line, it will be passed to your program as a "command line". Your PB/CC program can then read the string with the `COMMAND$` function.

TITLEBAR: (Optional): If you enter something on this line, it will be displayed in the console window's title bar. (It is also possible for your PB/CC program to use the Console Tools ConsoleTitle function to change the title, but pre-setting it with CWC makes sure that the title is correct when the console window first appears on the screen.) If you do not specify a title, CWC will use the executable program's name plus "(CWC)".

FONTTYPE: If you enter "`TT`" or "`TrueType`" on this line, the console window will be created using the console's TrueType font. If you type "`BMP`" or "`BitMap`" or "`Raster`", the console will be created using the console's non-TrueType font. If you leave this line blank or use a string that CWC does not recognize, the non-TrueType (BitMap/Raster) font will be used. (See notes below about choosing between TrueType and non-TrueType.)

FONTSIZE: If you enter the word "`Max`" on this line, CWC will automatically use the largest font (of the specified FONTTYPE) that it can, without creating a console that is too wide to fit on the current desktop. "`Max`" will usually, but not always, completely fill the screen from left to right. For example, if your desktop is in the 1024x768 mode, Windows does not provide a console font size that allows the screen to be filled completely. (Doing so would require a font that is 12.8 pixels wide, which does not exist.)

If you type a number with a percent sign after the word Max, like "`Max 50%`", CWC will use the largest font that it can, but only fill that percentage of the screen from left to right. For example, using "`Max 50%`" would create a console window that was half the width of the screen. The console would therefore fill roughly one-quarter of the screen. This technique is especially useful if your program uses console sizes other than 80x25. For instance, if your PB/CC program will be using the Console80x function to select 80x43 you could use "`Max 50%`" to create a console that would allow an 80x43 console to be displayed without scroll bars. You can use any number from 10% to 100%.

If you enter a single number like "`16`" on the FONTSIZE line, CWC will interpret it as a "point size" and will automatically calculate the appropriate TrueType font size (such as 10x16). You may use any number from 1 to 72. Please note that if you specify a non-TrueType FONTTYPE but you use a single number on this line, CWC will automatically switch to the TrueType font. Non-TrueType fonts cannot be specified with a single number.

If you enter something like "`10x18`" on the FONTSIZE line, CWC will use that font size. You are responsible for using a font size that is valid for the specified font type (see lists of font sizes below). If you use an invalid size, Windows will automatically attempt to use the closest valid size. But Windows has been known to make some unusual choices, so we recommend that you consult the lists below.

TOOL_BAR: If you type "`Y`" or "`Yes`" or the number "`1`" in this line, CWC will create a console that has a Windows 95/98/ME toolbar. If you type "`N`" or "`No`" or zero ("`0`"), or use a value that CWC does not recognize, it will create a window without a toolbar. (Windows NT, 2000, and

XP consoles do not have toolbars, so this line is ignored when CWC is used on NT computers.)

WINSTATE: If you type "`Max`" or "`Maximized`", a maximized console window will be created. The values "`Min`", "`Minimized`", "`Hide`", "`Hidden`", and "`Show`" are also recognized. You can also use a number from `0 to 10` that corresponds to the numeric value of one of the first ten ConsoleWindow equates (`%Maximize=3`, etc.) that are listed in the Console Tools INC files file. For more information about the many benefits of using "`Hidden`", see HideCons. IMPORTANT NOTE: If you use CWC to create a hidden console window, be sure to read **Making A Hidden Console Visible** at the very end of this section of this document.

FULLSCRN: If you type "`Y`" or "`Yes`" or the number "`1`" on this line, the console will be created in the fullscreen mode. If you type "`N`" or "`No`" or zero ("`0`") or use a value that CWC does not recognize, the console will be created in the window mode. If the fullscreen mode is selected, the FONTTYPE, FONTSIZE and other settings above will be irrelevant when the console is first displayed. But you should not ignore the rest of the `CWC.CWC` settings just because you use the fullscreen mode. The settings that you specify will be used if your program is switched from the fullscreen mode to the window mode with Alt-Enter or the ConsoleWindow function.

