Graphics Tools™

This is version 2.12 of the
Graphics Tools Help File
Build ID# 030606F

Copyright and Trademark Information

Perfect Sync, ***Graphics Tools***, ***Console Tools***, *and* ***SQL Tools*** *are trademarks of...*

PERFECT SYNC

8121 Hendrie Blvd., Suite C
Huntington Woods, Michigan (USA) 48070

You can visit us at http://perfectsync.com.

PowerBASIC®, PB/CC®, PB/DLL®, PB/Win®, and PB/DOS® *are trademarks of PowerBASIC, Inc.*

Microsoft®, Windows®, Windows 95®, Windows 98®, Windows ME®, Windows NT®, Windows 2000®, Windows XP®, Win32®, Internet Explorer®, Visual Studio®, *and* ***TrueType®*** *are registered trademarks of Microsoft Corporation.*

Netscape® *is a trademark of Netscape Communications Corporation.*

Intel® *is a registered trademarks of Intel Corporation.*

UniSys® *is a trademark of UniSys Corporation.*

MicroAngelo® *is a trademark of Impact Software.*

We apologize to the owners of other trademarks which may have been used in this document without recognition here. Please contact support@perfectsync.com and we will correct the omission in future versions of this file.

TABLE OF CONTENTS

Start Here!

This document is composed of four major parts.

The **User's Guide** is a series of short narratives that describe Graphics Tools. It covers several basic ideas that you'll need to understand. If possible, you should read the entire User's Guide because it includes "conceptual" information that will be valuable whenever you use Graphics Tools.

The **Reference Guide** lists all of the Graphics Tools functions, in alphabetical order, and provides complete, detailed information about each one. While it would be possible to read the Reference Guide from front to back -- and it would certainly teach you a *lot* about Graphics Tools -- most people simply look up the functions that they use, as they use them.

The **Appendices** cover stand-alone topics like Graphics Tools Error Codes and the numeric values that you need to use to access Windows Bitmaps and Icons.

The **Sample Programs** section contains descriptions of the various sample programs that are included with Graphics Tools.

When it is presented as a Help File, this document is extensively cross-referenced. Just click on a word or phrase that is highlighted like This and you'll jump directly to a more detailed discussion of that topic. Then you can click the Back button to return to where you were. The >> and << buttons at the top of the screen can be used to browse through the pages of the Help File, as if it was a book. At any time you may click the Contents or Index button, to see a complete Table of Contents or a dialog box that will enable you to search for a word or phrase.

What Is Graphics Tools?

What are "Graphics Tools" and "Console Tools Plus Graphics"?

Graphics Tools is a software development toolkit that allows Microsoft Visual Basic (VB) and PowerBASIC for Windows (PB/Win and PB/DLL) programmers to add easy-to-use graphics windows to their programs. Graphics windows can be used for just about any type of "visual output" that you can imagine, from the simple display of ready-to-use bitmaps, to bar charts, to sophisticated CAD programs.

Console Tools Plus Graphics is the combination of Graphics Tools and Perfect Sync's Console Tools product, which, together, provide PowerBASIC PB/CC programmers with functions for creating graphics windows that work well with console applications. For more information, see Console Tools Plus Graphics.

Also see What's the Difference Between Graphics Tools "Standard" and "Pro"?

What's the Difference Between Graphics Tools "Standard" and "Pro"?

Graphics Tools is available in two different versions, called "Standard" and "Pro".

The Standard version includes all of the *basic* functions that Visual Basic, PowerBASIC, and other programmers need to add high-quality graphics windows to their programs.

The Pro version includes everything that's in the Standard version, plus many additional functions and features.

Graphics Tools Pro Features

More Graphics Windows While Graphics Tools Standard allows your programs to create as many as four (4) graphics windows at the same time, Graphics Tools Pro can create up to 256.

Sizable and Movable Graphics Windows Create movable and resizable graphics windows with horizontal and vertical scroll bars, and much more. If desired, you can allow your users to resize and move graphics windows manually, using the mouse. You can even create "stretchable" graphics windows!

Improved Control Over the Graphics Window The GfxMove, GfxResize, and GfxScroll functions can be used to control the graphics window programmatically.

Gradients Most Graphics Tools figures (circles, rectangles, polygons, pies, etc.) can be filled using Gradients instead of solid colors. In addition to the familiar "blue fade", gradients provide a *huge* selection of very impressive color effects. Predefined gradients can be loaded with the GradientLoad function, or you can create your own with the UseGradient and GradientValue functions. Tell Graphics Tools which figures should be filled with a Gradient instead of a Brush by using the GradientBrush function, save your Gradients for future use with the GradientSave function, and access the colors of your gradients one by one with the GradientColor function.

The OverlayWindow Function OverlayWindow copies the contents of one graphics window into another much like DisplayWindow, but one color (which you specify) is treated as "transparent".

The DrawTube Function DrawTube draws a cylinder without an "end cap", resulting in a realistic "tube" effect when an appropriate Gradient is used.

Combine the Pro version's Gradients with the new 3D figures, and you won't believe your eyes!

Drag-Drop Capability The GfxDroppedFiles function can be used to obtain the names of disk files that have been dragged and dropped on a graphics window, as well as the exact location where the drop occurred.

Auto-Play of Animated Icons and Cursors. The new AutoPlayCursor, AutoPlayIcon, and AutoPlayControl functions allow your programs to play animations *continuously*, without any extra "play loop" code.

Printing of Images Graphics Tools Pro supports the printing of high-resolution, full-color graphics images. The GfxPrintWindow and GfxPrintArea functions can be used to print all or

part of any graphics window's image, and the GfxPrintSetup and GfxPrintPageSetup functions can be used to provide the user with standard printing-oriented dialog boxes. The GfxPrintStatus function can report the status of the most recently submitted print job, the GfxPrintParam function can be used to set certain printing parameters programmatically, and GfxPrintDefaults resets the printing functions to use the system default printer.

Display JPEG files The new DisplayJpeg, StretchJpeg, and CropJpeg functions allow the display of JPEG image files on any system. (Graphics Tools Standard can display JPEG, but only if Windows is configured to support JPEGs. Graphics Tools Pro can display JPEGs on any system.)

Save JPEG files The SaveGfxWindow and SaveGfxArea functions have been enhanced to allow Graphics Tools programs to save JPEG files in various formats and quality levels. (Graphics Tools Standard cannot save JPEG files.)

JPEG File Information The Pro version also includes the JpegParam function, which can retrieve "size" information from a JPEG file without displaying it.

Clip Areas: The flexible GfxClipArea function can be used to define areas of a graphics window where drawing will be allowed or rejected. A clip area can be any size and shape that you want to define -- you can even define multiple clip areas -- and any drawing operations that take place outside the specified area will not be visible. Clip areas are indispensable for creating a wide variety of otherwise hard-to-draw figures!

Software License Agreement & Runtime Distribution Rights

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

IMPORTANT NOTE: In this document, "Graphics Tools" refers to Perfect Sync's Graphics Tools Version 2, i.e. Graphics Tools Standard and Pro products with version numbers 2.00 through 2.99. This License Agreement is separate and distinct from the License Agreements for versions of Graphics Tools less than 2.00 and greater than 2.99.

Graphics Tools Standard License

The Graphics Tools Standard License allows you to install the Graphics Tools Standard development package (the contents of the Graphics Tools Installation File as originally received from Perfect Sync or an authorized distributor) on a single development computer, to use the package for software development, and to distribute the Graphics Tools Standard Runtime Files as described in "General Provisions" below.

Graphics Tools Pro License

The Graphics Tools Pro License allows you to install the Graphics Tools Pro development package (the contents of the Graphics Tools Installation File as originally received from Perfect Sync or an authorized distributor) on up to four (4) development computers, to use the package for software development, and to distribute the Graphics Tools Pro Runtime Files as described in "General Provisions" below.

General Provisions

You may distribute the Graphics Tools Runtime Files with applications which you develop, which require non-trivial functionality in the Runtime Files to operate properly, and which add significant functionality to the Runtime Files.

It is important to note that the right to distribute the Graphics Tools Runtime Files cannot be granted *by you* to another entity (a person or corporation). For example, if you use Graphics Tools to create a "charting and graphing" package that is intended for use by other software developers, you may distribute Graphics Tools with that package. But the entities to which you license your package are *not* authorized to distribute the Graphics Tools Runtime Files with *their* products. If they wish to do so, they must obtain a valid Graphics Tools license from Perfect Sync or an authorized distributor.

IMPORTANT NOTE: Each Graphics Tools Runtime File is serialized. The unique Authorization Code that is embedded in each copy of the Runtime Files 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 Files will self-deactivate or malfunction if tampering is detected. Perfect Sync cannot be held responsible for damage to databases or other files that may be caused by a Graphics Tools Runtime File that has been intentionally altered. (See LIMITED WARRANTY below.)

If you have not purchased a Graphics Tools Software License from Perfect Sync or an authorized distributor then you are not legally entitled to use the Graphics Tools Runtime Files for software development or to distribute the Graphics Tools Runtime Files 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

This Software License is an agreement between the Licensee ("you") and the Licensor (Perfect Sync, Inc.). By installing Graphics Tools (the "software") on a computer system and/or by using the Graphics Tools machine-executable files ("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 (Graphics Tools Standard License) or on four computer systems (Graphics 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 Graphics Tools License from Perfect Sync or from an authorized distributor may reproduce and distribute the Graphics Tools Runtime Files, and then only with application(s) that 1) are written by the licensee, 2) require the Runtime Files to operate, and 3) add significant functionality to the Runtime Files. 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 2002 Perfect Sync, Inc

...you may distribute the Graphics Tools Runtime Files royalty free.

The Perfect Sync Authorization Code which is provided in human-readable form with the Graphics Tools installation package is also embedded in the Graphics Tools Runtime Files and is considered to be part of the Runtime Files. 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 Graphics Tools Runtime Files by others, and is legally equivalent to the unauthorized distribution of the Runtime Files themselves.

No other portion of the Graphics 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.

Graphics Tools Authorization Codes

This is a topic that all Graphics Tools programmers should read and understand thoroughly. If you have any questions, 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 Graphics Tools licensee, intend to comply with the terms of the Graphics 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, Graphics 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 Graphics Tools Runtime File is serialized. That means that your copies of the Runtime Files contain a unique, embedded key number called an Authorization Code. Nobody else's Graphics Tools Runtime Files have the same Authorization Code as your copies of the Runtime Files. This allows Perfect Sync to identify a Graphics Tools Runtime File that is being used illegally (i.e. distributed in violation of the Graphics Tools License Agreement) and to determine the identity of the original licensee.

In order to use a Graphics Tools Runtime File, you must prove to the Runtime File that you know its correct Authorization Code by using the GraphicsToolsAuthorize function. This is done so that when you distribute the Graphics 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 Graphics Tools Runtime File(s) illegally. YOU are legally responsible for preventing that from happening!

Using the GraphicsToolsAuthorize Function

If you don't use the GraphicsToolsAuthorize function at all, the InitGraphicsTools, OpenGfx and other functions will refuse to work, making it impossible for your program to use Graphics Tools in any way.

If you use the GraphicsToolsAuthorize 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

GraphicsToolsAuthorize function to test all of the possible Authorization Codes one by one, until it found one that worked with your Runtime File. The GraphicsToolsAuthorize 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 GraphicsToolsAuthorize 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 GraphicsToolsAuthorize function to return SUCCESS. This makes it much more difficult to use the GraphicsToolsAuthorize 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 GraphicsToolsAuthorize function will return SUCCESS, the InitGraphicsTools and OpenGfx functions will work properly, and *all other Graphics Tools functions will appear to work properly*. But in reality, the Graphics Tools Runtime File will purposely malfunction. At random intervals, many different Graphics Tools functions will produce results that are completely or partially incorrect. For example, every so often a drawing function like DrawCircle might return SUCCESS when it actually -- purposely -- drew a circle of the incorrect size. This will make the Graphics Tools Runtime Files seem to work properly most of the time, but they will be unreliable.

Don't worry, the Graphics 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 Graphics 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 GraphicsToolsAuthorize 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 GraphicsToolsAuthorize will accept. If you mis-type three out of eight digits... well, you should probably take typing lessons before attempting to use Graphics Tools.

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

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

Please see Four Critical Steps For Every Program and GraphicsToolsAuthorize for more information.

Installing Graphics Tools on your Development Computer

IMPORTANT INFORMATION!

You must perform these steps before
using Graphics Tools for the first time!

Graphics 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 Graphics Tools will be installed. The default directory is `\GFXTOOLS`, and the rest of this section will assume that you used the default.

Graphics Tools Standard uses a runtime file called `GfxT_Std.DLL`, and Graphics Tools Pro uses `GfxT_Pro.DLL`. We refer to these as the `GfxT_*.DLL` files. *These files are used internally by all Graphics Tools programs, even Visual Basic programs that use the OCX version of Graphics Tools.*

In order for your programs to be able to *use* the appropriate Graphics 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 `GfxT_*.DLL` file somewhere on your computer's hard drive. (We recommend that you leave the original copy in the `\GFXTOOLS` directory, to serve as a backup.)

The ideal location for the `GfxT_*.DLL` file is *the same directory as the executable program that you are developing*, but keep in mind that you may be developing graphics 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 run a Graphics Tools program and you see a Windows message box that says something like...

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

...or you see a Visual Basic message box that says *File Not Found*, it means that Windows was unable to link Graphics Tools to your program's EXE. This almost *always* means that the `GFXT_STD.DLL` or `GFXT_PRO.DLL` file needs to be copied to a location where your program can find it.

Visual Basic Users

Microsoft Visual Basic has certain rules that govern where a DLL file must be placed. In some cases a copy of the Graphics Tools DLL must be placed in your "Vb" directory for use during development, and someplace else when it is used with a finished program. For this reason, we usually suggest that Visual Basic users place a copy of their Graphics Tools DLL in the Windows System directory (see above). This will insure that Windows -- and therefore Visual Basic -- will always be able to find it.

Your Authorization Code

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

```
MY_GFXT_AUTHCODE = &H........
```

Type your Graphics 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! Graphics 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 `\GFXTOOLS` directory. A variety of bitmaps, icons, cursors, sample programs, and other files are provided. You should also look at your system's Start Menu, where you will find several Graphics Tools components listed under the heading "Perfect Sync Development Tools".

See Distributing the Graphics Tools Runtime Files and Four Critical Steps For Every Program.

Distributing the Graphics Tools Runtime Files

IMPORTANT NOTE: You may duplicate and distribute the Graphics Tools Runtime Files only in accordance with the terms of the Graphics Tools Software License Agreement.

If you are using the DLL version of Graphics Tools with PowerBASIC (or another SDK-style language) you will need to distribute the `GfxT_Std.DLL` or `GfxT_Pro.DLL` file, depending on the version of Graphics Tools that you are using.

If you are using the OCX version of Graphics Tools with Visual Basic or another language that uses ActiveX controls, in addition to the DLL you will *also* need to distribute the `GfxT_Std.OCX` or `GfxT_Pro.OCX` file, plus a variety of other Visual Basic runtime files that are necessary for the OCX to operate. Your programming language can provide a list of "file dependencies" for the Graphics Tools OCX file, and your setup/installation program can install them at the same time it installs your program.

If you are using the Graphics Tools Pro JPEG features you may also need to distribute either 1) the Graphics Tools JPEG Library DLL or 2) the Intel JPEG Library. (This is true regardless of whether you use the DLL or OCX version of Graphics Tools.)

ALL PROGRAMS

When your program is installed and run on another computer, it will need to be able to find the Graphics Tools Runtime Files. The best place to put the files is in your application's own directory, i.e., in the directory where your application's primary EXE file is located. This will insure that your application will be able to locate the exact copy of the Graphics Tools Runtime Files that you supplied with your application.

Another alternative is to place the Runtime Files somewhere in the "system path" (such as in the System, System32, or root directory) but this technique is somewhat less reliable and is prone to several problems. For example, at some point in the future another application may install an *older* version of the Graphics Tools Runtime Files in the same location, and interfere with your application's ability to use features that are present only in later versions.

For best results and to avoid the symptoms of what Microsoft calls "DLL Hell", we strongly recommend that you place the Graphics Tools Runtime Files in your application's own directory.

Using Graphics Tools with Different Versions of Windows

For the most part, Graphics Tools will work very well with Microsoft Windows 95, 98, ME, NT4, 2000, and XP. Graphics Tools has been tested extensively using all of those versions of Windows and several sub-versions such as 95b and 98SE. But the various versions of Windows do not always behave *exactly* alike. Whenever possible, Graphics Tools automatically compensates for the differences, but this is not always possible.

Windows Platform 1 includes Windows 95, 98, and ME.

Windows Platform 2 includes Windows NT4, 2000, XP, and (according to Microsoft) all future versions of 32-bit Windows.

Under most circumstances the differences between Windows Platforms 1 and 2 will *not* have a significant effect on your program, but if you have a choice we recommend the use of Platform 2. In particular, if you are creating "production graphics" (bitmaps or JPEG files that will be distributed to other people) we recommend the use of Windows Platform 2. Platform 1 (the 9x/ME family) sometimes produces slightly less desirable results.

For example, Graphics Tools uses low-level API functions (functions that are built into Windows) to draw curved lines. The curved lines that are drawn by Windows Platform 2 are slightly smoother-looking than those drawn by Platform 1, especially when drawing ellipses that are much wider than they are tall. It is more likely that figures with curved edges (such as ellipses and pies) will have "flat areas" when drawn on Platform 1 systems.

This effect is not even noticeable on *most* systems, with *most* images, but if you are trying to create the best possible image quality, we do recommend Platform 2.

It is also important to choose Platform 2 if your program will use very *large* images. Windows Platform 1 has a built-in limit on the maximum size of the images that can be used, and Platform 2 does not. This is not usually an issue unless a program attempts to create graphics windows that are significantly larger than the desktop, i.e. larger than the screen.

Four Critical Steps for Every Program

The steps that you must follow to use Graphics Tools in a project will vary, depending on the programming language that you use. Please see...

Four Critical Steps for Visual Basic Programmers (OCX)

Four Critical Steps for Visual Basic Programmers (DLL)

Four Critical Steps for PowerBASIC Programmers

Four Critical Steps for Visual Basic Programmers (OCX)

Graphics Tools can be used with Visual Basic in two different ways: as an OCX control, and as a DLL. This section of this document describes the use of the OCX control. Because it allows the Microsoft ActiveX layer to be bypassed, using the DLL method can result in improved drawing speed and a smaller distribution package. For more information, see Four Critical Steps for Visual Basic Programmers (DLL).

These steps are not very complicated, but they *must* be performed or Graphics Tools will not function properly in a Visual Basic project.

STEP 1 After you open your project in the Visual Basic IDE, select Project > Components from the pulldown menu. You should see "Perfect Sync's Graphics Tools" on the list. (If it is not there, Graphics Tools has not been installed properly on your development computer.) Check the box next to that item in order to add the appropriate OCX file to your project. Note that when you select OK, an icon for a Graphics Tools graphics window -- a dark blue rectangle -- will appear in the VB toolbox.

STEP 2 Select Project from the pulldown menu again, but this time select Add Module. Choose the "Existing Module" tab, then navigate to the directory where you installed Graphics Tools (usually `\GfxTools`) and find either `modGfxToolsStd.BAS` or `modGfxToolsPro.BAS`, depending on the version of Graphics Tools you are using. After you have selected the file, click the Open button. This will add the Graphics Tools Runtime Files to your project, to allow you to use functions that are not controlled by the OCX file.

STEP 3 Using Visual Basic, edit the code for your main form's `_Initialize` event. It usually looks something like this:

```
Private Sub Form_Initialize()

End Sub
```

Please note that it is very important to use the `_Initialize` event, *not* the `_Load` event.

Add a single line of code to the `_Initialize` event, so that it looks like this:

```
Private Sub Form_Initialize()
      GraphicsToolsAuthorize MY_GFXT_AUTHCODE
End Sub
```

For more information, see Graphics Tools Authorization Codes.

STEP 4 To add a Graphics Tools Graphics Window to a form, simply click on the Graphics Tools icon in the VB toolbox, then position and size the graphics window on the form by using the mouse.

When it is first displayed in the VB environment, the graphics window will appear dark blue, with a black circle and white text that says "Sample Text". These are aids to allow you to select an initial brush, pen, and font. *They will not be displayed at runtime unless you use Graphics Tools to draw them.*

That's all there is to it! From this point forward you can use Graphics Tools properties, methods, and functions to control the graphics window.

See Using Graphics Tools With Visual Basic for more information.

Four Critical Steps For Visual Basic Programmers (DLL)

Graphics Tools can be used with Visual Basic in two different ways: as an OCX control, and as a DLL. This section of this document describes the use of the DLL. Because it allows the Microsoft ActiveX layer to be bypassed, using the Direct To DLL technique can result in improved drawing speed, a much smaller distribution package, unrestricted access to certain features, and other advantages. However, the OCX provides a more convenient programming environment that many programmers prefer. For more information, see Four Critical Steps for Visual Basic Programmers (OCX).

IMPORTANT NOTE: When the rest of this document refers to Visual Basic, it assumes that you are using the OCX version of Graphics Tools. If you use Visual Basic with the Graphics Tools Direct To DLL technique you should (generally speaking) follow the directions that are provided in this document for PowerBASIC programmers. When this document says "Visual Basic programmers must..." or "...cannot..." it is referring to VB programs that use the Graphics Tools OCX, not the DLL.

Direct To DLL

It is possible -- and often highly desirable -- to use Graphics Tools with languages such as Visual Basic *without* using the Graphics Tools OCX Control. Using the "Direct To DLL" technique has several advantages including improved speed, reduced memory requirements, and a significant reduction in the number of files that must be distributed with an application. If you use Direct To DLL, the only Graphics Tools file that you need to distribute with your application is the Graphics Tools DLL. It is not necessary to distribute the Graphics Tools OCX file or any of its support files.

The Direct To DLL sample programs demonstrate this technique. You can use them as templates for creating your own Direct To DLL projects, or...

Here are the four steps that you must follow when creating a new Direct To DLL project.

STEP 1 After you open your project in the Visual Basic IDE, select Project from the pulldown menu, then select Add Module. Select the "Existing Module" tab, then navigate to the directory where you installed Graphics Tools (usually `\GfxTools`) and select either `modGfxToolsStd.BAS` or `modGfxToolsPro.BAS`, depending on the version of Graphics Tools you are using. After you have selected the file, click the Open button. This will add the declaration file for the Graphics Tools DLL to your project.

STEP 2 Using Windows Explorer, make a copy of the file called `\GfxTools\Samples\VB\Callback.bas` using a new name of your choice. Back in VB, select Project > Add Module > Existing again, and this time add the file that you just created.

STEP 3 Using Visual Basic, edit the code for your main form's `_Initialize` event. It usually looks something like this:

```
Private Sub Form_Initialize()

End Sub
```

Please note that it is very important to use the `_Initialize` event, *not* the `_Load` event.

Add a single line of code to the `_Initialize` event, so that it looks like this:

```
Private Sub Form_Initialize()
      GraphicsToolsAuthorize MY_GFXT_AUTHCODE
End Sub
```

For more information, see Graphics Tools Authorization Codes.

STEP 4 You can create a graphics window either *with* or *without* a Picture Control as a "container".

With a Picture Control

This technique is demonstrated in the NoOCX1 *sample programs.*

To add a Graphics Tools graphics window to a form, use the VB toolbox to add a standard Picture control. This control will serve as a "container" for the graphics window and as a placeholder during the form-design process.

In your Form_Load event handler, add the following code:

```
Private Sub Form_Load()
    'Picture1 is a standard Picture control that is
    'used as the parent of the graphics window.
    OpenGfxEx 0, Picture1.hWnd, 0, 0, 0, 0, GFX_STYLE_TABSTOP
    GfxOptionEx 0, GFX_SET_CALLBACK, AddressOf GfxCallback
    GfxWindowEx 0, GFX_SHOW
    'other drawing code can go here
End Sub
```

Note the use of the Ex functions. When using Direct To DLL, your program must use the Ex functions for *all* Graphics Tools operations, and specify the graphics window number as the first parameter of every operation.

If you want to create two or more graphics windows in the same program you must use a different window number for each one. You must also use separate Picture controls, but it is not necessary to use separate Callback functions. All of the messages that are sent to the Callback function (usually called GfxCallback) contain the graphics window number, so your Callback function can easily determine which graphics window a message came from.

Without a Picture Control

This technique is demonstrated in the NoOCX2 *sample programs.*

This technique must be used when you create drag-resizable graphics windows. If you are not doing that, we recommend the use of a Picture control.

To add a Graphics Tools graphics window to a form, determine the desired location and size of the graphics window in VB twips, then convert those values to pixels. (The NoOCX2 sample programs demonstrate how to do this.) This example assumes you will use variables named lLeft, lTop, lWidth, and lHeight, and a main form called frmMain.

In your Form_Load event handler, add the following code:

```
Private Sub Form_Load()
    OpenGfxEx 0, _
              fmrMain.hWnd, _
              lLeft, _
              lTop, _
              lWidth, _
              lHeight, _
              GFX_STYLE_TABSTOP
    GfxOptionEx 0, GFX_SET_CALLBACK, AddressOf GfxCallback
    GfxWindowEx 0, GFX_SHOW
    'other drawing code can go here
End Sub
```

Note the use of the Ex functions. When using Direct To DLL, your program must use the Ex functions for *all* Graphics Tools operations, and specify the graphics window number as the first parameter of every operation.

If you want to create two or more graphics windows in the same program you must use a different window number for each one, but it is not necessary to use separate Callback functions. All of the messages that are sent to the Callback function (usually called GfxCallback) contain the graphics window number, so your Callback function can easily determine which graphics window a message came from.

Functions Which Return Strings

This applies to Microsoft Visual Basic only. It does not apply to any other languages the use the Graphics Tools DLL.

There are a few Ex functions which require special handling when Visual Basic programmers use the Graphics Tools DLL. Because of an obscure bug (acknowledged by Microsoft) in Visual Basic 5.0 and 6.0, if a function in a DLL returns a string value *you must retrieve the result*. For example, if you want to use the GfxFontNameEx function to change the name of the current font, you might be tempted to do this:

```
GfxFontNameEx 0, "Arial"
```

But the GfxFontNameEx function returns a string value, and if you ignore the return value Visual Basic will sometimes produce an error message that says "Variable uses an Automation type not supported in Visual Basic", usually on some *other* line of code that has nothing to do with GfxFontNameEx. This can, of course, be very confusing.

The solution to this problem is to do this:

```
sResult$ = GfxFontNameEx(0, "Arial")
```

The proper use of a string variable for the return value -- even if you ignore it -- allows VB to handle the string properly.

This is only necessary when you use Graphics Tools functions that return strings.

In most cases you will not *want* to ignore the return value of a function that returns a string. It would be pointless, for example, to do something like this.

```
GfxDroppedFilesEx 0
```

The whole point of using GfxDroppedFiles is to *obtain* the return value. But in those unusual circumstances when you do not need the return value of a function which returns a string (such as when using GfxFontNameEx) it is important to remember to *retrieve* the return value, even if you do not use it.

Again, this is the result of a bug in VB 5.0 and 6.0. It is not a bug in Graphics Tools.

Sample Program

Direct To DLL

Four Critical Steps for PowerBASIC Programmers

These steps are not very complicated, but they *must* be performed or Graphics Tools will not function properly in a PowerBASIC project.

If you're an experienced PowerBASIC For Windows, PB/DLL, or PB/CC programmer, click here to jump to a very brief overview of the four steps. Detailed descriptions are provided here...

STEP 1 Locate your program's WinMain, LibMain, or PBMain function. All PowerBASIC programs (EXEs and DLLs) contain one of those three functions. They tell Windows where to start when a program is executed. See the PowerBASIC documentation for more information about this.

At some point in your source code *before* the WinMain. LibMain, or PBMain function (i.e. as part of your program's "main" code, not as part of any sub or function), add *one* of the two following lines, depending on which version of Graphics Tools you are using:

```
#INCLUDE "\GFXTOOLS\GfxT_Std.INC"
```
...or...
```
#INCLUDE "\GFXTOOLS\GfxT_Pro.INC"
```

If you installed Graphics Tools somewhere other than the default `\GFXTOOLS` directory you will have to change the path (the part between the two backslashes) to reflect the correct location. You may also want (or need) to add a drive letter to the beginning of the path.

If you are using Console Tools Plus Graphics you will also need to add one of these lines, depending on the version of Console Tools that you are using...

```
#INCLUDE "\CONTOOLS\CT_Std.INC"
```
...or...
```
#INCLUDE "\CONTOOLS\CT_Pro.INC"
```

Please refer to the Console Tools documentation for more information about this and other steps that may be necessary before your program can use Console Tools.

STEP 2 Locate the point in your program where you want to begin using Graphics Tools. If you are not sure, the safest place is the *very beginning* of your WinMain, PBMain, or LibMain function. The exact placement is not important, as long as you add this line of code before your program attempts to use any other Graphics Tools functions:

```
GraphicsToolsAuthorize %MY_GFXT_AUTHCODE
```

For more information, see Graphics Tools Authorization Codes.

STEP 3 At the point in your program when you know how big the graphics window needs to be, use the OpenGfx function to create it. If your program will use two or more graphics windows, use the OpenGfxEx function instead of OpenGfx.

Console Tools Plus Graphics users should use the ConsoleGfx function instead of OpenGfx. Please refer to the Console Tools Help File's ConsoleGfx entry for detailed information about this process.

This step does not actually make the graphics window *visible*, so you may choose to perform this step during your program's "initialization" code.

This step can be performed (and repeated) at virtually any time, as long as it is performed before Step 4.

STEP 3A You may optionally use Graphics Tools drawing functions to "fill in" the graphics window before Step 4 is performed.

STEP 4 At the point in your program where you want the graphics window to become visible to the user, add this code...

```
GfxWindow  GFX_SHOW
```

That's it! When Step 4 is executed, the graphics window will be ready for drawing operations.

IMPORTANT NOTE: If you are using Graphics Tools to create a graphics window that is part of another window (a form, a dialog, etc.) and if you are using `GFX_TOOLSET_WINDOWS`, the graphics window may not be *visible* right away. `GFX_TOOLSET_WINDOWS` creates a graphics window with a gray background that is the same color as an empty window, so the background of the graphics window will blend in with the background of the parent window. If you draw in the graphics window you *will* see the results. To see the entire graphics window, do something like this:

```
BrushColor BLACK
GfxCLS
```

See The Graphics Window.

The Four Basic Steps

1) `#INCLUDE "\GFXTOOLS\GFXT_?.INC"`
   1a) (Console Tools Plus Graphics ONLY)
   `#INCLUDE "\CONTOOLS\CT_?.INC"`

2) `GraphicsToolsAuthorize %MY_GFXT_AUTHCODE`

3) `OpenGfx`

4) `GfxWindow  GFX_SHOW`

Other Programming Languages

This documentation specifically covers the use of Graphics Tools by Visual Basic and PowerBASIC programmers, but it contains information that applies to virtually *all* programming languages.

There are two basic ways to use Graphics Tools, as an OCX control and as a DLL.

If you want to use the OCX version of Graphics Tools with your programming language, follow the directions that are provided for Visual Basic.

If you want to use the DLL version of Graphics Tools with your programming language, follow the directions that are provided for PowerBASIC.

You will, of course, need to change the syntax of the example code to be compatible with your language, but that should be a relatively easy task. BASIC code is understood by most programmers, and we have purposely kept the example code as simple and "human-readable" as possible.

In many cases you will have your choice. For example, Visual C++ programmers can use either the OCX or DLL version of Graphics Tools. We recommend the DLL version for performance reasons -- bypassing the Microsoft ActiveX layer allows Graphics Tools to draw faster -- but your choice will (of course) depend on many factors.

Refreshing the Display

Microsoft Windows is required to perform the *very* complex task of making sure that the picture that is shown on your computer screen is displayed properly. While Graphics Tools does everything that it can to make sure that Windows is able to do its job properly, there are a few things that you can do to help.

Windows must constantly keep track of the status of every single pixel on the screen, and it periodically, automatically "refreshes" or "re-paints" parts of the screen as necessary. For example, if you perform an operation that is as outwardly simple as moving the mouse cursor, Windows must draw an image of the cursor in its new location, and re-display the background image that was previously covered up by the cursor. (Otherwise you would see a series of mouse cursors whenever you moved the mouse.) If you are moving the mouse very quickly, Windows must perform thousands of operations per second, just to keep up. Not only that, but it must constantly make decisions about *which* cursor to display. You have probably noticed how the cursor's shape changes as it moves over certain types of windows. Imagine writing a program that would perform just the simple task of displaying a moving mouse cursor and you'll understand how hard Windows is working (quite literally) "behind the scenes".

Fortunately, Windows does that job very well. It can experience some problems, however, if a graphics-oriented application isn't careful. You will need to consider two different situations when you're designing your program. We call them "Drawing Too Fast" and "Window Corruption".

Drawing Too Fast

If your program performs many drawing operations in a very small amount of time, it is possible for Windows to fall behind and for "glitches" to appear in the display. These glitches usually show up as small areas where drawing has taken place, but it is not immediately visible. For example, if you were to write a program that drew thousands of lines within just a few seconds, Windows might fail to display a certain percentage of those lines.

To fix the display, all you have to do is periodically "refresh" the graphics window (by using the GfxRefresh function) and Windows will instantly correct the problems. There are, however, some things that you can do to *prevent* graphics glitches from being a problem in the first place.

If your program is about to perform a rapid multiple-drawing operation, the most important thing that you can do is to "freeze" the graphics window with the GfxWindow `GFX_FREEZE` function. This effectively tells Windows "don't try to display anything for a few seconds". Your program can then draw virtually anything, as fast as it wants to, and the results won't become visible until you use the GfxWindow `GFX_UNFREEZE` function. When that happens, everything that your program has drawn will become visible, all at once.

Using `GFX_FREEZE` and `GFX_UNFREEZE` has another major advantage. Since Windows doesn't have to worry about updating the graphics window hundreds or thousands of times per second, your computer can concentrate on drawing. That means that most Graphics Tools drawing operations will be performed much faster -- between two and *fifty* times faster! -- if you `GFX_FREEZE`, draw, and then `GFX_UNFREEZE`.

Window Corruption

In addition to the problems that your program can cause by drawing too rapidly, *other*

programs can also cause problems for the Windows re-painting system.

Imagine a program, with a large graphics window, in the center of the computer screen. Also visible on the screen is another program, in a small window of its own. For this example we'll use the Windows NotePad program. If the user clicks on NotePad and drags its window *across* your graphics window, Windows will be required to paint and re-paint the screen dozens or hundreds of times. For example, if Notepad is moved one-quarter of an inch to the right, Windows will have to **1)** draw NotePad in its new location and then **2)** re-paint the one-quarter-inch stripe of your graphics window that was previously covered up.

And unfortunately, this system does not always work properly. It is fairly common for portions of the graphics window -- usually horizontal or vertical stripes -- to be refreshed incorrectly. This is especially true when programs are run on slow computers, and on Windows 95/98/ME computers.

Many different "external forces" can corrupt a graphics window: Here is another example...

Imagine the same program (with a graphics window) in the middle of the screen. Then somebody drags NotePad and drops it so that it covers part of the graphics window. If you click on the graphics window, the NotePad window will disappear and the graphics window will be refreshed correctly. But if you click on your application's *title bar* instead of the graphics window, the NotePad window will sometimes be replaced by a "hole" in the graphics window.

And one more example... If you use the mouse to drag your program so that the graphics window is partially obscured by the Windows Task Bar or by the edge of the desktop, when you drag it back into view it may not be refreshed properly. Gaps can appear.

If you experience any of these "Window Corruption" problems, we suggest that you have your program periodically use the GfxRefresh function to refresh the entire graphics window. For example, Console Tools Plus Graphics users can use the OnTimer function to automatically refresh the screen once per second. If you do that, the screen *can* still be corrupted but it will be corrected, on average, one-half second after the corruption becomes visible.

You can refresh the screen more often than once per second, but doing it *too* often will result in your computer slowing down, because so much processor time will be devoted to refreshing the screen even when is isn't necessary.

We have obtained good results by refreshing the screen every one-quarter of a second (every 250 milliseconds). This results in a relatively low "CPU load", and a very rapidly-refreshed display. But even that resulted in a 15% slowdown of our test program, so you'll need to decide for yourself which is more important: drawing speed, or rapid window-corruption repair.

Using (Calling) Graphics Tools Functions

PowerBASIC programmers (and other SDK-style programmers) perform all Graphics Tools operations using functions that are located in the Graphics Tools DLL.

Visual Basic programmers (and other programmers that use the ActiveX version of Graphics Tools) perform many operations using Properties and Methods, but can also use functions in the DLL. (**Tip:** For optimum performance you should use functions instead of Properties and Methods whenever possible, because using a function bypasses the Microsoft ActiveX layer and increases the speed at which your programs can draw.)

When this document show the syntax for using a Graphics Tools function, you should keep in mind that it is always the most *generic* form of the syntax. The actual syntax that you use will depend on 1) the programming language that you use and 2) whether or not your program needs to examine the return value of the function.

For example, here is the generic syntax for the DrawCircleEx function, which is used to draw a circle of radius *lRadius&* in graphics window number *lWindowNumber&*:

```
lResult& = DrawCircleEx(lWindowNumber&, lRadius&)
```

In practice, to draw a 100-unit circle in graphics window number 1, your actual source code would probably look something like this:

```
lRadius& = 100
lWindowNumber& = 1
lResult& = DrawCircleEx(lWindowNumber&, lRadius&)
```

Of course you can often simplify that code like this:

```
lResult& = DrawCircleEx(1, 100)
```

In fact, you can usually simplify your code even further. The DrawCircleEx function is virtually 100% reliable, so it is not usually necessary to check its return value. So if your programming language supports it, most programs can simply do this:

```
CALL DrawCircleEx(1, 100)
```

...and effectively discard the *lResult&* return value. Many languages also support the CALL-less syntax as well:

```
DrawCircleEx 1, 100
```

Keep in mind that there is very little point in writing code that looks like this:

```
lResult& = DrawFrom(0,0)
lResult& = DrawCircle(100)
lResult& = DrawFrom(20,20)
lResult& = DrawCircle(100)
lResult& = DrawFrom(40,40)
lResult& = DrawCircle(100)
```

If you never *examine* the return values to check for errors, it would be much more efficient to do this:

```
DrawFrom   0,0
DrawCircle 100
DrawFrom   20,20
DrawCircle 100
DrawFrom   40,40
DrawCircle 100
```

On the other hand, you really *should* examine the return value of certain functions. For example, you might want to examine the return value of the DisplayBitmap function to make sure that it does not return a "file not found" error code.

For quick-and-dirty programs, you can probably ignore the return values of most Graphics Tools functions. Even in more important applications, it is often safe to ignore the return value of most drawing functions because they are highly reliable.

Two Of Everything: The Ex Functions

When you look at the long list of drawing functions that Graphics Tools provides you will probably notice that most of the functions are available in two forms. For example, to draw a circle you can use either the DrawCircle function or the DrawCircle**Ex** function.

The Ex functions are 100% identical to the non-Ex functions, except that they allow you to specify a graphics window number.

To avoid a lot of redundancy, this document tends to refer to functions by their non-Ex name. It will usually say "the DrawCircle function" instead of something like "the DrawCircleEx function or the DrawCircle function or method". This document is indexed according to the non-Ex function names, so if you want to read about DrawCircleEx, look up DrawCircle.

If your program uses only a single graphics window, you can use the non-Ex functions for just about everything. Simply ignore the Ex functions. You can probably stop reading this section and skip to the next.

Visual Basic Programmers

Most of the *non*-Ex functions in Graphics Tools correspond to Properties and Methods. For example, VB programs don't use the DrawCircle function, they use the DrawCircle method like this:

```
GfxWindow1.DrawCircle 100
```

VB programs can, however, use the Ex functions. In fact, using the Ex functions can improve the drawing speed of Visual Basic programs. The various Properties and Methods that Graphics Tools provides are simply "wrappers" for functions in the Graphics Tools DLL, so calling the functions bypasses the Microsoft ActiveX layer and increases the speed at which drawing operations can be performed. So instead of the line of code just above, you would do this:

```
DrawCircleEx GfxWindow1.ID, 100
```

The `.ID` property is used as the graphics window number. It represents an *internal* value that Graphics Tools uses to identify its own windows. It is important to understand that the ID number is *not* the same thing as the number that Visual Basic appends to the control name. For example, when you create your first graphics window in a project it will be called GfxWindow1, but Graphics Tools will use an internal ID number of *zero*, not one. If you rename the control so that it is called MyDrawingWindow12345, Graphics Tools will *still* use the internal ID number zero, not 12345. The "number in the name" (if there is one) and the `.ID` property are not related to each other, so you must always use the `.ID` property to identify a graphics window when using an Ex function.

Tip: Use the `.ID` property once in your `Form_Load` event handler, and save the value in a variable, like this:

```
Private WindowNumber As Long

Private Sub Form_Load()
    WindowNumber = GfxWindow1.ID
End Sub
```

From then on, use the WindowNumber variable instead of the `.ID` property. Accessing a variable is faster than accessing a property, and this can result in faster drawing speeds.

Of course, if you use more than one graphics window in your project you will need to use more than one variable, or an array of variables, to track the window ID numbers.

Tip: If your program uses only *one* graphics window, it is safe to use the literal number zero for the window number, like this:

```
DrawCircleEx 0, 100
```

That's because Graphics Tools always assigns the number zero to the first graphics window that is created. If there is only one, it must be window zero.

And if you are very careful you can use literal numbers for two or more graphics windows. They are always numbered in the order in which they are created, starting with zero.

PowerBASIC Programmers

If your program uses two or more graphics windows at the same time, you have two options:

1) Use the Ex functions for everything. This allows you to specify a graphics window for each individual drawing operation. *Or...*

2) Alternatively, you can use the UseGfxWindow function to tell Graphics Tools which graphics window you want to use.

For example, to draw a 100-unit circle in window number 3 you can do this:

```
DrawCircleEx  3, 100
```

...or you can do this...

```
UseGfxWindow 3
DrawCircle 100
```

The nice thing about the UseGfxWindow function is that it is "sticky". It tells Graphics Tools "draw in this window until I tell you to change" so you can use the UseGfxWindow function once, and then use as many non-Ex drawing functions as you need.

Using Graphics Tools With Visual Basic

For the most part, the Graphics Tools graphics window will behave the way you expect. It was designed to be very similar to other ActiveX (OCX) controls.

The most significant difference is probably the way some of the properties work. For example, if you use a standard Label or TextBox control on a VB form to display some text, it will all be displayed in the same font. And if you use the control's Font properties to change the font, any text that is currently being displayed will instantly change to the new font.

But with Graphics Tools, many runtime properties set attributes *for future use only*.

If you use DrawTextRow to display some text in a graphics window, the current Font properties will be used and the text will appear in the intended font. But if you then change the Font properties the *already-drawn text will not change*. If you use DrawTextRow again the new font properties will be used. This allows you to use an unlimited number of Fonts, Pens, and Brushes, all in the same graphics window.

The general rule is "once something has been drawn, properties will no longer affect it."

Using Graphics Tools With PowerBASIC

Graphics Tools Version 1 was originally designed specifically for PowerBASIC, so the use of most Version 2 functions is very straightforward. PowerBASIC For Windows (including PB/DLL) and Graphics Tools work extremely well together.

PowerBASIC programmers should note that while separate, ready-to-use Sample Programs are provided for PowerBASIC, *this document* uses "generic" source code notation for most examples. In order to use the example code in this document with Visual Basic certain syntax changes must be made, and the same is true for PowerBASIC.

The most notable example of this is the use of the `%` prefix for numeric equates. The standard Microsoft notation for a "constant" (a word that represents a fixed numeric value) looks like this: `FIXED_VALUE`. Graphics Tools uses a large number of these values, such as `GFX_SOLID` and `GFX_SHOW`. In PowerBASIC, constants are called "equates" and you must add a percent-sign prefix to create equates such as `%GFX_SOLID` and `%GFX_SHOW` when using them in actual source code.

By the way, a large number of Graphics Tools users frequent the PowerBASIC Web BBS, which (as of this writing) is located at http://www.powerbasic.com/support/forums/Ultimate.cgi. It's a great place to ask questions and compare techniques with other Graphics Tools users. Please use the Third Party Addons forum when discussing Graphics Tools.

Also see Console Tools Plus Graphics.

The Graphics Window

A Graphics Tools "graphics window" is a rectangular area of the computer screen where drawing operations can be performed.

A graphics window is always the "child" of another window, meaning that it will appear to be *part of* another window called the "parent". If you are using Visual Basic or PowerBASIC functions to create a graphical user interface (a GUI), the graphics window will appear to be part of a form or dialog box. If you are using Console Tools Plus Graphics the graphics window will appear to be part of a console window.