QUICK_ED: All Windows console windows support a mode called "Quick Edit". The Quick Edit mode can be changed manually by using the console's Properties Menu. When the Quick Edit mode is enabled, the mouse can be used to highlight text within the console window, and mouse events (clicks, moves, etc) are no longer passed to your program's input functions. Windows 95, 98, ME, and NT all default to the "Quick Edit Disabled" mode (which most users prefer), but Windows 2000 and XP default to Quick Edit Enabled. The QUICK_ED line in the CWC file can be used to enable or disable the Quick Edit mode, regardless of the version of Windows that is being used. If you type "`Y`" or "`Yes`" or the number "`1`" on this line, the Quick Edit mode will be enabled. If you type "`N`" or "`No`" or zero ("`0`") or use a value that CWC does not recognize, the Quick Edit mode will be disabled.

COMMENTS: You may use this line for anything you like. It is ignored by CWC. IMPORTANT NOTE: Everything after COMMENTS in the `CWC.CWC` file will be ignored. If you rearrange the lines in the file so that COMMENTS comes before a line like FONTTYPE or FONTSIZE, those settings will be ignored.

One final note... The first and last lines of the `CWC.CWC` file (the ones with the dashes) are optional. In fact just about everything except the PROGNAME line is optional, and the lines in the file can appear in any order, as long as COMMENTS is last. If you use the appropriate keywords (like PROGNAME and FONTTYPE) and the colons, CWC will recognize them. That means that you can write simple programs that create CWC.CWC files without worrying about the exact file format.

<u>Using Different CWC Configuration Files</u>

It is possible to tell `CWC.EXE` (or a renamed `CWC.EXE` file) to use a specific configuration file instead of the default `CWC.CWC` file. The configuration file can have any name, and can be located in any directory that is accessible to the `CWC.EXE` program.

Simply pass the name of your console configuration file as the command line of `CWC.EXE`. For example, to use a console window configuration file called `C:\MyDir\MyConsole.CFG` you would do this:

```
CWC.EXE  C:\MyDir\MyConsole.CFG
```

You could accomplish this with a batch file, or by using the Windows Start/Run menu, or by creating a Windows Shortcut.

The file that you specify on the `CWC.EXE` command line must have the same internal format as `CWC.CWC` (see above), but it can have any name and extension.

<u>Choosing Between TrueType and Non-TrueType Fonts</u>

Generally speaking, Perfect Sync recommends the use of the *Non*-TrueType console font. While the TrueType font is available in more sizes and looks better in very large sizes, the non-TrueType font is bolder and is significantly more readable in normal sizes.

Also, the True Type font that is supplied with Windows NT/2000/XP computers does not support the entire 256-Character Extended ASCII Set. All of the control characters (below ASCII 32) and extended characters (above ASCII 127) appear as a nondescript "box".

Another consideration is the consistency of the Non-TrueType fonts. They exist in the same standard sizes on all Windows 95b, 98, ME, NT, 2000, and XP computers. And while the TrueType fonts are very similar on 98, ME, and NT computers -- they all use a sans serif font called "Lucida Console" -- on Windows 95 computers the TrueType console font is "Courier New", which is an old-fashioned-looking serif font that has a different "aspect ratio" from the 98/ME and NT fonts. (The 98/NT Lucida font has an optimal aspect ratio of 0.60 but the 95 Courier New font has a ratio of approximately 0.50.)

The Windows 95 TrueType font is also somewhat less predictable and less complete than all of the other fonts. For example, there is no such thing as a Windows 95 TrueType console font that is 9, 10, or 11 pixels high. (See tables of valid sizes below for more details).