Visual Basic programmers (and other programmers that use the ActiveX version of Graphics Tools) always create graphics windows by placing an instance of the Graphics Window control on a form. See Creating a Graphics Window with Visual Basic for more information.

PowerBASIC programmers (and other programmers that use SDK-style languages) usually use the OpenGfx method to create graphics windows, but other methods are also available. For more information, see...

Creating a Graphics Window with the OpenGfx Function

Creating a Graphics Window with PowerBASIC's DDT

Creating a Graphics Window with a Resource Script

Creating a Graphics Window with the CreateWindowEx API

If you have already decided which method to use, you can skip those sections and go directly to Graphics Window Styles or Drawing Units.

Creating a Graphics Window with Visual Basic

To create a graphics window in a Visual Basic program, simply click on the Graphics Tools icon in the VB toolbox, then use the mouse to locate and size the window. The toolbox icon is a dark blue rectangle with the tooltip "GfxWindow". If the Graphics Tools icon does not appear in your VB toolbox, Graphics Tools has not been installed properly on your system.

When the graphics window first appears on your form at design-time, it will be dark blue with a black circle and the words "Sample Text" in a white font. These elements will not actually appear at runtime, they are provided in the VB environment to make selecting an initial Pen, Brush, and Font easier. When you run your program the graphics window will appear as a blank gray rectangle until your program draws something.

After you locate and size the graphics window, you will probably want to adjust its "squareness". Squareness does not refer to the height and width of the graphics window, it refers to the ability of the window to produce round circles and square squares. See Drawing Units and Using Different "Worlds" for more information.

After that, you can use the graphics window's properties to change things like the Border Style, Brush Color and Pen Width, and use the graphics window's methods to draw various figures. You can also use many different Graphics Tools functions.

Basic Window Styles

Graphics Tools provides several properties that can affect the graphics window's "style". For example, the BorderStyle, CaptionState, CaptionText, FocusRect, ScrollBars, and TabStop properties all affect the graphics window in one way or another.

Advanced Window Styles

As you change the window style properties you will probably notice that one property is changing all by itself. The BorderStyle*Ex* property is a hex (base 16) value that summarizes all of the other styles, so when you change a window style property, the BorderStyleEx property will change automatically.

If you are familiar with window styles you can use different values for the BorderStyleEx property, to produce effects that cannot normally be selected. For example, if you enter the value `&h00820200` (which corresponds to the `GFX_STYLE_BORDER_LINE`, `GFX_STYLE_SUNKEN`, and `GFX_STYLE_SUNKEN_SHALLOW` values all added together) you can create a graphics window with a "double sunken" border. If you do that, you will see that the BorderStyle property will be automatically changed to "99 - BorderExtended".

The hex values that you will need to use the BorderStyleEx property are listed in the modGfxToolsStd.BAS or modGfxToolsPro.BAS file, depending on which version of Graphics Tools you are using.

It is important to note that the BorderStyleEx value affects several other named properties, so if you change it to a different number you should review all of the other window style parameters to make sure you didn't change something unintentionally. If, for example, you accidentally changed the TabStop property, you can use the TabStop property to correct the mistake, and the new value of BorderStyleEx will automatically reflect the change.

For more information, see Using Graphics Tools With Visual Basic.

Creating a Graphics Window with the OpenGfx Function

For SDK-style programmers, including PowerBASIC programmers, the easiest and most flexible way to create a Graphics Window is usually to use the OpenGfx function. (PowerBASIC PB/CC programmers who use Console Tools Plus Graphics are an exception to the rule. PB/CC programmers should use the ConsoleGfx function that is described in the Console Tools documentation.)

The OpenGfx function allows you to specify the size, location, and "styles" that a graphics window will have, as well as which window or dialog should be the graphics window's "parent".

See OpenGfx for a detailed description of this function.

Creating a Graphics Window with PowerBASIC's DDT

Most PowerBASIC programmers use the OpenGfx function to create graphics windows, but it is also possible to use other techniques.

PowerBASIC For Windows and PB/DLL programmers can also use the "DDT" system that is part of PowerBASIC to create a graphics window as a Custom Control. The code would look something like this:

```
GraphicsToolsAuthorize &h.......

InitGraphicsTools 1, %GFX_TOOLSET_WINDOWS

'(code to create your dialog goes here)

Control Add "GFXTOOLS2", _   'Required value
            hDlg&, _         'handle of parent dialog
            5000, _          'Control ID Number
            "", _            'window caption text
            lLeft&, _        'horizontal location
            lTop&, _         'vertical location
            lWidth&, _       'width of window
            lHeight&, _      'height of window
            %WS_CHILD, _     'Required value
            0                'extended styles

GfxWindow %GFX_SHOW
```

Some of that code is required for *any* Graphics Tools program. The InitGraphicsTools and `CONTROL ADD` code is our focus here.

InitGraphicsTools

Programs that use the DDT system to create graphics windows *must* use the InitGraphicsTools function before the first graphics window is created. The InitGraphicsTools function tells Graphics Tools **1)** how many graphics windows you intend to create, and **2)** which "toolset" to use when the windows are created. The toolset determines the colors of the initial Pen, Brush, and Font. See the InitGraphicsTools function for more information.

Failure to use the InitGraphicsTools function before CONTROL ADD will result in an error, possibly including an Application Error (General Protection Fault).

CONTROL ADD

For background information, see `CONTROL ADD` in the PowerBASIC documentation.

The string value `GFXTOOLS2` is the "class name" of a Graphics Tools graphics window. No other string can be used to create a graphics window. It must be used exactly as shown, in upper case letters. (Recent versions of PowerBASIC can use the string equate `$GFX_TOOLS_CLASS`.)

The Control ID Number should be the window number of the graphics window that you want to create, plus 5000. The example shown (ID 5000) would create graphics window number

zero, i.e. the default graphics window. To create graphics window number one (1) use 5001, and so on. (If it conflicts with your preferred numbering system, the "5000" value can be changed with the GfxOption `GFX_WINDOW_NUMBER_OFFSET` option.)

The ninth parameter of `CONTROL ADD` must always contain `%WS_CHILD`, and it may also contain other "styles" including standard `WS_` values and Graphics Tools `%GFX_STYLE_` values. (See Graphics Window Styles for a discussion of these values.) It should *not* include the `%WS_HSCROLL` and `%WS_VSCROLL` styles. They will have no effect if you use them.

The optional last (tenth) parameter of `CONTROL ADD` may contain "extended style" values. See Graphics Window Styles for more information about extended styles too.

All of the other parameters of the `CONTROL ADD` syntax are described in the PowerBASIC documentation.

Important Note: If you use `lWidth&` and `lHeight&` values that are the same, you will probably *not* produce a graphics window that is visually square. Keep in mind that unlike the OpenGfx function, the PowerBASIC `CONTROL ADD` function uses Dialog Units, not pixels, and on most systems a vertical dialog unit is *not* the same size as a horizontal dialog unit. See the PowerBASIC documentation under `DIALOG UNITS` for more information about this. Also see the OpenGfx entry in this document, because it contains information about converting Dialog Units to Pixels.

Creating a Graphics Window with Console Tools Plus Graphics

Console Tools Plus Graphics programmers normally use the ConsoleGfx function to create graphics windows. Instead of using pixels the ConsoleGfx function uses rows and columns, to make it easier to create a graphics window that fills a rectangular area of the console window.

By default, Console Tools Plus Graphics creates plain, borderless graphics windows. This is done to allow a large graphics window to fill the console window without creating the appearance of double borders. You can, however, use the following line of code...

```
GfxOption %GFX_WINDOW_DEFAULT_STYLE, lStyles
```

...to tell Console Tools Plus Graphics to create graphics windows with borders and other properties. The *lStyles* parameter (above) must contain a valid Graphics Window Style, and you must set the style *before* you create a graphics window with the ConsoleGfx function.

Please consult the Console Tools documentation for more information about ConsoleGfx.

Creating a Graphics Window with a Resource Script

The use of a Resource Editor is a complex topic that is beyond the scope of this document. Please refer to your Resource Editor's documentation for general instructions.

Programs that use a resource script to create graphics windows *must* use the InitGraphicsTools function before the first dialog is created. The InitGraphicsTools function tells Graphics Tools **1)** how many graphics windows you intend to create, and **2)** which "toolset" to use when the windows are created. The toolset determines the colors of the initial Pen, Brush, and Font. See the InitGraphicsTools function for more information.

Failure to use the InitGraphicsTools function before the first graphics window is created will result in an error, possibly including an Application Error (General Protection Fault).

Graphics Tools graphics windows can be added to a resource script as a "custom control". Using your Resource Editor, simply use the class name `GFXTOOLS2`.

The Control ID Number should be the window number of the graphics window that you want to create, plus 5000. For example, using ID 5000 would create graphics window number zero, i.e. the default graphics window. To create graphics window number one (1) use 5001, and so on. (If it conflicts with your preferred numbering system, the "5000" value can be changed with the GfxOption `GFX_WINDOW_NUMBER_OFFSET` option.)

Do not use the `WS_HSCROLL` and/or `WS_VSCROLL` styles when creating a graphics window in a resource script. They will have no effect if you use them. To control a graphics window's scroll bars, use `GFX_STYLE_SCROLLBARS_AUTO` (the default), `GFX_STYLE_SCROLLBARS_ALWAYS`, or `GFX_STYLE_SCROLLBARS_NEVER`. See Graphics Window Styles for more information about styles and extended styles.

Also see the discussion of "squareness" in the section titled Using Different "Worlds".

Creating a Graphics Window with the CreateWindowEx API

It is also possible to create a graphics window by using the CreateWindowEx API. This advanced technique should be used only by programmers who have previous experience using the Windows API to create windows, so this document will assume that you already know how to use the CreateWindowEx API.

Programs that use CreateWindowEx to create graphics windows *must* use the InitGraphicsTools function before the first graphics window is created. The InitGraphicsTools function tells Graphics Tools **1)** how many graphics windows you intend to create, and **2)** which "toolset" to use when the windows are created. The toolset determines the colors of the initial Pen, Brush, and Font. See the InitGraphicsTools function for more information.

Failure to use the InitGraphicsTools function before the first graphics window is created will result in an error, possibly including an Application Error (General Protection Fault).

The class name for a graphics window is `GFXTOOLS2`.

A graphics window must be created with the `WS_CHILD` style and a parent window or it will not function properly.

The Control ID Number should be the window number of the graphics window that you want to create, plus 5000. For example, using ID 5000 would create graphics window number zero, i.e. the default graphics window. To create graphics window number one (1) use 5001, and so on. (If it conflicts with your preferred numbering system, the "5000" value can be changed with the GfxOption `GFX_WINDOW_NUMBER_OFFSET` option. This must be done *before* any windows are created.)

Do not use the `WS_HSCROLL` and/or `WS_VSCROLL` styles when creating a graphics window with CreateWindowEx. They will have no effect if you use them. To control a graphics window's scroll bars, use `GFX_STYLE_SCROLLBARS_AUTO` (the default), `GFX_STYLE_SCROLLBARS_ALWAYS`, or `GFX_STYLE_SCROLLBARS_NEVER`. See Graphics Window Styles for more information about styles and extended styles.

Also see the discussion of "squareness" in the section titled Using Different "Worlds".

Graphics Window Styles

Graphics Tools graphics windows can have many different "styles" that affect the visual appearance of the window, its ability to receive the keyboard focus, its ability to receive files that are dragged and dropped on the window, and several other properties. Graphics Tools recognizes most of the standard Microsoft Windows window styles and extended styles (often called `EX_` styles), with a few exceptions and a few additions.

In order to simplify its use, Graphics Tools uses a number of "aliases" for the standard Windows names for certain values. For example, instead of the standard (and cryptic) name `WS_EX_CLIENTEDGE` Graphics Tools uses `GFX_STYLE_SUNKEN`, because it is much more descriptive. Those two constants have the same numeric value and can be used interchangeably, but this document will always use the more descriptive name.

Graphics Tools also provides a number of "combination" styles. For example, instead of requiring the combined use of the `WS_DLGFRAME` and `WS_EX_CLIENTEDGE` styles to produce a window with a "bump" border, Graphics Tools simply uses `GFX_STYLE_BUMP`.

The other primary difference between the Windows system and the Graphics Tools system is that Graphics Tools uses a single value to specify a window's style, and Windows requires two. The Windows "style" and "extended style" values are effectively combined into a single "graphics window style" value.

Border Styles

The basic graphics window border styles are:

```
GFX_STYLE_PLAIN
GFX_STYLE_BORDER_LINE
GFX_STYLE_RAISED
GFX_STYLE_BUMP
GFX_STYLE_SUNKEN
GFX_STYLE_SUNKEN_SHALLOW
GFX_STYLE_SUNKEN_DEEP
GFX_STYLE_CAPTION
```

All programmers can use those styles (or the Visual Basic properties that correspond to them). These additional styles are also defined:

```
GFX_STYLE_SUNKEN_LINE
GFX_STYLE_SUNKEN_DOUBLE
GFX_STYLE_CAPTION_BUMP
GFX_STYLE_CAPTION_SUNKEN
GFX_STYLE_CAPTION_SMALL
```

PowerBASIC programmers can use those styles directly. Visual Basic programmers must change the BorderStyleEx property to use them. (See Creating a Graphics Window with Visual Basic for more information.)

You can also create your own "combination" styles, such as...

```
GFX_STYLE_CAPTION OR GFX_STYLE_SUNKEN_DOUBLE
```

...which would produce a graphics window with a caption and a double-sunken border. Many

different combination styles are available. (The OR operator should be used to combine window styles. You can also use the plus-sign (+) operator, but it will not always produce the desired results.)

Focus Rectangles

In most Windows applications, the Tab key can be used to move the "keyboard focus" from control to control. The control that currently has the focus usually has a "focus rectangle" drawn around it. If you want a graphics window to be part of the Tab key system, add this style:

```
GFX_STYLE_TABSTOP
```

Borders and the Focus Rectangle

If the GFX_STYLE_TABSTOP style is used with a graphics window that has a border, a "focus rectangle" will be drawn around the graphics window whenever it has the Windows focus. (If the graphics window has no border, there is no room to draw a focus rectangle even if it has the GFX_STYLE_TABSTOP style.)

Some border styles work better than others when it comes to focus rectangles. You will find that if you use certain of the "sunken" styles (GFX_STYLE_SUNKEN, GFX_STYLE_SUNKEN_SHALLOW, etc.) the focus rectangle will not be easily visible on the left and top edges of the graphics window. This is because of the particular shades of gray that Windows uses when drawing sunken borders. If you would like to use a focus rectangle with a sunken border, you can use this code...

```
GfxOption GFX_FOCUS_RECT_COLOR, BLACK
```

...to change the color of the focus rectangle to a color that stands out against a sunken border. Black, of course, is only one of the many colors that you can use for the focus rectangle.

You can also use the GFX_**NO**_FOCUS_RECT_COLOR option to tell Graphics Tools to draw a colored rectangle around graphics windows that do *not* have the focus. For example, if your program uses four (4) graphics windows at the same time, all of them could have red borders except for the one that has the focus, which could have a green border.

If you want to *stop* Graphics Tools from displaying a rectangle, set the appropriate option to GFX_NONE instead of a color value. If you want Graphics Tools to use a standard focus rectangle instead of a color that you have specified, use GFX_AUTO.

Scrollbar options

Graphics window styles are also used to tell Graphics Tools what kind of scroll bars the graphics window should have. The default value is...

```
GFX_STYLE_SCROLLBARS_AUTO
```

...which tells Graphics Tools to add or remove scroll bars as necessary. You can also use one of these two values...

```
GFX_STYLE_SCROLLBARS_ALWAYS
GFX_STYLE_SCROLLBARS_NEVER
```

...to tell Graphics Tools that the graphics window should have scroll bars even when it doesn't need them, or that it should not have scroll bars even when it does need them.

Size/Location Options

By default, Graphics Tools does not allow graphics windows to change size or location. If anything -- a person, your program, or another program -- tries to resize or move the graphics window, the change will be refused. However you can add one or both of these styles...

```
GFX_STYLE_MOVABLE
GFX_STYLE_SIZABLE
```

...to a graphics window, in order to allow it to be sized and/or moved. But those styles do not allow the *user* to resize or move the window. You must add one or both of these styles to allow that:

```
GFX_STYLE_RAISED_DRAG_SIZE
GFX_STYLE_DRAG_MOVE
```

(The Windows window style that allows a window's size to be dragged always produces a raised border, so the style is called `GFX_STYLE_`*`RAISED`*`_DRAG_SIZE`. You should think of it as "Style Raised... Drag Size".)

When `GFX_STYLE_RAISED_DRAG_SIZE` is used, Graphics Tools automatically adds `GFX_STYLE_SIZABLE`. When `GFX_STYLE_DRAG_MOVE` is used, Graphics Tools automatically adds `GFX_STYLE_MOVABLE.`

Finally, the style...

```
GFX_STYLE_STRETCHABLE
```

...tells Graphics Tools that when a graphics window is resized, the image in the window should be stretched or compressed to fit the new size. Scroll bars are never displayed when you select `GFX_STYLE_STRETCHABLE` because they are never needed.

Before deciding to use `GFX_STYLE_STRETCHABLE` you should be aware that the process of stretching/compressing the image usually results in image distortions. Microsoft Windows uses a digital graphics system, not an analog one, so angled lines may become slightly more jagged, and other undesirable effects may be observed.

Dragging and Dropping Files

In some Windows applications, it is possible to drag and drop files to accomplish certain tasks. If you want a graphics window to be able to receive files in this way, add this style:

```
GFX_STYLE_DROP_FILES
```

If you do *not* add the `GFX_STYLE_DROP_FILES` style to a graphics window and the user tries to drag a file over the window, a special Windows icon that means "you can't drop that here" will be displayed. If you do add the `GFX_STYLE_DROP_FILES` style, an icon that

means "you may drop the file here" will be displayed. See the GfxDroppedFiles function for more information.

Rarely-Needed Options

The following styles, which correspond to standard Windows styles, are also supported by Graphics Tools. They are not needed by most programs. If you think you may need to use them, you should refer to the corresponding `WS_` or `WS_EX_` style in the Windows API documentation.

```
GFX_STYLE_CLIP_CHILDREN    (WS_CLIPCHILDREN)
GFX_STYLE_CLIP_SIBLINGS    (WS_CLIPSIBLINGS)
GFX_STYLE_DISABLED         (WS_DISABLED)
GFX_STYLE_NO_PARENT_NOTIFY (WS_EX_NOPARENTNOTIFY)
GFX_STYLE_TRANSPARENT      (WS_EX_TRANSPARENT)
GFX_STYLE_TOPMOST          (WS_EX_TOPMOST)
GFX_STYLE_RIGHT_ALIGN      (WS_EX_RIGHT)
GFX_STYLE_RTL_READING      (WS_EX_RTLREADING)
GFX_STYLE_LEFT_SCROLLBAR   (WS_EX_LEFTSCROLLBAR)
```

Using Multiple Graphics Windows

Graphics Tools Standard can be used to create two different graphics windows in the same application at the same time. Graphics Tools Pro can be used to create as many as 256 graphics windows at the same time.

With the exception of certain "global options", each graphics window is completely independent from all of the others. Each graphics window has its own Pen, Brush, Font, World, Border Style, Gradient, and so on.

Each graphics window is assigned a number when it is created. Generally speaking, the first graphics window that is created should be "window number zero", the second should be number one, and so on. The highest window number that Graphics Tools Pro can use is therefore 255.

When a multi-window program performs a drawing operation you will usually need to specify which window number should be used. This topic is covered in detail under Two Of Everything.

Keep in mind that not all of your program's graphics windows need to be *visible*. For example, you may find it useful to have one or more hidden "work windows" where you can perform certain drawing operations, and then transfer the finished drawing(s) to the visible window in a single operation. This is useful for "sprite-style" programming. Each work window can contain a different pre-drawn sprite.

Tip: For optimum performance, *visible* graphics windows should use the lowest possible window numbers. For example, a program that used windows 0-20 for "work windows" and window number 21 for a visible window would not perform as well as a program that used window zero (0) for the visible window and 1-21 for work windows.

Note also that a small number of operations such as holes are limited to graphics window zero, so using zero for your *main* visible window is usually a good idea.

Drawing Units

This is a very important section of this document. All Graphics Tools users should become familiar with this topic.

A picture element or "pixel" is the smallest element of the screen that a computer can control -- a *tiny* dot.

Computer programs that use pixels as their unit of measure can therefore draw very small picture details very accurately. But because the Microsoft Windows operating system is so flexible -- so *configurable* -- pixels are not a very convenient unit of measure when you want to draw something. Here's why...

Let's say that your development computer is using the popular 1024x768 screen resolution. That means that it has 1024 pixels from left to right, and 768 from top to bottom. If your program created a graphics window that was 800x600 pixels it would fill about two-thirds of the desktop. But if you ran that same program on a computer that was using the 640x480 screen mode, an 800x600 graphics window would *over*-fill the screen, and parts of it would not be visible. The same graphics window, created by the same program in exactly the same way, would *appear* to be much larger on the second computer.

So we need a way of measuring things that is "resolution independent", meaning that it works the same way regardless of the screen's current settings.

Different programming languages use different units of measurement. For example, Visual Basic uses "twips" and PowerBASIC (and other SDK-style languages) use "dialog units", which is Window's native system. But neither of those systems are appropriate for a *drawing* program because they do not produce consistent results vertically and horizontally. One vertical twip or dialog unit is not the same distance as one horizontal twip or dialog unit, and this makes certain operations like drawing round circles difficult.

So Graphics Tools uses a system called Drawing Units. When a graphics window is first created it has a physical (i.e. visual) size that you define, but it *always* has a measurement of 1024x1024 Drawing Units, numbered from zero (0) to 1023.

Important Point: In its default mode, the top-left corner of the graphics window is always treated as "location 0,0", meaning "horizontal location 0, vertical location 0". The top-right corner is 1023,0; the bottom-left corner is 0,1023; and the bottom-right corner is 1023,1023.

That means that no matter how large or small the graphics window really is, you can draw from coordinate 0,0 to coordinate 1023,1023 and you will see a line that goes from the top-left corner to the bottom-right corner of the graphics window.

If you attempt to draw outside the limits of the graphics window -- like drawing a line from 0,0 to 2000,2000 when the window is only 1024x1024 -- Windows simply won't show the portions of your drawing that extend past the edges of the window.

The use of Drawing Units creates a "Drawing World" that is independent of the size of the graphics window, the screen resolution, and everything else. You can always use the same numbers to draw things, and Graphics Tools will automatically "scale" your drawing to fit the available space. That means that if you change the size of your graphics window to make your interface look better, you don't need to change all of your drawing code.

Now, if your graphics window is not square but rectangular -- say, twice as wide as it is tall -- using a Drawing World of 1024x1024 will produce some interesting results. If you draw a

"circle" in that World, the circle will be twice as wide as it is tall, and it will look like an ellipse (an oval). So Graphics Tools allows you to specify the size of the Drawing World that you create. In this case you would probably change the World so that it was 2048x1024, so that the proportions matched the "physical" proportions of the graphics window. If you did that, circles would appear round and squares would appear square.

We call this a "Square World". A Square World is one where circles appear round and squares appear square. *It has nothing to do with the actual physical shape of the graphics window.*

Console Tools Plus Graphics users please note: Since most 80x25 console windows are rectangles that are roughly twice as wide as they are tall, Console Tools Plus Graphics *automatically* uses a default size of **1024x512** Drawing Units instead of 1024x1024. It is important to note that these values works well for most consoles, but they will not always produce a perfectly Square World because consoles can vary in size based on the number of rows/columns that you specify and the Console Font that is used. (See the Console Tools Help File under "CWC" for information about specifying a console font.)

Since drawing round circles and square squares is usually important, Graphics Tools provides an easy way for you to determine the "squareness" of your graphics window. See the GfxSquareness function for more information.

It is important to keep in mind that virtually *all* Graphics Tools drawing operations are affected by the squareness of the Drawing World If you tell Graphics Tools to draw a line or display some text at a 45-degree angle, that angle will be compressed or expanded if the World is not square.

The Graphics Tools "World" system is very flexible, and there's a lot more to it than just selecting appropriate numbers for the horizontal and vertical scale. See Using Different "Worlds" for a more complete discussion of this topic.

Using Different "Worlds"

When a graphics window is first created it has a physical (i.e. visual) size that you define, but it *always* has a measurement of 1024x1024 Drawing Units, numbered from zero (0) to 1023. This is called the Graphics Tools "World".

In its default World, the top-left corner of the graphics window is always treated as "location 0,0", meaning "horizontal location 0, vertical location 0". The top-right corner is 1023,0; the bottom-left corner is 0,1023; and the bottom-right corner is 1023,1023.

That means that no matter how large or small the graphics window really is, you can draw from coordinate 0,0 to coordinate 1023,1023 and you will see a line that goes from the top-left corner to the bottom-right corner of the graphics window.

If the default World is not convenient, you can specify your own World. Use the GfxWorld function (or the VB World properties) to specify the numbers you prefer.

IMPORTANT NOTE: Everything in this document -- sample code, function descriptions, etc. -- assume that you are using the default World. If you use the GfxWorld function to change the World, you must remember to mentally change the appropriate values when reading this document.

The parameters of the GfxWorld function are used to specify the numbers that should be used for the Left, Top, Right, and Bottom edges of the World, in that order. To specify the default World, use:

```
GfxWorld 0, 0, 1023, 1023
```

To specify an upside-down world where the *bottom*-left corner is 0,0 instead of the top-left corner, use:

```
GfxWorld 0, 1023, 1023, 0
```

To specify a 2048x2048 world where 0,0 is in the middle of the graphics window, use negative values like these:

```
GfxWorld -1023, 1023, 1023, -1023
```

You can use just about any coordinate system that you can imagine.

One warning: don't use numbers that are too small. If your graphics window is 500 pixels wide and you use a World with only 50 Drawing Units from left to right, you won't be able to draw fine details very easily because location 1 and location 2 will be 10 pixels apart.

And a tip: As noted elsewhere in this document, for optimum performance, bitmaps are required to have an even number of pixels in both the horizontal and vertical directions. It's natural to pick even numbers for Drawing Unit scales, such as the default 1024 units (0 to 1023). But keep in mind that if you do that, the graphics window will not have a true center. The exact center of a graphics window that has 1024 Drawing Units is half-way between the numbers 511 and 512. This can result in single-pixel errors. You may find that a "centered" circle, for instance, misses touching either the left or right side of the graphics window by one pixel. The best solution is to use scales like 0 to 1024. A scale like -1000 to +1000 will also work well. While doing this produce an odd number of Drawing Units, it can eliminate certain types of single-pixel errors.

Now For The Bad News

If your goal is to create a perfectly square World, keep in mind that your program has no control whatsoever over the computer monitor that will be used to display Graphics Tools images. For example, you can design a graphics window World that is *perfectly* square and *should* produce perfectly round circles, but if the computer monitor's "height" or "width" control is not adjusted properly, the image will not look right to the user. The Windows operating system does not provide any feedback between the hardware controls and your software, so if the user adjusts the monitor poorly, the image will be distorted and your program won't be able to detect it, much less do anything about it.

The best solution to this problem is usually to provide a "test pattern" image that contains one or more perfect circles, and instruct the user to adjust their monitor width and height controls until the image looks right.

Sample Programs

A World With Zero In The Center
A Rectangular Graphics Window

Angled Lines

Many different Graphics Tools functions draw angled lines. For example, the DrawAngle function can be used to draw a straight line at an angle that you specify.

Imagine a clock that is drawn in the center of the graphics window. By default, *all Graphics Tools functions that deal with angles use degrees (not radians) that are based on zero degrees at the nine o'clock mark.* Put another way, *an angle always represents a rotation of the X-axis* by a certain number of degrees. The twelve o'clock position corresponds to 90 degrees, three o-clock is 180 degrees, and six o'clock is 270 degrees. 360 degrees is the same as zero degrees, back at nine o'clock.

It is possible to use the GfxOption `GFX_BASE_ANGLE` function to tell Graphics Tools to use 12 o'clock, 3 o'clock, or 6 o'clock instead of 9 o'clock as the zero-degree mark. For clarity, the rest of this discussion and most of the examples in this document will use the default value of 9 o'clock.

It is also possible to use the GfxOption `GFX_USE_RADIANS` function to tell Graphics Tools to use radians instead of degrees for all angle measurements, but for clarity the rest of this discussion will use degrees. If you use the `GFX_USE_RADIANS` option, the concepts covered below are still completely valid.

If you use the DrawAngle function to draw an angled line at 45 degrees, starting from the center of the screen, it will point toward ten-thirty on the clock. If the graphics window is using a relatively Square World, the line will actually appear on the screen at a 45 degree visual angle, about half-way between nine and noon. *But...*

If the graphics window is *not* square -- if it is stretched or compressed horizontally or vertically -- the *apparent* angle of a line that is drawn at a 45-degree angle will change.

Imagine the clock and the 45-degree line again, but then picture the entire graphics window (including the clock) being compressed down, so that it was not very tall. The clock face will have become a flattened oval. That line that was drawn at a 45 degree angle would still point at ten-thirty, but because everything was compressed vertically, the angle of the line on the screen would appear to be much less than 45 degrees. Even though you told Graphics Tools to draw a line at a 45 degree angle, and even though it drew a 45-degree-angled line *inside its own "world"*, the resulting *visual* angle will not appear to be 45 degrees when the graphics window is viewed by the user.

Keep in mind that *all* Graphics Tools measurements, including angles, are scaled to conform to the current World. If you want a 45-degree angle to appear to be exactly 45-degrees on the screen, you must adjust the Graphics Tools World values to compensate for the squareness of your graphics window.

The LPR

The LPR or "Last Point Referenced" is the starting point for most drawing operations.

When the graphics window is first initialized, the LPR is located at the top-left corner of the screen, which is known as 0,0. (For information about the screen coordinate system, see Drawing Units.)

Most of the time you will move the LPR "manually", by using the DrawFrom function to set it. For example, you might use...

```
DrawFrom  100,100
```

...before using a function like DisplayIcon, so that the icon would be drawn starting at location 100,100.

Sometimes the LPR will be moved for you, as a convenience. For instance if you use the DrawTo function, the LPR is automatically moved to the end of the line that you draw, so that you may draw several connected lines simply by using DrawTo over and over, without using DrawFrom to explicitly set the LPR each time.

Various drawing operations use the LPR in different ways...

1) Some operations, such as DrawTo, use the LPR as the starting point of the figure that is drawn. The figure can be drawn in any direction from the LPR.

2) Some operations, such as DrawRect ("draw rectangle") use the LPR as both the starting and ending point of the figure. In most cases, the LPR is used as the *top-left corner* of a figure like a rectangle.

3) Some operations, such as DrawCircle, use the LPR as the *center* of the figure.

And finally, keep in mind that some drawing operations ignore the LPR completely. The DrawPixel function, for example, does not use or change the LPR. You simply tell DrawPixel the exact window location that you want to change, and the LPR is ignored.

Also remember that some operations, such as GfxCLS, can affect the LPR even though the function is not directly affected by the LPR. The location of the LPR does not matter when GfxCLS clears the graphics window, but the LPR is automatically reset to the top-left corner of the graphics window (0,0) when the clear-screen operation is finished.

Pens and Brushes

Pens and Brushes are the two most commonly used Windows drawing tools. They are used by many different Graphics Tools functions.

Pens are used to draw lines. A Pen can be any color, can have any width, and can have any one of several different "styles". The most common style is `GFX_SOLID`, which draws an unbroken line. Four different styles of broken-line Pens can also be created (for dotted or dashed lines), and a `GFX_CLEAR` pen is available for special situations. Because of a limitation of Windows, broken-line pens must be either zero (0) or one (1) pixel wide.

Brushes are used to fill areas of the screen. A Brush can be any single `GFX_SOLID` color, or `GFX_CLEAR`, or `GFX_HATCHED`. When you use a Hatched Brush, Windows automatically fills the area with a pattern of horizontal, vertical, or diagonal lines. You can also use a "Bitmap Brush", which fills an area with a pattern that is taken from an image that you provide. If the bitmap's image is too small, it is automatically "tiled" to allow the entire area to be filled. (Graphics Tools Pro users can also use a "Gradient Brush". See Gradients for more information.)

Clear Pens and Clear Brushes are used when you don't want the results of a drawing operation to be visible. For example, the DrawCircle function will draw a circle using the current Pen, and then fill the circle using the current Brush. If the Pen was `GFX_SOLID` white and the Brush was `GFX_SOLID` red, the result would be a white circular border, and the center of the circle would be filled with red. But...

If you were to use a `GFX_CLEAR` pen and a `GFX_SOLID` brush, the DrawCircle function would not actually draw the border, and the result would be a borderless red circle. Or...

If you were to use a `GFX_SOLID` pen and a `GFX_CLEAR` brush, the result would be a white circular border and the original contents of the graphics screen would be visible through the middle of the circle.

When Graphics Tools first creates a graphics window, a default Pen and Brush are also created. You can change the Pen and Brush with functions like PenColor, PenWidth, PenStyle, BrushColor, BrushStyle, BrushHatch, and BrushBitmap. If you are going to change two or more aspects of a Pen or Brush, such as changing the color and style at the same time, it is more efficient to use the Pen and Brush functions.

Drawing Modes

By default, Pens and Brushes are used in the `GFX_OPAQUE_COLORS` mode. But a total of sixteen (16) different drawing modes can be used, including `XOR` (reversible) drawing, and many different modes where the Pen/Brush and screen colors are combined in different ways. For more information, see the GfxDrawMode function.

More about Pens

Pens are round. If you are using a Pen that is relatively narrow, the shape of the pen won't make much difference, but the larger the Pen, the more you'll see the roundness.

For example, if you create a Pen with a width of 50 to draw a line you will clearly see that the ends of the resulting line are rounded.

The roundness is intentional, so that lines that are drawn at different angles will always connect smoothly, but the effect is not always desirable. Fortunately, several different techniques can be used to create different effects.

Some Graphics Tools drawing functions automatically "sharpen" corners where it is appropriate. If you were to use the DrawRect function instead of DrawLine, a rectangle with sharp corners would be produced.

You can also draw several narrow lines to "build up" a wide line. The resulting line could have square, angled, or tapered ends, depending on how you draw it.

Or you could use a narrow Pen, a Brush of the same color, and the DrawRect or DrawPgram function to draw a long, very narrow rectangle that looks like a line.

More about Brushes

If you are using a Hatched Brush or a Bitmap Brush, it is important to understand how they will be used to fill an area. They both work in basically the same way.

Windows always uses Hatches and Bitmaps to fill *portions* of the graphics window as if the *entire* window was being filled, starting at the top-left corner of the window. The LPR and the location of the area that is being filled are ignored. For example, imagine a graphics window that is 100x100, and a 100x100 bitmap that starts with black on the left side and gradually progresses to gray in the middle and white on the right side. If you were to simply display the bitmap in the graphics window (using the DisplayBitmap function) it would fill the window. Instead, if you were to use that bitmap to create a Bitmap Brush, and then use the brush to fill a small area that was on the right side of the screen, the area would be filled with the *white* portion of the bitmap, not the black portion.

While this default Windows behavior may seem inconvenient at first, it insures that if two filled areas touch each other, a seamless image will be produced. Another example: If you were to create a Bitmap Brush using a bitmap image of a person's face, and then use that brush to fill an area near the bottom of the graphics window, you would see only the person's mouth and chin. If you were to then use it to fill an area in the middle of the screen, the nose would appear, properly connected to the mouth.

That would happen unless, of course, the bitmap was too small to fill the entire window. In that case the bitmap would be tiled as if the entire screen was being filled, and then the portion of the image that happened to correspond to the area being filled would be displayed.

You should keep in mind that bitmaps are never stretched or compressed when they are used as Bitmap Brushes. The StretchBitmap function allows you "force fit" a bitmap into a rectangle that is too large or too small, but Bitmap Brushes always use the "native" size of the Bitmap.

Please also note that Hatched Brushes are produced by Windows, not Graphics Tools, so they are not "scaled" to conform with the World. Hatch patterns look the same regardless of the squareness of your graphics window.

The Windows 9x Hatched-Brush Bug

Under certain very specific circumstances, Windows will fail to draw the hatch-lines of a hatched brush. As far as Perfect Sync has been able to determine, this happens only when **1)** Windows 95, 98 or ME is being used on the runtime system, **2)** the video display is using

the TrueColor mode, and **3)** the brush color and the graphics window Background Color are the same. This does not refer to "the color of the background" upon which a hatched brush is being used, it refers to the actual Background Color setting, as defined by the GfxBkgdColor function. Under normal circumstances the GfxBkgdColor function has no effect on Graphics Tools drawing operations because the default setting of the GfxBkgdMode function is "transparent". But because of an apparent bug in Windows 95, 98, and ME, if the Background Color is the same as the current Brush Color, Windows will fail to draw the hatch-lines of a hatched brush.

If you encounter this problem, the solution is to change the GfxBkgdColor to a color that is different from the current brush color. Unless you have also changed the GfxBkgdMode function, changing the Background Color will have no effect whatsoever on your program (other than working around this Windows 9x bug).

Windows Colors

Microsoft Windows uses a color-numbering system that is very efficient... for *Windows* to use.

Unfortunately, it can be relatively difficult for programmers to use. (After we explain Windows Colors we'll be able to tell you about Graphics Tools HSW Colors, which are much less cumbersome for certain purposes.)

Windows defines a color by specifying how much Red, Blue, and Green the color contains. (A discussion of the physics of color perception is well beyond the scope of this document, but you are probably aware that any color, including all of the shades of gray from black to white, can be created by combining various amounts of Red, Green, and Blue.)

The Red, Green, and Blue values (commonly known as R, G, and B) are always integers between zero (0) and 255. That means that there are 256 possible values for each color, for a total of 256x256x256 = 16,777,216 colors. Virtually every Windows drawing function -- and therefore every Graphics Tools function -- requires that you use a color value that is between zero (0) and 16,777,216.

The number 16,777,216 is too large to be stored as a 16-bit INTEGER or WORD variable, so Windows uses 32-bit LONG integers. But LONG integers have a range of plus-or-minus 2.15 *billion*, so most of the values that will fit in a LONG integer are meaningless as color values. In fact less than 0.4% of the possible LONG integer values are meaningful as Windows Colors, so you have to be careful when you're calculating color numbers or you can produce an Invalid Parameter error. In some cases, using an invalid color value results in `MAXCOLOR` (white) being used, but not always.

The actual Windows Color number for a set of R, G, and B values can be obtained by using the Visual Basic, PowerBASIC, or Win32 RGB function. Please refer to your language's documentation for more information about the RGB function.

The values that are calculated by the RGB function are fairly easy to understand. RGB returns a four-byte LONG integer, where three of the four bytes represent the R, G, and B values, and the remaining byte is always zero.

A common method of denoting Windows Colors is to use a hex (base 16) notation, where the R, G, and B values are written as values from `00` to `FF`. For example, the color Blue would be...

```
FF 00 00
```

This stands for "FF Blue" (meaning 255 units of Blue), "zero green" and "zero red". In hex notation this would be written as `&hFF0000`. (Notice that Windows stores the bytes backwards, in the order BGR.) Green would be written as `&h00FF00` and red would be `&h0000FF`. If you were to add blue and red together to produce magenta you would get `&hFF00FF`. Add green to that magenta and you'd get `&hFFFFFF`, which equals `MAXCOLOR` (white). Of course you could create *many* different color values -- well over 16 million of them -- simply by using values other than `00` and `FF`.

Incidentally, this system is called "24-bit color" because 24 of the 32 bits in the LONG integer value are used to store color information. Another common name is "TrueColor".

As you can see, Windows colors are not always very intuitive. If you ask most people "what do you get when you mix magenta and green?" not many would say "white".

If the R, G, and B values are all exactly the same number (like 20, 20, and 20) , a "color" on the scale from black to white is created. That is to say, there is no "hue" or "tint" involved. The blackest black results when R, G, and B are all equal to zero (0), resulting in a Windows color value of zero. The whitest white results when all three values are 255 (`&hFF`), resulting in a Windows color value of `MAXCOLOR`.

As you'll see, the Windows Color system can be very cumbersome to use. For example, the second-darkest black that can be created requires R, G, and B values of one (1), and that creates a Windows RGB color value of 65,793. Here are numbers that correspond to the first eight shades of gray, starting with black...

```
      0
 65,793
131,586
197.379
263,172
328,965
394,758
460,551
```

There's a pattern there, but it's not very easy to see. (The difference between the numbers is always 65,793.)

It's even harder to use when you want to start with one color and create a *related* color. Consider color number 250, which is almost as red as you can get. The hex representation of 250 is `&h0000FA&`. Add one (1) to that value and you'll get color 251 (`&h0000FB&`) which would be very slightly redder. Then keep adding one until you reach color number 255 (`&h0000FF&`), which is the reddest red that Windows can display. You've just created several shades of red, where the color is gradually getting more intense. So far so good.

But if you then add one to 255 you get color 256, which is hex value `&h000100&`. That's a green that is so dark it is almost black! And adding one to that adds red, not green, and you get a very dark cyan color. Adding one again shifts the balance back to red.

It would be *very* difficult to use Windows Colors to create a "rainbow" effect where colors gradually blend from one to the next. Fortunately, Graphics Tools can use Windows Colors *and* our HSW colors, which make things like "spectrums" and "color wheels" very easy to create. More about HSW in a minute, but first we have to discuss a few other common problems with Windows Colors.

Color "Depth" Limitations

Because of the 24-bit color-numbering system that Windows uses, the native Windows drawing functions are limited to 16,777,216 colors. But *each individual computer* may be far more limited than that.

Most modern computers contain a video card (the circuit board that controls the monitor) which can handle 24-bit TrueColor, with over 16 million colors. But they can also be set up for 16-bit color, which allows the display of just 65,536 colors. Some use a modified 16-bit system that allows only 32,768 colors. Still others use 8-bit color that allows only 256 colors, and a few only allow 16 (yes, *sixteen*) colors to be displayed, out of the total of over sixteen million. Even the most expensive, sophisticated video card may be *set up* for as few as sixteen colors!

If you attempt to draw in a color that the video card's current configuration cannot display,

Windows will automatically substitute the closest color that it can display.

That means that if you use a function like DrawLine or DrawPixel to change the color of a certain pixel to a certain color, and then immediately check the color of the pixel with the PixelColor function, you may be unpleasantly surprised. It may or may not be the color that you specified.

Also see ActualColor.

HSW Colors

HSW Colors

In order to make certain color effects easier to create, Graphics Tools allows the use of HSW colors in addition to RGB (Windows) colors.

HSW colors can be used *anyplace* that a Windows color value is required, even if a function's Reference Guide entry does not specifically say that they can be used. The HSW function and the BASIC RGB function are simply different methods of creating valid Windows color values.

RGB colors are defined by three values called Red, Green, and Blue. HSW colors, on the other hand, are defined by three values called Hue, Saturation, and Whiteness.

Hue is the property of color that we see as Red, Yellow, Orange, and so on.

Saturation is the *amount* of the specified color. A "highly saturated" color is one that is very intense. Picture a dark, subtle blue that is almost black, next to a dark, rich blue with lots of color.

Whiteness is the amount of *white* that is added to the color. Some people think of Whiteness as the "brightness" of the color, but there are some important differences between those two terms. A "bright" color is not really the same thing as a color to which white has been added. (More about this in a minute.)

In the same way that a Windows color value can be created from Red, Green, and Blue values by using the BASIC RGB function, a Windows color value can be created from Hue, Saturation, and Whiteness values with the HSW function. Both functions produce Windows Color values between zero (0) and 16,777,216 which can be used in the graphics window.

Hue

All of the colors that can be produced by Windows can be pictured on a "color wheel". On the far left side of the wheel, at zero degrees, is red. Then, as you go clockwise around the wheel, the color gradually fades from red to orange, to yellow, to green, to blue-green (cyan), to blue, to purple (magenta), and finally back to red. Each direction from the center, each *angle*, defines a slightly different hue.

Red corresponds to exactly zero (0) degrees, green corresponds to exactly 120 degrees, blue corresponds to exactly 240 degrees, and every other numeric value between 1 and 359 corresponds to a slightly different hue.

Hue Angle	Primary Colors	Secondary Colors
0	Red	
60		Yellow
120	Green	
180		Cyan
240	Blue	
300		Magenta
360	Red (again)	

Additional colors are easy to figure out. For example, 30 degrees (half way between Red and Yellow) produces Orange. You should note, however, that the farther you get away from the

three primary and three secondary colors, the more variation you are likely to see from monitor to monitor.

When you specify the H (Hue) value of an HSW color, you should use a value between zero (0) and 360 degrees. If you use a value outside that range, it will be "normalized" into the 0-360 range. For example, 361 would be normalized to 1, and 370 would be normalized to 10. Negative ten (`-10`) degrees would be normalized to 350 degrees.

Even if you use the GfxOption `GFX_USE_RADIANS` option, you must use degrees, not radians, whenever you specify a hue. Similarly, the GfxOption `GFX_BASE_ANGLE` and `GFX_REVERSE_ANGLES` options have no effect on HSW colors. Zero degrees always corresponds to red, regardless of the option settings.

Saturation

Graphics Tools uses a scale of zero (0) to 255 for Saturation. A color with a Saturation of zero (0) is a shade of gray between black and white, with no color at all. A color with a saturation value of 255 is the most intense color that Windows can produce.

Whiteness

You should think of whiteness as the amount of white that is used to "dilute" a color. HSW colors use a Whiteness range of zero (0) to 255. For example, an HSW color with a Hue of zero degrees (red), a saturation of 128 (50%), and a whiteness of zero would be a dark, medium-rich "brick" red. Changing the whiteness to 128 (50%) without changing the Hue or Saturation would produce a much more pale, pinkish color. It would be exactly the same color as the brick red, but with quite a bit of white added.

Too Many Colors?

The HSW function can accept 360x256x256 = 23,592,960 different values. But Windows only recognizes 16,777,216 different colors. That means that if you make a very subtle change in an HSW color -- like changing the H, S, or W value by one or two -- you probably won't see a difference in the color that is actually displayed. There won't *be* a difference, since the HSW function must create color values that Windows will accept, and Windows can't display 23,592,960 different colors.

On the other hand, the HSW function can't *really* accept that many valid combinations of H, S, and W values...

Invalid HSW Values

What is the Hue of a color when the Saturation is zero?

That's a lot like asking a meteorologist "*What is the Wind Direction when the Wind Speed is zero?*" The question is meaningless.

There are certain circumstances under which one or two of the HSW values will be meaningless, and changing them will have no effect on the color that is produced.

If the Saturation value is zero, the Hue value has no meaning and the color that is produced will be completely dependent on the Whiteness value. A Saturation value of zero always

produces a shade of gray, which has no "hue".

And if the Whiteness value is 255, that means that the HSW color should contain 100% of the whiteness that Windows can produce, and that doesn't leave any room for *differences* between the Red, Green and Blue values. And since those differences are what create Hue and Saturation, using a Whiteness of 255 will always produce a pure white, regardless of the Hue and Saturation values.

Even beyond those extremes, using very *small* Saturation values or very *large* Whiteness values will produce a limited selection of Windows colors. For example, using HSW to create a dark red color with a very low Saturation might create a Windows color with Blue and Green values of zero (0), and a Red value of one (1). If you were to change the Hue value from zero to 20 degrees, the HSW function would not be able to create a different, valid RGB color, because Windows can't use *fractional* values for R, G, and B. When the Hue reached 40 degrees the HSW function would be able to change the Green value to one, and a different color would finally be produced, but the entire range from zero to 39 degrees would produce a single dark-red color.

The bottom line is that the HSW function will often produce *identical* colors when you give it *similar* parameters, especially when you are using values are that very close to the legal limits of a parameter. But it is possible to create all of the 16,777,216 Windows colors using HSW, so this is not really a "limitation" of the HSW system.

The HSW "Points" System

In its default mode, the HSW color system produces colors that are distributed in a way that is very similar to Windows' native colors. For example, if you use Graphics Tools to create an HSW "spectrum" and compare it to the spectrum that is displayed by the Windows ChooseColor dialog (see SelectGfxColor) they will look quite a bit alike.

Specifically, the areas of the spectrum around red, blue, and green will be expanded, and the colors in between -- yellow, cyan, and magenta -- will be compressed into narrower stripes. Because the three primary colors tend to dominate the spectrum, we call this a "three point color system". Again, this is the system that Windows uses, and it is Graphics Tools' default mode.

If you add this line of code to your program...

```
GfxOption GFX_HSW_POINTS, 6
```

...then Graphics Tools will use a *six*-point HSW system. The distribution of the colors around the *six* most important colors (red, green, blue, yellow, cyan, and magenta) will be distributed more evenly. This results in a spectrum that looks somewhat different. Some people find it more pleasing, and it can be used to produce some interesting effects. If you use 12 points, the HSW system will use the six most important colors and the six colors in between them, resulting in an increase in the "narrowing" effect.

View the image in `\GfxTools\Images\HSW_Points.jpg` to see the effect of using 3, 6, 12, and 24 points. Values larger than 24 can be used, but this usually results in distorted colors. This may or may not be useful to you.

Gradients

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

Gradients are very powerful tools, and learning to get the most out of them can be fairly complex. We have divided this document's discussion of gradients into several sections, progressing (roughly) from basic to complex topics. We recommend that you read *at least* the first few sections of this discussion before beginning your first experiments with gradients. To get the most out of your gradients -- to be able to fine-tune them to produce exactly the results you want -- you may need to read all of these sections

An Introduction to Gradients

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

A gradient is a field of color that is different in different places. The most common example of a very simple gradient is a "window caption" bar (at the top of a program's main window) that changes from dark blue on the left to light blue on the right. But gradients are *much* more powerful than that.

You can think of a gradient as an *array* of colors, where each color is slightly different from the one before. But the array is endless; Graphics Tools actually *calculates* the colors, rather than using an array of pre-calculated values.

When you tell it to use a gradient, Graphics Tools draws with the first color, then the second, then the third, and so on, automatically. The end result is a smooth transition from one color to another. In most cases the individual colors are not visible, because they appear to blend together.

Unfortunately, Microsoft Windows cannot display gradients when the display is limited to 256 colors, and Windows Help Files (and Adobe PDF Files) are limited to 256 colors, so this document does not contain pictures of gradients. We have provided several sample bitmaps, however, in the `\GfxTools\Images` directory. All of the `.bmp` and `.JPG` files that are mentioned in this document are located there. They can be viewed with Microsoft Paint, which is installed on all Windows computers, or they can be displayed by a Graphics Tools program that you write.

We would like to emphasize that the bitmaps that are provided in the `\GfxTools\Images` directory represent only a tiny fraction of all of the gradients that Graphics Tools can create. Literally billions of different gradients can be created.

See `BlueGradient.bmp` for an example of a very simple gradient.

Limitations of Gradients

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

While gradients provide an extremely powerful way to fill areas with color, they do have a few limitations that you should keep in mind.

Windows cannot display a gradient when the screen is limited to 256 or fewer colors. If your Graphics Tools programs will be run on systems that are set up for 16-color or 256-color operation, you should probably avoid using gradients unless you use gradients that are specifically designed for use in those environments. Displaying gradients requires a computer to display many different, subtle color variations, and 256 colors simply isn't enough for most types of gradients.

It is possible to display a good-quality gradient if a computer is configured for 32k or 64k colors. The best results, however, are produced on systems that are configured for TrueColor or TrueColor32.

If your system (like *most* modern systems) is configured for *at least* 32k Colors, gradients can add a very impressive dimension to your programs. And we mean "dimension" in more ways than one. Many of the realistic three-dimensional effects that can be produced with Graphics Tools rely on gradients.

Another limitation of gradients can be *speed*. As you can imagine, creating a complex gradient to fill a large three-dimensional figure (for example) is a very complex operation, much more complex than filling an area with a solid-color brush. So it may take up to several seconds for Graphics Tools to paint a large gradient. Some gradients can be created quickly, but others may take a relatively long time. This is normal and unavoidable.

The last noteworthy limitation of gradients is caused by Windows "digital" nature. Windows is not capable of drawing perfectly round circles without "jaggies", i.e. subtle steps in the edges of the circles. In fact if you look closely enough, even straight lines that are drawn at angles have a slightly "jagged" look. This is not a defect in Graphics Tools, this is a side-effect of the fact that Microsoft Windows is a digital system, not an analog system, and it can produce subtle imperfections in gradients. Most of the time these effects will not be noticeable, but there may be times when you will need to modify your gradient designs to reduce or eliminate these imperfections.

Basic Gradient Types

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

Most people are familiar with the "window caption" type of gradient, which changes from dark "navy" blue on the left to light "sky" blue on the right. This is called a Whiteness Gradient because the color itself (blue) and the a*mount* of color do not really change. Increasing amounts of white are added to the blue color to achieve the desired result.

Most programmers know that any color can be described in terms of the amounts of Red, Blue, and Green that the color contains. R, G, and B values are always specified as numbers between zero (0) and 255. For more information about RGB colors, see Windows Colors.

If you have read the section of this document about HSW Colors then you are also familiar with the terms Hue, Saturation, and Whiteness. Those three values can also be used to describe a color, and they are much more "user friendly" than Red, Green, and Blue values.

Hue is the basic color or "tint", such as Red, Green, Blue, Yellow, Orange, and so on. Hue is measured on a scale of zero (0) to 360 degrees, where 120 degrees is Green, 240 degrees is Blue, and Red can be represented by either zero (0) or 360 degrees. Every other color can be represented by a different angle around the "color wheel". For example, Purple is half-way between Blue and Red, at 300 degrees.

Saturation is the amount or "intensity" of the color, such as faint blue, rich blue, and so on. Saturation is measured on a scale of zero (0) to 255. A "fully saturated" color is one that is as intense as the system can reproduce. A color with "zero saturation" is a shade of gray, somewhere between black and white.

Whiteness is the amount of white that the color contains. Whiteness is also measured on a scale of zero (0) to 255.

Those three properties can be used to describe any color that Windows can display. For more information about Hue, Saturation, and Whiteness, see HSW Colors.

Gradients can be created using either Red, Green, and Blue values, or with Hue, Saturation, and Whiteness (HSW) values. While you may be tempted to "stick with what you know" and use only RGB gradients, we strongly encourage you to experiment with HSW gradients. The HSW system is much easier to use than RGB, and can produce *much* more useful gradients.

Using Predefined Gradients

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

The easiest way to use a gradient is to use one that is predefined. Graphics Tools has two built-in "default" gradients, and the directory `\GfxTools\Grads\` contains a number of predefined gradients stored as disk files.

The Default Gradients

If your Graphics Tools Pro program is already up and running (see Four Critical Steps for Every Program) then this is all you have to do to draw a rectangle using one of the default Graphics Tools gradients:

```
Dim tGradient As GT_HSW_GRADIENT

InitGradientHSW tGradient

UseGradient tGradient

GradientBrush GFX_ALL

DrawRect 500,500
```

The first step (the `DIM`) creates a variable (a UDT) that Graphics Tools will use to store the gradient. Most of the examples in this document use `tGradient` for the name of this structure, but you are free to use any variable name(s) you like. The choice of `GT_`**`HSW`**`_GRADIENT` indicates that you want to create an HSW gradient. You could also use `GT_`**`RGB`**`_GRADIENT` to create an RGB Gradient, to produce different effects.

The second step (`InitGradientHSW`) tells Graphics Tools to "initialize" the `tGradient` structure, and get it ready for use. This step sets up default values that are a good starting point for most gradients. If you used `GT_`**`RGB`**`_GRADIENT` in the first step, you would use `InitGradient`**`RGB`** in the second step.

The third step (`UseGradient`) tells Graphics Tools "use this gradient".

The fourth step (`GradientBrush`) tells Graphics Tools to use a gradient instead of a brush when filling certain types of figures. The use of `ALL` tells Graphics Tools to use a gradient for all of the figures that support gradients.

The fifth step (`DrawRect`) actually draws the rectangle. Because the DrawRect function supports gradients, Graphics Tools will fill the inside of the rectangle with the gradient instead of a brush.

If you add those five lines of code to a Graphics Tools program you will see a rectangle filled with a "spectrum"-style gradient. If you change "HSW" to "RGB" in that sample code, you will see a simple blue gradient. Those are the two Graphics Tools default gradients.

Loading Saved Gradients

Using a gradient that has been saved in a disk file is even easier than using a default

gradient. It is not necessary to initialize the gradient, and the process of "loading" a gradient also tells Graphics Tools to "use" it.

```
GradientLoad "\GfxTools\Grads\Spectrum.gtg"

GradientBrush GFX_ALL

DrawRect 500,500
```

The file extension `.gtg` stands for Gradient Tools Gradient. You are not required to use this extension when you save gradients that you create, but it is recommended.

If you load a gradient from a disk file you can use the GradientValue function to modify it before it is used. You can optionally use the GradientSave function to save those changes.

You can also obtain a copy of a gradient that is loaded from disk like this:

```
Dim tGradient As GT_HSW_GRADIENT

GradientLoad "\GfxTools\Grads\Spectrum.gtg"

tGradient.Gradient.lGradientType = QUERY

UseGradient tGradient
```

The UseGradient function will see the `QUERY` flag, and instead of *using* `tGradient` it will return a copy of the current gradient *in* `tGradient`. It is not necessary -- in fact it is not recommended -- to use one of the two InitGradient functions before using UseGradient in this way.

Using Gradients to Fill Specific Figures

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

As shown briefly in the previous section, the GradientBrush function is used to tell Graphics Tools that it should fill figures with a gradient instead of a brush.

If you use this line of code...

```
GradientBrush GFX_ALL
```

...Graphics Tools will fill many different figures (see below) with gradients whenever the figures are drawn.

You can also tell Graphics Tools to fill only certain types of figures. Instead of ALL you can use the following values:

`GRADIENT_FILL_SQUARE_FIGURES` tells Graphics Tools to use a gradient instead of a brush when your program uses the DrawArea, DrawRect, DrawXagon, DrawPgram, and DrawCube functions.

`GRADIENT_FILL_ROUND_FIGURES` tells Graphics Tools to use a gradient instead of a brush when your program uses the DrawCircle, DrawEllipse, DrawEllipseRotated, DrawPie, DrawCylinder, and DrawWedge functions.

`GRADIENT_FILL_TEXT_BODY` tells Graphics Tools to use a gradient instead of a solid color (GfxFontColor) when your program uses the DrawTextBox and DrawTextRow functions.

`GRADIENT_FILL_TEXT_EFFECTS` tells Graphics Tools to use a gradient instead of a solid color (see various GfxOption settings) when your program uses the DrawTextRow function to draw text that has a shadow or outline.

Figures that are not listed above cannot be filled with gradients.

You can tell Graphics Tools to fill only square figures with gradients, for example, by doing this:

```
GradientBrush GRADIENT_FILL_SQUARE_FIGURES
```

Or you can specify two or more types of figures like this:

```
GradientBrush GRADIENT_FILL_SQUARE_FIGURES OR
GRADIENT_FILL_ROUND_FIGURES
```

Using `ALL`, as you might expect, is the same as using all of the `GRADIENT_FILL_` values at the same time.

If you want to tell Graphics Tools to *stop* filling figures with gradients, you can use either one of the following two methods:

```
GradientBrush GFX_NONE

GradientBrush 0
```

Gradient Fill Types

Most Graphics Tools functions that support gradients allow two or more different "fill types" to be used.

The numbers zero (0) through three (3) are used to specify different fill types. The numbers have not been given "names" because they create different effects when they are used with different figures.

If you use the code on the previous page you will see a rectangle that is filled with a "spectrum" gradient with the lines of color running horizontally. The GradientValue function can be used to change that. In fact, as you will see, the GradientValue function can be used to change many different gradient properties. If you add this line of code...

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientValue GRADIENT_FILL_TYPE, 1
GradientBrush GFX_ALL
DrawRect 500,500
```

...and run the program again, the lines of color will run vertically. When you are drawing a rectangle, Gradient Fill Type 0 (the default) is "horizontal" and Gradient Fill Type 1 is "vertical". Types 2 and 3 are not defined for rectangles. (There are other ways to affect the "angle" at which a gradient is drawn, which will be discussed later.)

Now instead of a rectangle, let's draw a circle.

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientBrush GFX_ALL
DrawFrom 512,512
DrawCircle 250
```

If you use Fill Type 0 (the default) you will see a circle that is filled with a gradient from the inside out. In other words, if the gradient starts with red, the center of the circle will be red and the colors will change as the gradient moves toward the perimeter of the circle.

If you use Fill Type 1 like this...

```
UseGradient tGradient
GradientValue GRADIENT_FILL_TYPE, 1
GradientBrush GFX_ALL
```

...you will see a circle that is red around the perimeter and the colors will change as the gradient moves toward the center.

If you use Fill Type 2 you will see a circle that is filled "radially". A red line will appear at zero degrees (nine o'clock) and the colors will change as the gradient moves *around* the circle.

If you use Fill Type 3 you will see a circle that is filled "elliptically". A very narrow red ellipse will be drawn up and down along the middle of the circle, and the colors will change as the ellipse gets *wider* and fills the circle This results in a circle that has stripes that resemble a "beach ball".

To see these different DrawCircle fill types, see Gradient Circles sample program.

Each type of figure responds in its own way to the "fill type" numbers. For example, the DrawCube function will fill the cube from top to bottom, left to right, or front to back, depending on the number.

Changing The Gradient Angle

Some gradients that use straight lines, such as those that are used to fill rectangles and text, have an `ANGLE` property that can be adjusted.

As mentioned in the previous section you can cause a rectangle to be filled either "vertically" or "horizontally" by changing the gradient Fill Type. The `ANGLE` property gives you much finer control.

For example, using this code from the last section...

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientValue GRADIENT_FILL_TYPE, 1
GradientBrush GFX_ALL
DrawRect 500,500
```

...you can tell Graphics Tools to fill the rectangle using a gradient that uses vertical lines instead of horizontal lines.

Using this line of code will have exactly the same effect:

```
GradientValue GRADIENT_ANGLE, 90
```

Or, if you wanted the lines to be drawn at a 45 degree angle, you could use 45 instead of 90. Any number between zero (0) and 359 can be used, but keep in mind that the numbers between zero and 179, and the numbers between 180 and 359, have the same effect. (Actually, you can use any integer, positive or negative for this value. Graphics Tools automatically "normalizes" the value to be in the range 0-359.)

The `GRADIENT_ANGLE` property affects gradients that are drawn with the DrawRect, DrawArea, and DrawTextRow functions.

The `GRADIENT_ANGLE` property also affects the DrawXagon function, but not in the same way. An x-agon is really a "circular" figure, that is, it is drawn based on a center point and regular angles. (An x-agon with an infinite number of sides would be a circle.) The `GRADIENT_ANGLE` property cannot affect a gradient-filled x-agon in the normal way because an x-agon gradient requires lines that are drawn at many different angles. But that same use of multiple angles makes it difficult for Graphics Tools to "scale" a gradient x-agon properly. If you use gradient-filled x-agons in a world that is not square, you may need to set the `GRADIENT_ANGLE` property to 90 (degrees) to obtain proper gradient scaling.

All other figures should be drawn with a `GRADIENT_ANGLE` of zero (0). Failure to do this can result in distorted gradients.

Designing Your Own Gradients

The easiest way to design your own gradient is to modify one of the default gradients, so that is how we will begin. As described in the previous section, this code can be used to load the default HSW "spectrum" gradient and use it to draw a rectangle.

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientBrush GFX_ALL
DrawRect 500,500
```

If you add one line of code you can modify the default gradient, and change the rate at which the spectrum changes color (changes its Hue):

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientValue HSW_GRADIENT_HUE_RATE, 5000
GradientBrush GFX_ALL
DrawRect 500,500
```

Using values other than 5000 would produce gradients that change color at different rates. Larger numbers will cause the color to change more rapidly, smaller numbers more slowly. Using a rate of zero means "this property does not change".

The GradientValue function can be used to change *many* different aspects of a gradient. For example...

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientValue HSW_GRADIENT_HUE_RATE, 0
GradientValue HSW_GRADIENT_SATURATION_RATE, 1000
GradientBrush GFX_ALL
DrawRect 500,500
```

...would modify the default gradient so that the Hue did *not* change (rate = 0) and so that the Saturation value changed at a moderate rate (1000). This would result in a gradient of a single color, with a saturation (color intensity) that would change. The default starting color is red, so if you used that code you would see a rectangle that was filled with a red gradient. If you added this additional line of code...

```
GradientValue HSW_GRADIENT_HUE_START, 240
```

...the gradient would be blue instead of red, because "240 degrees" corresponds to blue on the HSW color wheel.

An Alternate Method

If you are going to make several different modifications to a gradient, or if you are designing a gradient from scratch, you may want to use a different method of setting values. Instead of (or in addition to) using the GradientValue function, you can do this:

```
  Dim tGradient As GT_HSW_GRADIENT
  InitGradientHSW tGradient
  tGradient.Hue.lRate = 0
  tGradient.Saturation.lRate = 1000
  '(many other changes, if desired)
  UseGradient tGradient
  GradientBrush GFX_ALL
  DrawRect 500,500
```

You can use this type of syntax to make changes in the `tGradient` structure, and then tell Graphics Tools "use this gradient". This method is more efficient, code-wise, than using the GradientValue function to set elements individually.

This technique also makes it easy to create a "complete" gradient with code, and to experiment with different values:

```
  Dim tGradient As GT_HSW_GRADIENT
  InitGradientHSW tGradient
'----------------------------------- GRADIENT
  tGradient.Gradient.lGradientType  = HSW_GRADIENT
  tGradient.Gradient.lFillType      = 0
  tGradient.Gradient.lAngle         = 0
  tGradient.Gradient.lTexture       = 0
  tGradient.Gradient.lTextureStart  = 0

  tGradient.Hue.lMin                = 0
  tGradient.Hue.lMax                = 0
  tGradient.Hue.lStart              = 0
  tGradient.Hue.lRate               = 1024
  tGradient.Hue.lOptions            = 0
  tGradient.Hue.lOffset             = 0
  tGradient.Hue.lTexture            = 0
  tGradient.Hue.lCutoff             = 50

  tGradient.Saturation.lMin         = 0
  tGradient.Saturation.lMax         = 255
  tGradient.Saturation.lStart       = 255
  tGradient.Saturation.lRate        = 0
  tGradient.Saturation.lOptions     = 0
  tGradient.Saturation.lOffset      = 0
  tGradient.Saturation.lTexture     = 0
  tGradient.Saturation.lCutoff      = 50

  tGradient.Whiteness.lMin          = 0
  tGradient.Whiteness.lMax          = 255
  tGradient.Whiteness.lStart        = 0
  tGradient.Whiteness.lRate         = 0
  tGradient.Whiteness.lOptions      = 0
  tGradient.Whiteness.lOffset       = 0
  tGradient.Whiteness.lTexture      = 0
  tGradient.Whiteness.lCutoff       = 50

  UseGradient tGradient

  GradientBrush GFX_ALL
  DrawRect 500,500
```

Using the "raw" structure also allows you to create two or more different gradients, and switch back and forth between them very quickly by using the UseGradient function:

```
Dim tGradient_1 As GT_HSW_GRADIENT
Dim tGradient_2 As GT_HSW_GRADIENT

GradientBrush GFX_ALL

InitGradientHSW tGradient_1

InitGradientHSW tGradient_2
tGradient_2.Hue.lRate = 0
tGradient_2.Saturation.lRate = 1000

UseGradient tGradient_1
DrawFrom 0,0
DrawRect 500,500

UseGradient tGradient_2
DrawFrom 512,512
DrawRect 500,500
```

You could even create an *array* of gradients!

Gradient RATE Values

The rest of the documentation about Gradients assumes that you are using the GradientValue *method of setting gradient properties. If is also possible to use the* "alternate method" *to do everything that is described here.*

If you are using an HSW gradient you can use these properties:

```
HSW_GRADIENT_HUE_RATE
HSW_GRADIENT_SATURATION_RATE
HSW_GRADIENT_WHITENESS_RATE
```

...to change the speeds at which those gradient properties change. If you are using an RGB gradient, you can use:

```
RGB_GRADIENT_RED_RATE
RGB_GRADIENT_GREEN_RATE
RGB_GRADIENT_BLUE_RATE
```

Different effects can be produced by having two or three properties change at the same rate, or at different rates. For example, you might create a gradient where both the Hue and Saturation change.

Note that it is NOT possible to mix HSW and RGB properties in the same gradient. You cannot, for example, create a gradient with both a `GREEN_RATE` and a `HUE_RATE`.

Gradient MIN and MAX Values

Each of the gradient properties (Hue/Saturation/Whiteness or Red/Green/Blue) has a `MIN` and `MAX` property associated with it, such as `HSW_GRADIENT_SATURATION_MAX`. The default `MIN` value is zero (0) and the default `MAX` value is 255, except for the Hue property. Hue is a special case that is discussed separately. See Why Hue Is Different.

Normally a gradient will "bounce" between the `MIN` and `MAX` values. For example, if the `GRADIENT_SATURATION_RATE` is set to a value like 1000, the Saturation value will start at zero (0) and increase until it reaches the default `MAX` value (255), then it will begin decreasing. When it reaches the default `MIN` value it will begin increasing again. It will continue to "bounce" between the `MIN` and `MAX` values indefinitely.

If you change the various `MIN` and/or `MAX` values, you will change the gradient accordingly.

Here are the valid `MIN` and `MAX` properties:

```
HSW_GRADIENT_HUE_MIN
HSW_GRADIENT_HUE_MAX
HSW_GRADIENT_SATURATION_MIN
HSW_GRADIENT_SATURATION_MAX
HSW_GRADIENT_WHITENESS_MIN
HSW_GRADIENT_WHITENESS_MAX

RGB_GRADIENT_BLUE_MIN
RGB_GRADIENT_BLUE_MAX
RGB_GRADIENT_GREEN_MIN
RGB_GRADIENT_GREEN_MAX
RGB_GRADIENT_RED_MIN
RGB_GRADIENT_RED_MAX
```

Gradient START Values

All of the various gradient `START` properties start at zero (0) by default. This creates, for example, gradients that start out with zero Saturation (no color) and increase to full Saturation. Because the color red corresponds to zero degrees on the HSW color wheel, the `HSW_GRADIENT_START_HUE` start color is red.

Here are the valid `START` properties:

```
HSW_GRADIENT_HUE_START
HSW_GRADIENT_SATURATION_START
HSW_GRADIENT_WHITENESS_START

RGB_GRADIENT_RED_START
RGB_GRADIENT_GREEN_START
RGB_GRADIENT_BLUE_START
```

Note that it is possible to set up a conflict between a `START` value and the corresponding `MIN` and `MAX` values. Consider this code:

```
GradientValue HSW_GRADIENT_SATURATION_MIN,     0
GradientValue HSW_GRADIENT_SATURATION_MAX,   128
GradientValue HSW_GRADIENT_SATURATION_START, 255
```

That's an impossible situation, because you are telling the gradient to start "out of range". Graphics Tools resolves all conflicts like that internally, as best it can.

There are several other gradient properties that can be adjusted. They are described below, but first we must pause to explain...

Why Hue Is Different

The Red, Green, Blue, Saturation, and Whiteness gradient properties all have a maximum range of zero (0) to 255, and the gradient's values normally "bounce" between those values as described above. The gradient Hue property is different in two very important ways.

1) Hue is measured in "degrees", between zero and 360, not zero and 255, and...

2) Hue can be visualized as a "continuous" range.

For example, a Hue value of zero degrees corresponds to Red, 120 degrees is Green, and 240 degrees is Blue. But if you continue around the "color wheel" to 360 degrees, you get Red again. Zero and 360 degrees represent exactly the same color, so it is not *necessary* for a Hue value to "bounce" when it reaches a MIN or MAX value. It can simply "keep going".

In fact, the default value for `HSW_GRADIENT_HUE_MIN` is zero, and the default value for `HSW_GRADIENT_HUE_MAX` is *also* zero, not 360. This results in Hue gradients that "rotate" rather than "bounce". Here is a more specific example...

Let's say you have a gradient with an `HSW_GRADIENT_HUE_RATE` of 1000. (The actual value is not important, as long as it is not zero.) The gradient's Hue would start out at zero (Red), progress along to 120 (Green), then to 240 (Blue), and then it would reach 360, which is Red again. The Hue would then *continue to rotate* around to 120, then 240, and so on.

But if you were to specify an `HSW_GRADIENT_HUE_MAX` value of 360, the Hue would start at zero, progress to 120 and then 240, but when it reached 360 it would "bounce" and begin moving back toward 240, then 120, then zero. Finally, when it reached zero, it would "bounce" again and begin moving toward 120... and so on indefinitely.

Using different MIN and MAX values with the Hue property can produce some interesting effects. For example, using a MIN of 240 and a MAX of 360 produces a gradient that "bounces" between Blue (240 degrees) and Red (360 degrees). It is important to note that using a `MAX` of zero degrees (also Red) would *not* work. When Graphics Tools sees a `HUE_MAX` value of zero, it interprets that to mean "allow continuous Hue rotation".

Please note that Hue is *always* measured in degrees from 0-360, even if you use the GfxOption `GFX_USE_RADIANS` option (which tells Graphics Tools to use radians instead of degrees for most angles).

Gradient OPTIONS Values

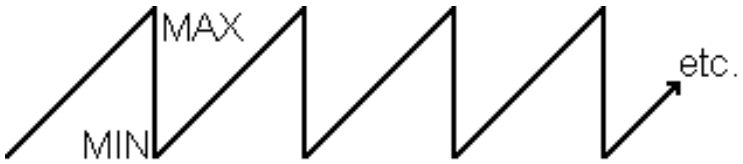
Each gradient property has an additional value called `OPTIONS`.

```
HSW_GRADIENT_HUE_OPTIONS
HSW_GRADIENT_SATURATION_OPTIONS
HSW_GRADIENT_WHITENESS_OPTIONS

RGB_GRADIENT_RED_OPTIONS
RGB_GRADIENT_GREEN_OPTIONS
RGB_GRADIENT_BLUE_OPTIONS
```

The `OPTIONS` values can be used to affect the *way* in which the gradient's values change. Normally, properties like Red, Green, Blue, Hue, Saturation, and Whiteness are "linear". That is, they progress very smoothly, bouncing back and forth between their `MIN` and `MAX` values. You can picture this normal progression as a "triangle wave" that looks something like this:

The `OPTIONS` properties can be used to create a "sawtooth gradient" with values that would be graphed like this:

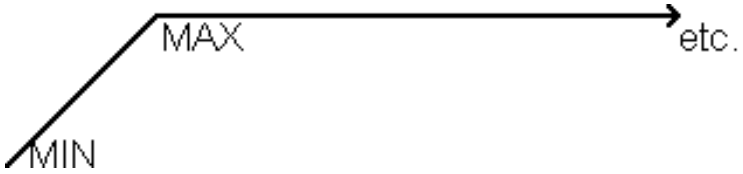


You would create a "Sawtooth Saturation" gradient with this line of code:

```
GradientValue HSW_GRADIENT_SATURATION_OPTION, _
              GRADIENT_SAWTOOTH
```

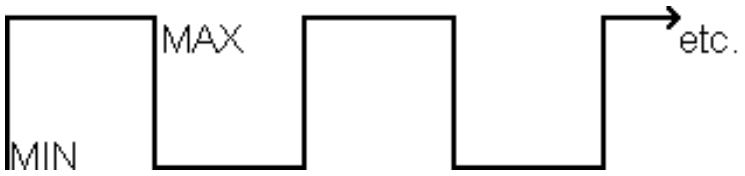
For visual examples of how the `GRADIENT_SAWTOOTH` option affects gradients, see the `Gradients.jpg` image in the `\GfxTools\Images` directory.

The `OPTIONS` properties can also be used to create "single-cycle" gradients that would have graphs like this:



A single-cycle gradient progresses from the `MIN` to the `MAX` value, but then *stays* at the `MAX` value no matter how far the gradient progresses. (Alternatively, a single-cycle gradient can progress from `MAX` down to `MIN` and stay at the `MIN` value.) Use the `GRADIENT_SINGLE` option to create a single-cycle gradient.

The OPTIONS properties can also be use to create "binary" or "two-state" gradients:



A binary gradient does not progress smoothly from one color to another, it switches abruptly between two colors. Use the GRADIENT_BINARY option to create a binary gradient. The gradient properties that end in CUTOFF (such as RGB_GRADIENT_SATURATION_CUTOFF) are used to define the point where a binary gradient switches, from 0% to 100%. The default value is 50%.

The OPTIONS properties can also be used to "sharpen" the gradient so that a graph of the values would look like this:

Or you can "soften" the gradient like this:

Many different levels of sharpening and softening are available.

```
GRADIENT_SOFTEN
GRADIENT_SOFTEN_x2
GRADIENT_SOFTEN_x3
GRADIENT_SOFTEN_x4
GRADIENT_SOFTEN_x5
GRADIENT_SOFTEN_x6
GRADIENT_SOFTEN_x7

GRADIENT_SHARPEN
GRADIENT_SHARPEN_x2
GRADIENT_SHARPEN_x3
GRADIENT_SHARPEN_x4
GRADIENT_SHARPEN_x5
GRADIENT_SHARPEN_x6
GRADIENT_SHARPEN_x7
```

Those values are used like this:

```
GradientValue HSW_GRADIENT_SATURATION_OPTION, _
              GRADIENT_SOFTEN
```

For visual examples of how sharpening and softening affect a gradient, see the `Gradient_Sharpness.jpg` image in the `\GfxTools\Images` directory.

You can also use many of the `OPTIONS` value in various combinations. The `SOFTEN` and `SHARPEN` values cannot be combined with each other (because they would cancel each other our) but you can, for example, use the `GRADIENT_SHARPEN` and `GRADIENT_SAWTOOTH` options together, like this:

```
GradientValue HSW_GRADIENT_RED_OPTION, _
              GRADIENT_SHARPEN OR GRADIENT_SAWTOOTH
```

Many, many different combinations are possible, and each one produces a different visual effect.

Using Gradient "Textures"

Graphics Tools can add a "texture" to a gradient by adding a small amount of "randomness" to the colors that are generated. For example, if you do this...

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientBrush GFX_ALL
DrawRect 500,500
```

...you will see a rectangle that is filled with a normal, smooth 'spectrum" gradient. If you add this line of code...

```
Dim tGradient As GT_HSW_GRADIENT
InitGradientHSW tGradient
UseGradient tGradient
GradientValue HSW_GRADIENT_HUE_TEXTURE, 5
GradientBrush GFX_ALL
DrawRect 500,500
```

...the gradient's colors (hues) will be less smooth. The hue of each part of the spectrum will be "randomized" by plus-or-minus five (5) degrees. Using larger or smaller values will add more or less randomness. Using a texture value that is equal to range of the value (such as using a texture of 360 for Hue or 255 for other properties) will result in a gradient that is completely randomized. There will be no pattern to the gradient.

Using a texture value of five (5) or less is a good way to make a gradient appear slightly "less perfect". This can actually enhance the illusion that a drawing is a "real" object.

If you use a *negative* value such as negative five (`-5`) Graphics Tools will add plus or minus five *percent* to the value, instead of plus-or-minus five "units". This produces a slightly different visual effect.

Gradients can also enhance certain 3D effects by giving the eye additional lines to follow.

The following gradient textures can be used:

```
HSW_GRADIENT_HUE_TEXTURE
HSW_GRADIENT_SATURATION_TEXTURE
HSW_GRADIENT_WHITENESS_TEXTURE

RGB_GRADIENT_RED_TEXTURE
RGB_GRADIENT_GREEN_TEXTURE
RGB_GRADIENT_BLUE_TEXTURE
```

In addition to the textures that can be added by randomizing the various gradient properties, you can add a texture to the *overall* gradient in a different way by using...

```
GRADIENT_TEXTURE
```

That will randomize everything at once, using a somewhat different technique.

As described in Introduction To Gradients, you can picture a gradient as being "an array of colors". Normally Graphics Tools uses the colors in precise numeric order... 1, 2, 3, 4, and so on. But using something like this

```
GradientValue GRADIENT_TEXTURE, 5
```

...tells Graphics Tools to use slightly randomized numbers. For example, if the value 5 was used for `GRADIENT_TEXTURE`, instead of starting with element 1 the gradient would start out with a color that corresponds to a random color between -4 and +6. Instead of color 2 it would use a random color between -3 and +7. Instead of color 3 it would use a color between -2 and +8. The "plus or minus 5" value tells Graphics Tools "randomize the gradient color *number* that you use". This results in a texture that is different from textures that are created with the individual gradient properties.

Changing the Texture

By default, Graphics Tools uses the same texture every time it draws something. This is done so that textures can "match" properly. For example, you may want all of the pies in a pie chart to have exactly the same texture so that when they are displayed together the "texture lines" will form circles that go all the way around the pie chart.

If you don't like the texture that is being created -- if, for example, it creates a dark or light line where you don't want one -- you can change the texture. Use this line of code:

```
GradientValue GRADIENT_TEXTURE_START, x
```

...where *x* is an integer in the `LONG` range.

If you want each figure that is drawn to have a different texture, use `GRADIENT_TEXTURE_START` with a different number for each figure. If you want gradients that are different each time a figure is drawn, use the Visual Basic or PowerBASIC `TIMER` function like this:

```
GradientValue GRADIENT_TEXTURE_START, TIMER*1000
```

Gradient OFFSET Values

The gradient `OFFSET` values are not used very often, but they can provide some useful effects.

For example, let's say that you are creating a spectrum-style gradient and you want to have continuously "rotating" colors. If you use `HUE START` and `HUE STOP` values of zero (0) the colors will rotate continuously around the color wheel. (See Why Hue Is Different for more information about this.) But what if you don't want the gradient to start with red (Hue angle zero)? That's where the `OFFSET` value can help.

If you use a `HUE OFFSET` value of 120, for example, the gradient will look exactly the same but the hue will be "offset" by 120 degrees, so instead of red the gradient would start with green (Hue angle 120).

An `OFFSET` value tells Graphics Tools "calculate the gradient normally, but add *x* to the final value".

You can use these properties to obtain different effects:

```
HSW_GRADIENT_HUE_OFFSET
HSW_GRADIENT_SATURATION_OFFSET
HSW_GRADIENT_WHITENESS_OFFSET

RGB_GRADIENT_RED_OFFSET
RGB_GRADIENT_GREEN_OFFSET
RGB_GRADIENT_BLUE_OFFSET
```

Experimenting with Gradients

Please note that gradients are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support gradients.

It would be impossible for this document to explain all of the different gradients that can be created with Graphics Tools, but we can give you a few ideas. Once you understand the basics of `RATE`, `MAX`, `MIN`, `START`, and so on, you are free to experiment with thousands of different combinations.

For example, if you create a gradient where the Hue and Saturation both change you can experiment with allowing the Hue to change at one rate and the Saturation to change at another rate.

Or you could create a gradient where the Whiteness and Saturation both change. Not only could they change at different rates, but one could be going "up" while the other one was going "down", to produce a completely different effect.

You can use an `OPTION` like `GRADIENT_SOFTEN` for one property (like Saturation) and `GRADIENT_SHARPEN` for another (like Whiteness).

Here are some other ways to change the way your gradients look..,

- Use the GfxOption `GFX_HSW_POINTS` option to change the way HSW colors are generated. See The HSW "Points" System.

- Use the GfxClipArea function to fill part of a figure with one gradient, and another part with a different gradient.

- Use the GfxClipArea function to create a circular clipping area, then fill it with a straight-line gradient by using DrawArea. Or create a rectangular clipping area, and fill it with one of the DrawCircle gradient patterns.

- Experiment with different ways to accomplish the same thing. For example, you might create a gradient-filled circle by using the DrawCircle function. If you use gradient Fill Type 3 you will get a "beach ball" effect where the stripes go up and down. By using DrawEllipseRotated instead of DrawCircle you can change the angle at which the stripes are drawn. (A "rotated circle" usually looks exactly like the original circle, but not in this case.)

- Remember that you can use gradients for things other than filling figures like circles and wedges. By using the GradientColor function and simple functions like DrawTo and DrawEllipse you can write programs that will draw just about anything you can imagine.

- Use *purposely* out-of range values. For example, the normal range of most gradient properties is 0-255, but if you use a `MAX` value of 300 the gradient will be "flattened" in the areas that have values between 256 and 300. Windows is strictly limited to a range of 0-255, so if you tell Graphics Tools to use a value of 300 (for example) it will be automatically "cut off" at 255. This will result in gradients with areas of color that do not appear to change, in between areas that do change. You can also use `MIN` values less than zero.

There are literally *billions* of possible gradient effects that can be created with Graphics Tools!

Displaying Images from Files

Images can be stored in many different types of disk files:

- Icons
- Cursors
- Bitmaps
- JPEG and JPG Files
- Windows Metafiles (WMF)
- TIFF Files
- PNG Files
- GIF Files (see below)

...and many others. Graphics Tools provides support for various image formats in three different ways.

1) Graphics Tools provides "direct support" for Icons, Cursors, and Bitmaps. That means that all of the functions for handling these formats are built into Graphics Tools, and they will work on all versions of 32-bit Microsoft Windows. Both Graphics Tools Standard and Pro can *display* icons, cursors and bitmaps, and can *create* bitmap files on any normal Windows system. See the DisplayBitmap, DisplayIcon, DisplayCursor, and SaveGfxWindow functions, among many others.

2) Graphics Tools Pro provides "external support" (via the Graphics Tools JPEG Library and the Intel JPEG Library) for displaying *and* creating JPEG files on all versions of 32-bit Windows.

3) Graphics Tools Standard and Pro both provide "indirect support" for *displaying* additional image formats. These formats typically include WMF and JPEG, and possibly others, but the actual image formats that are supported depends on the versions of Windows and Microsoft Internet Explorer that are installed on the runtime system. The code for displaying these image formats is not built into Graphics Tools, it simply tells Windows "please display this image". If Windows recognizes the file format, the image will appear in the graphics window. In not, an error code will be returned. The great majority of Windows computers support WMF and JPEG files. Early versions of Windows 95 and Internet Explorer, however, do not support those formats, so using the use of the DisplayImage and StretchImage functions may require Internet Explorer to be updated.

> PLEASE NOTE: Graphics Tools cannot be used to display GIF files even if the runtime system is configured to do so. The GIF file format is patented by UniSys Corporation, and UniSys does not allow GIF files to be used by any computer program without a license from UniSys. If GIF functionality was included in Graphics Tools, Perfect Sync would be required to obtain a license, and *you* would be required to obtain a license in order to use that functionality, even to simply *display* GIF files. Because of the license costs and legal complexities involved, Graphics Tools locks out the use of GIF files.

See Bitmaps and JPEG Files and Cursors and Icons.

Also see DisplayBitmap, StretchBitmap, CropBitmap, DisplayCursor, StretchCursor, DisplayIcon, StretchIcon, DisplayJpeg, StretchJpeg, CropJpeg, DisplayImage, and StretchImage.

Bitmaps and JPEG Files

Bitmaps are the "universal language" of graphics images. Almost every program that can use images can use bitmap files, including most drawing programs, word processors, and spreadsheets. Support for bitmaps is actually a part of Window itself, so many, many Windows programs can use them.

The most notable exception is the internet, and internet-related software. Bitmap files, while they produce very high-quality images, are usually quite large. On the internet, "big" means "slow", so a new image format called JPEG has been developed.

Many other graphic-image formats are used in the world today, such as GIF, PIFF, PCX, PNG, and so on, but none of them are as universally accepted as bitmaps as JPEGs. Some formats, such as GIF, have additional disadvantages such as restrictive licensing terms.

Bitmaps

The primary advantage of bitmap files is their high quality. In a bitmap, every pixel in an image is perfectly preserved. Several different bitmap file formats can be used, but all of them can store images without the loss of any details at all.

Bitmap Type	Bits Per Pixel
Monochrome	1
16 Colors	4
256 Colors	8
32/64k Colors	16
TrueColor	24
TrueColor32	32

It is important to keep the "number of colors" and "bits per pixel" numbers straight. For example, "16 colors" is *not* the same thing as "16 bits per pixel".

TrueColor bitmaps can reproduce 16,777,216 different colors. TrueColor is sometimes called "Millions Of Colors".

It is important to note that not all versions of Windows support the TrueColor32 format. Just as importantly, Windows itself is limited to 16,777,216 colors so using the TrueColor32 format does *not* result in a better quality image than standard TrueColor, and the file size is 33% *larger* than standard TrueColor. For those reasons, we recommend the use of the standard TrueColor format instead of TrueColor32.

The 16-bit-per-pixel format can be implemented differently by different video drivers. Some provide 32,768 colors, and some provide 65,536 colors. This difference usually has very little effect on the final result, and it is beyond the control of the programmer.

The 16-color and 256-color formats can be "compressed" using what is called Run Length Encoding (RLE) but not all software (or even all versions of Windows) can read RLE-compressed files, so Graphics Tools does not support the *saving* of RLE files.

The larger the number of bits-per-pixel that a bitmap uses, the more colors a bitmap can contain. But unfortunately, more bits-per-pixel means larger files. Here is a chart that shows the file sizes for a 400-by-400-pixel image saved as various types of bitmaps.