If your program is limited to NT/2000/XP and 98/ME computers, the TrueType and Non-TrueType fonts will both work reasonably well for you. But because of the readability of the two font types, we still recommend the use of the Non-TrueType console fonts.

<u>NON-TRUETYPE (Bitmap/Raster) Console Font Sizes</u>

Most console window Properties Menus can be used to select the following font sizes.

`4x6`, `5x12`, `6x8`, `7x12`, `8x8`, `8x12`, `10x18`, and `12x16`.

The WxH notation, like "`10x18`", indicates a console font's Width and Height. A `10x18` font would be 10 pixels wide and 18 pixels high.

Windows NT/2000/XP computers also allow the selection of `16x8` and `16x12`.

In almost all cases, CWC will allow you to use all of those font sizes, regardless of whether or not your program is being run on 95/98/ME or NT. (Note: Some Windows 95a computers only support the `8x12` non-TrueType font. As far as we know, Windows 95b, 98, and NT support all of the sizes that are listed here.)

CWC also allows the use of many different font sizes that cannot normally be selected from the Properties Menu. Specifically, you can usually use "multiples" of the standard sizes. For example, the standard `5x12` font can be multiplied by two to get a `10x24` font.

Here is a table that shows the "base sizes" for the non-TrueType console fonts, and some of their known multiples. (The word "no" means that Windows does not accept the font size that would normally appear at that location on the table.)

BASE	x2	x3	x4	x5	x6	x7
4x6	8x12	no	16x24	no	32x48	28x42
5x12	10x24	15x36	20x48	25x60	no	
6x8	12x16	18x24	24x32	30x40		
7x12	14x24	21x36	28x48			
8x8	16x16	24x24	32x32			
10x18	20x36	30x54	40x72			
16x8	32x16	no				
16x12	32x24					

Other types of multiples are also available. For example, the `8x8` font can be multiplied to `24x36`. Notice that the W and H numbers were not multiplied by the same number. And using the `5x12` font as a base, `5x24` and `10x36` fonts can be used. (In fact, non-equal multiples are the basis of the `16x8` and `16x12` fonts that are listed on the Properties Menus of NT computers: they are multiples of `8x8` and `8x12`, respectively. `16x8` and `16x12` are not really "base" font sizes.)

But don't be surprised if you try a multiple and it doesn't work. Four fonts are listed as "no" in the table above because they do not produce the expected results. And it seems like you should be able to use the `4x6` font as a base and get `12x12`, but for some reason Windows does not accept `12x12`.

Please note that if you use a nonstandard font size, the Window Menu Properties Font menu and the Windows 95/98/ME toolbar will not be able to display the size correctly. (To be clear, the font itself will be displayed correctly in the console window, but the font-size *displays* will not show the correct numbers.)

Warning about Windows 95a

During the testing of CWC we encountered a version of Windows 95a that only supported the 8x12 non-TrueType font! Perfect Syncs strongly recommends that you upgrade to Windows 95b, Windows 98, Windows NT, Windows 2000, or Windows XP, to make sure that a complete selection of font sizes is available.

WINDOWS 98 AND NT TRUETYPE Console Font Sizes

Any integer between 1 and 72 can be used to specify the size of a TrueType font. If you use this "single number" technique, CWC will automatically calculate the correct font size for you. For example, if you use `16`, CWC will use `10x16` on Windows 98 and NT computers, and `10x18` on Windows 95 computers.

You should be aware that not all 98/NT TrueType fonts have exactly the same proportions. If you examine the table below, you will notice that most font-widths are associated with two different font heights. For example, if your desktop is in the `800x600` mode you will find that using a font that is 10 pixels wide will produce a console which fills the screen from left to right. (80 characters times 10 pixels per character = 800 pixels.) That means that you could use either 16 or 17 for the FONTSIZE value, because both of those numbers produce a font-width of 10 pixels. Using 17 will produce a slightly taller console window than using 16. (And remember that non-TrueType fonts in `10x18` and `10x24` sizes are available, so you really have a number of possible choices.)