```
Type         File Size
-----------  ---------
Monochrome     20,862
16 Color       80,118
256 Color     161,078
32/64k Color  320,054
TrueColor     480,054
TrueColor32   640,054
```

Size is the main thing that makes bitmaps impractical for use on the internet. A 640x480 standard TrueColor bitmap, for example, would require a file larger than 920,000 bytes. Even with a high-speed connection to the internet, a file of that size would produce an unacceptable delay.

RLE Compressed Bitmaps

The standard 16-color and 256-color bitmap formats allow something called Run Length Encoding (RLE) compression. RLE-compressed bitmaps can be significantly smaller than normal bitmaps, with no loss of detail, especially if a bitmap contains large areas (or horizontal lines) of uniform colors.

Graphics Tools can *display* RLE-encoded bitmaps. No additional steps are necessary; Graphics Tools handles RLE bitmaps automatically.

For a number of reasons, Graphics Tools doesn't provide a method for *creating* RLE bitmaps:

- The 16- and 256-color bitmap formats produce relatively small files even without RLE compression (see above).
- The 16- and 256-color bitmap formats are gradually becoming less common, with the increasing popularity of TrueColor.
- External programs are available for converting standard bitmaps into RLE-compressed bitmaps, as well as many other image-file formats.
- Most importantly, *not all programs can use and display RLE-compressed bitmaps*.

JPEGs

JPEG stands for Joint Photographic Experts Group, the original name of the committee that wrote the standard. The primary advantage of JPEG files is their relatively small size. Their primary disadvantage is their somewhat reduced image quality.

JPEG is, by design, a "lossy" image compression technique. That means that some of the details of the original image are always lost in the process of creating a JPEG file. When you are dealing with "photographic" type images this loss of detail is often undetectable. In fact it can sometimes make the image more pleasing to the eye. But when you are dealing with "flat" images -- charts, graphs, cartoons, maps, screen shots, etc. -- you can almost always see the subtle imperfections that are caused by the JPEG encoding.

It is important to note that the *repeated* loading and saving of a JPEG image degrades the picture quality even more. The process of saving a JPEG file always involves a loss of quality, even if the original image was loaded from a JPEG file.

JPEG files can be produced in 100 different "qualities" from one to one hundred (1-100). A one-quality JPEG image will be very small, but the image will not usually be recognizable. A

100-quality JPEG image will usually look very good, but the file size will be considerably larger.

The final size of a JPEG file is not nearly as predictable as with bitmaps. The file size will depend very heavily on the *content* of the image. To give you an idea of how image quality and file size are related, here are the results of a *very simple* test that we performed with Graphics Tools Pro. The test image was 400x400 pixels, and consisted of a series of red lines and circles in a "crosshairs" pattern on a flat gray background. The compression percentages shown are comparisons to a 400x400 TrueColor bitmap (see above).

```
QUALITY   SIZE   COMPRESSION
-------  ------  -----------
   1      4,201     99.1%   (NOT RECOGNIZABLE)
  10      8,520     98.2%
  20     13,386     97.2%
  30     17,691     96.3%
  40     20,715     95.7%
  50     23,869     95.0%
  60     27,713     94.2%
  70     32,308     93.3%
  80     40,216     91.6%
  90     56,412     88.2%
 100    135,274     71.8%   (VERY GOOD QUALITY)
```

Remember, your results *will* vary, depending on the image that you are saving. And even a 100-quality JPEG will not usually look quite as good as a bitmap. That is the nature of JPEG encoding.

Many people consider 75% quality to be the "sweet spot" in the trade-off between size and quality. Our simple test produced these results:

```
  75     35,475     92.6%
```

But keep in mind that many of the images that you are likely to create with Graphics Tools will be closer to "charts and graphs" than "photographs", so you may wish to consider using 90- or even 100-quality JPEGs. The difference between the 75- and 90-quality images (in our simple test) was the difference between 92.6% and 88.2% compression. Not much of a difference. But if you look at it another way, the 90-quality file would take 1.6 times as long to download as the 75-quality file. On the internet, file size can be very important. But of course, so is image quality.

For reference, at the other end of the scale is the "barely recognizable" 5-quality JPEG image.

```
   5      5,445     99.1%
```

This may be appropriate if the final image will be displayed in a very small size. We do not recommend the use of zero-to-four (0-4) quality images for any purpose.

Bitmaps vs. JPEGs

The larger the image, the more of an advantage JPEG files will have. In fact, for small images, JPEG files are sometimes *larger* than bitmaps. Compare the two files called `Locator.bmp` and `Locator.jpg` in the `\GfxTools\Images` directory, and you will see that the `Locator.jpg` image is 767 bytes, compared to 630 bytes for `Locator.bmp`.

But keep in mind that many modern internet browsers, including Netscape and Microsoft Internet Explorer, cannot display bitmap files so *you must always use JPEG files when creating images for the internet*.

Other Image Size Considerations

In order to work well under all circumstances, bitmaps must contain an even number of pixels in both the horizontal and vertical directions. For that reason, Graphics Tools will not allow you to save a bitmap file that contains an odd number of pixels. If you attempt to save an image that contains an odd number of pixels, Graphics Tools will automatically decrease the width and/or height of the image by one pixel.

Similarly, JPEG images must have a height that is an even number of pixels and a width that is a multiple of four (4) pixels. Graphics Tools will automatically reduce the size of a saved image to avoid creating JPEG files that will not work well under all circumstances.

The Graphics Tools JPEG Library

Graphics Tools Standard and Pro both provide the ability to display JPEG files, but only if the runtime version of Windows supports JPEG. See Displaying Images From Files for more information about this.

Graphics Tools Pro provides the ability for your programs to display and create standard JPEG-format files by using either 1) the Graphics Tools JPEG Library (GTJL) or 2) the Intel JPEG Library (IJL). In almost all cases we recommend the use of the Graphics Tools JPEG Library. The GTJL is freely distributable (see below) and less than half of the size of the IJL. The Intel JPEG Library is provided primarily for backward compatability with programs written for Graphics Tools before version 2.10.

Distribution of the Graphics Tools JPEG Library

The Graphics Tools JPEG Library is a small (132k), freely distributable file (GfxT_Jpg.DLL) which provides Graphics Tools Pro programs with the ability to load and save JPEG files. The GfxT_Jpeg file is covered by the Graphics Tools License Agreement, which means that you can distribute it under the same terms that you distribute the main Graphics Tools DLL (GfxT_Std.DLL or GfxT_Pro.DLL).

If your program does not use any JPEG functions, or if it uses *only* the DisplayImage and/or StretchImage functions to display JPEG images, you do *not* need to distribute the Graphics Tools JPEG Library with your programs, and you do not need to read the rest of this section.

But if your program **1)** uses any of the Graphics Tools JPEG-loading functions (DisplayJpeg, StretchJpeg, CropJpeg, or JpegParam) or **2)** uses the SaveGfxArea or SaveGfxWindow function to save JPEG images, you *will* need to distribute the Graphics Tools JPEG Library runtime file as well as the Graphics Tools Standard or Pro runtime file. Failure to do so will result in an error (`ERROR_LIBRARY_NOT_FOUND`) at runtime.

The Intel® JPEG Library

Graphics Tools Standard and Pro both provide the ability to display JPEG files, but only if the runtime version of Windows supports JPEG. See Displaying Images From Files for more information about this.

When Graphics Tools Version 2.00 was introduced the Intel JPEG Library (IJL) was a package of freely-distributable JPEG-related functions that were provided, at no cost, by Intel Corporation's web site. Unfortunately, in 2002, Intel withdrew the IJL and replaced it with a much larger package which was not free. Perfect Sync then introduced a replacement for Graphics Tools programmers who use the IJL, called the Graphics Tools JPEG Library (GTJL).

By default, Graphics Tools Pro versions 2.10 and above use the Graphics Tools JPEG Library. If you want your programs to use the Intel JPEG Library instead, add this line of code immediately following the GraphicsToolsAuthorize function:

```
GfxOption %GFX_JPEG_LIB_VERSION, %INTEL_JPEG
```

Distribution of the Graphics Tools JPEG Library

If your does not use any JPEG functions or if your program uses *only* the DisplayImage and/or StretchImage functions to display JPEG images, you do *not* need to distribute a JPEG Library with your programs, and you do not need to read the rest of this section.

But if your program **1)** uses any of the Graphics Tools JPEG-loading functions (DisplayJpeg, StretchJpeg, CropJpeg, or JpegParam) or **2)** uses the SaveGfxArea or SaveGfxWindow function to save JPEG images, and if you choose to use the Intel JPEG Library instead of the Graphics Tools JPEG Library, thenyou *will* need to distribute the Intel JPEG Library runtime file as well as the Graphics Tools Standard or Pro runtime file. Failure to do so will result in an error (`ERROR_LIBRARY_NOT_FOUND`) at runtime.

IMPORTANT NOTE: This document is not intended to provide legal advice. You, the Graphics Tools licensee, are solely responsible for reading, understanding, and following the terms of the Intel JPEG Library License Agreement, which is included with the package.

As we understand the terms of the Intel JPEG Library License Agreement, if you (a Graphics Tools licensee) wishes to distribute the Intel JPEG Library you must download it *directly* from Intel. You may not *re*-distribute the copy that is included with Graphics Tools. Unfortunately, Intel no longer allows the IJL package to be downloaded. So to avoid possible legal problems related to the distribution of the Intel JPEG Library, Perfect Sync now recommends the use of the Graphics Tools JPEG Library instead of the Intel JPEG Library.

If you downloaded the IJL package from Intel *before* it was discontinued you will, just as before, need to decide whether or not your use of the package complies with the Intel License Agreement.

Cursors and Icons

As the terms are commonly used, "cursors" are small images (such as an arrow or hourglass) that are displayed to represent the current location of the mouse on the screen, and "icons" are small images that are double-clicked to start a program or perform some other function.

A Graphics Tools graphics window has a "mouse cursor" just like any other window. You can change each graphics window's mouse cursor individually, by using the GfxCursor function.

And your Graphics Tools programs can have "program icons" just like any other program. If you embed an icon in your program, Windows Explorer and Windows Shortcuts will be able to display that icon.

But Graphics Tools also allows your programs to display cursors and icons as *part of your graphics windows*. This is a very convenient technique for displaying pre-drawn images. Functions called DisplayCursor, DisplayIcon, StretchCursor, StretchIcon, AnimateCursor, AnimateIcon, AutoPlayCursor, AutoPlayIcon, and AutoPlayControl are provided.

Cursors and Icons can be treated much like bitmaps, but they have certain advantages over bitmaps. For example, you can create animated cursors and icons, and cursors and icons can have areas that are defined as "transparent". Bitmaps can't do those things.

Cursors and icons can contain only a limited number of colors -- usually either 16 or 256 -- and one of those colors is reserved to mean "transparent". Any portion of a cursor or icon that is drawn using that color will *not* be drawn on the screen, the background will be shown instead.

As far as Microsoft Windows is concerned, cursors and icons are nearly identical. In fact, in many cases the Graphics Tools cursor functions and icon functions can be used interchangeably. For example, the sample animated cursor called `\GfxTools\Images\Spinner.ani` can be displayed using any of the cursor *or* icon functions listed above. Sometimes, however, you may encounter an image that requires the use of *either* the cursor or icon functions. Generally speaking, files with the extension `ICO` should be displayed with the icon functions, and files with the extension `CUR` should be displayed with the cursor functions. Files with the extension `ANI` are usually animated *cursors*.

Graphics Tools functions can optionally load cursors and icons using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE` for more information.

Perfect Sync uses (and recommends) a program called MicroAngelo by Impact Software for creating cursors and icons. For information about MicroAngelo, you can visit http://www.impactsoft.com/. (Perfect Sync is not affiliated with Impact Software in any way.) Many other cursor-creation and icon-creation programs are also available.

Animated Cursors and Icons

See Cursors and Icons for background information.

An animated cursor or icon is actually a *series* of cursors or icons, contained in a single file. The individual images are called "frames", and the frames are displayed sequentially to create the illusion of movement. If you play all of the frames of an animated cursor or icon, it is called one "cycle".

As far as Microsoft Windows is concerned, cursors and icons are nearly identical. This is especially true about animated cursors and icons. In fact, in most cases the Graphics Tools cursor functions and icon functions can be used interchangeably. So for the sake of simplicity, the rest of this section will discuss only animated cursors. All of the information applies to animated icons, as well.

Animated cursor files usually have the extension `.ANI`, but it is not required. Perfect Sync uses (and recommends) a program called MicroAngelo by Impact Software for creating animated cursors. For information about MicroAngelo, you can visit http://www.impactsoft.com/. (Perfect Sync is not affiliated with Impact Software in any way.) Many other animated-cursor-creation programs are also available.

Display Your Animations In a Jiffy

Each frame of an animated cursor is allowed to have a different display time. The display time for each frame is defined when the cursor is created, and stored in the `ANI` file along with the frame's image.

The official Microsoft unit of measurement for display times is called (believe it or not) a "jiffy". One jiffy is equal to one one-sixtieth (1/60) of a second, or approximately 16.6667 milliseconds.

One problem with the Microsoft system is that most computers are only able to measure time in increments of 10 milliseconds, so jiffies are often rounded up to the next multiple of 10 milliseconds. For example, using a display time of one (1) jiffy may actually result in a frame being displayed for 20 milliseconds instead of 16.6667. Using two (2) jiffies will result in a display time of 40 milliseconds instead of 33.3, using three (3) will result in 50 milliseconds, and so on.

Problems with Animated Cursor "Overdraw"

Many animated cursors have relatively static outer borders. For example, an hourglass animation's "sand" might appear to change, but the outline of the hourglass itself will not. Or a rotating globe's sphere might appear to rotate, but the outline of the globe itself will not change. The same would be true for images of clocks, rotating wheels, "smiling faces" that change expression, blinking light bulbs, thermometers, progress bars, and many other types of common animated cursors.

For animations like those, the Graphics Tools default mode will work very well. By default, Graphics Tools simply draws the frames of the cursor directly onto the graphics window, one on top of the other. And since the outline will not change, the frames will cover each other up.

But if the animated cursor that you want to display has an outline that changes, Graphics Tools will be required to re-paint the cursor's background between each frame. For example,

a "running horse" cursor might not be displayed properly because the outline of the shape -- the horse's legs -- would be different from frame to frame. If the background is not re-drawn between each frame, the images will be superimposed on top of each other, and the horse will appear to have too many legs.

In some cases this can be a very useful "special effect", but in other cases it will be undesirable. To tell Graphics Tools to re-paint the background between the frames of animated cursors and icons, use this line of code...

```
GfxOption  GFX_ANIMATE_BKGD, TRUE
```

After the GFX_ANIMATE_BKGD mode has been activated, Graphics Tools will use the current Brush to re-paint the background between each frame of the animation. You may, of course, be required to create a Brush with a color and style that matches the area of the graphics window where the animation will be displayed.

IMPORTANT NOTE: It is usually not practical to use a Hatched Brush or a Bitmap Brush for the background of an animated cursor or icon. Because of the way in which Windows displays animation backgrounds, the incorrect portion of the hatch or bitmap may be used.

Please note that the use of the GFX_ANIMATE_BKGD mode will cause most animations to slow down very slightly. This is because Windows has to perform the additional task of re-painting the background between each frame, and that takes time. In most cases the effect will not be noticeable.

TIP: If you are using an animated cursor that will always be displayed on a certain background, you might want to modify the cursor so that it does not use the "transparent background" color. *Change the animated cursor to use the background color(s) on which the cursor will be displayed.* That way, all "background repainting" issues will be eliminated, because the entire cursor frame will always be displayed. The previous frame will always be covered up properly.

AutoPlaying Animated Cursors and Icons

When the AnimateCursor and AnimateIcon functions are used, Graphics Tools plays one "cycle" of the animation and then stops and returns control to your program, so it can do other things.

If you want an animation to run continuously for a period of time, you should use the AutoPlayCursor or AutoPlayIcon function. Animations that are started in this way can be controlled (paused, restarted, and stopped) using the AutoPlayControl function.

Using Embedded Resources

In addition to using image resources (cursors, icons, and bitmaps) that are located in disk files, many Graphics Tools functions can use resources that are "embedded" in an EXE or DLL file, right along with your program's executable code. Some programmers like to do this because it makes a program more "monolithic" -- there are fewer files to distribute -- and because it reduces the chances that a necessary image will be damaged, replaced, or deleted.

Most programming languages provide a mechanism for embedding resources in an EXE or DLL file. Please consult your programming language's documentation for the appropriate method(s) for your circumstances.

Once they are embedded, Graphics Tools can use the resources for many different operations. Simply specify the Resource ID number of the resource instead of a file name. For example, to display a bitmap that was embedded in your EXE with Resource ID #201, do this:

```
DisplayIcon "201"
```

Using the Resource ID number is sufficient if the resource was embedded in your program's EXE file. If the resource is embedded in a DLL, you will need to obtain the handle of the DLL (by using the LoadLibrary API function) and pass that value to Graphics Tools this way:

```
GfxOption GFX_MODULE_INSTANCE, hLibrary
```

After you do that, Graphics Tools will look in the specified DLL for all numbered resources, until you tell it to use another module. To return to using resources in the EXE, set the `GFX_MODULE_INSTANCE` option to zero (0).

Three-Dimensional Figures

Computer display screens are, of course, effectively two dimensional. But properly drawn figures can create the *illusion* of depth and provide a realistic three-dimensional effect.

Please note that Graphics Tools uses certain specific terms to describe generic three-dimensional figures. For example, the term "cube" is used to describe any cube-like or box-like figure. The use of the word "cube" is not meant to imply that the figure has three equal dimensions, as with the strict definition of "cube". Graphics Tools cubes can have any combination of height, width, and depth measurements.

You'll probably also notice that this document always refers to "figures" instead of "objects". The term "object" can imply that something is separate from the overall drawing and can be manipulated separately. Graphics Tools three-dimensional figures are just like everything else that Graphics Tools draws: when they are drawn, they become part of the graphics window.

The phrase "three-dimensional object" also implies (to some people) that a figure can be rotated in three dimensions, and that is not the case with Graphics Tools figures. A "wedge" (a three-dimensional pie slice) for example, can be drawn only in the horizontal and vertical orientations. Angled rotation is not supported, primarily because it is not very useful and it would greatly complicate the drawing process, slowing it down.

Cubes

The DrawCube function can be used to draw figures that resemble three-dimensional cubes or boxes. Cubes are very useful for things like three-dimensional bar charts.

You can specify the height and width of the cube, which correspond to the dimensions of the front of the cube, i.e. the side of the cube that appears to be closest to the viewer. You can also specify the depth of the cube -- the apparent distance between the front and the back -- and the angle at which the "depth" should be drawn. Most boxes are drawn with 45- or 135-degree angles because those values produce the most realistic effects, but any angle can be used. Cubes that are drawn with angles greater than 180 degrees have visible bottoms instead of tops.

The cube's depth measurement is automatically scaled by Graphics Tools to provide a visually more accurate drawing. For example, if you use DrawCube to draw a cube that is 100 x 100 x 100 it will appear to the eye to be a true cube, but if you measure the depth with a ruler, you will see that it is smaller than the width and height. This is done to provide an easy-to-use system that gives the illusion of normal, physical, three-dimensional objects. Graphics Tools uses a default scaling factor of 62%, which gives good results under a variety of circumstances. You can use the GfxOption `GFX_CUBE_DEPTH_SCALING` option to specify any value between 1% and 100%. To use other scaling factors, use 100% and have your program perform the scaling of the values using program code.

Cylinders

The DrawCylinder function can be used to draw figures that resemble cylinders and disks. These too are very useful for creating three-dimensional bar charts.

You start by specifying the height and width of the rectangle that defines the round end of the cylinder. This effectively defines the visual angle at which the cylinder will appear to be

viewed. If you specify a rectangle that is twice as wide as it is tall, for example, the end of the cylinder will be a 2:1 ellipse and a fairly normal-looking cylinder will be drawn. For best result we recommend using a ratio between 2:1 and 3:1. If you use a rectangle that is too square, the top of the cylinder will resemble a circle and the results will be unrealistic. If you use a rectangle that is not tall enough you will create a cylinder that is viewed from the side, and the top will not be visible. In other words you will draw a plain flat rectangle, not a cylinder with a three-dimensional appearance.

You can also specify the depth of the cylinder, i.e. the distance between the two round ends of the cylinder. If a cylinder is drawn with a vertical orientation, depth refers to the height of the cylinder from top to bottom. If a cylinder is drawn horizontally, depth refers to the length of the cylinder from left to right.

Tubes

The Graphics Tools Pro DrawTube function works very much like the DrawCylinder function, except that no "flat top" is drawn on the figure. Instead, Graphics Tools draws the "inside" of the tube, giving the illusion that you can see inside the tube through its open end. (This effect is most realistic when a gradient is used. See below.)

Wedges

The DrawWedge function can be used to draw figures we call "wedges" that resemble a three-dimensional slice of pie. They are most commonly used for three-dimensional pie charts.

Wedges are somewhat more complex than cubes or cylinders, because in order to look realistic they must often "interact" visually with other wedges. For example, if you are creating a bar chart with DrawCube you don't usually have to worry about the relative positions of the bars, you simply draw them next to each other, with no overlap. But if you are creating a three-dimensional pie chart you will usually want to draw two or more wedges arranged in a circle, to create a complete "pie".

For this reason, wedges (and two-dimensional DrawPie figures for that matter) are defined by the rectangle that surrounds *the entire pie*, not just one wedge. You specify a rectangle that defines how a complete pie would be drawn, and then you specify which "slice" should be drawn by specifying the start- and end-angles of the wedge. In this way, you can use the same rectangle over and over, and define different angles to define different wedges, and the end result will be a pie that "makes sense" visually.

We recommend that you use a rectangle with a width:height ratio that is somewhere in the 2:1 or 3:1 range, for the most realistic-looking wedges. Please note that Windows Platform 2 (95/98/ME) sometimes "flattens" the edges of wedges that are drawn with low ratio rectangles. This effect can be minimized by using larger ratios, i.e. taller rectangles.

Exploded Pies

Graphics Tools also supports a process called "exploding" a pie or wedge chart. This involves moving all of the slices away from the center of the overall pie, at appropriate angles.

You can adjust the size of the "pie explosion" for pie and wedge charts by using the GfxOption `GFX_PIE_EXPLOSION_DISTANCE` option. IMPORTANT NOTE: This is a global setting that affects the drawing of *all* pies and wedges. It cannot be set window-by-window.

"Spheres" and Other Figures

Graphics Tools does not provide a "DrawSphere" function, but it is possible to use the DrawCircle function with a Type 3 "elliptical" gradient fill to produce surprisingly realistic-looking spheres. (Gradients are available only to Graphics Tools Pro users.)

It is also possible to create some very interesting figures like the "3D doughnut" chart shown in `\GfxTools\Images\Donut.bmp`. That image was created with the DrawPie function (not DrawWedge) and an appropriate gradient.

Drawing 3D Figures with the Current Pen and Brush

By default, as with *all* Graphics Tools figures, three-dimensional figures are drawn with the current Pen and Brush. But to improve the visual impression of depth, Graphics Tools automatically "highlights" and "shades" the Pen and Brush when drawing cubes, cylinders, tubes, and wedges.

For example, when the DrawCube function is used to draw a cube, Graphics Tools automatically shades the right side of the cube to make it look like it is slightly "shadowed". (The bottom of a cube, if it is visible, will also be shaded.) The amount of shading can be controlled with the GfxOption `GFX_CUBE_SHADING_LEVEL` option. Cylinders, tubes, and wedges are also shaded automatically, and have options that control their shading: `GFX_CYLINDER_SHADING_LEVEL`, `GFX_TUBE_SHADING_LEVEL`, and `GFX_WEDGE_SHADING_LEVEL`.

If you use a Clear Pen (see PenStyle) when drawing a three-dimensional figure, Graphics Tools will automatically use a *non-clear* Pen that is a "highlighted" (slightly brighter) version of the current Brush. For example, if you use a red Brush and a clear Pen to draw a cube, the edges of the cube will be drawn with a Pen that is a slightly brighter red than the Brush. This results in a pleasing "highlighting" effect that enhances many figures. The amount of highlighting can be controlled with the GfxOption `GFX_3D_HIGHLIGHT` option. If you use a highlight option of zero (0) the Brush color will be used for the Pen, resulting in the appearance that a Clear Pen was actually used, but this almost always results in unrealistic three-dimensional figures.

Tip: Highlighting works best when you choose Brush colors that are not fully saturated. For example, if you use a Brush Color of pure red (`h0000FF`) the highlight will be less pleasing than if you use a very-slightly less red color, such as `h0000F8`. The Brush color will appear to the eye to be virtually the same, but Graphics Tools will be able to create a more pleasing highlight color. To create non-saturated colors easily, use the HSW function with an S (saturation) value less than 255. We have found that using 248 produces very good results if you want vivid colors.

By the way, the current Pen Width (see PenWidth) is *ignored* when Graphics Tools draws a three-dimensional figure. A one-pixel pen is always used, because that provides the most realistic three-dimensional effects. It is not necessary for your programs to select a one-pixel pen before drawing a 3D figure. Graphics Tools handles it automatically.

Using "Cap Colors"

If you want to create, for example, a red cylinder with a blue round end, it is easy to do.

Simply use the GfxOption GFX_CYLINDER_CAP_COLOR option to specify the color that should be used for the "cap".

The color of the pie-shaped face of a Wedge can be controlled with the GFX_WEDGE_CAP_COLOR option.

The color of the top of a cube can be controlled with the GFX_CUBE_CAP_COLOR option. You can also use the GFX_CUBE_FRONT_COLOR and GFX_CUBE_SIDE_COLOR options to specify colors for the other faces of the cube. If you use all three, the current Brush will be ignored when a cube is drawn.

If you specify a Cap Color for a surface and then want to resume using the default color (i.e. the Brush color or a Gradient) simply set the appropriate option to GFX_AUTO instead of a color.

Drawing Three Dimensional Figures using Gradients

Some of the most impressive figures that Graphics Tools can create are three-dimensional figures that are drawn with gradients. Gradients can provide both bold and subtle shading effects that can greatly enhance the visual appeal of 3D figures.

For example, \GfxTools\Images\GradWedges.bmp contains an image of a "wedge chart" that is very realistic.

Please note: Gradient-filled figures take longer for Graphics Tools to draw than brush-filled figures. Please see Limitations Of Gradients for more information.

Drawing With Text

When Windows draws angled and curved lines, it sometimes creates "jaggies", i.e. lines with jagged edges (single-pixel imperfections) that can vary from barely noticeable to objectionable. The effect will vary depending on many different factors such as the colors that are being used and the angles of the lines. *This is a side-effect of the fact that Microsoft Windows is a digital system, not an analog system, and all Windows drawing programs experience these effects.*

Windows does provide *limited* support for something called "anti-aliasing" which is used to minimize these effects. Pixels of certain colors are automatically added to curved and angled edges, making them appear smoother. For example, if you draw with black on a white background, pixels of various shades of gray will be added to minimize the jagged appearance of the edges. The result is a somewhat thicker line -- less-precise drawing -- but a more pleasing visual effect.

While Windows does not provide anti-aliasing for *most* drawing operations, it always uses it -- automatically -- when text is drawn. If you look very closely at a character that is drawn with DrawTextRow or DrawTextBox --especially a large, curved character -- you will see a few pixels that are a different color from the one you selected with GfxFontColor.

If your program draws small circles, diamonds, arrows, and other simple shapes you may not be completely happy with the result, because the one-pixel jagged edges will be relatively large compared to the size of the figure, and therefore more noticeable. You may be able to produce better results if you "draw with text" to take advantage of Windows anti-aliasing feature.

For example, the Wingdings font contains many useful "drawing" characters such as circles, squares, and arrows. Depending on your system configuration, other useful fonts may also be present, such as Symbol and Wingdings 2 and 3. You can, of course, use very large width and height values when creating the fonts, to draw text-based figures of any size.

Keep in mind that text can be drawn at different angles by using the GfxFontAngle function, so it is even possible to draw anti-aliased straight lines at various angles by choosing an appropriate font and character.

Using Clip Areas

Please note that Clip Areas are available only when Graphics Tools Pro is used. Graphics Tools Standard does not support Clip Areas.

Clip Areas can be used to restrict the portion(s) of the graphics window where drawing operations take place.

For example, if you set up a Clip Area that encloses just the right half of a graphics window, and then draw a circle that is centered in the window, only the right half of the circle will be drawn.

If a Clip Area has been defined for a graphics window, *any* drawing operations that take place outside the area will be discarded.

To create a Clip Area, use the DrawFrom function to define its starting point. Then use the GfxClipArea function like this:

```
GfxClipArea AREA_DEFINE
```

Then use one or more of the following functions to draw the outline of the desired Clip Area.

DrawTo
DrawMultiLine
DrawRect
DrawCircle
DrawEllipse
DrawEllipseRotated
DrawPolygon
DrawXagon
DrawPgram
DrawBezier

It is important to note that the DrawFrom function is *not* on that list. You must set the starting point of the Clip Area *before* you use GfxClipArea `AREA_DEFINE`, and you may not use it again until after the area has been fully defined.

After you have drawn the outline of the clip area using those functions, add this line of code to your program:

```
GfxClipArea AREA_USE
```

That tells Graphics Tools to begin using the Clip Area that you have defined.

Note that the list above shows the functions that can be used to *create* a Clip Area. After a Clip Area has been created, you may use *any* Graphics Tools drawing function and it will respect the Clip Area.

If the lines that you draw to create a Clip Area cross each other, the final Clip Area will be determined by the setting of the PolyFillMode function. For example, if you use the DrawXagon function to draw a five-pointed star, the PolyFillMode function will determine whether 1) the Clip Area will be the entire star, or 2) the center pentagon (where the lines cross) will be excluded.

To stop using a Clip Area, use...

```
GfxClipArea AREA_RESET
```

Using AREA_RESET tells Graphics Tools to resume drawing in the entire graphics window, and to discard the current Clip Area.

Complex Clip Areas

If you want to create and combine two or more Clip Areas in the same graphics window, create the first one as shown above, using AREA_USE. Then repeat the process for the second area, but instead of AREA_USE you should use AREA_ADD. This tells Graphics Tools to add the second area to the existing Clip Area. Two or more simultaneous Clip Areas can be created in this way.

If you want to create two Clip Areas and tell Graphics Tools to draw only in the area where the two Clip Areas overlap, finish the first one with AREA_USE and the second with AREA_OVERLAP.

If you want to create two Clip Areas and tell Graphics Tools to draw only in the areas that do *not* overlap, use AREA_XOR to finish the second area.

If you want to create two Clip Areas and tell Graphics Tools to draw only in the area that is included in the first area but not in the second, use AREA_DIFFERENCE to finish the second area.

The various AREA_ functions can be used in any combination. For example, you can create an area with AREA_USE, then add a second area with AREA_ADD, then use AREA_ADD again to add a third, then use AREA_DIFFERENCE to overlay a fourth area, and so on. Once two areas have been combined in some way, they are considered to be the "current" area, and future operations treat them as a single unit.

In all cases, AREA_RESET is used to tell Graphics Tools to resume drawing in the entire graphics window, and to discard the Clip Area.

You can also use AREA_INACTIVE to tell Graphics Tools to resume drawing in the entire graphics window, but to *save* the Clip Area. You can then begin using the saved clip area again by using AREA_ACTIVE.

Creating "Holes" in the Display

Graphics Tools is capable of creating "holes" in the default graphics window. (Graphics Windows other than window zero are not allowed to have holes.) The graphics window's parent window will be visible through the holes, resulting in an area that appears to be "transparent". If you attempt to draw something in a hole, the results will not be visible.

The DrawHole function is used to create holes. (If you're using Console Tools Plus Graphics, the GfxTextHole function lets you create holes that correspond to console rows and columns, so the results of PB/CC `PRINT` statements can be seen though the holes.) The FillHole function can be used to eliminate a hole that was created with DrawHole or GfxTextHole.

Whenever possible, you should use the DrawHole (or GfxTextHole) function immediately *after* the graphics window has been created, but *before* you make it visible with the GfxWindow `GFX_SHOW` function. That's because showing the graphics window will cause the parent window to be covered up. A hole will not be immediately visible if it is created when the graphics window is already showing. The parent window will only become visible through the new hole if the parent window is refreshed in some way.

For example, let's assume that you are using Console Tools Plus Graphics to display a graphics window on top of a console application. (If you're using Graphics Tools to display a graphics window in a GUI-type program, these same basic ideas apply.) If you create and show a graphics window, the console window (the parent window) will be covered up. If you then create a hole in the graphics window, Graphics Tools will not perform any *further* drawing operations in the area that you specify. But the console window that is underneath the graphics window will not become visible until something causes the console window to be refreshed. This can be accomplished in many different ways, including **1)** using the PB/CC `PRINT` statement to display some text (or even some spaces) in the hole, or **2)** hiding and then showing the graphics window, or **3)** hiding and then showing the console window, or **4)** minimizing and then restoring the application, or **5)** ...anything else that causes the application's parent window and the graphics window to be redisplayed.

Please Note...

If you use holes, you will need to be especially careful to avoid refreshing the graphics window in a way that temporarily "covers up" the holes.

For example, if you perform a large number of rapid drawing operations, Windows may not refresh the graphics window properly and your holes may be covered up. If you minimize your application and then restore it (or hide it and then show it again), the hole will reappear. That proves that the hole is not really "filled in" as it would be if you used the FillHole function. It is simply "covered up" by Windows, so that it is *temporarily* disabled.

(Console Tools Plus Graphics users please note: You can also "uncover" a hole by using the PB/CC `PRINT` statement to re-display the information that is in the hole.)

If you follow the guidelines provided in Refreshing the Display, with particular attention to the GfxWindow `GFX_FREEZE/GFX_UNFREEZE` technique, you shouldn't have any problems with covered-up holes.

Saving a Graphics Window With Holes

It is not possible to "save the holes" when using the SaveGfxWindow or SaveGfxArea function. If you use one of those functions to create a bitmap file, and then view the file, the holes will be missing and the entire graphics window will be visible.

The Graphics Window and the Mouse

This section of this document applies only to Console Tools Plus Graphics *programmers who are using the PowerBASIC PB/CC compiler. All other programmers should read* Graphics Window Messages *instead of this section.*

At some point you will probably need to detect that your program's user has clicked on the graphics window with the mouse, or that the mouse has been moved.

When Console Tools Plus Graphics detects that the graphics window has been clicked, it first determines the exact screen location that was clicked. It then calculates the console location (row/column) that corresponds to the screen location, and sends a message about the event to PB/CC. The PB/CC input functions (`INSTAT`, `INKEY$`, etc.) can then detect that message and produce values that are very similar to the ones that are returned when a user clicks the console directly, rather than the graphics window.

The following graphics window mouse events can be detected by Console Tools Plus Graphics:

- Left Button Down
- Left Button Up
- Left Button Double Click
- Right Button Down
- Right Button Up
- Right Button Double Click
- Mouse Move

Because of a limitation of Microsoft Windows, the middle mouse button is not supported by Console Tools Plus Graphics. (Middle button events *may* be detectable on some Windows 95, 98, and ME computers, but we recommend against relying on them.)

Your PB/CC program should detect graphics window mouse events using exactly the same techniques (`MOUSE ON`, `INSTAT`, `INKEY$`, etc.) that it uses to detect console window mouse events. For more information about these functions, please consult the PowerBASIC documentation.

After your program has detected a mouse event, it can use the MouseOverX and MouseOverY functions to determine the exact X and Y locations of the mouse cursor, and use those values for drawing operations.

You should keep in mind that the PB/CC "keyboard buffer" stores mouse events in the same way that it stores keypresses. For example, if you type "ABCD" while your program is busy, and a few seconds later the program uses `INKEY$` to find out which keys were pressed, all four keypresses would be reported. Similarly, if a user clicks several different screen locations and your program does not use `INKEY$` right away, all of the events will be reported at a later time. And the PowerBASIC `MOUSEX` and `MOUSEY` functions will accurately report the rows and columns that were clicked earlier. But the MouseOverX and MouseOverY functions report only the *current* location of the mouse cursor, not "where it was when the last click was detected".

Here is a specific example. Let's say that your program is busy doing something and not monitoring `INKEY$` for mouse events. If somebody clicks on Row 1, Column 1, and then clicks on Row 10, Column 1, those events would be stored by PB/CC. If your program then examines those events with `INKEY$`, `MOUSEX`, and `MOUSEY`, it could determine that Row 1, Column 1 and Row 10, Column 1 were clicked. But the MouseOverX and MouseOverY

functions would only return the *current* location of the mouse cursor, not "where the cursor was when the user clicked the graphics window the first time".

Whenever possible, if your PB/CC program needs to watch for mouse events it should do so in "real time". During the times that your program is too busy to check for mouse events -- for even a few seconds -- it should use the `MOUSE OFF` function to temporarily disable the detection of mouse events.

Graphics Tools and PB/CC (*without* Console Tools)

It is not possible for PB/CC programs to detect graphics window mouse events without using Console Tools Plus Graphics. The Graphics Tools Runtime Files will attempt to send a message to the console window, but without Console Tools the console window will not recognize the message, and it will be ignored.

Graphics Window Messages

PB/CC PROGRAMMERS: This section of this document does not apply to Console Tools Plus Graphics. Console Windows cannot process normal window messages, so PB/CC programs cannot use this feature of Graphics Tools. Console Tools Plus Graphics *user should read* The Graphics Window And The Mouse*)*

VISUAL BASIC PROGRAMMERS: Most of this section does not apply to Visual Basic. VB programs handle Graphics Window Messages with standard-format Events. VB programmers may read this section for background, but you can safely skip it.

A Graphics Tools graphics window can provide information about many different "events" that take place in the window. This is done through standard Windows Messages that are sent to the graphics window's parent window.

The rest of this section will assume that you are familiar with "callback functions" and the *basic* techniques of processing window messages. If you are not, we recommend that you acquire a good book about the way the Windows 32-bit GUI system works, and return to this Graphics Tools topic when you are comfortable with those core concepts. (PowerBASIC programmers who use the DDT or SDK system and Visual Basic programmers who are comfortable with events like Click should have no problem with this topic.)

PLEASE NOTE: This section refers to a large number of standard windows constants, such as the values below that start with `WM_`, `VK_`, and `SC_`. In order for your program to understand those values you may need to perform an extra step. For example, PowerBASIC users will usually need to add the `WIN32API.INC` file to their program. (Visual Basic programmers may need to add the `\VB\WIN32API.TXT` file to their project, but this is usually not necessary.)

Graphics Window Events

When certain events take place in a graphics window, Graphics Tools can react to them. In some cases, Graphics Tools can even tell your program "X is about to happen... Do you want to allow it?"

Graphics window events include

- Mouse movement and button-clicks
- Keyboard key-presses
- Graphics window scrolling operations
- Receipt or loss of focus
- Changes in the graphics window size and location
- Files being dragged from other programs and dropped on the graphics window
- `WM_NOTIFYPARENT` messages

As you probably know, each Window message has been assigned a number by Microsoft. The range of numbers from zero (0) to 1,023 is used by Windows itself. The range of numbers from 1,024 to 32,767 (h400-h7FFF) is called the `WM_USER` range and it is available for modules like Graphics Tools to send its own messages.

To help avoid conflicts with other modules that may use `WM_USER` messages, Graphics Tools window messages start with number 4,096 (h1000). This value is called `WM_GFX_EVENT`. If you find that Graphics Tools window messages conflict with another

module, you can change the value of WM_GFX_EVENT and thereby change the window message numbers that Graphics Tools uses. To do that, use GfxOption GFX_EVENT_MESSAGE. The rest of this section will assume that you have *not* done that. If you *have*, you will need to use your own value wherever you see WM_GFX_EVENT below.

Graphics Window Messages

All Graphics Tools window messages are based on existing Windows window message, but they are not identical to them. You should be careful to note the differences.

For example, let's examine the WM_DROPFILES message that is used by Windows. When a user drags a file (usually from another application) and drops it on a window, Windows generates a WM_DROPFILES message to alert the program that a drop has taken place. The program must then decode a complex message and use a number of API function to find the name(s) of the file(s) that were dropped.

When a file is dropped on a Graphics Tools graphics window that was created with the GFX_STYLE_DROP_FILES style, this message number is generated:

```
WM_GFX_EVENT + WM_DROPFILES
```

When your program's callback function detects that event, it will know that a file has been dropped on a graphics window. It should then use the GfxDroppedFiles function to obtain the file name(s). Graphics Tools takes care of all the messy API details for you.

Note that the graphics window message is *similar but not identical* to the original Windows message. A complete list of the supported messages and their differences is provided below.

Your program's callback function can examine the "details" of the window message to obtain other information about the event. For example, if a program has two or more graphics windows it can examine the high word of *wParam* to get the number of the graphics window where the file-drop took place. See High Words and Low Words for more information.

Tip: PowerBASIC programmers can use the built-in HIWRD and LOWRD functions to obtain high- and low-word values. Also, PowerBASIC DDT users should substitute CBLPARAM and CBWPARAM wherever you see *lParam* and *wParam* in this section.

Here's another example of a window message. If you create a graphics window with the GFX_STYLE_DRAG_MOVE style, the user will be able to click on the graphics window's caption bar and drag the window to a new location. Whenever that happens, this message will be generated:

```
WM_GFX_EVENT + WM_WINDOWPOSCHANGING
```

Graphics Tools will send that message to your program *before* the move actually takes place. If your program responds to the message by using the GfxResponse function it can "refuse" the operation and prevent the window from moving. You might do this selectively, for example, if you want to keep the graphics window from being moved too far in any given direction.

Mouse Events

The following messages are sent when the user manipulates the mouse:

```
WM_GFX_EVENT + WM_MOUSEMOVE
WM_GFX_EVENT + WM_LBUTTONDOWN
WM_GFX_EVENT + WM_LBUTTONUP
WM_GFX_EVENT + WM_LBUTTONDBLCLK
WM_GFX_EVENT + WM_RBUTTONDOWN
WM_GFX_EVENT + WM_RBUTTONUP
WM_GFX_EVENT + WM_RBUTTONDBLCLK
WM_GFX_EVENT + WM_MBUTTONDOWN
WM_GFX_EVENT + WM_MBUTTONUP
WM_GFX_EVENT + WM_MBUTTONDBLCLK
WM_GFX_EVENT + WM_MOUSEWHEEL
```

The high word of the message's *wParam* value will contain the graphics window number which generated the message.

The low word of *wParam* will contain bit-flags (`MK_SHIFT` and/or `MK_CONTROL`) indicating whether or not a Shift or Ctrl key was being held down when the mouse event occurred. If the message is `WM_MOUSEMOVE` these additional flags may be present, indicating that one or more mouse buttons were being held down while the mouse cursor was being moved: `MK_LBUTTON`, `MK_RBUTTON`, `MK_MBUTTON`, `MK_XBUTTON1`, and `MK_XBUTTON2`.

Except for `WM_MOUSEWHEEL` messages (see below) the low word of *lParam* will always contain the horizontal position of the mouse cursor at the time of the event. The high word of *lParam* will contain the vertical position. Note that the mouse position is given in Drawing Units, not pixels. If an `WM_LBUTTONDOWN` or `WM_LBUTTONUP` message has an *lParam* value of `&hFFFF` it means that the user clicked the graphics window's caption, not the drawing area.

Note also that the Microsoft system of using the high and low words of a value effectively limits the possible mouse locations to a range of 65,535 drawing units. If you use a graphics window world with scales that are larger than that, the *lParam* value will not contain useful information and you will need to use the MouseOverX and MouseOverY functions to obtain the mouse position.

Note also that under most circumstances, mouse events will be generated only for the "drawing" area of a graphics window. If you create a graphics window with a caption, for example, no mouse-move messages will be generated if the mouse cursor is moved over the caption or border. There are two exceptions to this rule: single- and double-clicks. Double-clicks on a caption are always sent to the parent window, and single-clicks are sent unless the window is movable. (In the case of movable windows, a single-click means that a drag-move operation is beginning.)

Using GfxResponse with mouse-event messages has no effect. There is no way to "refuse" a mouse click.

WM_MOUSEWHEEL Events

The `WM_GFX_EVENT + WM_MOUSEWHEEL` event is slightly different from all of the other mouse events. Instead of containing the mouse cursor location, the *lParam* value will contain a value which indicates the "direction and distance" that the mouse wheel was moved. If you need the mouse cursor location as well, you should use the MouseOverX and MouseOverY functions.

Keyboard Events

Windows normally generates several messages for each key that is pressed. For the sake of simplicity Graphics Tools supports these two:

```
WM_GFX_EVENT + WM_KEYDOWN
WM_GFX_EVENT + WM_CHAR
```

The high word of the message's *wParam* parameter will contain the graphics window number that generated the message, i.e. the graphics window that had the keyboard focus when the key was pressed.

The low word of *wParam* will contain a number that corresponds to the key that was pressed. These values correspond to the standard Windows constants that start with `VK_`, such as `VK_ESCAPE`, `VK_A`, `VK_B`, `VK_C`, etc.

The *lParam* value will contain a bit-masked value that contains additional information about the keypress. It is identical to the information that Windows provides in a `WM_KEYDOWN` message. Please consult the Windows API documentation for details. (This information is rarely useful.)

If your program responds to a `WM_KEYDOWN` or `WM_CHAR` message with GfxResponse, that tells Graphics Tools "*I have handled this keypress and you should not act upon it*". For example, you could tell a graphics window to ignore Page-Down and Page-Up keypresses. Then, instead of scrolling the graphics window when those keys were pressed, your program could perform some other action.

Graphics Window Scrolling

Whenever a graphics window is scrolled, this message will be generated:

```
WM_GFX_EVENT + WM_SYSCOMMAND
```

The high word of the message's *wParam* parameter will contain the graphics window number that generated the message, i.e. the graphics window that is being scrolled.

The low word of wParam will contain either `SC_VSCROLL` or `SC_HSCROLL`, to indicate the type of scrolling (Vertical or Horizontal) that has taken place.

The value of *lParam* will always be zero.

Using GfxResponse with this messages has no effect.

Focus Events

If a graphics window was created with the `GFX_STYLE_TABSTOP` style it can receive and lose the windows "focus". When this happens, these messages are sent:

```
WM_GFX_EVENT + WM_SETFOCUS
WM_GFX_EVENT + WM_KILLFOCUS
```

The high word of the message's *wParam* parameter will contain the graphics window number

that generated the message, i.e. the graphics window that is getting or losing the focus.

The low word of *wParam* and the value of *lParam* will always be zero.

Using GfxResponse with these messages will have no effect.

It is important to note that these message are sent *before* the actual focus-change takes place. This is to allow your program to change the focus rectangle colors so that (for example) each graphics window could have a different color focus rectangle. See GfxOption GFX_FOCUS_RECT_COLOR and GFX_NO_FOCUS_RECT_COLOR for more information.

Dropped Files

When a file is dropped on a Graphics Tools graphics window that was created with the GFX_STYLE_DROP_FILES style, this message number is generated:

```
WM_GFX_EVENT + WM_DROPFILES
```

The high word of the message's *wParam* parameter will contain the graphics window number that generated the message, i.e. the graphics window where the files were dropped.

When your program's callback function detects this event, it should use the GfxDroppedFiles function to obtain the name(s) of the file(s) that were dropped.

Using GfxResponse with this messages has no effect.

Window Location and Size

Both location *and* size changes cause this window message to be generated:

```
WM_GFX_EVENT + WM_WINDOWPOSCHANGING
```

Microsoft calls both location *and* size changes "position" changes.

This message will be generated only if a graphics window was created with the GFX_STYLE_MOVABLE and/or GFX_STYLE_SIZABLE style, or if those styles have been added with the GfxWindow function.

The *lParam* and *wParam* values that are associated with a normal WM_WINDOWPOSCHANGING. message are relatively complex (they involves pointers) so Graphics Tools uses a simplified system.

The high word of the message's *wParam* parameter will contain the graphics window number that generated the message, i.e. the graphics window that is being moved or sized.

The low word of *wParam* will contain either WM_SIZE or WM_MOVE, depending on the operation that is being performed. Keep in mind that certain operation such as dragging the left or top edge of a graphics window can produce changes in both the size *and* location of the window. If that happens, two different WM_WINDOWPOSCHANGING messages will be sent.

If the window is being moved, the low word of the *lParam* value will contain the proposed new horizontal size. If the window is being resized, the low word of *lParam* will contain the proposed new horizontal location. The high word of *lParam* will contain either the proposed vertical size or location.

Note that all of these values are given in *pixels*, not Drawing Units. (Drawing Units are used only *inside* a graphics window.) Your program may need to convert the pixel values into Dialog Units in order to use them effectively.

If your program responds to a `WM_WINDOWPOSCHANGING` message with GfxResponse, it tells Graphics Tools "do not allow this change in the window position or size to take place".

The following message are generated *after* a location or resize operation has been completed:

```
WM_GFX_EVENT + WM_SIZE
WM_GFX_EVENT + WM_MOVE
WM_GFX_EVENT + WM_EXITSIZEMOVE
WM_GFX_EVENT + WM_WINDOWPOSCHANGED
```

In all cases, the high word of the message's *wParam* parameter will contain the graphics window number that generated the message, i.e. the graphics window that was moved or sized. The low word of *wParam* will always be zero (0).

In the case of `WM_EXITSIZEMOVE` the value of *lParam* will also be zero.

For `WM_MOVE` the low word of *lParam* will contain the new horizontal location of the window, and the high word will contain the new vertical location.

For `WM_SIZE` the low word of *lParam* will contain the new horizontal size of the window, and the high word will contain the new vertical size.

For `WM_WINDOWPOSCHANGED` (not `CHANGING`) *lParam* will contain a pointer to a Windows `WINDOWPOS` structure, describing the window's new size and location. Refer to the Win32 API documentation for more information about this structure.

WM_PARENTNOTIFY Messages

This is the only Window Message that was supported by Graphics Tools Version 1. It is supported by Graphics Version 2 primarily for backward compatibility. It is much more limited than other Window Messages, and we recommend that you use other messages if possible.

It is important to note that `WM_PARENTNOTIFY` messages are managed by Windows itself, not by Graphics Tools, so the usual system of identifying the graphics window (see above) is not used.

When the following events take place, a `WM_PARENTNOTIFY` message will be sent to the parent window of a graphics window. The Low Word of the message's wParam parameter will contain the appropriate `WM_` value:

- Graphics Window Created (`WM_CREATE`)
- Graphics Window Destroyed (`WM_DESTROY`)
- Left Mouse Button Down (`WM_LBUTTONDOWN`)
- Middle Mouse Button Down (`WM_MBUTTONDOWN`)
- Right Mouse Button Down (`WM_RBUTTONDOWN`)

The High Word of the message's wParam parameter will contain the "identifier" of the graphics window. This is *not* the same as the Graphics Tools window number, it is the numeric identifier that was assigned to the window when it was created. (This is usually 5000

greater than the window number, but the offset can be changed with GfxOption `GFX_WINDOW_NUMBER_OFFSET`.)

If the graphics window is being created or destroyed, the message's lParam parameter will contain the handle of the graphics window. Otherwise, lParam will contain the coordinates of the mouse cursor, in pixels, at the time of the click. The x-coordinate will be in the Low Word and the y-coordinate will be in the High Word.

Sample Program

Handling Window Messages

Printing Graphics Tools Images

Please note that printing is available only when Graphics Tools Pro is used. Graphics Tools Standard does not support printing.

For the person who is using a computer program, the process of printing *anything* in Windows is usually a multi-step process. The same is true for printing a Graphics Tools image.

For example, in addition to a simple "Print" button or menu item, your program will probably need to provide a Page Setup dialog and a Print Setup dialog so the user can select a printer, the number of copies to be printed, and so on. Don't worry, Graphics Tools provides all of the dialogs you'll need.

In most cases, you will want to provide these basic functions:

1) Identify the portion of the graphics window that will be printed. Your program could allow the user to mouse-select a rectangular area of the graphics window (the DrawFocus function is useful for this) or you may simply provide a Print button that prints the entire image.

2) *Optional:* You may wish to pre-set certain options by using the GfxPrintParam function. For example, you may want to pre-select the landscape orientation.

3) Allow the user to access the Page Setup dialog to set the paper size and source, the portrait or landscape orientation, and the margins. The GfxPrintPageSetup function provides all of those options. It also displays a "thumbnail" of the graphics image that represents the current settings, based on the portion of the image that was selected in step 1. The Page Setup dialog's "Printer" button displays a dialog that is similar to the one provided in step 4...

4) Display the Print Setup dialog using the GfxPrintSetup function. This allows the user to select a printer and the number of copies that should be printed. By selecting the Print Setup dialog's Properties button, the user can access a large number of other settings, including some of those that can be set in step 3.

5) *Optional:* After the user has made their selections, your program can force the use of certain printer settings by using the GfxPrintParam function. For example, if you may wish to override the selection of the landscape orientation by forcing the use of the portrait mode.

6) Actually print the image, using the GfxPrintWindow or GfxPrintArea function.

7) Optional: You may also wish to provide a function that resets the printer settings to their default values, effectively un-doing any changes that the user has made. This can be done with the GfxPrintDefaults function.

As you can see, the various steps can interact quite a bit. Graphics Tools simply keeps track of the last settings that were selected, and it uses those settings when GfxPrintWindow or GfxPrintArea is finally used to print an image. You do not have to use the functions in any particular order, except of course that the first step must be selecting the portion of the image to print. In fact it is not absolutely necessary to display any dialogs at all.

The actual implementation is up to you, based on the needs of your program. For example, when the user wants to print something, you might display the Print Setup dialog. Then, unless the user selects the dialog's Cancel button, your program would proceed automatically and use the GfxPrintWindow function to print the image.

Many programs allow the user to access the Page Setup menu via a pulldown menu item, but

do not automatically display it. But actually, Microsoft now recommends that you display the Page Setup dialog *instead* of the Print Setup dialog whenever the user wants to print something. It really doesn't matter, because all printing functions can be accessed from either dialog. However the Page Setup dialog is more attractive and impressive than the Print Setup dialog, so we (like Microsoft) recommend that you normally display that dialog.

Slow Display Of Printing Dialogs

The first time that either GfxPrintSetup or GfxPrintPageSetup is used, Graphics Tools must initialize the printing subsystem, and that can take a few seconds. To speed up the initial display you can pre-initialize the system by using GfxPrintDefaults in your program's startup code. If you do that, you will find that the GfxPrintSetup and GfxPrintPageSetup dialogs will display roughly as fast as they do with other Windows applications which use the same standard dialogs.

Rough Edges

Modern printers are capable of reproducing very finely detailed images. For example, a 600-dot-per-inch printer can reproduce over 30 million dots (pixels) on a single page. Compare that to the number of pixels on a typical computer screen (1024x768 = 786,432) and you'll understand why, when a screen image is "blown up" to fill a piece of paper, you will often see rough edges. This is especially true when images with curved and angle lines are printed.

For "perfect" image printing you can use this line of code:

```
GfxPrintParam PRINTER_SETUP_OPTIONS, PRINT_ACTUAL_SIZE
```

That tells Graphics Tools to print the image pixel-for-pixel, without stretching it.

Slow Printing

When a word processor prints a document, the process of preparing the page is relatively fast. Preparing a graphics image, however. can take significantly longer.

The largest single factor that impacts printing speed is the size of the printed image. For example, in order to print an 8-inch-by-8-inch image on a 600-dot-per-inch printer, Windows must create a 4800x4800-pixel temporary image. That's a *very large* image that must be processed, and it can require many megabytes of memory and/or disk space.

The actual printing speed will depend on many different factors including the size of the printed image, the size of the graphics window, the speed of the computer, the amount of available memory, the speed of the printer driver, and the number and types of other programs that are running.

It is not unusual for a full-page image on a slow computer to take 20-30 seconds to begin printing. During that time, while the image is being prepared, your program will be temporarily unable to do anything else.

So Graphics Tools can display a Progress Box that shows how the preparation is proceeding. If you use a non-empty string for the *sCaption$* parameter of the GfxPrintArea or GfxPrintWindow function, Graphics Tools will display a small dialog with a Progress Bar and a Cancel button. If the user selects Cancel before the preparation reaches the last step, no printing will take place.

Printers That Won't Print Images

If the GfxPrintWindow or GfxPrintArea function returns `ERROR_FIRST_PRINTER_ERROR + 800`, the computer does not have sufficient memory to print an image in the specified size. This should be relatively rare.

If `ERROR_FIRST_PRINTER_ERROR + 900` is returned it means that the target printer does not support the printing of images. You will find that if you attempt to use Microsoft Paint to print an image, it will also fail.

These problems can *sometimes* be remedied by freeing up some space on the Windows System hard drive, installing more memory, and/or installing an updated printer driver.

Getting Technical Support

Please note that Technical Support for all Perfect Sync products is provided by Perfect Sync, Inc. even if you obtained the product from one of our Authorized Distributors.

The following policies apply to Graphics Tools, Console Tools, SQL Tools, and the registered version of DOSBox. Perfect Sync does not provide technical support for our free and Unregistered Shareware products.

Perfect Sync reserves the right to change these Policies without notice. For the most recent version of this document, visit http://perfectsync.com/DevToolSupport.htm.

We have worked very hard to make sure that our Development Tool documentation contains everything that you'll need to know about using our tools. Before contacting Perfect Sync for help, please search the documentation for words and phrases that might be related to your question. (For example, when the documentation is presented as a Help File, use the Windows Help Contents, Index, and Find features. The printable PDF files can also be searched.) Most topics are covered twice: once in the User's Guide and once in the Reference Guide.

If you don't find an answer in the documentation, Perfect Sync provides free Technical Support via electronic mail to all developers who license our tools. Please send all pertinent technical questions to support@perfectsync.com. Be sure to include your name, an email address where we can send our response, and a detailed description of the problem. If possible please include sample source code for a small, compilable program that demonstrates the problem, If you are using SQL Tools, please include a Trace File showing the problem.

If you contact us and it turns out that the answer to your question is given in the documentation, that is probably the answer that you will receive: a polite suggestion that you read a particular section of the Help File. After all, an informal email message from our Tech Support department wouldn't be able to explain a topic nearly as thoroughly as the documentation. If you feel that the documentation does not explain a topic well enough, please cut and paste the unclear help text into your message, and ask a specific question. We'll be glad to try to clarify!

If the answer to your question is not covered in the documentation but does fall within the bounds that we have established, we will do our best to 1) answer your question quickly and completely via email and 2) add the answer to the next release of the documentation, 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.

In the end, a Development Tool function either works properly or it doesn't. If it doesn't work, or if the documentation is flawed, we will endeavor to provide a bug fix for the tool. If the tool is working properly and the documentation is accurate, that's where you, as a programmer, take over.

As the president of PowerBASIC, Inc. is fond of saying, "*When you buy a hammer it doesn't come with instructions for building a house*". Don't get us wrong: we will be very glad to help you learn to use our "toolkits"! But we can't possibly provide free training in the rest of the skills that you will need to complete a project, whether it's a birdhouse or a (data) warehouse.

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 Development Tool Licensee.)

Questions about the licensing and distribution of a Perfect Sync Development Tool runtime file (such as a DLL or EXE) and other components should be directed to sales@perfectsync.com.

Microsoft Visual Basic support is available from a wide variety of internet and phone-based sources.

For PowerBASIC questions, please contact support@powerbasic.com. PowerBASIC also sponsors several excellent peer-support forums on their "Web BBS" site at http://www.powerbasic.com/support/forums/Ultimate.cgi.

Conventions Used In the Reference Guide

Most of the text in this document will appear in a non-bold Arial font.

Important Warnings are shown in **bold red**. Less urgent warnings are shown in **bold dark red**.

Source code, numeric values, string values, and BASIC keywords are shown in the `Courier New` font. Source code "remarks" are shown in `this color`.

Constants -- words that represent *fixed* numeric values -- appear in `UPPER_CASE` letters, like `SUCCESS`, `GFX_SOLID`, and `GFX_SHOW`. (PowerBASIC users will need to add the `%` prefix to all constants in order to create PowerBASIC "equates", such as `%SUCCESS`, `%GFX_SOLID`, and `%GFX_SHOW`. Most example source code is shown without the prefix, to keep it consistent with the standard Microsoft notation.)

If this document is presented in electronic (Help File) form, Graphics Tools function names will often be highlighted like THIS to indicate that you can click on the highlighted word to jump to the function's Reference Guide entry.

Variable Naming Conventions

The BASIC variable names that are used in example code are completely optional.

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 PowerBASIC Long Integer. Other programmers, particularly those who program for 32-bit Windows, often use a convention 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 prefix standing for Long. (Hungarian notations vary. For example, some use `i` for Integer, others use `n`.)

For maximum readability by both groups of people, this document uses both prefixes and suffixes, so every variable you see will look like `lSomething&`. The following prefixes and suffixes are used in this document:

`lSomething&`	LONG (Signed) Integer
`dwSomething???`	DWORD (Unsigned) Integer
`qSomething&&`	QUAD integer
`sSomething$`	Dynamic String
`lpzSomething`	ASCIIZ string (no suffix defined)
`spSomething!`	Single precision floating point
`dpSomething#`	Double precision floating point
`epSomething##`	Extended precision floating point

AcceptDrop

Purpose

This Visual Basic Property specifies whether or not the graphics window will accept files that are dragged onto the window and dropped. (Non-VB programmers should use the `GFX_STYLE_DROP_FILES` flag that is described in Graphics Window Styles.)

Availability

Graphics Tools Pro Only (OCX version only)

Warning

None.

Remarks

This property can be set to True or False.

If it is set to True, the graphics window will generate a DropFiles Event when a file is dragged and dropped onto the window.

See Also

Using Graphics Tools With Visual Basic

AccessKeys

Purpose

This Visual Basic Property specifies the hot-key(s) that will cause the keyboard focus to be given to the graphics window. (Non-VB programmers should use the GfxCaption function.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

This Visual Basic Property specifies the hot-key(s) which, when pressed, will cause the keyboard focus to be given to the graphics window. For example, if you use "se" for this property, pressing either Alt-S or Alt-E (at runtime) will cause the focus to be given to the graphics window.

See Also

Using Graphics Tools With Visual Basic

ActualColor

Purpose

Returns the color that Windows will actually use if you attempt to draw in the color that you specify.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = ActualColor(lColor&)

VB method: lResult& = GfxWindow*.ActualColor(lColor)

lResult& = ActualColorEx(lWindowNumber&, _
                         lColor)
```

Parameters

lColor&

A Windows Color value.

Return Value

If you specify an invalid value for the *lColor&* parameter, negative one (-1) will be returned. Otherwise. the value of *lResult&* will be a Windows Color value between zero (0) and MAXCOLOR.

Remarks

It is possible for your programs to specify 16,777,216 different Windows Colors, but Windows will not always *use* exactly the color that you specify. For example, if your computer's video card is currently configured to use a limited number of colors (such as 16, 256, or 64k) and if you attempt to draw in a color that is not supported by that mode, Windows will substitute a color that is as close as possible to the color that you specified.

The ActualColor function can be used to determine which color Windows will use to represent any given color.

Example

```
If lColor& <> ActualColor(lColor&) Then
      'Windows will not use the exact
      'color that you specified.
End If
```

See Also

Windows Colors

AnimateCursor

Purpose

Plays a single cycle of (i.e. all the frames of) or optionally displays a single frame of, an animated cursor. (Also see AutoPlayCursor.)

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = AnimateCursor(sCursor$, _
                         lFrame&, _
                         lWidth&, _
                         lHeight&)
```

VB method: *GfxWindow**.AnimateCursor

```
lResult& = AnimateCursorEx(lWindowNumber&, _
                           sCursor$, _
                           lFrame&, _
                           lWidth&, _
                           lHeight&)
```

(See Syntax Options)

Parameters

sCursor$

To auto-play an animated cursor using the pre-defined frame delays, you must use a *file name* for this parameter. If you simply want to display the cursor's frame(s) manually you can also use either **1)** an empty string, to display the default Graphics Tools cursor, or **2)** a string with a numeric value between one (1) and ninety-nine (99) to display a standard Graphics Tools cursor (see Appendix B), or **3)** a string with a numeric value between 32,500 and 32,767 to display a standard Windows cursor (see Appendix A), or **4)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a cursor that is embedded in an EXE or DLL module.

lFrame&

Use either **1)** the value GFX_AUTO to automatically play all of the frames of an animated cursor *file* using the pre-defined frame delays that are embedded in the file, or **2)** a negative number which indicates the number of *milliseconds* that Graphics Tools should display each frame of the cursor, or **3)** a positive number between zero (0) and the maximum frame number (which is zero-based) to display a single frame without any delays.

lWidth& and *lHeight&*

The size of the image that should be displayed, in Drawing Units. Use zero (0) for both of these parameters if you want to display the cursor in a size that is chosen by Windows, i.e. the animation's "native" size.

Return Value

If a *single frame* is displayed and no errors are detected, the return value will be SUCCESS (zero).

If the cursor is an animated cursor and no errors are detected, the return value of this function will be highest frame number that was played. For example, if frames zero (0) though nine (9) are played (i.e. ten frames in all), the return value would be nine.

Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For general information about cursors, see Cursors and Icons.

For general information about animated cursors, including solutions for some common problems, see Animated Cursors And Icons.

To play an animation *continuously*, see AutoPlayCursor.

This function displays a cursor *in a fixed location*, as part of the graphics window. The cursor's image will be displayed with its top-left corner located at the LPR. (If you want to change the cursor that is displayed by Windows when the mouse cursor is located over the graphics window, see GfxCursor.)

If you use this function to auto-play an animated cursor *file*, Graphics Tools will use the frame-delay information that is embedded in the file to determine how long each frame should be displayed. Each frame can have a different delay time.

If you use this function to play an animated cursor that is embedded in a resource file, the frame-delay information is not available so Graphics Tools will use the number of milliseconds that you specify with the *lFrame&* parameter (see above).

If you use this function to display a single frame of a cursor, Graphics Tools will not use any delays at all.

This function can optionally load cursors using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE`.

Note that using the AnimateCursor function to display a single frame of an animated cursor is not quite the same as using DisplayCursor or StretchCursor. The AnimateCursor function provides the ability to paint the cursor's background using the current brush.

Example

```
'Play an animated cursor, and stretch
'it to 100x100 drawing units.
AnimateCursor "\GfxTools\Images\Spinner.ani", GFX_AUTO, 100,100
```

Sample Program

Drawing Animated Cursors and Icons

See Also

DisplayCursor, StretchCursor, DisplayIcon, StretchIcon
DisplayBitmap, StretchBitmap, CropBitmap

Also see AutoPlayCursor.

AnimateIcon

Purpose

Plays a single cycle of (i.e. all the frames of) or optionally displays a single frame of, an animated icon. (Also see AutoPlayIcon.)

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = AnimateIcon(sIcon$, _
                       lFrame&, _
                       lWidth&, _
                       lHeight&)
```

VB method: *GfxWindow**.AnimateIcon

```
lResult& = AnimateIconEx(lWindowNumber&, _
                         sIcon$, _
                         lFrame&, _
                         lWidth&, _
                         lHeight&)
```

(See Syntax Options)

Parameters

sIcon$

To play an animated icon using the pre-defined frame delays, you must use a *file name* for this parameter. If you simply want to display the icon's frame(s) manually you can also use either **1)** an empty string, to display the default Graphics Tools icon, or **2)** a string with a numeric value between one (1) and ninety-nine (99) to display a standard Graphics Tools icon (see Appendix B), or **3)** a string with a numeric value between 32,500 and 32,767 to display a standard Windows icon (see Appendix A), or **4)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of an icon that is embedded in an EXE or DLL file.

lFrame&

Use either **1)** the value GFX_AUTO to automatically play all of the frames of an animated icon *file* using the pre-defined frame delays that are embedded in the file, or **2)** a negative number which indicates the number of *milliseconds* that Graphics Tools should display each frame of the icon, or **3)** a positive number between zero (0) and the maximum frame number (which is zero-based) to display a single frame without any delays.

lWidth& and *lHeight&*

The size of the image that should be displayed, in Drawing Units. Use zero (0) for both of these parameters if you want to display the icon in a size that is chosen by Windows, i.e. the animation's "native" size.

Return Value

If a *single frame* is displayed and no errors are detected, the return value will be SUCCESS (zero).

If the icon is an animated icon and no errors are detected, the return value of this function will be highest frame number that was played. For example, if frames zero (0) though nine (9) are played (i.e. ten frames in all), the return value would be nine.

Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For general information about icons, see Cursors and Icons.

For general information about animated icons, including solutions for some common problems, see Animated Cursors And Icons.

To play an animation *continuously*, see AutoPlayIcon.

This function displays an icon *in a fixed location*, as part of the graphics window. The icon's image will be displayed with its top-left corner located at the LPR.

If you use this function to play an animated icon *file*, Graphics Tools will use the frame-delay information that is embedded in the file to determine how long each frame should be displayed. Each frame can have a different delay time.

If you use this function to auto-play an animated icon that is embedded in a resource file, the frame-delay information is not available so Graphics Tools will use the number of milliseconds that you specify with the *lFrame&* parameter (see above).

If you use this function to display a single frame of an icon, Graphics Tools will not use any delays at all.

This function can optionally load icons using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE`.

Example

```
'Play an icon, and stretch it
'to 100x100 drawing units.
AnimateIcon "\GfxTools\Images\Spinner.ani", GFX_AUTO, 100,100
```

Sample Program

Drawing Animated Cursors and Icons

See Also

DisplayCursor, AnimateCursor, DisplayIcon, StretchIcon
DisplayBitmap, StretchBitmap, CropBitmap

Also see AutoPlayIcon

AutoPlayControl

Purpose

Allows you to control an animated cursor or animated icon that has been started with the AutoPlayCursor or AutoPlayIcon function.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = AutoPlayControl(lNumber&, _
                           lFunction&)
```

VB method: *GfxWindow**.AutoPlayControl

```
lResult& = AutoPlayControlEx(lWindowNumber&, _
                             lNumber&, _
                             lFunction&)
```

(See Syntax Options)

Parameters

lNumber&

Either **1)** The number of the animated cursor or icon that you wish to control, between one (1) and the maximum number that Graphics Tools supports, or **2)** the value GFX_ALL, to control all currently-running animations at the same time.

lFunction&

One of the following values: AUTOPLAY_START, AUTOPLAY_PAUSE, AUTOPLAY_ERASE, or AUTOPLAY_END. See **Remarks** below for details.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

AUTOPLAY_START can be used to start an animated cursor or icon that has been paused with AUTOPLAY_PAUSE.

AUTOPLAY_PAUSE can be used to pause an animated cursor or icon that has been started with AUTOPLAY_START. Note that AUTOPLAY_PAUSE takes affect only when the current animation cycle has been completed, i.e. when Frame Zero is about to be played again. This allows AUTOPLAY_PAUSE to be used to synchronize two or more animations.

AUTOPLAY_ERASE can be used to "clear" an animated cursor or icon that is currently paused, i.e. to erase the background area of the paused cursor or icon using the Brush that was in use when the cursor or icon was first started with AutoPlayCursor or AutoPlayIcon. (The GFX_ANIMATE_BKGD option must have been activated before the cursor or icon was first displayed. See GfxOption.)

AUTOPLAY_END can be used to permanently stop the auto-playing of an animated

cursor or icon. Once AUTOPLAY_END has been used, no other AutoPlayControl functions can be used to re-start the animation, you must use AutoPlayCursor or AutoPlayIcon again. AUTOPLAY_END affects an animation immediately, it does not wait for the cycle to be completed.

If you want an animation to complete its current cycle and then stop, use AUTOPLAY_PAUSE and then AUTOPLAY_STOP.

If you want an animation to complete its current cycle and then disappear, use AUTOPLAY_PAUSE, then AUTOPLAY_ERASE, and then AUTOPLAY_STOP.

Example

```
AutoPlayCursor 1, _
               "\GfxTools\Images\Spinner.ani", _
               GFX_AUTO, _
               0, _
               0, _
               AUTOPLAY_START

(other code usually goes here)

AutoPlayControl 1, AUTOPLAY_PAUSE
```

See Also

AutoPlayCursor, AutoPlayIcon, AnimateCursor, AnimateIcon

AutoPlayCursor

Purpose

Automatically plays an animated cursor, and continuously repeats the animation.

Availability

Graphics Tools Pro Only

Warning

The use of this function can place an unusually high load on the system CPU. See Animated Cursors And Icons for more information.

Syntax

```
lResult& = AutoPlayCursor(lNumber&, _
                          sCursor$, _
                          lFrame&, _
                          lWidth&, _
                          lHeight&, _
                          lInitState&)
```

VB method: *GfxWindow**.AutoPlayCursor

```
lResult& = AutoPlayCursorEx(lWindowNumber&, _
                            lNumber&, _
                            sCursor$, _
                            lFrame&, _
                            lWidth&, _
                            lHeight&, _
                            lInitState&)
```

(See Syntax Options)

Parameters

lNumber&

A value indicating the "control number" that should be assigned to this cursor, between one (1) and the maximum number that Graphics Tools supports. Graphics Tools Standard supports two (2) auto-play cursors/icons and Graphics Tools Pro supports sixteen (16). The control number is used by the AutoPlayControl function to control the animation.

sCursor$

Either **1)** The file name (with optional drive and path) of the animated cursor that you wish to play, or **2)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a cursor that is embedded in an EXE or DLL file..

lFrame&

Either **1)** GFX_AUTO to automatically play the animated cursor using the delays that are specified in the cursor file, or **2)** a negative number, to indicate the number of milliseconds that Graphics Tools should pause between each frame of the animation.

lWidth& and lHeight&

The width and height of the desired image, in drawing units. Use zero (0) for these parameters if you wish to display the cursor in its native size.

lInitState&

The initial state of the animation, either AUTOPLAY_START or AUTOPLAY_PAUSE. If you use AUTOPLAY_PAUSE the animated cursor will

not become visible until the AutoPlayControl function is used to change the state to AUTOPLAY_START.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For general information about icons, see Cursors and Icons.

For general information about animated icons, including solutions for some common problems, see Animated Cursors And Icons.

This function is very similar to AnimateCursor, except that it plays *and automatically repeats* the animation until you tell it to stop by using the AutoPlayControl function or by closing your program. We suggest that you familiarize yourself with the AnimateCursor function before using AutoPlayCursor.

You should also become familiar with the AutoPlayControl function, to gain an additional level of control over auto-play animations.

Example

```
AutoPlayCursor 1, _
               "\GfxTools\Images\Spinner.ani", _
               GFX_AUTO, _
               0, _
               0, _
               AUTOPLAY_START
```

See Also

AutoPlayControl, Animated Cursors and Icons

AutoPlayIcon

Purpose

Automatically plays an animated icon, and continuously repeats the animation.

Availability

Graphics Tools Pro Only

Warning

The use of this function can place an unusually high load on the system CPU. See Animated Cursors And Icons for more information.

Syntax

```
lResult& = AutoPlayIcon(lNumber&, _
                        sIcon$, _
                        lFrame&, _
                        lWidth&, _
                        lHeight&, _
                        lInitState&)
```

VB method: *GfxWindow**.AutoPlayIcon

```
lResult& = AutoPlayIconEx(lWindowNumber&, _
                          lNumber&, _
                          sIcon$, _
                          lFrame&, _
                          lWidth&, _
                          lHeight&, _
                          lInitState&)
```

(See Syntax Options)

Parameters

lNumber&

A value indicating the "control number" that should be assigned to this icon, between one (1) and the maximum number that Graphics Tools supports. Graphics Tools Standard supports two (2) auto-play cursors/icons and Graphics Tools Pro supports sixteen (16). The control number is used by the AutoPlayControl function to control the animation.

sIcon$

Either **1)** The file name (with optional drive and path) of the animated icon that you wish to play, or **2)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a icon that is embedded in an EXE or DLL file.

lFrame&

Either **1)** GFX_AUTO to automatically play the animated icon using the delays that are specified in the icon file, or **2)** a negative number, to indicate the number of milliseconds that Graphics Tools should pause between each frame of the animation.

lWidth& and lHeight&

The width and height of the desired image, in drawing units. Use zero (0) for one or both of these parameters if you wish to display the icon in its native size.

lInitState&

The initial state of the animation, either AUTOPLAY_START or

AUTOPLAY_PAUSE. If you use AUTOPLAY_PAUSE the icon will not become visible until the AutoPlayControl function is used to change the state to AUTOPLAY_START.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For general information about icons, see Cursors and Icons.

For general information about animated icons, including solutions for some common problems, see Animated Cursors And Icons.

This function is very similar to AnimateIcon, except that it plays *and automatically repeats* the animation until you tell it to stop by using the AutoPlayControl function or by closing your program. We suggest that you familiarize yourself with the AnimateCursor function before using AutoPlayIcon.

You should also become familiar with the AnimateIcon function, to gain an additional level of control over auto-play animations.

Example

```
AutoPlayIcon 1, _
             "\GfxTools\Images\Spinner.ani", _
             GFX_AUTO, _
             0, _
             0, _
             AUTOPLAY_START
```

See Also

AutoPlayControl, Animated Cursors and Icons

BitmapParam

Purpose

Provides information about a bitmap.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = BitmapParam(sBitmap$, _
                       lInfoType&)
```

Parameters

sBitmap$

Use either **1)** the name of a bitmap file, or **2)** an empty string, for the default Graphics Tools bitmap, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools bitmap or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows bitmap (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a bitmap that is embedded in an EXE or DLL file.

lInfoType&

Use one of the values listed in **Remarks** below.

Return Value

The value of *lResult&* will be the requested value, or a Graphics Tools Error Code.

Remarks

By using different values for *lInfoType&*, this function can be used to obtain five different pieces of information about a bitmap.

`IMAGE_WIDTH` and `IMAGE_HEIGHT` can be used to obtain the width and height of the bitmap, in pixels. (These values indicate the *native* size of the bitmap, without regard to the size of the graphics window, the screen resolution, the Graphics Tools World, or the displayed size of the bitmap.)

You can also obtain the width and height of a bitmap in a single call to BitmapParam by using `IMAGE_WIDTH_HEIGHT`. This is significantly more efficient than using BitmapParam twice. The return value *lResult&* will contain the width of the bitmap in the Low Word and the height of the bitmap in the High Word. See High Words and Low Words.

`IMAGE_BITS_PER_PIXEL` indicates the number of bits that are required to store the color value of one pixel. Common values include one (monochrome), four (16 colors), eight (256 colors), sixteen (32k or 64k colors), twenty-four (TrueColor) and thirty-two (TrueColor32).

`IMAGE_PLANES` is the number of color planes in the bitmap. Monochrome bitmaps will always return one (1).

`IMAGE_BYTES_PER_LINE` is the number of bytes that are used for each scan line.

Microsoft's documentation provides conflicting information about this value, saying that "*This value must be divisible by 2, because Windows assumes that the bit values of a bitmap form an array that is word aligned*" but also saying that "*Each scan is a multiple of 32 bits*" which would imply that the value will always be divisible by *four*.

Example

```
lResult& = BitmapParam("\GfxTools\Images\Locator.bmp", _
                       IMAGE_HEIGHT)
```

See Also

DisplayBitmap, StretchBitmap, CropBitmap

BlueValue

Purpose

Returns the amount of blue in a Windows Color.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = BlueValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and MAXCOLOR.

Return Value

This function will always return a value between zero (0) and 255 which indicates how much blue the specified color contains.

Remarks

The RedValue, GreenValue, and BlueValue functions can be used to obtain the amounts of Red, Green, and Blue that a Windows Color contains.

Example

```
If BlueValue(lColor&) > 0 Then
      'The color contains some blue
End If
```

See Also

Windows Colors

BorderStyle and BorderStyleEx

Purpose

These Visual Basic Properties specify the type of border that the graphics window will have, and certain other window characteristics. See **Remarks** below. (Non-VB programmers should see Graphics Window Styles.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

The BorderStyle Property can be set to any of the following values: NoBorder, BorderLine, BorderRaised, BorderSunken, BorderShallow, BorderDeep, BorderBump, or BorderCaption.

As the BorderStyle Property is changed, the BorderStyle**Ex** Property will change automatically. The BorderStyleEx Property is a *numeric* representation of the current border style, but it also includes other "style" values such as whether or not the graphics window is a Tab Stop, has Scroll Bars, and so on.

If you are familiar with Graphics Window Styles you can edit the BorderStyleEx value directly, to create window styles that cannot be created by selecting other property values. If you do that, the BorderStyle Property will be automatically changed to BorderExtended to indicate that an unnamed style is being used.

See Also

Using Graphics Tools With Visual Basic

Brush

Purpose

Creates a Brush that will then be used by certain other drawing functions.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = Brush(lColor&, _
                 lStyle&, _
                 lHatch&)
```

VB: Use the BrushColor, BrushStyle, and BrushHatch properties instead of the Brush function.

```
lResult& = BrushEx(lWindowNumber&, _
                   lColor&, _
                   lStyle&, _
                   lHatch&)
```

(See Syntax Options)

Parameters

lColor&

The Windows Color of the brush, from zero (0) to `MAXCOLOR`. You can also use the HSW function for this parameter, as shown in the second **Example** below.

lStyle&

Use one of the values described in **Remarks** below.

lHatch&

Use one of the values described in **Remarks** below.

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the requested brush was created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Brushes are used by various Graphics Tools functions to fill areas of the graphics window.

The *lStyle&* parameter must be one of the following values:

`GFX_SOLID` creates a solid-color brush

`GFX_HATCHED` creates a hatched brush (see the *lHatch&* parameter below).

`GFX_CLEAR` creates a "hollow brush" (also called a "null brush") which is transparent.

If the *lStyle&* value is GFX_HATCHED, the *lHatch&* parameter must be one of the following values:

GFX_HLINE specifies a Horizontal Line hatch pattern.

GFX_VLINE specifies a Vertical Line hatch pattern.

GFX_HVLINES specifies a hatch pattern with both Horizontal and Vertical lines.

GFX_DIAGBACK specifies a hatch pattern with diagonal lines that go from Northwest to Southeast. It was named DIAGBACK because it resembles the angle of the backslash character.

GFX_DIAGFWD specifies a hatch pattern with diagonal lines that go from Northeast to Southwest. It was named DIAGFWD because it resembles the angle of the forward-slash character.

GFX_DIAGBOTH specifies a hatch pattern with both types of diagonal lines.

The hatch pattern will be created using lines of the color specified by the *lColor&* parameter. The second color (i.e. the color of the *spaces* between the hatch lines) will depend on the current background mode. If the default background mode is being used, the spaces between the lines will be transparent, so the current contents of the graphics window will be visible. If the GfxBkgdMode function has been used to change the background mode to OPAQUE, the color of the spaces between the lines will be determined by the GfxBkgdColor function.

If the *lStyle&* value is not GFX_HATCHED, the *lHatch&* parameter is ignored. It can, however, have an effect later. For example, if you use the Brush function to create a green Brush like this...

```
Brush HIGREEN, GFX_SOLID, GFX_DIAGBOTH
```

...the last parameter would be ignored and a solid green Brush would be created. However, if the BrushStyle function was then used to change the brush style to GFX_HATCHED, the GFX_DIAGBOTH pattern would be used automatically.

If you need to change only one parameter of the current Brush, it is more efficient to use the BrushColor, BrushStyle, or BrushHatch function.

If the BrushBitmap function is used to create a Bitmap Brush, all of the Brush function parameters are cleared to default values.

Example

```
'Create a black, solid brush:
Brush 0, GFX_SOLID, 0

'Create a solid brush using the HSW function
'to specify a color value by Hue, Saturation, and Whiteness...
Brush HSW(0,255,0), GFX_SOLID, 0
```

See Also

BrushColor, BrushStyle, BrushHatch, BrushBitmap

BrushBitmap

Purpose

Creates a Bitmap Brush that can then be used by other drawing functions.

Availability:

Graphics Tools Standard and Pro

Warning

Windows 95 does not fully support this feature. See **Remarks** below for details.

Syntax

```
lResult& = BrushBitmap(sBitmap$)
```

VB method: *GfxWindow**.BrushBitmap

```
lResult& = BrushBitmapEx(lWindowNumber&, _
                         sBitmap$)
```

(See Syntax Options)

Parameters

sBitmap$

Use either **1)** the name of a bitmap file, or **2)** an empty string, for the default Graphics Tools bitmap, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools bitmap or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows bitmap (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a bitmap that is embedded in an EXE or DLL file.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested brush was created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

If your Graphics Tools programs will be run on Windows 95 computers, be sure to read "Windows 95 Limitations" at the end of these Remarks.

Brushes are used by various Graphics Tools functions to fill areas of the graphics window. A Brush Bitmap can be used to fill an area, or the entire screen, with an image.

It is not possible to scale (i.e. stretch or compress) a bitmap image when it is used as a Brush. Windows always uses the default or "native" size of the bitmap. Brush Bitmaps will therefore produce somewhat different results on different systems and you may wish to create different bitmaps for use with different screen resolutions, in order to produce consistent results.

If the specified bitmap is not large enough to fill a given area, Windows will automatically "tile" the bitmap so that the entire area is filled. (If a bitmap is intended to be tiled, it will usually be created so that the edge-colors are matched in a way that produces a seamless appearance.)

Windows always fills *portions* of the graphics window as if the *entire* window was

being filled, starting at the top-left corner of the window. The LPR and the location of the area that is being filled are ignored. For example, imagine a graphics window that is 100 pixels high and 100 pixels wide, and a 100x100 bitmap that starts with black on the left side and gradually progresses to gray in the middle and white on the right side. If you were to simply display the bitmap in the graphics window (using the DisplayBitmap function) it would fill the window. Instead, if you were to use that bitmap to create a Bitmap Brush, and then use the brush to fill a small area that was on the right side of the screen, the area would be filled with the white portion of the bitmap, not the black portion.

While this default Windows behavior may seem inconvenient at first, it insures that if two filled areas touch each other, a seamless image will be produced. Another example: If you were to create a Bitmap Brush using a bitmap of a person's face, and then use that brush to fill an area near the bottom of the graphics window, you would see only the person's mouth and chin. If you were to then use it to fill an area in the middle of the screen, the nose would appear, properly connected to the mouth.

That would happen unless, of course, the bitmap was too small to fill the entire window. In that case the bitmap would be tiled as if the entire screen was being filled, and then the portion of the image that happened to correspond to the area being filled would be displayed.

If you use a Bitmap Brush and then need to return to using a "normal" brush, you should usually use the Brush function to create it. If you use the BrushColor, BrushStyle, or BrushHatch functions instead of Brush, Graphics Tools will be forced to use certain default values. For example, if you are using a Bitmap Brush and use the BrushColor function to replace it, Graphics Tools will assume that you want a `GFX_SOLID` brush.

Windows 95/98/ME Limitations

Microsoft Windows 95, 98, and ME systems (also known as Windows Platform 1) do not support bitmap brushes as fully as Windows NT, 2000, and XP do.

1) In most cases, Platform 1 is limited to using bitmap brushes that are 8x8 pixels. If you attempt to use a larger bitmap for a brush, only the top-left 8x8 pixels of the bitmap will be used. If you attempt to use a smaller bitmap for a brush, portions of the image may be black.

2) Platform 1 does not fully support color mapping of bitmap brushes, so bitmaps with more than 16 colors may not be faithfully reproduced when they are used for brushes.

Please note that these are documented limitations of Windows Platform 1. This is not a flaw in Graphics Tools.

You may find that *certain* functions *are* able to use larger bitmaps on some Platform 1 systems. For example, on all of the Windows 98 and ME systems that we tested, the GfxCls function can be used with bitmap brushes of any size. However you should probably not count on this support being consistent from system to system.

If you want to use a bitmap as the background for a graphics window on Windows Platform 1 computers, we suggest that you use the DisplayBitmap or StretchBitmap function.

Example

```
BrushBitmap "MyBrush.BMP"
```

See Also

Brush, BrushColor, BrushStyle, BrushHatch, Pens and Brushes

BrushColor

Purpose

Changes the color of the current brush, without changing the style or hatch pattern.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = BrushColor(lColor&)
```

VB property: *GfxWindow**.BrushColor = lColor&

```
lResult& = BrushColorEx(lWindowNumber&, _
                        lColor&)
```

(See Syntax Options)

Parameters

lColor&

The Windows Color of the brush, from zero (0) to MAXCOLOR. You can also use the HSW function for this parameter, as shown in the second **Example** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested brush was created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Brushes are used by various Graphics Tools functions to fill areas of the graphics window.

See Brush for complete information. Also see Pens and Brushes.

Example

```
'Change the brush color to black.
BrushColor 0

'Change the brush color using the HSW function to specify
'a color in terms of Hue, Saturation, and Whiteness...
BrushColor HSW(90, 255, 0)
```

See Also

Brush, BrushStyle, BrushHatch

BrushHatch

Purpose

Changes the hatch pattern of the current brush, without changing the color or style.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = BrushHatch(lHatch&)
```

VB property: `GfxWindow*.BrushHatch = lHatch&`

```
lResult& = BrushHatchEx(lWindowNumber&, _
                        lHatch&)
```

(See Syntax Options)

Parameters

lHatch&

Use one of the values described in **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested brush was created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Brushes are used by various Graphics Tools functions to fill areas of the graphics window.

The *lHatch&* parameter must be one of the following values:

`GFX_HLINE` specifies a Horizontal Line hatch pattern.

`GFX_VLINE` specifies a Vertical Line hatch pattern.

`GFX_HVLINES` specifies a hatch pattern with both Horizontal and Vertical lines.

`GFX_DIAGBACK` specifies a hatch pattern with diagonal lines that go from Northwest to Southeast. It was named `DIAGBACK` because it resembles the angle of the backslash character.

`GFX_DIAGFWD` specifies a hatch pattern with diagonal lines that go from Northeast to Southwest. It was named `DIAGFWD` because it resembles the angle of the forward-slash character.

`GFX_DIAGBOTH` specifies a hatch pattern with both types of diagonal lines.

The hatch pattern will be created using the current brush color and the background color, as specified with the GfxBkgdColor function, at the time that the brush is *used*. In other words, if the background color is changed after the hatch pattern is created,

the new background color *will* be used.

If the current brush style is not `GFX_HATCHED`, this function will have no immediate effect. It can, however, have an effect later. For example, consider the following code...

```
BrushStyle GFX_SOLID

BrushHatch GFX_DIAGBOTH

BrushStyle GFX_HATCHED
```

In the end, this example would create a hatched brush with the `GFX_DIAGBOTH` pattern. The hatch type would be *set* by the BrushHatch function, but the setting would not be used because at that time the brush was solid. But then, when the brush was switched to a `GFX_HATCHED` brush, the `GFX_DIAGBOTH` pattern *would* be used.

See Brush for more information. Also see Pens and Brushes.

Example

```
'Switch the brush pattern to Horizontal Lines
BrushHatch GFX_HLINE
```

See Also

Brush, BrushStyle, BrushColor, BrushBitmap

BrushStyle

Purpose

Changes the style of the current brush, without changing the color or hatch pattern.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = BrushStyle(lStyle&)
```

VB property: *GfxWindow**.BrushStyle = lStyle&

```
lResult& = BrushStyleEx(lWindowNumber&, _
                        lStyle&)
```

(See Syntax Options)

Parameters

lStyle&

Use one of the values described in **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested brush was created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Brushes are used by various Graphics Tools functions to fill areas of the graphics window.

The *lStyle&* parameter must be one of the following values:

`GFX_SOLID` creates a solid-color brush

`GFX_HATCHED` creates a hatched brush (also see BrushHatch).

`GFX_CLEAR` creates a "hollow brush", also called a "null brush", which is transparent.

See Brush for complete information. Also see Pens and Brushes.

Example

```
'Change the brush style to "clear"...
BrushStyle GFX_CLEAR
```

See Also

Brush, BrushHatch, BrushColor, BrushBitmap

CalcPoint

Purpose

Calculates the location of a point in the graphics window, based on an angle and a distance from a specified location. (This function does not draw anything or change the LPR, it merely *calculates* a new location.)

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
CalcPoint  XPos&, _
           lYPos&, _
           dpAngle#, _
           lDistance&

CalcPointEx  lWindowNumber&, _
             XPos&, _
             lYPos&, _
             dpAngle#, _
             lDistance&
```

Parameters

lXPos& and *lYPos&*

These are input *and* output parameters. In other words, these parameters of the CalcPoint function must be *variables* that contain the starting point (the X and Y location) for the calculation. Graphics Tools will modify the values of the variables in order to return the new location to your program, so if you use literal numbers for these parameters the resulting values will be lost.

dpAngle#

The angle, in degrees based on zero degrees at nine o'clock, at which an imaginary line should be drawn. (If you use the GfxOption `GFX_USE_RADIANS` function, you must specify this value in radians instead of degrees. And if you use the GfxOption `GFX_BASE_ANGLE` function, zero degrees may represent a different visual angle.) Note that this is a floating-point parameter, so fractional angles can be specified.

lDistance&

The length of the imaginary line.

Return Value

The value of *lResult&* will always be `SUCCESS` (zero), so it is safe to ignore the return value of this function. The *useful* values that are returned by this function are the *lXPos&* and *lYPos&* parameters (see just above).

Remarks

This function starts at the point *lXPos&*,*lYPos&* and draws an imaginary line of length *lDistance&*, at angle *dpAngle#*. It then modifies the *lXPos&* and *lYPos&* parameters to indicate the results of the calculation, i.e. the location of the end of the imaginary line.

This function is useful when you need to calculate the location of a point without actually drawing a line to it with the DrawAngle function.

It is important to note that working with angled lines in a digital system is an inexact process. The location of the end of the line must be rounded to the nearest pixel, both horizontally and vertically, so one-pixel errors are common. For example, if you start at 0,0 and draw a 100 Drawing Unit line at 225 degrees (down and to the right), the end-point of the line *should* be at exactly 70.7, 70.7 but that's not possible in a digital system. The location will be rounded to 71,71, resulting in a line that may be one pixel longer than expected. Because of the way the math must be performed, these rounding errors can occur even with horizontal and vertical lines.

Example

```
'Calculate the location of a point that is
'50 drawing units, at an angle of 120 degrees
'(one o'clock), from the location 512,256...

lXPos& = 512
lYPos& = 256
CalcPoint lXPos&, lYPos&, 120, 50

'The variables lXPos& and lYPos& now contain
'the location of that point.
```

See Also

DrawAngle

CaptionClick

Purpose

This Visual Basic Event is fired when the user clicks on the graphics window's caption. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

None.

Remarks

This event will, of course, be fired only if a graphics window *has* a caption.

See Also

Using Graphics Tools With Visual Basic

CaptionState

Purpose

This Visual Basic Property specifies the state (Active, Inactive, or Not Visible) of the graphics window's caption bar. (Non-VB programs should use the GfxCaption function.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

The default value of this property is CaptionNotVisible.

If you specify CapitonActive, the graphics window will have a caption with the "Active Window" color, usually dark blue.

If you specify CaptionInactive, the graphics window will have a caption with the "Inactive Window" color, usually gray.

If the caption is visible, it will contain the text specified by the CaptionText Property.

See GfxCaption for additional details.

See Also

Using Graphics Tools With Visual Basic

CaptionText

Purpose

This Visual Basic Property specifies the text that will be displayed in the graphics window's caption bar, if it has one. (Non-VB programs should use the GfxCaption function.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

The default value of this property is...

Graphics Tools: GfxWindow_

...where the underscore (_) is the graphics window's VB-assigned ID number.

Also see the CaptionState Property.

See GfxCaption for additional details.

See Also

Using Graphics Tools With Visual Basic

Changed

Purpose

This Visual Basic Event is fired when the graphics window's size or location has been changed. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Pro Only (OCX version only)

Warning

The current OCX version of Graphics Tools does not support this Event. It is reserved for future use. In order to create drag-sizable graphics windows Visual Basic programmers must use the Direct To DLL technique, which provides event notification using Window Messages not VB events. Drag-movable windows can be created using the technique shown in the Creating a Draggable Graphics Window sample program, but that technique does not raise an event when the user moves the window.

Parameters

None.

Remarks

This Visual Basic Event is fired when the graphics window's size or location has been changed. It will, of course, be fired only if a graphics window is movable and/or sizable.

See Also

Using Graphics Tools With Visual Basic

Click

Purpose

This Visual Basic Event is fired when the user clicks on the graphics window, i.e. presses and then releases the left mouse button. Also see MouseDown and MouseUp. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

None.

Remarks

The Click Event is fired when the user presses and then releases the left mouse button while the mouse cursor is located over the graphics window.

To process the down- and up-actions separately, see the MouseDown and MouseUp events.

See Also

Using Graphics Tools With Visual Basic

ConsoleColor

See the Console Tools Documentation for information about the ConsoleColor function.

ConsoleGfx

See the Console Tools Documentation for information about the ConsoleGfx function.

CropBitmap

Purpose

Displays a bitmap, optionally stretching and/or "cropping" the image, i.e. displaying only part of the original image.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = CropBitmap(sBitmap$, _
                      lWidth&, _
                      lHeight&, _
                      lLeft&, _
                      lTop&, _
                      lRight&, _
                      lBottom&)
```

VB method: *GfxWindow**.CropBitmap

```
lResult& = CropBitmapEx(lWindowNumber&, _
                        sBitmap$, _
                        lWidth&, _
                        lHeight&, _
                        lLeft&, _
                        lTop&, _
                        lRight&, _
                        lBottom&)
```

(See Syntax Options)

Parameters

sBitmap$

Use one of the following:

1) The name of the standard Windows bitmap file that you wish to display, or

2) The word `"CLIPBOARD"` in uppercase letters, to display a bitmap that was previously placed in the Windows clipboard by the SaveGfxArea or SaveGfxWindow function, or by an external program, or by the Print Screen key, or by any one of several other techniques.

3) An empty string, for the default Graphics Tools bitmap, or

4) A string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools bitmap or

5) A string with a numeric value between 32,500 and 32,767 for a standard Windows bitmap (see Appendix A), or

6) A string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a bitmap that is embedded in an EXE or DLL file.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for both of these parameters if you want the bitmap to be displayed in its native size. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

lLeft&, lTop&, lRight&, and *lBottom&*

The left, top, right, and bottom *pixel* locations (not Drawing Units) that define the portion of the bitmap that is to be displayed.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display a rectangular *portion* of a bitmap. The top-left corner of the displayed image will be located at the LPR. Except for the cropping parameters, this function is identical to StretchBitmap.

In order to use this function effectively, you must know a few things about the bitmap. If you do not know the width and height of the bitmap, for example, it will be difficult to use values for *lLeft&, lTop&, lRight&,* and *lBottom&* that will make any sense. The BitmapParam function can be used to obtain this information.

The CropBitmap function can optionally display bitmaps that are "flipped" vertically and/or horizontally. For more information, see GfxOption GFX_IMAGE_FLIP_V and GFX_IMAGE_FLIP_H.

This function can also optionally load bitmaps using many different options which control how the image is added to the graphics window. The default mode is called SRCCOPY and it causes the bitmap image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the bitmap are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds a bitmap to the graphics screen in a way that allows it to be "un-drawn" simply by using the CropBitmap function a second time, with exactly the same parameters. See GfxOption GFX_IMAGE_DRAW_MODE for more information.

This function can also optionally load bitmaps using three different "special effects" modes, including monochrome (black and white) loading. See GfxOption GFX_IMAGE_LOAD_MODE.

Example

```
'Display the rectangular area of the "Locator.bmp" bitmap
'defined by the pixel locations 0,0 (top-left corner)
'and 10,20 (bottom-right corner). Display it as an image
'that is 100 by 200 drawing units.
CropBitmap "\GfxTools\Images\Locator.bmp",100, 200, 0, 0, 10,
20
```

See Also: DisplayBitmap

CropJpeg

Purpose

Displays a JPEG image file, optionally stretching and/or "cropping" the image (i.e. optionally displaying only a portion of the image).

Availability:

Graphics Tools Pro Only

Warning

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL, this function will return `ERROR_LIBRARY_NOT_FOUND`.

Syntax

```
lResult& = CropJpeg(sBitmap$, _
                    lWidth&, _
                    lHeight&, _
                    lLeft&, _
                    lTop&, _
                    lRight&, _
                    lBottom&)
```

VB method: *`GfxWindow*`*`.CropJpeg`

```
lResult& = CropJpegEx(lWindowNumber&, _
                      sBitmap$, _
                      lWidth&, _
                      lHeight&, _
                      lLeft&, _
                      lTop&, _
                      lRight&, _
                      lBottom&)
```

(See Syntax Options)

Parameters

sFilename$

The name of the standard JPEG image file that you wish to display.

and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for both of these parameters if you want the image to be displayed in its native size. Use `GFX_AUTO` for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and `GFX_AUTO` for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

lLeft&, lTop&, lRight&, and *lBottom&*

The left, top, right, and bottom *pixel* locations (not Drawing Units) that define the portion of the JPEG image that is to be displayed.

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display a rectangular *portion* of a JPEG file. The top-left corner of the displayed image will be located at the LPR. Except for the cropping parameters, this function is identical to StretchJpeg.

In order to use this function effectively, you must know a few things about the JPEG file. If you do not know the width and height of the image, for example, it will be difficult to use values for *lLeft&, lTop&, lRight&,* and *lBottom&* that will make any sense. The JpegParam function can be used to obtain this information.

The CropJpeg function can optionally display images that are "flipped" vertically and/or horizontally. For more information, see GfxOption `GFX_IMAGE_FLIP_V` and `GFX_IMAGE_FLIP_H`.

This function can also optionally load images using many different options which control how the image is added to the graphics window. The default mode is called `SRCCOPY` and it causes the image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the image are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds an image to the graphics screen in a way that allows it to be "un-drawn" simply by using the CropJpeg function a second time, with exactly the same parameters. See GfxOption `GFX_IMAGE_DRAW_MODE` for more information.

This function is available only in Graphics Tools Pro.

Example

```
'Display a rectangle 10 pixels wide and
'20 pixels high, from the top-left corner
'(0,0) of the Jpeg file.  Display it as an
'image that is 100x200 drawing units.
CropJpeg "\GfxTools\Images\Locator.jpg",100, 200, 0, 0, 10, 20
```

See Also

DisplayJpeg, StretchJpeg

CropWindow

Purpose

Displays the contents of a graphics window (i.e. copies the contents of one graphics window to another) optionally stretching and/or "cropping" the image.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = CropWindow(lSourceWindow&, _
                      lWidth&, _
                      lHeight&, _
                      lLeft&, _
                      lTop&, _
                      lRight&, _
                      lBottom&)
```

VB method: *GfxWindow**.CropWindow

```
lResult& = CropWindowEx(lWindowNumber&, _
                        lSourceWindow&, _
                        lWidth&, _
                        lHeight&, _
                        lLeft&, _
                        lTop&, _
                        lRight&, _
                        lBottom&)
```

(See Syntax Options)

Parameters

lSourceWindow&

The graphics window number of the source image, i.e. the graphics window where the image should be copied *from*.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for *both* of these parameters if you want the image to be displayed in its native size. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

lLeft&, lTop&, lRight&, and *lBottom&*

The left, top, right, and bottom locations, in Drawing Units, that define the portion of the graphics window that is to be displayed.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See Using Multiple Graphics Windows for background information.

This function can be used to display a rectangular *portion* of a graphics window in *another* graphics window. The top-left corner of the displayed image will be located at the LPR. Except for the cropping parameters, this function is identical to StretchWindow.

The CropWindow function can optionally display images that are "flipped" vertically and/or horizontally. For more information, see GfxOption `GFX_IMAGE_FLIP_V` and `GFX_IMAGE_FLIP_H`.

This function can also optionally display images using many different options which control how the image is added to the graphics window. The default mode is called `SRCCOPY` and it causes the image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the new image are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds an image to the graphics screen in a way that allows it to be "un-drawn" simply by using the CropWindow function a second time, with exactly the same parameters. See GfxOption `GFX_IMAGE_DRAW_MODE` for more information.

Example

```
'Display a rectangle 10 Drawing Units wide and
'20 Drawing Units high, from the top-left corner
'(0,0) of window #2.  Display it as an image that
'is 100x200 Drawing Units.
CropWindow 2,100, 200, 0, 0, 10, 20
```

See Also

DisplayWindow, StretchWindow

CursorParam

Purpose

Provides information about a cursor.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = CursorParam(sCursor$, _
                       lInfoType&)
```

Parameters

sCursor$

Use either **1)** the name of a cursor file, or **2)** an empty string, for the default Graphics Tools cursor, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools cursor or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows cursor (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a cursor that is embedded in an EXE or DLL file.

lInfoType&

Use one of the values listed in **Remarks** below.

Return Value

The value of *lResult&* will be the requested information about the cursor, or a Graphics Tools Error Code.

Remarks

By using different values for *lInfoType&*, this function can be used to obtain five different pieces of information about a cursor.

`IMAGE_WIDTH` and `IMAGE_HEIGHT` can be used to obtain the width and height of the cursor, in pixels. (These values indicate the *native* size of the cursor, without regard to the size of the graphics window, the screen resolution, the Graphics Tools World or the displayed size of the cursor.)

You can also obtain the width and height of a cursor in a single call to CursorParam by using `IMAGE_WIDTH_HEIGHT`. This is significantly more efficient than using CursorParam twice. The return value *lResult&* will contain the width of the cursor in the Low Word and the height of the cursor in the High Word. See High Words and Low Words.

The following values are "synthesized" values that are derived from the bitmap that is internally associated with each Windows cursor.

`IMAGE_BITS_PER_PIXEL` indicates the number of bits that are required to store the color value of one pixel. Common values include one (monochrome), four (16 colors), and eight (256 colors).

`IMAGE_PLANES` is the number of color planes in the cursor. Monochrome cursors will always return one (1).

`IMAGE_BYTES_PER_LINE` is the number of bytes that are used for each scan line. Microsoft's documentation provides conflicting information about this value, saying that "*This value must be divisible by 2, because Windows assumes that the bit values of a bitmap form an array that is word aligned*" but also saying that "*Each scan is a multiple of 32 bits*" which would imply that the value will always be divisible by *four*.

Example

```
lResult& = CursorParam("\GfxTools\Images\LargeX.cur", _
                       IMAGE_HEIGHT)
```

See Also

BitmapParam, IconParam, JpegParam

CustomColor

Purpose

Sets or returns the value of one of sixteen Custom Colors.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = CustomColor(lNumber&, _
                       lColor&)
```

(See Syntax Options)

Parameters

lNumber&

A number from 1-16 indicating which of the sixteen Custom Color values the function should use.

lColor&

Either `GFX_QUERY` to return the current value of Custom Color number *lNumber&*, or a valid color value between zero (0) and `MAXCOLOR`, to set the value of Custom Color number *lNumber&*.

Return Value

If you use `GFX_QUERY` for *lColor&*, this function will return the current value of Custom Color number *lNumber&*.

If you use a valid color value for *lColor&*, this function will set the value of Custom Color number *lNumber&* and return the previous value, i.e. the value of Custom Color *lNumber&* before it was changed.

If you use an invalid value for *lNumber&* or *lColor&*, this function will return `ERROR_BAD_PARAM_VALUE`.

Remarks

Graphics Tools uses the sixteen Custom Colors when it displays the SelectGfxColor dialog. The Custom Colors can be displayed and modified by the dialog.

You are free to use the CustomColor function to store and retrieve custom color values for other purposes.

Example

```
'Set the value of custom color #3
'to the color value &h123465
CustomColor 1, &h123456
```

See Also

SelectGfxColor

DblClick and DblRightClick

Purpose

These Visual Basic Events are fired when the user double-clicks on the graphics window. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

X and *Y*

The location of the mouse cursor, in Drawing Units, when the double-click occurred.

Shifts

The status of the keyboard shift-keys when the double-click occurred. See **Remarks** below.

Remarks

The DblClick and DblRightClick Events are fired when the user double-clicks on the graphics window using the left or right mouse button, respectively.

The *Shifts* parameter may contain one or more of the following values, indicating that one or more shift-keys were being held down when the user double-clicked the graphics window:

```
MK_SHIFT
MK_CONTROL
```

See Also

Using Graphics Tools With Visual Basic

DisplayBitmap

Purpose

Displays a bitmap in a size that is chosen by Windows. (This is usually the actual size of the bitmap, without regard to screen resolution or the Graphics Tools World.)

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DisplayBitmap(sBitmap$)
```

VB method: `GfxWindow*.DisplayBitmap`

```
lResult& = DisplayBitmapEx(lWindowNumber&, _
                           sBitmap$)
```

(See Syntax Options)

Parameters

sBitmap$

Use either...

1) the name of the standard Windows bitmap file that you wish to display, or

2) the word `"CLIPBOARD"` in uppercase letters to display a bitmap that was previously placed in the Windows clipboard by the SaveGfxArea or SaveGfxWindow function or by an external program or by the Print Screen key, or

3) an empty string, for the default Graphics Tools bitmap, or

4) a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools bitmap or

5) a string with a numeric value between 32,500 and 32,767 for a standard Windows bitmap (see Appendix A), or

6) a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a bitmap that is embedded in a resource file that has been embedded in an EXE or DLL file.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays bitmaps in their "native" size, with the top-left corner of the displayed image located at the LPR.

This function is very similar to StretchBitmap, except that it allows Windows to determine the size of the displayed image. Also see CropBitmap.

This function can optionally display bitmaps that are "flipped" vertically and/or horizontally. For more information, see GfxOption GFX_IMAGE_FLIP_V and GFX_IMAGE_FLIP_H.

This function can also optionally load bitmaps using many different options which control how the image is added to the graphics window. The default mode is called SRCCOPY and it causes the bitmap image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the bitmap are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds a bitmap to the graphics screen in a way that allows it to be "un-drawn" simply by using the DisplayBitmap function a second time, with exactly the same parameters. See GfxOption GFX_IMAGE_DRAW_MODE for more information.

This function can also optionally load bitmaps using three different "special effects" modes, including monochrome (black and white) loading. See GfxOption GFX_IMAGE_LOAD_MODE.

Example

```
'Display the default Graphics Tools bitmap
DisplayBitmap ""
```

See Also

BitmapParam

DisplayCursor

Purpose

Displays a cursor in a size that is chosen by Windows. This is usually (but not always) the actual size of the cursor.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DisplayCursor(sCursor$)
```

VB method: *GfxWindow*.DisplayCursor*

```
lResult& = DisplayCursorEx(lWindowNumber&, _
                           sCursor$)
```

(See Syntax Options)

Parameters

sCursor$

Use either **1)** the name of a cursor file, or **2)** an empty string, for the default Graphics Tools cursor, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools cursor or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows cursor (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a cursor that is embedded in an EXE or DLL file.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays cursors in their "native" size, in a fixed location, as part of the Graphics Window. (If you want to change the cursor that is displayed by Windows when the mouse cursor is located over the graphics window, see GfxCursor.)

The top-left corner of the displayed image will be located at the LPR.

This function is very similar to the StretchCursor function, except that it allows Windows to decide the displayed size of the cursor.

Also see AnimateCursor and AutoPlayCursor.

This function can optionally load cursors using three different "special effects" modes. See GfxOption GFX_IMAGE_LOAD_MODE.

It is common for programmers to want to draw cursors such as the Windows "hourglass" (also known as OCR_WAIT), and then to un-draw that cursor later. It is usually much easier to use a bitmap than to use a cursor when un-drawing is required. Please refer to the DisplayBitmap function, and the GfxOption

GFX_IMAGE_DRAW_MODE function.

Example

```
'Display the default Graphics Tools cursor
DisplayCursor  ""
```

See Also

DisplayIcon, DisplayBitmap

DisplayIcon

Purpose

Displays an icon in a size that is chosen by Windows. This is usually (but not always) the actual size of the icon.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DisplayIcon(sIcon$)
```

VB method: *GfxWindow*.*`DisplayIcon`

```
lResult& = DisplayIconEx(lWindowNumber&, _
                         sIcon$)
```

(See Syntax Options)

Parameters

sIcon$

Use either **1)** the name of an icon file, or **2)** an empty string, for the default Graphics Tools icon, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools icon or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows icon (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of an icon that is embedded in an EXE or DLL file.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays icons in their "native" size. The top-left corner of the displayed image will be located at the LPR.

This function is very similar to the StretchIcon function, except that it allows Windows to decide the displayed size of the icon.

Also see AnimateIcon.

This function can optionally load icons using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE`.

Example

```
'Display the default Graphics Tools icon
DisplayIcon ""
```

See Also

DisplayCursor, DisplayBitmap

DisplayImage

Purpose

Displays an image from a disk file.

Availability

Graphics Tools Standard and Pro

Warning

This function's ability to display various image formats is dependent on the version of Windows that is installed on the runtime system, and other factors. See Displaying Images From Files for more information. Also see DisplayBitmap, DisplayIcon, DisplayCursor, and DisplayJpeg for functions that are *not* system-dependent.

Syntax

```
lResult& = DisplayImage(sFileName$)
```

VB method:: *GfxWindow*.DisplayImage*

```
lResult& = DisplayImageEx(lWindowNumber&, _
                          sFileName$)
```

(See Syntax Options)

Parameters

sFileName$

The file name (and optional drive/path) of a disk file that contains an image.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays an image from a disk file in its native size. Also see StretchImage.

This function's ability to display various image formats is dependent on the version of Windows that is installed on the runtime system, and other factors. It is usually (but not always) capable of displaying WMF, JPEG, and JPG, among others. See Displaying Images From Files for more information about this.

Graphics Tools Pro licensees can use the DisplayJpeg, StretchJpeg, and CropJpeg functions to display JPEG files regardless of system configuration.

PLEASE NOTE: This function cannot be used to display GIF files even if the runtime system is configured to do so. The GIF file format is patented by UniSys Corporation, and UniSys does not allow GIF files to be used by any computer program without a license from UniSys. If GIF functionality was included in Graphics Tools, Perfect Sync would be required to obtain a license, and *you* would be required to obtain a license in order to use that functionality, even to simply *display* GIF files. Because of the license costs and legal complexities involved, Graphics Tools locks out the use of GIF files.

Example

```
DisplayImage "MyImage.WMF"
```

Sample Program

A Simple Image Viewer

See Also

Bitmaps and JPEG Files

DisplayJpeg

Purpose

Displays a Jpeg image (from a disk file) in its native size.

Availability

Graphics Tools Pro Only

Warning

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL, this function will return `ERROR_LIBRARY_NOT_FOUND`.

Syntax

```
lResult& = DisplayJpeg(sFileName$)
```

VB method: *`GfxWindow*`*`.DisplayJpeg`

```
lResult& = DisplayJpegEx(lWindowNumber&, _
                         sFileName$)
```

(See Syntax Options)

Parameters

sFileName$

The name of the disk file, with optional drive and path, where the JPEG image is located.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays a JPEG image without stretching or cropping it. Compare StretchJpeg and CropJpeg.

For general information about JPEG images, see Bitmaps and JPEG Files.

This function is available only in Graphics Tools Pro.

Example

```
DisplayJpeg "\GfxTools\Samples\Images\NASA\PIA03190.JPG"
```

See Also

StretchJpeg and CropJpeg

DisplayWindow

Purpose

Displays the contents of a graphics window (i.e. copies the contents of one graphics window to another.)

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DisplayWindow(lSourceWindow&)
```

VB method: *`GfxWindow*`*`.DisplayWindow`

```
lResult& = DisplayWindowEx(lWindowNumber&, _
                           lSourceWindow&)
```

(See Syntax Options)

Parameters

lSourceWindow&

The graphics window number of the source image, i.e. the graphics window where the image should be copied *from*.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See Using Multiple Graphics Windows for background information.

This function displays graphics windows in their "native" size, with the top-left corner of the displayed image located at the LPR.

This function is very similar to StretchWindow except that it does not stretch the image. Also see CropWindow.

This function can optionally display images that are "flipped" vertically and/or horizontally. For more information, see GfxOption `GFX_IMAGE_FLIP_V` and `GFX_IMAGE_FLIP_H`.

This function can also optionally load images using many different options which control how the image is added to the graphics window. The default mode is called `SRCCOPY` and it causes the image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the new image are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds an image to the graphics screen in a way that allows it to be "un-drawn" simply by using the DisplayWindow function a second time, with exactly the same parameters. See GfxOption `GFX_IMAGE_DRAW_MODE` for more information.

Example

```
'Display the contents of graphics window #2
'in the current graphics window.
DisplayWindow 2
```

See Also

BitmapParam

DrawAngle

Purpose

Draws a line, using the current Pen, from the LPR to the point that is defined by the specified angle and distance.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
DrawAngle  dpAngle#, _
           lDistance&
```

VB method: *GfxWindow**.DrawAngle

```
DrawAngleEx lWindowNumber&, _
            dpAngle#, _
            lDistance&
```

Parameters

dpAngle#

The angle, in degrees based on zero degrees at nine o'clock, at which the line should be drawn. (If you use the GfxOption GFX_USE_RADIANS function, you must specify this value in radians instead of degrees. And if you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.) Note that this is a floating-point parameter, so it is possible to specify fractional values.

lDistance&

The length of the line, in Drawing Units.

Return Value

The value of *lResult&* will always be SUCCESS, so it is safe to ignore the return value of this function.

Remarks

This function is very similar to CalcPoint, except that it actually draws a line.

It is important to note that drawing angled lines in a digital system is an inexact process. The location of the end of the line must be rounded to the nearest pixel, both horizontally and vertically, so one-pixel errors are common. For example, if you start at 0,0 and draw a 100 Drawing Unit line at 225 degrees (down and to the right), the end-point of the line *should* be at exactly 70.7, 70.7 but that's not possible in a digital system. The location will be rounded to 71,71, resulting in a line that may be one pixel longer than expected. Because of the way the math must be performed, these rounding errors can occur even with horizontal and vertical lines.

Example

```
'Draw a line 50 drawing units long, at an angle
'of 120 degrees (one o'clock) from location 512,256:
DrawFrom 512,256
DrawAngle 120,50
```

See Also: DrawTo

DrawArc

Purpose

Draws an arc, which is a portion of an ellipse. The line is drawn using the current Pen.

Availability

Graphics Tools Standard and Pro

Warning

See note about a known Windows bug, in **Remarks** below.

Syntax

```
lResult& = DrawArc(lWidth&, _
                   lHeight&, _
                   dpStartAngle#, _
                   dpEndAngle#)
```

VB method: *GfxWindow**.DrawArc

```
lResult& = DrawArcEx(lWindowNumber&, _
                     lWidth&, _
                     lHeight&, _
                     dpStartAngle#, _
                     dpEndAngle#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters specify the dimensions of an imaginary rectangle, inside which the arc will be drawn. See DrawEllipse and **Remarks** below for more information.

dpStartAngle# and *dpEndAngle#*

The angle (in degrees, based on zero degrees at nine o'clock) from the center of the ellipse, at which the arc will start and stop. (If you use the GfxOption GFX_USE_RADIANS function, you must specify these values in radians instead of degrees. And if you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.) Note that these are floating-point parameters, so fractional values can be specified. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested arc is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws an arc by first drawing an imaginary ellipse with its center at the LPR, inside a rectangle with the dimensions *lWidth&* and *lHeight&*. (See DrawEllipse for more information about this process.) It then draws two imaginary straight lines starting from the LPR, one at *dpStartAngle#* and one at *dpEndAngle#*. It then *actually* draws the portion of the ellipse starting where the first imaginary line crosses the imaginary ellipse, and going clockwise until it reaches the point where the second imaginary line crosses the ellipse.

Please note: Microsoft has confirmed that a bug exists in the Windows API which may cause subtle errors in the way arcs, chords, and pies are drawn. Specifically, the drawing function that Windows uses may not be precise, and small errors are possible. This is usually only obvious when, for example, two pies should precisely meet each other along one line, but instead there is a small gap or the lines overlap very slightly.

Example

```
'Draw an arc from 30 degrees (ten o'clock) to
'120 degrees (one o'clock), centered on the
'location 512,256, inside an imaginary rectangle
'that is 100 drawing units wide and 50 high.
DrawFrom 512,256
DrawArc 100, 50, 30, 120
```

See Also

DrawEllipse, DrawPie, DrawChord

DrawArea

Purpose

Draws a rectangular area, filled by the current Brush or a Gradient Brush. (Unlike DrawRect, the current Pen is not used and no line is drawn around the area.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
DrawArea  lWidth&, _
          lHeight&
```

VB method: *GfxWindow**.DrawArea

```
DrawAreaEx lWindowNumber&, _l
           lWidth&, _
           lHeight&
```

Parameters

lWidth& and *lHeight&*

The width and height of the rectangular area. See **Remarks** below for more information.

Return Value

The value of `lResult&` will always be SUCCESS (zero) so it is safe to ignore the return value of this function.

Remarks

If *lWidth&* and *lHeight&* are positive numbers, a rectangular area of that size will be drawn (and filled using the current brush) with the LPR as the top-left corner of the area.

If *lWidth&* and *lHeight&* are negative numbers, the LPR will be the bottom-right corner of the area.

If one parameter is positive and one is negative, the area will be drawn accordingly.

Unlike most functions that use the current Brush, the DrawArea function is *not* affected by the GfxDrawMode function. If you need to use an unusual GfxDrawMode setting to fill a rectangular area with the current Brush, we suggest that you use the DrawRect function with a clear Pen, so that the area will be filled but a border will not be drawn.

Example

```
'Draw a filled area 50 drawing units square,
'starting from location 10,20

DrawFrom 10,20
DrawArea 50,50
```

See Also: GfxCLS, DrawRect

DrawBezier

Purpose

Draws a Bezier curve, or a series of Bezier curves, based on an array of locations. The curve is drawn using the current Pen.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawBezier(lPoint&(0,0), _
                      lPointCount&)
```

VB method: *GfxWindow*.*DrawBezier

```
lResult& = DrawBezierEx(lWindowNumber&, _
                        lPoint&(0,0), _
                        lPointCount&)
```

(See Syntax Options)

Parameters

lPoint&(0,0)

The first element of an array of X and Y locations. See **Remarks** below for details.

lPointCount&

The number of points in the *lPoint&()* array that DrawBezier should use for drawing. (This does not have to be the actual number of points in the array, but it can never be *larger* than the number of points in the array or memory errors are likely.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested Bezier curve is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

A Bezier is a curved line that is defined by a series of "control points".

To draw a single Bezier curve, four (4) control points are required. The first and last control points precisely define the start and end points of the curve. The second and third control points "influence" the curve. In most cases the Bezier curve will not *pass through* the second and third control points, the curve will simply be "pulled toward" those points.

Imagine a straight line that connects the first and last points. Placing the second and third control points close to that imaginary line will produce a Bezier with a subtle curve.

The farther away the second and third points are placed, the more strongly the Bezier will be pulled toward them, and the more the resulting line will be curved.

Multiple Bezier curves can be defined by adding points. The end point of one Bezier

serves as the starting point of the next, so points are always added in sets of three (3).

In practice, the points are usually numbered beginning with zero (0), not one (1). So, for example, if you define seven (7) points...

Point 0 is the start of the first curve.

Points 1 and 2 influence the first curve.

Point 3 is the end of the first curve and the start of the second.

Points 4 and 5 influence the second curve.

Point 6 is the end point of the second curve.

This pattern can be extended indefinitely, creating literally hundreds of connected Bezier curves.

The *lPoint&()* array must therefore contain an appropriate number of elements, or the Bezier function will not work properly. The formula for calculating valid numbers of elements is *(X times 3) plus 1* where *X* is the number of curves to be drawn. Assuming that the array always starts with element zero (0), the upper bound of the array can be 3, 6, 9, 12, 15, etc.

The *lPoint&()* array must have two dimensions, because each point is defined by an X and a Y location. See the **Example** below for a sample array.

Functionally, DrawBezier is very similar to the DrawMultiLine function, except that it draws Bezier curves instead of straight lines. Please refer to that function's **Remarks** for more information, with particular attention to the correct formatting of the *lPoint&()* array.

Example

```
'Create an array, 2 elements (X/Y) by 4 elements (4 points)
Dim lPoint&(1,3) As Long

'Start Point
lPoint&(0,0) = 256
lPoint&(1,0) = 512

'Control Point
lPoint&(0,1) = 512
lPoint&(1,1) = 256

'Control Point
lPoint&(0,2) = 512
lPoint&(1,2) = 768

'End Point
lPoint&(0,3) = 768
lPoint&(1,3) = 512

'Draw the Bezier curve
DrawBezier lPoint&(0,0), 4
```

```
'To display the four control points...
For lCounter& = 0 To 3
      DrawFrom lPoint&(0,lCounter&), lPoint&(1,lCounter&)
      DrawCircle 10
      DrawTextRow Format$(lCounter&),0
Next
```

Sample Program

Drawing A Bezier Curve

See Also

DrawPolygon

DrawButton

Purpose

Draws the image of a standard Windows Push Button, Radio Button, or Checkbox. (This function does not create a fully functioning control, it simply displays an image of one.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawButton(lWidth&, _
                      lHeight&, _
                      lType&)
```

VB method: *GfxWindow*.*DrawButton

```
lResult& = DrawButtonEx(lWindowNumber&, _
                        lWidth&, _
                        lHeight&, _
                        lType&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The width and height of the button image that is to be displayed. (In the case of the circular Radio Button, these parameters define a rectangle inside which the circle is drawn.)

lType&

Use *one* of the following four values: GFX_PUSHBUTTON for a standard Windows pushbutton, or GFX_RADIOBUTTON for a standard radio button, or GFX_CHECKBOX for a standard check box, or GFX_THREESTATE for a three-state checkbox.

You can also add GFX_INACTIVE, GFX_PUSHED, or GFX_CHECKED to specify the button state that is to be displayed:

You can also add GFX_FLAT_BORDER or GFX_MONO_BORDER to change the way the border looks.

You can also add GFX_ADJUST_RECT to exclude the surrounding edge of a push button from the size calculations.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested button is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function is used to draw images of standard Windows "button" controls, including checkboxes, radio buttons, and push buttons. The top-left corner of the displayed image will be located at the LPR.

While it would be possible to use this function to simulate standard Windows controls within the graphics window, its primary function is to display familiar-looking checkboxes and raised areas that resemble unlabeled pushbuttons.

It is important to note that this function uses the current Windows colors (as defined by the Windows Desktop Properties "Appearance" dialog) so if a computer is configured for nonstandard colors this function will not necessarily produce the black/white/gray images that you may be expecting.

Example

```
DrawButton 200, 100, PUSHBUTTON
```

See Also

DrawFrame

DrawChord

Purpose

Draws a chord, which is a portion of an ellipse that is similar to an arc but with the endpoints connected by a straight line. The chord is drawn using the current Pen and filled using the current Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawChord(lWidth&, _
                     lHeight&, _
                     dpStartAngle#, _
                     dpEndAngle#)
```

VB method: *GfxWindow**.DrawChord

```
lResult& = DrawChordEx(lWindowNumber&, _
                       lWidth&, _
                       lHeight&, _
                       dpStartAngle#, _
                       dpEndAngle#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters specify the dimensions of an imaginary rectangle, inside which the chord will be drawn. See DrawEllipse and **Remarks** below for more information.

dpStartAngle# and *dpEndAngle#*

The angle (in degrees, based on zero degrees at nine o'clock) from the center of the ellipse, at which the chord will start and stop. (If you use the GfxOption GFX_USE_RADIANS function, you must specify these values in radians instead of degrees. And if you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.) Note that these are floating-point parameters, so fractional values can be specified. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested chord is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a chord by first drawing an imaginary ellipse with its center at the LPR, inside a rectangle with the dimensions *lWidth&* and *lHeight&*. (See DrawEllipse for more information about this process.) It then draws two imaginary straight lines starting from the LPR, one at *dpStartAngle#* and one at *dpEndAngle#*. It then *actually* draws the portion of the ellipse starting where the first imaginary line crosses the imaginary ellipse, and going clockwise until it reaches the point where the second

imaginary line crosses the ellipse. Finally, it draws a straight line between the endpoints of the arc, to form a chord.

A chord cannot be filled with a Gradient Brush, but if you need to do that you can use the GfxClipArea function to create a chord-shaped shaped Clip Area, then use DrawArea to draw the gradient.

Please note: Microsoft has confirmed that a bug exists in the Windows API which may cause subtle errors in the way arcs, chords, and pies are drawn. Specifically, the drawing function that Windows uses may not be precise, and small errors are possible. This is usually only obvious when, for example, two pies should precisely meet each other along one line, but instead there is a small gap or the lines overlap very slightly.

Example

```
'Draw a chord from 30 degrees (ten o'clock) to
'120 degrees (one o'clock), centered on the
'location 512,256, inside an imaginary rectangle
'that is 100 drawing units wide and 50 high.

DrawFrom 512,256
DrawChord 100, 50, 30, 120
```

See Also

DrawEllipse, DrawPie, DrawArc

DrawCircle

Purpose

Draws a circle, using the current Pen. The circle is filled using the current Brush or a Gradient Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawCircle(lRadius&)
```

VB method: *GfxWindow*.DrawCircle*

```
lResult& = DrawCircleEx(lWindowNumber&, _
                        lRadius&)
```

(See Syntax Options)

Parameters

lRadius&

The radius of the circle that is to be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested circle was drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a circle with its center at the LPR, with radius *lRadius&*.

The apparent roundness of a circle in a graphics window will depend on several factors. See Drawing Units and Using Different "Worlds" for more information.

Example

```
DrawFrom 512,256
DrawCircle 100
```

Sample Programs

Drawing Circles
Gradient Circles

See Also

DrawEllipse

DrawCube

Purpose

Draws a figure that resembles a three-dimensional cube or box. (The word "cube" is not meant to imply that all of the sides of the figure have equal length.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawCube(lWidth&, _
                    lHeight&, _
                    lDepth&, _
                    dpEdgeAngle#
```

VB method: *GfxWindow**.DrawCube

```
lResult& = DrawCubeEx(lWindowNumber&, _
                      lWidth&, _
                      lHeight&, _
                      lDepth&, _
                      dpEdgeAngle#
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The width and height (in Drawing Units) of the "front" face of the cube or box, i.e. the face that appears to be closest to the viewer.

lDepth&

The apparent depth (in Drawing Units) of the cube or box, i.e. the apparent distance between the front and the back.

dpEdgeAngle#

The angle at which the depth of the cube or box is drawn, usually 135 degrees.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the figure is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The front of the cube or box will be drawn using the exact size that you specify, with the top-left corner at the LPR. The angled lines that produce the 3D effect will be drawn "scaled" to produce a more realistic effect. See Three-Dimensional Figures for more information about this. The amount of scaling can be adjusted with the GfxOption GFX_CUBE_DEPTH_SCALING option.

The outline of the cube or box is normally drawn with the current Pen. If the current Pen is GFX_CLEAR, the outline of the cube or box will be drawn with a highlighted version of the current brush. The amount of highlight can be adjusted with the GfxOption GFX_3D_HIGHLIGHT option. If the Pen is clear and GFX_3D_HIGHLIGHT is set to zero (0) then the outline of the figure will not be drawn. This usually results in an unrealistic figure.

The visible faces of the cube or box are normally filled with the current Brush. One or more faces will be shaded slightly, to enhance the 3D effect. The amount of shading can be adjusted with the GfxOption `GFX_CUBE_SHADING_LEVEL` option.

If one or more of the following GfxOption values are set:

```
GFX_CUBE_CAP_COLOR
GFX_CUBE_SIDE_COLOR
GFX_CUBE_FRONT_COLOR
```

...the corresponding face(s) will be drawn with the specified color(s). Note that if a face color is specified, exactly that color will be used. `GFX_CUBE_SHADING_LEVEL` will be ignored for that face. If a face color has been specified and you wish to return to using the current Brush, set the corresponding GfxOption value to `GFX_AUTO`.

The DrawCube function can also be used with a Gradient Brush, resulting in some very attractive effects.

Tip: When using a Gradient Brush, try using a clear pen. The figure will be automatically highlighted using colors that match the gradient, usually producing a very attractive and realistic result.

Tip: When you are drawing cubes or boxes for certain purposes such as Bar Charts, it is often convenient to use *negative* values for the height and/or width of the figure. Example: When a positive value is used for the height, the front face of the bar is drawn with its top-left corner at the LPR, and the figure (as with most Graphics Tools figures) is drawn "down and to the right". The larger the height value, the farther *down* the bottom of the figure is drawn. But if you use a *negative* height value, the front face will be drawn with its *bottom*-left corner at the LPR, and it will be drawn *up* and to the right. This is a more natural way to draw a bar for a chart, because using larger height values causes the bar to grow taller without moving its "base".

Example

```
DrawCube 300,300,300,135
```

Sample Programs

Drawing Cubes and Boxes
Simple Bar Chart
Gradient-Bar Chart

See Also

Three-Dimensional Figures

DrawCylinder

Purpose

Draws a figure that resembles a three-dimensional cylinder.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawCylinder(lWidth&, _
                        lHeight&, _
                        lThickness&, _
                        dpRotation#)
```

VB method: *GfxWindow**.DrawCylinder

```
lResult& = DrawCylinderEx(lWindowNumber&, _
                          lWidth&, _
                          lHeight&, _
                          lThickness&, _
                          dpRotation#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters define a rectangle, inside which the ellipse that forms the top or "cap" of the cylinder is drawn.

lThickness&

The distance between the top or "cap" of the cylinder and the other end of the cylinder.

dpRotation#

Either 0, 90, 180, or 270. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested cylinder is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The visible round end or "cap" of the cylinder will be drawn inside a rectangle that is defined by the *lWidth&* and *lHeight&* parameters. For the most realistic results we recommend a width:height ratio of between 2:1 and 3:1. For example, a width of 300 and a height of 100 would correspond to a ratio of 3:1.

The *lThickness&* parameter determines the "length" of the cylinder, i.e. the apparent distance between the two round ends.

The *dpRotation#* parameter determines the angle at which the cylinder will be drawn. An angle of zero (0) produces a vertical cylinder with a visible top, and 180 produces a vertical cylinder with a visible bottom. An angle of 90 produces a horizontal cylinder with a visible left end, and 270 produces a horizontal cylinder with a visible right end.

The outline of the cylinder will normally be drawn with the current Pen. If the current Pen is GFX_CLEAR, the outline of the cylinder will be drawn with a highlighted version of the current brush. The amount of highlight can be adjusted with the GfxOption GFX_3D_HIGHLIGHT option. If the Pen is clear and GFX_3D_HIGHLIGHT is set to zero (0) then the outline of the figure will not be drawn. This usually results in an unrealistic figure.