Pt	Size	Pt	Size
5	3x5	20	12x20
6	4x6	21	13x21
7	4x7	22	13x22
8	5x8	23	14x23
9	5x9	24	14x24
10	6x10	25	15x25
11	7x11	26	16x26
12	7x12	27	16x27
13	8x13	28	17x28
14	8x14	29	17x29
15	9x15	30	18x30
16	10x16	31	19x31
17	10x17	32	19x32
18	11x18	33	20x33
19	11x19	34	20x34

When it has a choice, CWC will alway uses the largest possible font. For example, if your desktop is using the `800x600` mode and you tell CWC to use...

```
FONTTYPE: TrueType
FONTSIZE: Max
```

...CWC will use `10x17` instead of `10x16`.

WINDOWS 95 TRUETYPE Console Font Sizes

Windows 95 TrueType Console Fonts are much less flexible and predictable than 98/NT TrueType fonts. Here are several examples:

1) There are no Windows 95 TrueType console fonts that produce characters that are 9, 10, 11, 19, 26, or 28 pixels high.

2) Using a point size of `13` produces the unexpected result of a `6x12` font (instead of something-by-13), and using `34` produces `19x33`.

3) The "aspect ratio" (the Width-to-Height ratio) of the Windows 95 TrueType console font varies wildly. While 98/NT TrueType fonts have a very consistent aspect ratio of `0.6`, the 95 fonts vary from `0.42` to `0.57`.

Pt	Size	Pt	Size
7	4x7	21	11x21
8	5x8	22	12x22
9	(none)	23	13x23
10	(none)	24	13x24
11	(none)		
		25	14x25
12	5x12		
		26	(none)
13	6x12		
		27	14x27
14	7x14		
15	7x15	28	(none)
16	8x16	29	15x29
17	8x17		
		30	16x30
18	10x18		
		31	17x31
19	(none)	32	17x32
20	10x20	33	18x33
		34	19x33

<u>Making a Hidden Console Visible</u>

Because of the way Windows works, it is not possible to simply use `ConsoleWindow %SHOW` to un-hide an application that has been hidden with CWC. It will work sometimes, but it is not reliable. (The reasons are obscure, complicated, and unimportant.)

We recommend that you use `ConsoleWindow %RESTORE` or `ConsoleWindow %MAXIMIZE` when you want an application that has been hidden by CWC to become visible. It is also possible to use `ConsoleWindow %SHOW` <u>twice</u>.

Appendix U: Updating from Console Tools 2.xx to 2.50+

Effective with Version 2.50, Perfect Sync "relaxed" certain portions of the Console Tools Software License Agreement with regard to the distribution of the Console Tools DLL with "Shareware, Freeware, Public Domain, or Free Demo software".

Console Tools Versions 2.50 and above (which we will call **2.50+**) also allow the use of Graphics Tools Version 2. Console Tools Versions below 2.50 (**<2.50**) are not compatible with Graphics Tools Version 2, and vice versa.

Console Tools Version 2.50+ also includes a number of features that were not present in versions <2.50. See the list at the end of this section for details.

If you have been using a version of Console Tools below 2.50 you *can* continue to do so. The installation of 2.50+ does not delete the old `ConTools.DLL` or `ConTools.INC` file, so you can continue to use them if you like. For example, you may have existing programs that you do not wish to update to 2.50+. Those programs will continue to operate exactly as before, but keep in mind that the *old* Software License Agreement, with its different terms, still applies to the `ConTools.DLL` file. The new license terms apply only to the new `CT_Pro.DLL` and `CT_Std.DLL` (**`CT_*.DLL`**) files. To review the License Agreement for versions of Console Tools before version 2.50, please visit http://perfectsync.com/SoftwareLicenseAgreement.htm. To review the 2.50+ License Agreement, click here.