The visible surfaces of the cylinder are normally filled with the current Brush. The "tube" part of the cylinder will be shaded slightly, to enhance the 3D effect. The amount of shading can be adjusted with the GfxOption GFX_CYLINDER_SHADING_LEVEL option.

If one the GfxOption GFX_CYLINDER_CAP_COLOR value is set, the specified color will be used instead of the current Brush. If a cap color has been specified and you wish to return to using the current Brush, set the GfxOption GFX_CYLINDER_CAP_COLOR value to GFX_AUTO.

The DrawCylinder function can also be used with a Gradient Brush, resulting in some very attractive effects.

Tip: When using a Gradient Brush, try using a clear pen. The figure will be automatically highlighted using colors that match the gradient, usually producing a very attractive and realistic result.

Tip: When you are drawing cylinders for certain purposes such as Bar Charts, it is often convenient to use *negative* values for the height and/or width of the figure. Example: When a positive value is used for the height, the cylinder is drawn with the center of the cap at the LPR, and the figure is drawn "down" from that point. The larger the height value, the farther *down* the bottom of the figure will be drawn. But if you use a *negative* height value, the cylinder will be drawn with its invisible *bottom*-cap centered on the LPR, and it will be drawn *up* from that point. This is a more natural way to draw a bar for a chart, because using larger height values causes the bar to grow taller without moving its "base".

Example

```
DrawCylinder 300,100,500,0
```

Sample Program

Gradient-Filled Cylinders

See Also

Three-Dimensional Figures

DrawEllipse

Purpose

Draws an ellipse, which is an oval-like figure (a flattened or stretched circle) with specific mathematical properties. The ellipse is drawn using the current Pen, and filled using the current Brush or a Gradient Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawEllipse(lWidth&, _
                       lHeight&)
```

VB method: *`GfxWindow*`*`.DrawEllipse`

```
lResult& = DrawEllipseEx(lWindowNumber&, _
                         lWidth&, _
                         lHeight&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters define an imaginary rectangle inside which the ellipse will be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested ellipse is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Technically, in mathematical terms, an ellipse is defined by "two points and one distance". The total distance from the first point *to any point on the ellipse* and then back to the second point is a constant value. The resulting shape resembles a flattened or stretched circle. (Put another way, a circle and an ellipse have roughly the same relationship as a square and a rectangle.)

In Microsoft Windows drawing terms, an ellipse is defined by one point and one rectangle. The point (in our case the LPR) defines the center of the rectangle, and the rectangle's width and height define the size and shape of the ellipse that is drawn inside it. In the end, the LPR is the point that falls exactly half-way between the two points that mathematically define the ellipse. It is, in effect, the exact center of the ellipse.

(To be clear, the rectangle is never actually drawn. It is an *imaginary* rectangle, inside which the ellipse is drawn.)

It is also possible to draw a portion of an ellipse by using the DrawArc function. Also see DrawChord and DrawEllipseRotated.

Example

```
'Draw an ellipse that resembles the number zero (0).
DrawEllipse 100,200
```

See Also

DrawPie, DrawChord

DrawEllipseRotated

Purpose

This function is identical to DrawEllipse, except that it allows an ellipse to be drawn with its axis rotated to a specific angle.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawEllipseRotated(lWidth&, _
                              lHeight&, _
                              dpAngle#)
```

VB method:: *GfxWindow*.*DrawEllipseRotated

```
lResult& = DrawEllipseRotatedEx(lWindowNumber&, _
                                lWidth&, _
                                lHeight&, _
                                dpAngle#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters define an imaginary rectangle inside which the ellipse will be drawn.

dpRotation#

The angle to which the ellipse will be rotated.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested ellipse is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For background information, see DrawEllipse. Except for the *dpRotation#* parameter, DrawEllipseRotated is identical to DrawEllipse.

It is important to note that when you start *rotating* ellipses, the terms "height and width" take on different meanings. For example, if you draw and ellipse with width 1000 and height 500 that is not rotated, it will be 1000 units wide and 500 units high, just as you requested. But if you redraw that same ellipse rotated 90 degrees it will then be 1000 units high and 500 units wide.

When drawing a rotated ellipse, the *lWidth&* and *lHeight&* parameters indicate the size of the imaginary rectangle that encloses the ellipse *before* the ellipse is rotated.
'

Other confusing situations can arise if you use a graphics window that is not a Square World. For example, if you draw an ellipse that is 1000 units wide and 1000 units high in a square world, you see a circle. If you draw that same ellipse in a world with a 2-to-1 aspect ratio, you 'see a "compressed" circle, i.e. an ellipse that is twice as wide as it is tall. If you then change your code to draw that same 1000,1000 ellipse at

a 90 degree angle, the end result *will not change*. Remember, the object that is being drawn is a circle. If you rotate it 90 degrees you still get... a circle! Graphics Tools *then* compressed the circle to fit in the non-square world, and you will see a 2:1 ellipse that is not rotated.

Example

```
'Draw an ellipse that is rotated 45 degrees
DrawEllipseRotated 100,200,45
```

See Also

DrawPie, DrawChord

DrawFlood

Purpose

Floods an area (i.e. fills it with a color or a bitmap pattern) using the current Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawFlood(lMode&, _
                     lColor&)
```

VB method:: *GfxWindow*.*DrawFlood

```
lResult& = DrawFloodEx(lWindowNumber&, _
                       lMode&, _
                       lColor&)
```

(See Syntax Options)

Parameters

lMode&

Either FLOOD_TO_BORDER or FLOOD_SURFACE. See **Remarks** below.

lColor&

This is the color that defines the area to be flooded, *not* the color of the resulting flood. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used in two different ways.

If *lMode&* is FLOOD_TO_BORDER the flooding will start at the LPR and flood in all directions using the current Brush, until it reaches **1)** the color specified by *lColor&* or **2)** the edge of the graphics window.

If *lMode&* is FLOOD_SURFACE, this function will check to make sure that the color at the LPR is *lColor&* and if it is, it will replace it with the color specified by the current brush. It will then continue checking adjacent pixels and continue replacing them with the current brush until it can no longer find adjacent pixels with the color *lColor&*. In other words, when it encounters a border that is *not* the color specified by *lColor&*, it will stop.

In the case of FLOOD_TO_BORDER the *border* is assumed to be a single color, and in the case of FLOOD_SURFACE the *area to be filled* is assumed to be a single color.

Both methods have their own advantages and their own potential problems.

For example, let's assume that you are using a Bitmap Brush so that it is easy to tell

which areas have been flooded. Imagine a pattern of multi-colored random shapes in the graphics window, with a white circle drawn near the middle of the screen. The circle is simply a white line. It is not filled, so the various colored shapes "show through" the area inside and outside the white circle. If you were to use the DrawFrom function to place the LPR inside the circle, you could then use DrawFlood like this...

```
DrawFlood FLOOD_TO_BORDER, HIWHITE
```

That line of code effectively says "flood from the LPR in all directions until you reach a border with the color MAXCOLOR (white)". That would work in most cases, and the entire circle would be flooded using the Bitmap Brush.

But if one of the multicolored random shapes inside the circle had the color MAXCOLOR, that portion of the circle would not be filled because it would count as a "border", so you might not get the results that you expect. You could solve this problem by drawing the circle with a color that is not used anywhere else in the graphics window, but that might not always be possible. This is especially true if you are dealing with multicolored bitmaps (such as digitized photographs), or with a 16- or 256- color system where your selection of possible drawing colors is limited.

Another potential problem with the FLOOD_TO_BORDER mode is that if the color at the LPR is already *lColor&*, ERROR_GT_UNKNOWNERROR will be returned and no flooding will take place. This would be the case if you accidentally located the LPR *on* the border, instead of inside it.

The FLOOD_SURFACE mode, on the other hand, is useful if you need to fill a single-color area that has a single- or multi-colored boundary. If the color at the LPR is *not* the color specified by *lColor&* ERROR_GT_UNKNOWNERROR will be returned and no flooding will take place.

Both flood methods can experience problems that are caused by the fact that Graphics Tools "scales" the graphics window to make it easier to use. In most cases the World will be configured so that the number of Drawing Units is larger than the actual number of pixels in the graphics window. For example, a graphics window that is 800x400 pixels on a given computer might be scaled so that 1024x512 Drawing Units could be used. That means that there would be 60% more possible drawing locations (524,288) than actual pixels (320,000). *So two or more adjacent screen locations, as specified with Drawing Units, will often be mapped to exactly the same "real" pixel*. In other words, if you use the DrawPixel function to change the pixel at 100,100 (in Drawing Units) to blue, Graphics Tools will figure out exactly which *actual* pixel must be changed, given the current window size, screen resolution, and scaling. If you then make the assumption that the pixel at location 101,101 is *not* blue you may be unpleasantly surprised, because 101,101 may map to exactly the same *actual* pixel that 100,100 does. You can avoid problems like this by avoiding the use of nearby pixels, but this is not always possible when you're trying to locate a point inside the figure to be filled.

Example

See **Remarks**.

See Also

Pens and Brushes

DrawFocus

Purpose

Draws (or un-draws) the type of dotted-line rectangle that Windows uses to indicate that an area of the screen (or a control such as a pushbutton) has the Windows focus.

Availability

Graphics Tools Standard and Pro

Warning

This function will not draw anything if the current Font Color is black. Black is often the default font color, so you will *usually* need to change the font color before using DrawFocus. See **Remarks** below for more information.

Syntax

```
lResult& = DrawFocus(lWidth&, _
                     lHeight&)
```

VB method: `GfxWindow*.DrawFocus`

```
lResult& = DrawFocusEx(lWindowNumber&, _
                       lWidth&, _
                       lHeight&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The width and height of the focus rectangle that is to be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested focus rectangle is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a focus rectangle with its top-left corner at the LPR.

A focus rectangle is a standard Windows GUI element which usually looks like a gray, dotted line that is drawn around an area. Focus rectangles are useful for marking an area of a graphics window that is to be cut, copied, cleared, printed, saved, etc.

Windows uses XOR drawing to draw focus rectangles. That means that **1)** a focus rectangle will almost always be easily visible, regardless of the color(s) on which it is drawn, and **2)** drawing the same focus rectangle a second time, in exactly the same location, effectively *un-draws* the rectangle and restores the original background.

Surprisingly, Windows draws focus rectangles using the current *font* color, not the current pen color. So changing the font color with the GfxFontColor function will cause DrawFocus to draw focus rectangles of different colors. The actual color that is displayed will be the XOR combination of the font color and the image upon which the rectangle is drawn. See GfxDrawMode (`GFX_XOR_DRAW`) for more information about XOR drawing.

It is important to note that most graphics windows use a default font color of black,

and performing an XOR-draw operation with black results in *nothing* being drawn. (The numeric value of black is zero, and any value XOR 0 produces the original value.) So in most cases it will be necessary for you to use the GfxFontColor function to change the graphics window's font color before you use DrawFocus.

Example

```
GfxFontColor HIWHITE
DrawFrom 512,256
DrawFocus 100,100
```

See Also

DrawRect

DrawFrame

Purpose

Draws a frame with a three-dimensional appearance, similar to those that are used as Windows GUI elements.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawFrame(lWidth&, _
                     lHeight&. _
                     lType&, _
                     lOptions&)
```

VB method: *GfxWindow**.DrawFrame

```
lResult& = DrawFrameEx(lWindowNumber&, _
                       lWidth&, _
                       lHeight&, _
                       lType&, _
                       lOptions&)
```

(See Syntax Options)

Parameters

lWidth& and lHeight&

The width and height of the frame that is to be drawn.

lType&

One of the following values: FRAME_RAISED, FRAME_SUNKEN, FRAME_BUMP, or FRAME_ETCHED. If zero (0) is used for this parameter, the default value FRAME_RAISED will be used.

lOptions&

See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested frame is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function is used to draw a Windows "frame" with its top-left corner at the LPR.

The default frame type is FRAME_RAISED, which produces a standard Windows raised frame. You can also use FRAME_SUNKEN, FRAME_BUMP, or FRAME_ETCHED, to produce different effects.

Frame Options

If you use zero (0) for the lOptions& parameter, FRAME_OPT_NORMAL will be used. This creates a normal frame with four (4) sides.

You can also use the following values, or *combinations* of these values by adding them together.

IMPORTANT NOTE: Keep in mind that if you specify a value for *lOptions&* you must specify *all* of the options that you want to use. For example, simply using FRAME_OPT_SOFT would result in *nothing* being displayed, because no edges are specified. You would have to use FRAME_OPT_SOFT + FRAME_OPT_NORMAL to produce a soft rectangular frame with four edges.

FRAME_OPT_NORMAL

Specifies a normal rectangular frame, with all four edges.

FRAME_OPT_LEFT
FRAME_OPT_TOP
FRAME_OPT_RIGHT
FRAME_OPT_BOTTOM

These options tell the DrawFrame function to draw only the specified edge(s).

FRAME_OPT_SE_NW
FRAME_OPT_SW_NE
FRAME_OPT_NE_SW
FRAME_OPT_NW_SE

These options tell the DrawFrame function to draw diagonal lines from, for example, Northwest (NW) to Southeast (SE).

FRAME_OPT_FILL fills the middle of the frame.

FRAME_OPT_SOFT creates "soft" frames by using slightly different shades of gray. This effect is very subtle and, depending on the background color, can actually make frames appear *less* soft.

FRAME_OPT_FLAT creates flat borders.

FRAME_OPT_MONO creates, according to the Microsoft documentation, "a one-dimensional border". We assume that they mean *two*-dimensional, since a one-dimensional line (i.e. a line with no width) would not be visible.

Example

```
DrawFrom 512,256
DrawFrame 100,100, FRAME_SUNKEN, FRAME_OPT_NORMAL
```

See Also

DrawRect

DrawFrom

Purpose

Sets the LPR, which is the point from which most future drawing operations will originate.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
DrawFrom lXPos&, _
         lYPos&
```

VB method: *GfxWindow**.DrawFrom

```
DrawFromEx lWindowNumber&, _
           lXPos&, _
           lYPos&
```

Parameters

lXPos& and *lYPos&*

The X (left-right) and Y (up-down) graphics window locations, in Drawing Units, where future drawing operations should originate. You may also use the value GFX_WINDOW_CENTER to specify the center of the graphics window, or GFX_WINDOW_EDGE to specify the right or bottom edge of the graphics window. You may also use GFX_SAME for *one* of the *lXPos&* or *lYPos&* parameters, in order to change only the X or Y position. (Using GFX_SAME for both parameters would cause the LPR to remain in the same location.)

Return Value

The value of lResult& will always be SUCCESS, so it is safe to ignore the return value of this function.

Remarks

This function affects *most* drawing operations. For example, if you use this function to change the LPR to 100,100 many different functions such as DrawCircle, DrawRect, DrawTo and DisplayIcon will draw figures using 100,100 as their starting location. But some functions, such as DrawPixel, do not use the LPR.

If you're not sure whether or not a function uses the LPR, refer to the function's Reference Guide entry in this document.

For a complete discussion of this function, see The LPR.

Example

```
DrawFrom 512,256
```

See Also

DrawTo

DrawHole

Purpose

Creates a transparent area (a "hole") in the default graphics window (window number zero) through which the parent window can be seen.

PB/CC PROGRAMMERS: Console Tools Plus Graphics users should usually use the GfxTextHole function instead of DrawHole, to create holes that allow certain console rows/columns to be seen.

Availability

Graphics Tools Standard and Pro

Warning

Holes can be created only in the default graphics window (window number zero) and only if the graphics window is not sizable or movable. See Graphics Window Styles for more information.

Syntax

```
lResult& = DrawHole(lNumber&, _
                    lLeft&, _
                    lTop&, _
                    lRight&, _
                    lBottom&)
```

VB: Use the DrawHole function. There is no DrawHole method or property.

There is no Ex *version of this function because it is limited to window number zero.*

(See Syntax Options)

Parameters

lNumber&

The number of the hole that is to be created, from one (1) to thirty-two (32), or from one to the number that was specified with GfxOption GFX_MAX_HOLES. See **Remarks** below for more information about hole numbers.

lLeft& and *lTop&*

The location of the top-left corner of the hole, in Drawing Units.

lRight& and *lBottom&*

The location of the bottom-right corner of the hole.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested hole is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Holes can be created only in the default graphics window (window number zero) and only if the graphics window is not sizable or movable. See Graphics Window Styles for more information.

This function creates a "hole" in the default graphics window, through which the parent window is visible. Each hole is assigned an arbitrary number, so that it can be

easily deleted ("filled") later, if necessary.

You may use any hole number between one (1) and thirty-two (32). If you need more than 32 holes, you can use GfxOption GFX_MAX_HOLES to increase the number of holes, up to a maximum of 2048.

The order in which you use the hole numbers is not important, and it is not necessary to use hole number consecutively. although doing to can speed up certain drawing operations.

Holes may overlap, but this is somewhat inefficient. Still, using two holes that overlap to create an odd-shaped area will be more efficient than using three or more non-overlapping holes to create the same-shaped area. ('Efficiency", in this case, refers to a *very* brief delay that is produced by each hole. Whenever a drawing operation is performed, each hole imposes a very brief delay because it makes updating the screen more complicated. If your program performs very rapid or repeated drawing operations, this delay can become significant.)

To eliminate virtually all of the inefficiencies related to holes, we recommend that you use the GfxWindow GFX_FREEZE function to freeze the display, perform your drawing operations, and then use GfxWindow GFX_UNFREEZE. This process will also eliminate the possibility that a hole will incorrectly updated. For more information, see Refreshing The Display.

Example

```
'Create a hole with the top-left corner
'at 100,100 and the bottom-right corner
'at 200,200...
DrawHole 100,100,200,200
```

See Also

FillHole

DrawLine

Purpose

Draws a straight line between the specified two points, using the current Pen.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
DrawLine lStartX&, _
         lStartY&, _
         lEndX&, _
         lEndY&
```

VB method:: *GfxWindow**.DrawLine

```
DrawLineEx lWindowNumber&, _
         lStartX&, _
         lStartY&, _
         lEndX&, _
         lEndY&
```

Parameters

lStartX& and *lStartY&*

The starting point of the line, in Drawing Units, relative to 0,0 at the top-left corner of the graphics window.

lEndX& and *lEndY&*

The ending point of the line.

Return Value

The value of `lResult&` will always be SUCCESS, so it is safe to ignore the return value of this function.

Remarks

The DrawLine function is equivalent to the following code:

```
DrawFrom lStartX&, lStartY&
DrawTo lEndX&, lEndY&
```

The LPR is updated by DrawLine, just as if DrawFrom and DrawTo had been used.

Example

```
'Draw a line from 100,100 to 200,200
DrawLine 100,100,200,200
```

See Also

DrawFrom, DrawTo, DrawMultiLine

DrawMultiLine

Purpose

Draws one or more lines, based on an array of locations, using the current Pen. The LPR is not used.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawMultiLine(lPoint&(0,0), _
                         lPointCount&)
```

VB method: *GfxWindow**.DrawMultiLine

```
lResult& = DrawMultiLineEx(lWindowNumber&, _
                           lPoint&(0,0), _
                           lPointCount&)
```

(See Syntax Options)

Parameters

lPoint&(0,0)

The first element of an array of X and Y locations. See **Remarks** below for details.

lPointCount&

The number of points in the *lPoint&()* array that DrawMultiLine should use for drawing. (This does not have to be the actual number of points in the array, but it can never be *larger* than the number of points in the array or memory errors are likely.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested lines are drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

PLEASE NOTE: The following information applies to the DrawMultiLine, DrawPolygon, and DrawBezier functions, all of which use an array of X and Y locations to specify two or more points. We will refer to these three functions as the "PolyDraw" functions.

It is very important to note that the three PolyDraw functions require a properly-designed *array* for a parameter. If a non-array variable is used, or if the array is not dimensioned properly, a Windows Application Error (also known as a General Protection Fault) is possible.

You should use the following prototype, or its equivalent, to create your array of points:

```
Dim lPoint&(1,x)
```

...where *x* is the number of points (screen locations) that you wish to use. If desired, you may use an array with elements that are not zero-based, such as this prototype...

```
Dim lPoint&(1 TO 2, 1 To x)
```

It is very important for the array **1)** to use LONG integers, **2)** to have two and only two dimensions (i.e. one comma in the `DIM` statement), and **3)** to have a first dimension with two and only two elements, such as 0 and 1, or 1 and 2.

Using the first prototype above as an example, and assuming that four points (numbered zero through three) will be used, you would create the array like this...

```
DIM lPoint&(1,3)
```

...and assign the location of the first point like this...

```
'point 0 (the first point)
lPoint&(0,0) = 10
lPoint&(1,0) = 20
```

The `0,` element holds the X value, and the `1,` element holds the Y location, so that code would assign the graphics window location 10,20 as the first point. Now we'll assign the rest...

```
'point 1
lPoint&(0,1) = 360
lPoint&(1,1) = 240

'point 2
lPoint&(0,2) = 101
lPoint&(1,2) = 1

'point 3
lPoint&(0,3) = 70
lPoint&(1,3) = 260
```

Then, if one of the three PolyDraw functions was used that array, the specified points (10,20 and so on) would be used by the function. For example, the DrawMultiLine function...

```
DrawMultiLine lPoint&(0,0), 4
```

...would draw a line from 10,20 to 36,24, then from there to 101,1, and so on. The DrawPolygon function would draw the same lines but would connect the last point back to the first, and then fill the finished structure using the current Brush. And the DrawBezier function would draw a Bezier curve based on those same points.

It is important to note what was used for the parameters of the function. The first parameter is *the first element of the array*, usually (0,0). The second parameter is *the number of locations* in the array, in this case four (4).

As you can see if you run the example above, it is possible for the lines that are drawn to cross each other. The DrawMultiLine and DrawBezier functions are completely unaffected by crossed lines, but the filling of a crossed-line polygon that is created with the DrawPolygon function *will* be affected. See DrawPolygon and PolyFillMode for more information about this.

Using An Array That's "The Wrong Size"

Now let's assume that your program is going to perform several different PolyDraw operations, with different numbers of points each time, and that (for performance reasons) you don't want to `REDIM` the array every time the point-count changes.

All you have to do is dimension the array so that is has the largest number of points that you will need, and simply change the second parameter (*lPointCount&*) each time you use a PolyDraw function, to indicate how many locations should be used. The four-location example above would work exactly the same way if the array was dimensioned for 100 locations instead of 4. As long as the *lPointCount&* parameter was 4, only four points would be used, regardless of the actual size of the array.

Example

See **Remarks** above.

```
Dim lPoint(1,3) As Long
lPoint(0,0) = 500
lPoint(1,0) = 500
lPoint(0,1) = 600
lPoint(1,1) = 600
lPoint(0,2) = 700
lPoint(1,2) = 500
lPoint(0,3) = 800
lPoint(1,3) = 600
DrawMultiLine lPoint(0,0), 4
```

See Also

DrawBezier, DrawPolygon

DrawPgram

Purpose

Draws a parallelogram (Pgram), using the current Pen. The figure is then filled with the current Brush or a Gradient Brush. A parallelogram is a four-sided figure in which the pairs of opposite sides are parallel, but adjacent sides are not (necessarily) perpendicular to each other. (You can think of a parallelogram as a "skewed" or "italic" rectangle, but a true rectangle also qualifies as a parallelogram and can be drawn with this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawPgram(dpAngle1#, _
                     lDistance1&, _
                     dpAngle2#, _
                     lDistance2&)
```

VB method: *GfxWindow**.DrawPgram

```
lResult& = DrawPgramEx(lWindowNumber&, _
                       dpAngle1#, _
                       lDistance1&, _
                       dpAngle2#, _
                       lDistance2&)
```

(See Syntax Options)

Parameters

dpAngle1# and *lDistance1&*

The angle (in degrees) and the distance (in Drawing Units) that define the first and third sides of the figure.

dpAngle2# and *lDistance2&*

The angle and distance that define the second and fourth sides of the figure.

Note that both "angle" parameters are floating-point values, so fractional values can be specified. Also note that if you use the GfxOption GFX_USE_RADIANS function, you must specify the angle values in radians instead of degrees. If you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested parallelogram is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a parallelogram by starting at the LPR and drawing a line, using the current Pen, at angle *dpAngle1#* for distance *lDistance1&*. (For more information about this process, see DrawAngle.)

It then draws a second line, starting at the end of the first line, at angle *dpAngle2#* for distance *lDistance2&*.

It then draws a third line, the same length as the first, but at an angle that is opposite to *dpAngle1#*, (i.e. 180 degrees different from *dpAngle1#*) so that it is drawn parallel to the first line.

It then completes the outline of the figure by drawing a fourth line that is the same length as (and parallel to) the second line, thereby arriving back at the LPR.

Finally, it fills the resulting parallelogram by using the current Brush or a Gradient Brush.

Example

```
'Draw a parallelogram that
'is slanted to the right...
DrawPgram  135,100,180,100
```

See Also

DrawRect

DrawPie

Purpose

Draws a pie, using the current Pen. A pie is a portion of an ellipse (similar to an arc) with the endpoints connected to the center of the ellipse by two straight lines. The resulting pie shape is filled with the current Brush or a Gradient Brush.

For three-dimensional pies, see DrawWedge.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawPie(lWidth&, _
                   lHeight&, _
                   dpStartAngle#, _
                   dpEndAngle#)
```

VB method: *GfxWindow**.DrawPie

```
lResult& = DrawPieEx(lWindowNumber&, _
                     lWidth&, _
                     lHeight&, _
                     dpStartAngle#, _
                     dpEndAngle#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters specify the dimensions of an imaginary rectangle, inside which the pie will be drawn. See DrawEllipse and **Remarks** below for more information.

dpStartAngle# and *dpEndAngle#*

The angle (in degrees, based on zero degrees at nine o'clock) from the center of the ellipse, at which the pie will start and stop. (If you use the GfxOption GFX_USE_RADIANS function, you must specify these values in radians instead of degrees. And if you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.) Note that these are floating-point parameters, so fractional values can be specified. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested pie is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a pie by first drawing an imaginary ellipse with its center at the LPR, inside a rectangle with the dimensions *lWidth&* and *lHeight&*. (See DrawEllipse for more information about this process.) It then draws two imaginary straight lines starting from the LPR, one at *dpStartAngle#* and one at *dpEndAngle#*. It then

actually draws the portion of the ellipse starting where the first imaginary line crosses the imaginary ellipse, and going clockwise until it reaches the point where the second imaginary line crosses the ellipse. Finally, it draws two straight lines from the center of the ellipse to the endpoints of the arc.

Pies are somewhat more complex than figures like rectangles and circles because they must often "interact" visually with other pies. For example, if you are creating a bar chart with DrawRect you don't usually have to worry about the relative positions of the bars, you simply draw them next to each other, with no overlap. But if you are creating a pie chart you will usually want to draw two or more "pie slices" arranged in a circle, to create a complete "pie".

For this reason, pies are defined by the rectangle that surrounds *the entire pie*, not just one slice. You specify a rectangle that defines how a complete pie would be drawn, and then you specify which slice should be drawn by specifying the start- and end-angles of the slice. In this way, you can use the same rectangle over and over and define different angles to define different slices, and the end result will be a pie that "makes sense" visually.

Additional complexity is added it you want your pie chart to be "exploded". When all of the slices of a pie chart are moved away from the center, separating the slices, it is called an Exploded Pie. You can adjust the distance by which the pies are moved by using the GfxOption `GFX_PIE_EXPLOSION_DISTANCE` option. The default distance is zero (0).

Please note: Microsoft has confirmed that a bug exists in the Windows API which may cause subtle errors in the way arcs, chords, and pies are drawn. Specifically, the drawing function that Windows uses may not be precise, and small errors are possible. This is usually only obvious when, for example, two pies should precisely meet each other along one line, but instead there is a small gap or the lines overlap very slightly.

Example

```
'Draw a pie slice from 30 degrees (ten o'clock) to
'120 degrees (one o'clock), centered on the
'location 512,256, inside an imaginary rectangle
'that is 100 drawing units wide and 50 high.
DrawFrom 512,256
DrawPie 100, 50, 30, 120
```

See Also

DrawWedge, DrawChord, DrawArc

DrawPixel

Purpose

Changes a single pixel to the specified color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawPixel(lXPos&, _
                     lYPos&, _
                     lColor&)
```

VB method: *GfxWindow**.DrawPixel

```
lResult& = DrawPixelEx(lWindowNumber&, _
                       lXPos&, _
                       lYPos&, _
                       lColor&)
```

(See Syntax Options)

Parameters

lXPos& and *lYPos&*

The X and Y locations of the pixel, in Drawing Units, that is to be changed.

lColor&

The color that the pixel will be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the color of the requested pixel is changed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to change the color of a single pixel. A pixel (a Picture Element) is the smallest screen element that your computer's monitor can control.

It is important to keep in mind that, because of the automatic "scaling" that is performed by Graphics Tools, two or more screen locations (expressed in Drawing Units) may be mapped to the same *actual* pixel. In most cases, graphics window scaling is configured so that the number of Drawing Units is larger than the actual number of pixels in the graphics window. For example, a graphics window that is 800x400 pixels on a given computer would usually be scaled so that 1024x512 Drawing Units could be used. That means that there would be 60% more possible drawing locations (524,288) than actual pixels (320,000). *So two or more adjacent screen locations, as specified with Drawing Units, will often be mapped to exactly the same "real" pixel*. In other words, if you use the DrawPixel function to change the pixel at 100,100 (in Drawing Units) to blue, Graphics Tools will figure out exactly which *actual* pixel must be changed, given the current window size, screen resolution, and scaling. If you then attempt to change the pixel at location 101,101 to green, it is possible that exactly the same pixel will be changed.

If you ran that same program on a computer with a 640x480 screen resolution, the graphics window would probably be created as 640x320, but Graphics Tools would still scale it to 1024x512. That means that there would be 204,800 pixels in the window, but 524,288 possible Drawing Units locations, so the odds are virtually 100% that any given pixel would be shared by two different graphics window locations. The smaller the graphics window, and the larger the scaling factors, the greater the odds that you'll have problems with pixel-accurate graphics functions like DrawPixel.

Problems of this type can be eliminated by disabling the graphics window scaling function. Use the GfxWorld function to create a World that has the same number of Drawing Units as pixels, so each drawing location will correspond to a single pixel. This will, however, make *many* drawing operations *much* more complex, because your program will have to compensate for the size of the graphics window, the current screen resolution, and other factors. But it will make pixel-perfect drawing possible.

It is also important to keep in mind that Windows will not always use *exactly* the color that you request for *any* drawing operation, and that includes the DrawPixel function. For example, the pre-defined color `HIGREEN` (Windows color `&h00FF00`) will be reproduced accurately on virtually any computer that has a color monitor. But the color value that is one greater than `HIGREEN` (i.e. `&h00FF01`) can be displayed accurately only on computers that are using 24-bit or 32-bit "TrueColor". If a program that uses the Windows color value `&h00FF01` is run on a computer system that is using 16-color, 256-color, 32k-color, or 64k-color resolution, Windows will automatically substitute the closest approximation of that color that it can reproduce, which would be `&h00FF00`.

That means that if you set a pixel's color with DrawPixel and then immediately check the color of the pixel with the PixelColor function, the results may surprise you.

Example

```
DrawPixel 100, 100 HIGREEN
```

See Also

PixelColor

DrawPolygon

Purpose

Draws a polygon (any multi-sided, closed figure) using an array of locations. The current Pen is used to draw the figure, and it is filled using the current Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawPolygon(lPoint&(0,0), _
                       lPointCount& )
```

VB method:: *GfxWindow*.*DrawPolygon

```
lResult& = DrawPolygonEx(lWindowNumber&, _
                         lPoint&(0,0), _
                         lPointCount& )
```

(See Syntax Options)

Parameters

lPoint&()

The first element (0,0) of an *array* of X and Y locations. See **Remarks** below for details.

lPointCount&

The number of points in the *lPoint&()* array that DrawPolygon should use for drawing. (This does not have to be the actual number of points in the array, but it can never be *larger* than the number of points in the array or memory errors are likely.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested polygon is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

A Polygon is simply a multi-sided, closed figure. It is defined by a series of points, which are connected by straight lines to form the shape of the desired polygon. The lines may cross each other, and they do not have to be "regular" in any way, i.e. they are not required to form a symmetrical shape like a rectangle, pentagon or hexagon.

This function is virtually identical to the DrawMultiLine function, except that in addition to drawing a series of straight lines, DrawPolygon completes the figure by automatically connecting the last point back to the first, and filling the resulting shape using the current Brush or a Gradient Brush.

Please refer to the **Remarks** section of DrawMultiLine for complete information, with particular attention to the correct design of the *lPoint&()* array.

If you intend to draw polygons with lines that cross each other, you should also familiarize yourself with the PolyFillMode function, which affects the way complex polygons are filled.

A polygon cannot be filled with a Gradient Brush, but if you need to do that you can use the GfxClipArea function to create a polygon-shaped shaped Clip Area, then use DrawArea to draw the gradient.

Example

See DrawMultiLine.

See Also

DrawBezier

DrawRect

Purpose

Draws a rectangle, using the current Pen. The rectangle is filled using the current Brush or a Gradient Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawRect(lWidth&, _
                    lHeight&)
```

VB method: *GfxWindow**.DrawRect

```
lResult& = DrawRectEx(lWindowNumber&, _
                      lWidth&, _
                      lHeight&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The width and height, in Drawing Units, of the rectangle that is to be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested rectangle is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function is used to draw simple rectangles, which are four sided figures where opposite sides have equal length and are parallel to each other, and adjacent sides are perpendicular.

The rectangle is drawn starting at (and returning to) the LPR. In other words, the LPR defines the top-left corner of the rectangle.

Rectangles are drawn with the current Pen, and filled with the current Brush or a Gradient Brush.

The DrawSoftRect function can be used to draw rectangles with rounded corners.

The DrawPgram function can also be used to draw rectangles, and it has the ability to draw rectangles that are rotated so that their sides are not parallel with the edges of the graphics window. The DrawXagon function can also be used to draw rotated rectangles.

The DrawArea function can also be used to draw rectangles, but without using the current Pen.

The DrawFocus function can also be used to draw rectangles, but without using the current Pen or Brush. The DrawFocus function also has the ability to "un-draw" a rectangle.

Example

```
DrawFrom 100,100
DrawRect 20,50
```

See Also

DrawPgram, DrawSoftRect

DrawRhombus

This Graphics Tools Version 1 function is no longer supported. A rhombus is a parallelogram with four equal-length sides, so you can use the DrawPgram function to draw one.

DrawSoftRect

Purpose

Draws a rectangle with rounded corners, using the current Pen. The rectangle is filled using the current Brush.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawSoftRect(lWidth&, _
                        lHeight&, _
                        lCornerWidth&, _
                        lCornerHeight&)
```

VB method: *GfxWindow**.DrawSoftRect

```
lResult& = DrawSoftRectEx(lWindowNumber&, _
                          lWidth&, _
                          lHeight&, _
                          lCornerWidth&, _
                          lCornerHeight&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The width and height, in Drawing Units, of the rectangle that is to be drawn.

lCornerWidth& and *lCornerHeight&*

The width and height, in Drawing Units, of the ellipse that will be used for the rounded corners of the rectangle.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested rectangle is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to draw a rectangle with rounded corners. The rectangle will be drawn with the current Pen, and filled with the current Brush.

Soft rectangles are drawn using the LPR as an imaginary starting point. If you use zero (0) for the *lCornerWidth&* and *lCornerHeight&* parameters, a square-cornered rectangle will be drawn, with its top-left corner located at the LPR. If you use small nonzero values for those parameters, a rectangle with slightly rounded corners will be drawn, and the top-left corner of the rectangle will not quite touch the LPR. Using larger values will result in increasingly rounded corners, and an increasing distance between the LPR and the round corner of the rectangle.

Using *lCornerWidth&* and *lCornerHeight&* parameters that are equal to (or greater than) the *lWidth&* and *lHeight&* parameters will result in an ellipse being drawn.

A soft rectangle cannot be filled with a Gradient Brush, but if you need to do that you can use the GfxClipArea function to create a soft-rectangle-shaped shaped Clip Area, then use DrawArea to draw the gradient.

Example

```
DrawFrom 100,100
DrawSoftRect 200,300,50,50
```

See Also

DrawPgram

DrawTextBox

Purpose

1) Displays one or more lines of text, using the current font, in a rectangular area of the graphics window. (If you need to display a *single line* of text, it is usually preferable to use the DrawTextRow function because it requires fewer parameters and it provides several "special effects".)

2) The DrawTextBox function can also be used to determine how large an area would be required to display a text string. This option does not cause text to actually be displayed.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawTextBox(sText$, _
                       lOptions&, _
                       lWidth&, _
                       lHeight&)
```

VB method: *`GfxWindow*`*`.DrawTextBox`

```
lResult& = DrawTextBoxEx(lWindowNumber&, _
                         sText$, _
                         lOptions&, _
                         lWidth&, _
                         lHeight&)
```

(See Syntax Options)

Parameters

sText$

The string of text that is to be displayed.

lOptions&

See **Remarks** below for a list of the valid options.

lWidth& and *lHeight&*

The width and height (in Drawing Units) of the rectangle inside which the text is to be displayed.

Return Value

The value of `lResult&` will be the height of the text that is actually displayed. (It is also possible to obtain height and width information by using the `TEXT_CALC_SIZE` option.)

Remarks

The DrawTextBox function is most often used to display text within the rectangle that you specify. The LPR normally defines the location of the top-left corner of the rectangle, although this can be changed.

The DrawTextBox function can also be used to obtain information about text without actually displaying it. These options are described at the very end of these Remarks.

Our first example displays a single, short line of text, using a relatively small font. (If you attempt to duplicate this example, your results will vary depending on the size of the font and the size of your graphics window.)

```
DrawFrom 100,100

DrawTextBox "Graphics Tools", _
            0, _
            200, _
            100
```

The text that is to be displayed is "`Graphics Tools`", the zero indicates that no options are being used (more about options later), and the `200,100` defines a rectangle that is large enough to hold the text.

First, an imaginary rectangle (200 by 100 drawing units) would be drawn, starting at the LPR. As with virtually all Graphics Tools functions, the imaginary rectangle would be drawn "down and to the right", so that the LPR was at the top-left corner of the rectangle.

Then the text would be drawn inside the imaginary rectangle, using the current font. (See the various GfxFont functions for more information about changing the font.) As a result, the words "Graphics Tools" would appear on the screen.

If Graphics Tools was using the default (Transparent) background mode, no further drawing would be performed. However, if the GfxBkgdMode function has been used to change the background mode to Opaque, the background of the rectangle would be filled with the current background color, as specified with the GfxBkgdColor function.

Then the DrawTextBox function would calculate how much of the rectangle (from top to bottom) had actually been *used* to display text, and that number would be passed back to your program as the return value of the function. In our case, since it would be possible to display "Graphics Tools" using a single row of text, the return value would be the height, in Drawing Units, of one row of text using the current font.

The Text Origin

Let's go back and examine that process in more detail. Most first-time Graphics Tools users will probably notice that the text was not displayed quite where they expected. Here's why... If you were to use a pencil to put a small x on a piece of paper and ask a person to write their name "at the x", they would probably use the x as the *bottom*-left corner of the area where they would write. In other words, most people are used to signing "on the dotted line" -- and that means *above* the dotted line -- so many first-time DrawTextBox users expect text to be displayed *above* and to the right of the LPR. That is not the case.

The DrawTextRow function provides a method for changing the text origin, but because of the way in which Windows draws text, the DrawTextBox function does not.

Text Options

Now let's change our first example slightly. If you change the text that is to be displayed to "`Graphics Tools is great!`" and run the example code, you would probably see the words "Graphics Tools" and part of the letter `i` from the word "is", but the rest of the text would not be displayed. That's because a 200x100 rectangle is not wide enough to display the entire sentence. If you then add the **`TEXT_WORD_BREAK`** option like this...

```
DrawTextBox, "Graphics Tools is great!", _
             TEXT_WORD_BREAK, _
             200, _
             100
```

...the DrawTextBox function will automatically "word wrap" the text and attempt to make it fit inside the rectangle, so you would see "Graphics Tools" on one line and "is great!" on the next. If you were to make the text too long to be displayed within the 200x100 rectangle, like "`Graphics Tools is the greatest thing to happen to computers since sliced time!`" the DrawTextBox function would word-wrap and display as much of the text as possible, but the rest would not be displayed.

You might have noticed that the DrawTextBox function allowed the *partial* letter `i` (split vertically) to be displayed, and that a partial row of text (split horizontally) can be also displayed if the text won't fit in the rectangle that you specify. But if you use the **`TEXT_NO_PARTIALS`** option like this...

```
DrawTextBox, "Graphics Tools is great!", _
             TEXT_WORD_BREAK OR TEXT_NO_PARTIALS, _
             200, _
             100
```

... the DrawTextBox function will display only complete characters, words, and rows of text.

It is also possible to use the **`TEXT_CENTER`** option like this...

```
DrawTextBox, "Graphics Tools is great!", _
             TEXT_WORD_BREAK + TEXT_CENTER, _
             200, _
             100
```

...to cause each line of text to be centered horizontally within the rectangle. In this case, "Graphics Tools" would be centered on the first line, and "is great!" would be centered on the second.

The **`TEXT_V_CENTER`** option (which centers text vertically) can also be used, but *only* if the `TEXT_SINGLE_LINE` option is used at the same time. **`TEXT_SINGLE_LINE`** is basically the opposite of `TEXT_WORD_BREAK`, i.e. it forces the display of a single line of text.

The **`TEXT_ALIGN_RIGHT`** option can be used to display text that is aligned with the right edge of the rectangle. The **`TEXT_ALIGN_BOTTOM`** option can also be used, but only with `TEXT_SINGLE_LINE`. **`TEXT_ALIGN_TOP`** and **`TEXT_ALIGN_LEFT`** are the default modes, so it is never necessary to specify those options.

The **TEXT_RTL_READING** option can be used when the current font is a Hebrew or Arabic typeface, to tell the DrawTextBox function to use "right to left reading".

If the text is too long to fit in the rectangle that you provide, the **TEXT_END_ELLIPSIS**, **TEXT_WORD_ELLIPSIS**, and **TEXT_PATH_ELLIPSIS** options can be used to tell DrawTextBox to add an ellipsis (three dots...) to the text, in the appropriate places. TEXT_END_ELLIPSIS adds an ellipsis at the point where a string is truncated, TEXT_PATH_ELLIPSIS adds an ellipsis to the middle of a drive/path/file string that is too long to be displayed, and while TEXT_PATH_WORD is not formally documented by Microsoft, it appears to perform the same function as TEXT_END_ELLIPSIS.

If you use one of the TEXT...ELLIPSIS options and *also* use the **TEXT_MODIFY_STRING** option, the *sText$* parameter will actually be *modified* by the DrawTextBox function, to indicate where the dots were added.

The **TEXT_EXPAND_TABS** option expands embedded tab characters (CHR$(9)) to eight spaces.

The **TEXT_NO_PREFIX** option tells the DrawTextBox function *not* to interpret leading ampersand (&) characters as an "underline" prefix. In its default mode, DrawTextBox will interpret a string like "select a &File" and display "select a File", with the letter F underlined. If you use the TEXT_NO_PREFIX option, DrawTextBox will display ampersand characters instead of treating them as prefixes. (Only one underscore per string is interpreted in this way, so to produce underlined text you should usually use GfxFontEffects to create an entire *font* that is underlined.)

The **TEXT_NO_CLIP** option can be used to tell the DrawTextBox function to allow text to be drawn outside of the rectangle that you specify. This is usually only necessary when displaying rotated text, which may extend above or to the left of the LPR.

The **TEXT_EXTERNAL_LEADING** option tells the DrawTextBox function to include the font's "external leading" value when calculating the line height. Normally, the external leading value (which is the amount of extra space that Windows displays between rows of text) is not included in the height of a line of text.

Getting Information About Text

It is also possible to use the DrawTextBox function to determine how large text *would* be *if* it were to be displayed, without actually displaying the text in the graphics window. If you use the **TEXT_CALC_SIZE** option like this...

```
lHeight& = 0
lWidth&  = 100

DrawTextBox "Graphics Tools", _
            TEXT_CALC_SIZE, _
            lWidth&, _
            lHeight&
```

...nothing would actually be displayed. Instead, the value of the *lHeight&* variable would be changed by the DrawTextBox function, to indicate how much vertical space the text would require if it were to be drawn in a 100 drawing-unit-wide rectangle using the current font. (The value 100 is only an example. You may use any value.)

If you use TEXT_CALC_SIZE and TEXT_SINGLE_LINE together, the width of the text will be calculated instead of the height, and the *lWidth&* parameter will be changed.

Note that it is not usually possible to calculate both the height and width at the same time, because any given text string could be displayed in a tall, narrow rectangle or a short, wide rectangle. You must specify either the width or the height, and the DrawTextBox function will calculate the other one. The only exception to this rule is when the TEXT_SINGLE_LINE option is used. In that case, the height is automatically calculated as "the height of one row of text using the current font."