If you wish to use the new version of Console Tools with your programs, you will need to update your program(s) by following this checklist:

1) Use Windows Explorer to locate the directory where Console Tools is installed. It is *usually* `C:\ConTools\`.

2) If you are using Console Tools Standard, locate the new file called `CT_Std.DLL`. If you are using Console Tools Pro, locate `CT_Pro.DLL`.

3) Right-click on the file and select Copy from the popup menu.

4) Use Windows Explorer's Tools > Find > Files function to locate the `ConTools.DLL` file on your system. Keep in mind that two or more copies may be present in different locations. Also check network drives, if applicable. Make a list of the directories where the file is found, and then close the Find window.

5) If the `ConTools.DLL` file is present only in the `\ConTools\` directory, you can skip this step. Otherwise, use Windows Explorer to select each of the directories from step 4, one by one. Right-click on a directory name, then select Paste to place a copy of the new DLL in that directory. Repeat this process until a copy of `CT_Std.DLL` or `CT_Pro.DLL` has been placed in all of the directories that contain `ConTools.DLL`.

The new Console Tools DLL is now ready for use.

Repeat the following steps for each program that you wish to covert to Console Tools Version 2.50+.

6) In a Console Tools program's source code file, locate the line that looks like this:

```
$INCLUDE "ConTools.INC"
```

If you are using Console Tools Pro, replace that line with this line:

```
$INCLUDE "CT_Pro.INC"
```

If you are using Console Tools Standard, replace the original line with this:

```
$INCLUDE "CT_Std.INC"
```

The original line might say `#INCLUDE` instead of `$INCLUDE`, and you may have added a drive and/or path to the quoted part of the line. If so, make the same changes in the new line.

7) Also in the source code file, locate the line that contains the `InitConsoleTools` function. That function should be used only once in each Console Tools program.

8) Just *before* the `InitConsoleTools` line, add this line:

```
ConsoleToolsAuthorize %MY_CT_AUTHCODE
InitConsoleTools (etc.)
```

9) If you are using Console Tools Plus Graphics (Graphics Tools), then you will need to change one parameter of the `InitConsoleTools` line. You must change the *fourth parameter* (not necessarily the fourth *zero*) of `InitConsoleTools` to either **1)** the number 1 to tell Console Tools that your program is using Graphics Tools Version 1, or **2)** the number 2 to tell Console Tools that you are using Graphics Tools Version 2 STANDARD, or **3)** the number 3 to tell Console Tools that you are using Graphics Tools Version 2 PRO. *If you leave this parameter as zero (0) then Graphics Tools will not function properly with your program.* See InitConsoleTools and the Graphics Tools documentation for more information about this.

10) Re-compile your program.

That's it! The re-compiled program will now use the `CT_Std.DLL` or `CT_Pro.DLL` file instead of `ConTools.DLL`.

New Features in Console Tools 2.50+

- The ConsoleInputBox, ProgressBoxShow, and ConsoleListBox function can now create GUI elements with nonstandard button names. The button names can be defined by using the *sTitle$* parameters of those functions. For example, instead of displaying a "Cancel" button a Progress Box can now display "Stop" or "Quit", or any other label (in any language) as long as it will fit in the space that is provided. (Please note that the local-language button names that ConsoleMessageBox displays are defined by Windows, not by Console Tools.)

- The new CustomIcon function can be used to load an icon from a disk file. The icon can then be used by other Console Tools functions that use icons, such as ConsoleIcon and SplashBoxShow.

- A small number of minor bugs have been fixed, primarily related to unusual mouse problems when Windows 95/98/ME is used on certain systems.

- The Windows "message box and FPU" bug has been addressed.

- **Console Tools Plus Graphics Users:** You no longer need to distribute the `CT_Gfx.DLL` file with your applications. All of the functions from that DLL are now located in the main `CT_Pro.DLL` or `CT_Std.DLL` file.