It is important to note that the font angle (see GfxFontAngle) is ignored when TEXT_CALC_SIZE is used. For example, if a given string returns *lWidth&* and *lHeight&* value of 300 and 20 when the font angle is zero, you might expect the values to be 20 and 300 when the font angle is changed to 90 degrees. i.e. when the text is drawn vertically. But the values will still be 300 and 20. The *lHeight&* value is always the "tallness of the letters" and the *lWidth&* value is always "the length of the string". If the font is rotated, *lWidth&* and *lHeight&* will be the measurements of a *rotated* rectangle that is drawn around the text. (Consider the case where text is drawn at a 45-degree angle and you'll understand why the measurements must be made in this way. Which is the width and which is the height?)

Example

See **Remarks** above.

See Also

DrawTextRow, GfxFont

DrawTextRow

Purpose

Displays a single row of text, using the current font, optionally using one or more special effects.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawTextRow(sText$, _
                       lOptions&)
```

VB method:: *GfxWindow**.DrawTextRow

```
lResult& = DrawTextRowEx(lWindowNumber&, _
                         sText$, _
                         lOptions&)
```

(See Syntax Options)

Parameters

sText$

The string of text that is to be displayed.

lOptions&

See **Remarks** below for a list of the valid options.

Return Value

The value of `lResult&` will be the height of the text that is actually displayed.

Remarks

The DrawTextRow function displays a single row of text. To display two or more rows at the same time, see DrawTextBox.

The DrawTextRow function draws characters by using the current font. The font color is usually determined by the GfxFontColor function, but it is also possible to use a Gradient Brush. See **Gradient Text Effects** below.

Our first example displays a single, short line of text, using a relatively small font. (If you attempt to duplicate these examples, your results will vary depending on the size of the font, the size of your graphics window, and other factors.)

```
DrawFrom 100,100
DrawTextRow "Graphics Tools", 0
```

The text that is to be displayed is "`Graphics Tools`", and the zero indicates that no options are being used. (More about options later).

First, an imaginary rectangle would be drawn, starting at the LPR. As with virtually all Graphics Tools functions, the imaginary rectangle would be drawn "down and to the right", so that the LPR would be at the top-left corner of the rectangle. The rectangle would extend from there to the bottom-right corner of the graphics window.

Then the text would be drawn inside the imaginary rectangle, using the current font. (See the various GfxFont functions for more information about changing the font.) The words "Graphics Tools" would appear on the screen.

If Graphics Tools was using the default (Transparent) background mode, no further drawing would be performed. However, if the GfxBkgdMode function had been used to change the background mode to Opaque, the background of the rectangle in the area where text was drawn would be filled with the current background color, as specified with the GfxBkgdColor function.

Then the DrawTextRow function would calculate how tall the displayed text was (depending on the font) and that number would be passed back to your program as the return value of the function.

The Text Origin

Let's go back and examine that process in more detail. Most first-time Graphics Tools users will notice that the text was not displayed quite where they expected. Here's why... If you were to use a pencil to put a small x on a piece of paper and ask a person to write their name "at the x", they would probably use the x as the *bottom*-left corner of the area where they would write. Most people are used to signing "on the dotted line" -- and that means *above* the dotted line -- so many first-time DrawTextRow users expect text to be displayed *above* and to the right of the LPR. That is not the case.

You can, however, use the GfxOption `GFX_TEXT_ROW_ORIGIN` option to change this behavior. You can set that option to include any of the following values, including logical combinations:

```
ORIGIN_TOP
ORIGIN_LEFT
ORIGIN_RIGHT
ORIGIN_BOTTOM
ORIGIN_H_CENTER
ORIGIN_V_CENTER
ORIGIN_CENTER
```

For example, using this...

```
GfxOption GFX_TEXT_ROW_ORIGIN, ORIGIN_BOTTOM OR
ORIGIN_LEFT
```

...would tell Graphics Tools to use the bottom-left corner of the rectangle as the text origin. (Actually, since `ORIGIN_LEFT` is a default value, all you really need is `ORIGIN_BOTTOM`.)

It is important to note that the two `ORIGIN...CENTER` options work differently from the `TEXT_CENTER` option, which is described below. All of the `ORIGIN_` options affect *where the rectangle is placed relative to the LPR*, not where the text is drawn inside the rectangle.

Moving the LPR

The DrawTextRow function also has the ability to optionally move the LPR after it draws text. Specifically, it can place the LPR at the *end* of the displayed text, so that

the next drawing operation can add text more easily. For example, doing this...

```
DrawTextRow "Hello", TEXT_MOVE_X
```

...would display the word "Hello" and then change the LPR's X (horizontal) location so that it was at the *end* of the word. Using TEXT_MOVE_Y would display the text and then move the LPR *down*.

Text Options

Now let's add some special effects, to make the text more impressive. If you use the TEXT_SHADOW option like this...

```
DrawTextRow "Graphics Tools", TEXT_SHADOW
```

...a black 3D shadow will be automatically drawn behind the text. (Of course if you try this with a black font or a graphics window that has a black background, you won't be able to see the black shadow.) You can change the color of the shadow by using the GfxOption GFX_TEXT_SHADOW_COLOR setting, and the GfxOption GFX_TEXT_SHADOW_SIZE setting can be used to change the "depth" of the shadow.

If you use the TEXT_OUTLINE option like this...

```
DrawTextRow "Graphics Tools", TEXT_OUTLINE
```

...the text will be drawn with a black outline around the letters. The GfxOption GFX_TEXT_OUTLINE_SIZE and GFX_TEXT_OUTLINE_COLOR settings can be used to change the size and color of the outline.

The TEXT_REVERSE option...

```
DrawTextRow "Graphics Tools", TEXT_REVERSE
```

...is very similar to TEXT_OUTLINE except that the outline and fill colors are swapped.

The TEXT_RAISED and TEXT_ETCHED options can be used like this...

```
DrawTextRow "Graphics Tools", TEXT_RAISED
```

...to produce text that appears to be slightly raised from, or slightly etched into, the graphics window. If you use these effects to display text in the same color as the background, an "embossed" effect can be created.

The TEXT_B_EDGES and TEXT_W_EDGES options can be used to add very thin white (W) or black (B) edges to the displayed text. The TEXT_C_EDGES option can be used to add thin edges with other colors (C) which can be controlled with the GfxOption GFX_TEXT_EDGE_COLOR option.

Keep in mind that many of these effects can be combined. For example...

```
DrawTextRow "Graphics Tools", _
            TEXT_SHADOW OR TEXT_B_EDGES
```

...produces a particularly attractive effect. So does...

```
DrawTextRow "Graphics Tools", _
            TEXT_SHADOW OR TEXT_RAISED
```

The TEXT_ANIMATE option can be used to display text character-by-character as if it was being "typed", with a speed that can be controlled with GfxOption GFX_TEXT_ANIMATE_SPEED.

Using the TEXT_CENTER option...

```
DrawTextRow, "Graphics Tools", TEXT_CENTER
```

...will cause the text to be centered horizontally between the LPR and the right edge of the graphics window. If you need to center text in a different rectangle, you can use DrawTextBox. (The ORIGIN...CENTER options that are described above provide yet another method of centering text.)

IMPORTANT NOTE: Generally speaking, the TEXT_SHADOW, TEXT_OUTLINE, TEXT_RAISED, TEXT_x_EDGES, and other "special effects" options (see above) should not be used with TEXT_CENTER. Combining those options with TEXT_CENTER often results in shadows/outlines/etc. with slightly rough edges. We recommend that if you are using special effects, you should avoid using TEXT_CENTER and use DrawFrom to specify the left-right position of the text.

The TEXT_V_CENTER option can be used to center the text vertically between the LPR and the bottom of the graphics window.

Certain combinations of effects, such as TEXT_ANIMATE and TEXT_CENTER will not produce useful results when they are used together. The TEXT...CENTER options cannot be used with rotated text, i.e. text that is displayed when the GfxFontAngle is not zero.

The TEXT_ALIGN_RIGHT option can be used to display text that is aligned with the right edge of the rectangle instead of the left. The TEXT_ALIGN_BOTTOM option can also be used. TEXT_ALIGN_TOP and TEXT_ALIGN_LEFT are the default modes, so it is not necessary to specify those options.

The TEXT_RTL_READING option can be used when the current font is a Hebrew or Arabic typeface, to tell the DrawTextRow function to use "right to left reading".

If the text is too long to fit in the rectangle that you provide, the TEXT_END_ELLIPSIS, TEXT_WORD_ELLIPSIS, and TEXT_PATH_ELLIPSIS options can be used to tell DrawTextRow to add an ellipsis (three dots...) to the text, in the appropriate places. TEXT_END_ELLIPSIS adds an ellipsis at the point where a string is truncated, TEXT_PATH_ELLIPSIS adds an ellipsis to the middle of a drive/path/file string that is too long to be displayed, and while TEXT_PATH_WORD is not officially documented by Microsoft, it appears to perform the same function as TEXT_END_ELLIPSIS.

If you use one of the TEXT...ELLIPSIS options and *also* use the TEXT_MODIFY_STRING option, the *sText$* parameter will actually be *modified* by the DrawTextRow function, to indicate where the dots were added.

The TEXT_EXPAND_TABS option expands embedded tab characters (CHR$(9)) to eight spaces.

The TEXT_NO_PREFIX option tells the DrawTextRow function *not* to interpret leading ampersand (&) characters as an "underline" prefix. In its default mode, DrawTextRow will interpret a string like "&File" and display only "File", with the letter F underlined. If you use the TEXT_NO_PREFIX option, DrawTextRow will display ampersand characters instead of treating them as prefixes. (Only one underscore per string is interpreted in this way, so to produce underlined text you should usually use GfxFontEffects to create an entire *font* that is underlined.)

The TEXT_EXTERNAL_LEADING option tells the DrawTextRow function to include the font's "external leading" value when calculating the line height. Normally, external leading (which is the amount of extra space that Windows displays between rows of text) is not included in the height of a line of text.

The TEXT_NO_CLIP option (see DrawTextBox) is used automatically by the DrawTextRow function, so it is not necessary to specify it when using DrawTextRow.

Gradient Text Effects

The DrawTextBox and DrawTextRow functions display text using the current font, using the current GfxFontColor value. If special effects such as TEXT_SHADOW, TEXT_OUTLINE, and TEXT_C_EDGES are used (see above), they are usually drawn with solid colors that can be specified with the GfxOption function.

But it is also possible to use a Gradient Brush when drawing text. Gradient-Filled text can provide a wide variety of striking effects, and gradients can also be used for shadows, outlines, and other effects. See the Gradient-Filled Text sample program for several examples.

To use Gradient Filled Text, use the following line of code:

```
GradientBrush GRADIENT_FILL_TEXT_BODY
```

To use a gradient for shadows and outlines, use:

```
GradientBrush GRADIENT_FILL_TEXT_EFFECTS
```

It is usually not practical to use both of those values at once, because two different gradients would be required to produce an attractive effect. If you want to produce text that uses one gradient for the body and another for shadows and outlines, we suggest that you draw the shadow or outline first, using one gradient, and then "overlay" the body of the text using another gradient.

Example

See **Remarks** above.

Sample Program

Gradient-Filled Text

See Also

DrawTextBox, GfxFont

DrawTo

Purpose

Draws a straight line from the LPR to the specified point, using the current Pen, then changes the LPR to the specified point.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawTo(lXPos&, _
                  lYPos&)
```

VB method:: *GfxWindow**.DrawTo

```
lResult& = DrawToEx(lWindowNumber&, _
                    lXPos&, _
                    lYPos&)
```

(See Syntax Options)

Parameters

lXPos& and *lYPos&*

The endpoint of the line that is to be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested line is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The DrawTo function is usually used with the DrawFrom function, like this...

```
DrawFrom 100,100
DrawTo 200,200
```

...to draw a straight line. However, since the LPR is automatically changed by DrawTo, it may not be necessary for you to use DrawFrom every time you use DrawTo. For example, the following code...

```
DrawFrom 100,100
DrawTo 100,200
DrawTo 200,200
DrawTo 200,100
DrawTo 100,100
```

...would draw a square. And because the LPR ended up back at `100,100` if you added...

```
DrawTo 200,200
```

...Graphics Tools would then add a diagonal line from the top-left corner to the bottom right corner. To add a similar diagonal line from the top-right to bottom left corner would require the use of DrawFrom...

```
DrawFrom 200,100
DrawTo 100,200
```

Example

See **Remarks** above.

See Also

DrawLine

DrawTube

Purpose

Draws a figure which resembles a three-dimensional tube.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = DrawTube(lWidth&, _
                    lHeight&, _
                    lThickness&, _
                    dpRotation#)
```

VB method:: *GfxWindow*.*DrawTube

```
lResult& = DrawTubeEx(lWindowNumber&, _
                      lWidth&, _
                      lHeight&, _
                      lThickness&, _
                    dpRotation#)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters define a rectangle, inside which the ellipse that forms the open end of the tube is drawn.

lThickness&

The distance between the two ends of the tube.

dpRotation#

Either 0, 90, 180, or 270. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested tube is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function is nearly identical to the DrawCylinder function, except that no "end cap" is drawn so the "inside" of the tube is visible through the open end. As with cylinders, the visible round end of the tube will be drawn inside a rectangle that is defined by the *lWidth&* and *lHeight&* parameters. For the most realistic results we recommend a width:height ratio of between 2:1 and 3:1. For example, a width of 300 and a height of 100 would correspond to a ratio of 3:1.

The *lThickness&* parameter determines the "length" of the tube, i.e. the apparent distance between the two round ends.

The *dpRotation#* parameter determines the angle at which the tube will be drawn. An angle of zero (0) produces a vertical tube with a visible top, and 180 produces a vertical tube with a visible bottom. An angle of 90 produces a horizontal tube with a visible left end, and 270 produces a horizontal tube with a visible right end.

The outline of the tube will normally be drawn with the current Pen. If the current Pen is GFX_CLEAR, the outline of the tube will be drawn with a highlighted version of the current brush. The amount of highlight can be adjusted with the GfxOption GFX_3D_HIGHLIGHT option. If the Pen is clear and GFX_3D_HIGHLIGHT is set to zero (0) then the outline of the figure will not be drawn. This usually results in an unrealistic figure.

The visible surfaces of the tube are normally filled with the current Brush. The "inside" part of the tube will be shaded slightly, to enhance the 3D effect. The amount of shading can be adjusted with the GfxOption GFX_TUBE_SHADING_LEVEL option.

Please note that it is difficult to create realistic-looking tubes without using Gradients. When no Gradient is used, a tube tends to resemble a cylinder with a dark end cap and the eye is not fooled into seeing it as a tube. See the sample program called Gradient-Filled Tubes.

Tip: When using a Gradient Brush, try using a clear pen. The figure will be automatically highlighted using colors that match the gradient, usually producing a very attractive and realistic result.

Tip: When you are drawing tubes for certain purposes such as Bar Charts, it is often convenient to use *negative* values for the height and/or width of the figure. Example: When a positive value is used for the height, the tube is drawn with the center of the open end at the LPR, and the figure is drawn "down" from that point. The larger the height value, the farther *down* the bottom of the figure will be drawn. But if you use a *negative* height value, the tube will be drawn with its invisible *bottom* end centered on the LPR, and it will be drawn *up* from that point. This is a more natural way to draw a bar for a chart, because using larger height values causes the bar to grow taller without moving its "base".

Example

```
DrawCylinder 300,100,500,0
```

Sample Program

Gradient-Filled Tubes

See Also

Three-Dimensional Figures

DrawWedge

Purpose

Draws a figure that resembles a three dimensional wedge or "slice of pie".

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawWedge(lWidth&, _
                     lHeight&, _
                     lThickness&, _
                     dpStartAngle#, _
                     dpEndAngle#
```

VB: *GfxWindow**.DrawWedge

```
lResult& = DrawWedgeEx(lWindowNumber&, _
                       lWidth&, _
                       lHeight&, _
                       lThickness&, _
                       dpStartAngle#, _
                       dpEndAngle#
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

These parameters specify the dimensions of an imaginary rectangle, inside which the visible pie (i.e. the top surface of the wedge) will be drawn. See DrawEllipse and **Remarks** below for more information.

lThickness&

The distance between the top and bottom pies that define the wedge.

dpStartAngle# and *dpEndAngle#*

The angle (in degrees, based on zero degrees at nine o'clock) from the center of the ellipse, at which the wedge will start and stop. (If you use the GfxOption GFX_USE_RADIANS function, you must specify these values in radians instead of degrees. And if you use the GfxOption GFX_BASE_ANGLE function, zero degrees may represent a different visual angle.) Note that these are floating-point parameters, so fractional values can be specified. See **Remarks** below for more information. If *dpStartAngle#* and *dpEndAngle#* are within 1/100 of a degree of being equal, nothing will be drawn.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested wedge is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

A "wedge" is the three-dimensional version of a pie.

This function draws a wedge by first drawing a pie. The pie is drawn inside an imaginary ellipse with its center at the LPR, inside a rectangle with the dimensions *lWidth&* and *lHeight&*. (See DrawPie and DrawEllipse for more information about this process.) It then draws two imaginary straight lines starting from the LPR, one at *dpStartAngle#* and one at *dpEndAngle#*. It then *actually* draws the portion of the ellipse starting where the first imaginary line crosses the imaginary ellipse, and going clockwise until it reaches the point where the second imaginary line crosses the ellipse. Finally, it draws two straight lines from the center of the ellipse to the endpoints of the arc. This forms the visible pie, i.e. the top surface of the wedge. Additional lines are then drawn to form the "thickness" of the wedge, and then the figure is filled with the current Brush or a Gradient Brush.

The outline of the wedge is normally drawn using the current Pen. If the current Pen is clear (`GFX_CLEAR`) the outline will be drawn using a highlighted version of the current Brush. The amount of highlighting can be adjusted with the GfxOption `GFX_3D_HIGHLIGHT` option. If that option is set to zero (0) a clear pen will be used, usually resulting in an unrealistic figure.

Tip: When using a Gradient Brush, try using a clear pen. The figure will be automatically highlighted using colors that match the gradient, usually producing a very attractive and realistic result.

Wedges are somewhat more complex than cubes or cylinders, because in order to look realistic they must often "interact" visually with other wedges. For example, if you are creating a bar chart with DrawCube you don't usually have to worry about the relative positions of the bars, you simply draw them next to each other, with no overlap. But if you are using wedges to create a three-dimensional pie chart you will usually want to draw two or more wedges arranged in a circle, to create a complete "pie".

For this reason, wedges (and two-dimensional DrawPie figures for that matter) are defined by the rectangle that surrounds *the entire pie*, not just one wedge. You specify a rectangle that defines how a complete pie would be drawn, and then you specify which "slice" should be drawn by specifying the start- and end-angles of the wedge. In this way, you can use the same rectangle over and over and define different angles to define different wedges, and the end result will be a pie chart that "makes sense" visually.

When using wedges we recommend that you use a rectangle with a width:height ratio that is somewhere in the 2:1 or 3:1 range, for the most realistic-looking results. Please note that Windows Platform 2 (95/98/ME) sometimes "flattens" the edges of wedges that are drawn with low ratio rectangles. This effect can be minimized by using larger ratios, i.e. taller rectangles.

Additional complexity is added it you want your three-dimensional pie chart to be "exploded". When all of the slices of a pie are moved away from the center, separating the slices, it is called an Exploded Pie. You can adjust the distance by which wedges are moved by using the GfxOption `GFX_PIE_EXPLOSION_DISTANCE` option. The default distance is zero (0).

For additional information about wedges, see Three-Dimensional Figures.

Example

```
'Draw a wedge from 30 degrees (ten o'clock) to
'120 degrees (one o'clock), inside an imaginary
'rectangle that is 100 drawing units wide and 33 high.
'The pie will be 200 drawing units thick.
DrawFrom 512,256
DrawWedge 100, 33, 200, 30, 120
```

See Also

DrawChord, DrawArc

DrawXagon

Purpose

Draws an x-sided figure, such as a pentagon, hexagon, or octagon, using the current Pen. The shape is then filled using the current Brush or a Gradient Brush. This function can also be used to draw other shapes, such as pentagrams and other x-pointed "stars".

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = DrawXagon(lSides&, _
                     lRadius&, _
                     dpRotation#, _
                     lEffect&)
```

VB method: *`GfxWindow*`*`.DrawXagon`

```
lResult& = DrawXagonEx(lWindowNumber&, _
                       lSides&, _
                       lRadius&, _
                       dpRotation#, _
                       lEffect&)
```

(See Syntax Options)

Parameters

lSides&

The number of sides that the figure will have. This number must be greater than or equal to three (3). Using a number that is too large will result in a figure that resembles a circle.

lRadius&

The radius of the circle inside which the figure will be drawn. This number must be greater than or equal to one (1). See **Remarks** below for more information.

dpRotation#

Defines the location of the starting point of the first line that will be drawn. The default angle is "straight up" (i.e. 12 o'clock). You may *offset* the default angle by using a value between zero and plus-or-minus 360 degrees for this parameter. For example, using a *dpRotation#* value of 90 degrees would result in the first point being drawn at the 3 o'clock mark. Note that this is a floating-point parameter, so fractional values can be specified. (If you use the GfxOption `GFX_USE_RADIANS` function, you must specify this offset value in radians instead of degrees. This function is *not* affected by the GfxOption `GFX_BASE_ANGLE` function because it is an *offset*.)

lEffect&

See **Remarks** below. (This value must be between zero (0) and the value of the *lSides&* parameter, minus one.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested figure is drawn without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function draws a symmetrical x-sided figure, such as an equilateral triangle (3 sides), a square (4 sided), a pentagon (5 sides), a hexagon (6 sides), and so on. The figure is drawn using the current Pen, and filled using the current Brush or a Gradient Brush.

First, an imaginary circle with radius *lRadius&* is drawn, with its center at the LPR. Then the point on the circle at 90 degrees (i.e. straight up) from the center is found, and that point is used as the starting point for the first line. The DrawXagon function then calculates the remaining points on the circle. For example, if a square was being drawn, the locations of points at 180, 270, and 360 degrees would be calculated. Then straight lines would be drawn to connect the points, and the resulting shape would be filled using the current Brush.

The *dpRotation#* parameter can be used to specify a starting point other than 90 degrees. In this way, figures such as rotated squares can be easily drawn.

The *lEffect&* parameter can be used to specify the order in which the points are connected with lines. For example, if you were to use this code...

```
DrawFrom 200,200
DrawXagon 5, 100, 0, 0
```

...a simple pentagon (a five-sided figure) would be drawn. The *lEffect&* parameter of zero (0) tells the DrawXagon function to drawn the figure normally. But if you were to use an *lEffect&* value of two (2), like this...

```
DrawXagon 5, 100, 0, 2
```

...the DrawXagon function would start at the 90 degree point, and then draw lines to *every other* point. In other words, instead of drawing the lines from 1 to 2, 2 to 3, 3 to 4, 4 to 5, then from 5 back to 1, it would draw from 1 to 3, 3 to 5, 5 to 2 (because a full circle had been completed), 2 to 4, then from 4 back to 1. The resulting figure would be a pentagram instead of a pentagon. (A pentagram is a five-pointed star-shaped figure.)

Using an *lEffect&* value of three (3) or four (4) would drawn the same pentagram. Figures with more than five sides, however, will usually produce different results when different *lEffect&* values are used.

If the *lEffect&* parameter is zero, the figure will always be drawn with lines that do not cross each other. A nonzero *lEffect&* value will create a figure with crossed lines, and that makes the "filling" operation more complex. Windows provides two different polygon-filling (or "PolyFill") modes. In its default mode, Graphics Tools will produce polygons that are completely filled with the current Brush, but if the PolyFillMode function is used to change the mode, different effects can be obtained. For example, a pentagram can either be completely filled, or just the "points" of the star can be filled, leaving the pentagon in the middle of the figure unfilled.

Not all combinations of *lSides&* and *lEffect&* values will produce the expected results, especially when an even number is used for *lSides&*. For example, if you create a

six-sided figure with an *lEffect&* value of two, you will get two triangles, one on top of the other, in a "Sheriff's Star" or "Star Of David" pattern. It isn't possible to draw that figure without "lifting the pen" so when the second triangle is finished and filled with the Brush, some of the lines of the first triangle will be covered up.

Example

See **Remarks** above.

Sample Program

Drawing X-Sided Figures and Stars

See Also

DrawPolygon

DropFiles

Purpose

This Visual Basic Event is fired when the user drops a file on a graphics window that has the AcceptDrop Property. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Pro Only (OCX version only)

Warning

None.

Parameters

X and *Y*

The window location, in Drawing Units, where the drop occurred.

FileList

A string containing one or more file names. If the string contains two or more file names, they will be separated by Carriage Return characters (CHR$(13)).

Remarks

The DropFiles Event is fired when the user drops a file on the graphics window, but only if the window's AcceptDrop Property is True.

See Also

Using Graphics Tools With Visual Basic

FillHole

Purpose

Fills a "hole" in the default graphics window (which was previously created with the DrawHole function) so that it is no longer transparent.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
FillHole  lNumber&
```

VB: Use the FillHole function. There is no FillHole method or property.

There is no Ex *version of this function because it is limited to window number zero.*

Parameters

lNumber&

The number of the hole that is to be filled. This number must match the number that was previously used to create a hole with the DrawHole function.

Return Value

The value of `lResult&` will always be SUCCESS, so it is safe to ignore the return value of this function.

Remarks

This function is used to "fill" or "close" a hole that was "opened" by using the DrawHole function. See DrawHole for more information.

Example

```
FillHole 1
```

See Also

Creating "Holes" in the Display

FocusRect

Purpose

This Visual Basic Property determines whether or not a Focus Rectangle will be displayed around a graphics window when it has the keyboard focus. (Non-VB programs should use the Graphics Window Style GFX_STYLE_TABSTOP.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

This Property can be set to either True or False. If it is set to True, a Focus Rectangle will be displayed around the graphics window whenever it has the keyboard focus. The color of the focus rectangle can be specified with the GfxOption GFX_FOCUS_RECT_COLOR option.

See Also

Using Graphics Tools With Visual Basic

FontAngle, FontColor, FontFaceName, FontHeight, FontItalic, FontStrikeOut, FontUnderline, FontWeight, and FontWidth

Purpose

These Visual Basic Properties correspond to the various GfxFont functions. See **Remarks** below.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

These Visual Basic Properties correspond to the font settings that are provided by the following Graphics Tools functions. Please refer to the documentation for the functions for complete information about the valid values for each Property.

FontAngle	GfxFontAngle
FontColor	GfxFontColor
FontFaceName	GfxFontName
FontHeight	GfxFontHeight
FontItalic	GfxFontEffects
FontStrikeOut	GfxFontEffects
FontUnderline	GfxFontEffects
FontWeight	GfxFontWeight
FontWidth	GfxFontWidth

See Also

Using Graphics Tools With Visual Basic

GfxAllocBuffer

This function is exported by the Graphics Tools DLL, but it is intended for internal use by Graphics Tools.

Do not attempt to use this function except under instructions from Perfect Sync Technical Support.

GfxBkgdColor

Purpose

Sets the graphics background color. Note: In the default Graphics Tools background mode (see GfxBkgdMode TRANSPARENT) this setting no effect on the display.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxBkgdColor(lColor&)
```

VB method: *GfxWindow**.GfxBkgdColor

```
lResult& = GfxBkgdColorEx(lWindowNumber&, _
                          lColor&)
```

(See Syntax Options)

Parameters

lColor&

The background color that Windows should use for certain drawing operations, between zero (black) and MAXCOLOR (white)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the background color is changed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The default background *mode* is "Transparent", so the background *color* is not normally important. It only becomes important if the GfxBkgdMode function is used to change the background mode to "Opaque".

If that is done, the color that you specify with this function will be used **1)** as the background color for text that is drawn with the DrawTextBox and DrawTextRow functions, **2)** as the "in between" color for lines that are drawn with non-solid Pens, and **3)** as the "in between" color for hatched Brushes.

Example

```
GfxBkgdColor LOBLUE
```

See Also

GfxBkgdMode

GfxBkgdMode

Purpose

Determines whether the background of the graphics window is transparent (the default mode) or opaque (meaning that the GfxBkgdColor is displayed under certain circumstances).

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxBkgdMode(lTransparent&)
```

VB method: *GfxWindow**.GfxBkgdMode

```
lResult& = GfxBkgdModeEx(lWindowNumber&, _
                         lTransparent&)
```

(See Syntax Options)

Parameters

lTransparent&

Use a nonzero value (such as TRUE) for this parameter if you want the background mode to be Transparent, or zero (0) for the Opaque background mode.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the background mode is changed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See GfxBkgdColor for complete information about this function.

Example

See GfxBkgdColor.

See Also

Pens and Brushes

GfxCaption (Caption properties)

Purpose

This function can be used to specify a hot-key for a graphics window. If a graphics window has a *visible* caption (see Graphics Window Styles) this function can be used to change the text that appears in the caption and the apparent "state" of the caption bar.

Availability

Graphics Tools Standard and Pro

Warning

Console Tools Plus Graphics programs must use zero (0) for the *lState&* parameter.

Syntax

```
lResult& = GfxCaption(sCaption$, _
                      lActive&)
```

VB properties: (note the use of `Caption` not `GfxCaption`)

```
GfxWindow*.CaptionText = sCaption$
GfxWindow*.CaptionState = lActive&
```

```
lResult& = GfxCaptionEx(lWindowNumber&, _
                        sCaption$,
                        lActive&)
```

(See Syntax Options)

Parameters

sCaption$

The text (including an optional hot-key) that should appear in the graphics window's caption bar, or an empty string to leave the current text unchanged. See **Remarks** below.

lState&

Either **1)** the value `GFX_SAME` to leave the state unchanged, or **2)** False (zero) to give the caption bar the appearance of the "inactive" (grayed) state, or **3)** True (a nonzero value) to give it the appearance of the "active" state, or Because Microsoft Windows does not allow console applications to have two active captions, Console Tools Plus Graphics programs must use zero (0) for this parameter.

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the caption bar or hot-key is changed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Visual Basic users please note: For ease of use, the GfxCaption function is implemented as two different VB properties called CaptionText *and* CaptionState*. They behave just like the GfxCaption function.*

The GfxCaption function will have a *visible* effect only when a graphics window has a window style that provides a caption. This function will not, however, produce an error if the window does not have a caption. Microsoft Windows allows caption-less windows to have a "hidden" caption string, so Graphics Tools allows the caption to be changed even when it is not visible.

NON-VB PROGRAMS: If you use a string for *sCaption$* that contains an ampersand (&) the character that follows the ampersand defines the hot-key for the graphics window *even if the caption is not visible*. For example, if you use `&S` for *sCaption$* then pressing Alt-S at runtime will cause the keyboard focus to be given to the graphics window. Unfortunately, Window will actually *display* the ampersand in the caption, so using GfxCaption to specify both a hot-key and a visible caption is usually not practical.

VB PROGRAMS: Adding this code to a Visual Basic program...

```
GfxWindow*.CaptionText = "My Graphics Window"
GfxWindow*.CaptionState = CaptionActive
GfxWindow*.AccessKeys = "g"
```

...would change the graphics window caption to say "My Graphics Window", set the caption bar to the Active state, and specify a hot-key of Alt-G.

ALL PROGRAMS: If you use an empty string for *sCaption$* or the value `GFX_SAME` for *lActive&*, the corresponding property will be left unchanged. This is to allow you to easily change one without affecting the other.

Please note that the *lState&* parameter is used to *simulate* the appearance of an active or inactive window. This will correspond to the *actual* state of the window if and only if you write code to change the caption state as necessary. This is to allow programs to have two or more window that appear to be active at the same time, or to allow a window to be active/inactive without displaying the corresponding color.

In a Visual Basic program, you could add this code so that whenever the graphics window has the focus, the caption will be switched to the active state:

```
Private Sub GfxWindow*_Initialize()
    GfxWindow*.CaptionState = CaptionInactive
End Sub

Private Sub GfxWindow*_GotFocus()
    GfxWindow*.CaptionState = CaptionActive
End Sub

Private Sub GfxWindow*_LostFocus()
    GfxWindow*.CaptionState = CaptionInactive
End Sub
```

Here is the corresponding code for the callback function of a PowerBASIC DDT program:

```
If CbMsg = %WM_GFX_EVENT + %WM_SETFOCUS Then
      GfxCaption "", %TRUE
ElseIf CbMsg = %WM_GFX_EVENT + %WM_KILLFOCUS Then
      GfxCaption "", %FALSE
End If
```

Or you might add this code to your DDT callback function to synchronize the graphics window's caption state with the main dialog's caption state:

```
If CbMsg = %WM_NCACTIVATE Then
      'the main dialog is being
      'activated or deactivated.
      GfxCaption "", CbWParam
End If
```

The actual code you add to your program will, of course, depend on the effect that you are trying to achieve.

Example

See examples in **Remarks** above.

See Also

Graphics Window Styles

GfxClipArea

Purpose

This function is used to define and clear "clip areas", in order to restrict *other* drawing operations to certain portions of the graphics window.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxClipArea(lOperation&)
```

VB method: *GfxWindow**.GfxClipArea

```
lResult& = GfxClipAreaEx(lWindowNumber&, _
                         lOperation&)
```

(See Syntax Options)

Parameters

lOperation&

One of the following values: AREA_DEFINE, AREA_USE, AREA_RESET, AREA_ADD, AREA_XOR, AREA_DIFFERENCE, AREA_OVERLAP, AREA_ACTIVE, or AREA_INACTIVE. See **Remarks** below for details.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The GfxClipArea function can be used to *prevent* Graphics Tools from drawing in certain areas of a graphics window. For example, if you were to define a Clip Area that included just the right half of a graphics window and then draw a circle in the center of the window, only the right half of the circle would actually be drawn.

For a detailed discussion of this function, including several examples, see Using Clip Areas.

Example

See Using Clip Areas.

Sample Program

Using Clip Areas to Restrict Drawing

See Also

The Graphics Window

GfxCLS

Purpose

Clears the entire graphics window, using the current solid, hatched, or bitmap brush. (To clear a smaller area, use the DrawArea function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
GfxCLS
```

VB method: *`GfxWindow*`*`.GfxCls`

```
GfxClsEx lWindowNumber&
```

Parameters

None.

Return Value

The value of `lResult&` will always be SUCCESS (zero) so it is safe to ignore the return value of this function.

Remarks

This function clears the current graphics window, using the current Brush. It also resets the LPR to 0,0.

Unlike most functions that use the current Brush, the GfxCLS function is *not* affected by the GfxDrawMode function. It will always clear the screen *completely*, using the current Brush.

Example

```
BrushBitmap "MyScreen.BMP"
GfxCLS
```

See Also

DrawArea

GfxConvert

Purpose

Converts a "Pixels" value to a "Drawing Units" value, or vice versa.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxConvert(lType&, _
                      lUnits&)

lResult& = GfxConvertEx(lWindowNumber&, _
                        lType&, _
                        lUnits&)
```

Parameters

lType&

One of the following constants:

```
X_PIXELS_TO_UNITS
Y_PIXELS_TO_UNITS
X_UNITS_TO_PIXELS
Y_UNITS_TO_PIXELS.
```

lUnits&

The value to be converted.

Return Value

The value of *lResult&* will be the converted value. (The value of the *lUnits&* parameter is not changed by this function. See last **Example** below.)

Remarks

This function is used to convert a screen measurement (typically a size or a location) from one system to another.

Example

To convert a horizontal (X-axis) Pixel measurement to Drawing Units:

```
lUnits& = GfxConvert(X_PIXELS_TO_UNITS, lPixels&)
```

To convert a vertical (Y-axis) Pixel measurement to Drawing Units:

```
lUnits& = GfxConvert(Y_PIXELS_TO_UNITS, lPixels&)
```

To convert a horizontal (X-axis) Drawing Units measurement to Pixels:

```
lPixels& = GfxConvert(X_UNITS_TO_PIXELS, lUnits&)
```

To convert a vertical (Y-axis) Drawing Units measurement to Pixels:

```
lPixels& = GfxConvert(Y_UNITS_TO_PIXELS, lUnits&)
```

Note that the *lUnits&* parameter itself is not actually changed or "converted" by this function. If that is what you want to accomplish, you should use the same variable

name on both sides of the equation, like this:

```
lValue& = GfxConvert(Y_UNITS_TO_PIXELS, lValue&)
```

See Also

The Graphics Window

GfxCursor

Purpose

Changes the type of cursor that is displayed when the Windows mouse cursor is located over a graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
GfxCursor lCursor&
```

VB method: *GfxWindow**.GfxCursor

```
GfxCursorEx lWindowNumber&, _
            lCursor&
```

Parameters

lCursor&

You must use one of the values that are described in **Remarks** below.

Return Value

None. Always returns SUCCESS (zero).

Remarks

If you do not use this function in your program, the default Windows "arrow" cursor will be displayed whenever the mouse cursor is located over the graphics window.

You may use the following values with this function, to change the cursor:

0	The default Windows arrow cursor
1	The Graphics Tools Large X cursor
2	The Graphics Tools Large Arrow cursor

For example, using this...

```
GfxCursor  1
```

...would select the Large X cursor.

You may also use any of the IDC_ values that are listed in Appendix A: Windows Bitmaps, Icons, and Cursors such as IDC_ARROW, IDC_IBEAM, IDC_WAIT, IDC_CROSS, and so on. For example, using this...

```
GfxCursor  IDC_CROSS
```

...would select the standard Windows "cross" cursor. Or you may simply use the numbers that correspond to the constants, like this:

```
GfxCursor  32515    'same as IDC_CROSS
```

You may also embed a cursor (.CUR) file in your program's EXE file, using a numeric

ID number between `100` and `31,999` and then use that ID number with the GfxCursor function. For more information see Embedded Resources.

Advanced users: You may also use the Windows API to create or load a cursor (using the API functions called CreateCursor, LoadCursor, and/or LoadImage) and use the *handle* of your cursor with the GfxCursor function. Warning #1: To avoid problems, we recommend that if you use an API function to load a cursor, that you not *also* embed cursors in your resource file. Conflicts between handle-based and resource-based cursors are possible on Windows 95, 98, and ME. Warning #2: Using an invalid handle will produce unpredictable results, possibly including Application Errors.

Advanced Users: If your program sets the cursor by using the `WM_SETCURSOR` window message you should not *also* use the GfxCursor function. If you take over the handling of the cursor in this way you must do so completely.

Example

See **Remarks** above for several examples.

See Also

DisplayCursor

GfxDrawMode

Purpose

Changes the basic way in which Graphics Tools draws with Pens and Brushes.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxDrawMode(lMode&)
```

VB method: *GfxWindow**.GfxDrawMode

```
lResult& = GfxDrawMode(lWindowNumber&, _
                       lMode&)
```

(See Syntax Options)

Parameters

lMode&

The drawing mode that should be used for future drawing operations. See **Remarks** below for a list of valid values.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested draw mode is selected without errors. Otherwise, a Graphics Tools Error Code will be returned.

If a valid *lMode&* parameter is used, the value of *lResult&* will almost always be SUCCESS (zero), so it is usually safe to ignore the return value of this function.

Remarks

This function controls the way Graphics Tools draws with all Pens and almost all Brushes. The only Brush functions that are *not* affected by the GfxDrawMode setting are GfxCLS and DrawArea. The GfxCLS and DrawArea functions always fill the area that you specify with the current Brush, regardless of the GfxDrawMode setting. *Other than that, all Pen and Brush drawing is affected by GfxDrawMode.*

You must use one of the following values for *lMode&*:

Common Drawing Modes

GFX_OPAQUE_COLORS

This is the default Graphics Tools drawing mode, also known as GFX_NORMAL_DRAW_MODE. The colors of the current Pen and Brush are opaque, i.e. they completely replace the colors that are currently on the screen.

GFX_INVERSE_COLORS

Drawing will be performed with the "inverse colors" of the current Pen and Brush. Inverse colors are produced by taking the bit-wise NOT value of the Pen and Brush colors. For example, using the NOT

value of a blue pen (which contains blue but no red or green) produces yellow (which is red plus green, with no blue)

`GFX_INVERSE_SCREEN`

The current Pen and Brush colors will be ignored, and drawing will be performed by inverting the current screen colors. For example, drawing a line across a blue background would result in a yellow line. If the screen colors vary from area to area, the Pen and Brush colors will vary when drawing is performed in different areas.

`GFX_BLACK_ONLY` *and* `GFX_WHITE_ONLY`

These drawing modes provide an easy way to switch to a black or white Pen and Brush, and then return to the previous Pen and Brush color and style simply by returning to the `GFX_OPAQUE_COLORS` mode.

Draw and Un-Draw Modes

IMPORTANT NOTE: When you want to draw and then un-draw something, you should use the `GFX_XOR_DRAW` mode for *both* operations. Drawing with `GFX_XOR_DRAW` and then attempting to un-draw with `GFX_XOR_DRAW_INVERSE` *will not work*. These two modes are related to each other, but they are not intended to be used together.

`GFX_XOR_DRAW`

This mode draws using a combination of the Pen/Brush colors and the current screen colors. The colors are combined by using the bitwise `XOR` operator. The `XOR` operator is unique (and *very* useful) because it draws in a "reversible" way. For example, if you switch to this drawing mode and draw a line in a certain location, it will be easily visible on almost any background. And if you then *re-draw* the same line (using *exactly* the same drawing mode, colors, style, width, screen locations, etc.) the line will be un-drawn and the original background will be restored. This is an extremely useful technique.

The XOR mode will probably not, however, produce the colors that you expect. For example, a red pen will not produce a red line unless the background is black. If the background is white it will produce a cyan line. To really understand the way the colors are combined you should research XOR "bitwise" operations, which are beyond the scope of this document.

It is important to note that if you use a black pen or brush with the XOR draw mode, the results will not be visible. The XOR combination of black and any color is the original color. Generally speaking, the darker the color, the less visible the results of XOR drawing will be. Drawing with white in the XOR mode is visible against any background.

`GFX_XOR_DRAW_INVERSE`

This mode is very similar to the `GFX_XOR_DRAW` mode (just above)

except that it uses colors that are created by talking the XOR combination of the Pen/Brush and screen colors, and then inverting the resulting color with a bitwise NOT operation.

Color-Merge Operations

GFX_MERGE_COLORS

This drawing mode mixes the current Pen/Brush colors with the current screen colors. The resulting colors are a *combination* of the original background and the Pen/Brush. For example, if the screen was blue and a line was drawn with a red Pen, the resulting line would be magenta (red plus blue).

GFX_MERGE_COLORS_INVERSE

This drawing mode is similar to GFX_MERGE_COLORS except that 'inverse" colors are used, i.e. the Pen/Brush and screen colors are combined, and then the bitwise NOT operator is used to create an inverse color. Using the MERGE_COLOR example just above, the resulting line would be "NOT magenta". i.e. green.

GFX_MERGE_COLORS_INV_PEN

This drawing mode is similar to GFX_MERGE_COLORS except that the current screen color is merged with the inverse of the current Pen or Brush.

GFX_MERGE_COLORS_INV_SCRN

This drawing mode is similar to GFX_MERGE_COLORS except that the inverse of the background color is merged with the current Pen or Brush.

Color-Mask Operations

GFX_MASK_COLORS

This mode draws in the colors that are *common* to the Pen/Brush and the screen. For example, if the background is magenta (red plus blue) and you draw with a yellow (red plus green) Pen, the resulting line would be red because that is the color they have in common.

GFX_MASK_COLORS_INVERSE

This mode is similar to MASK_COLOR, but it creates colors by using the colors that are common to the Pen/Brush and the screen, and then using the bitwise NOT operator to create an inverse color. Using the MASK_COLOR example just above, the line would be drawn in "NOT red", i.e. cyan (blue plus green).

GFX_MASK_COLORS_INV_PEN

This mode is similar to MASK_COLOR, but it creates colors by using the colors that are common to the screen and the inverse of the

Pen/Brush.

`GFX_MASK_COLORS_INV_SCRN`

This mode is similar to `MASK_COLOR`, but it creates colors by using the colors that are common to the Pen/Brush and the inverse of the screen.

Misc. Drawing Modes

`GFX_NO_DRAW`

This mode effectively disables the current Pen and Brush. It is not particularly useful but it is a standard Windows drawing mode so Graphics Tools supports it.

Example

```
'Turn on the draw/un-draw mode:
GfxDrawMode GFX_XOR_DRAW
```

See Also

GfxOption, DrawFocus

GfxDroppedFiles

Purpose

Returns the names of files that have been dragged and dropped on a graphics window which has the `GFX_STYLE_DROP_FILES` style.

Availability

Graphics Tools Pro Only

Warning

This function is not supported by Console Tools Plus Graphics.

Syntax

```
sResult$ = GfxDroppedFiles
```

VB: Use the DropFiles event instead of this function.

```
sResult$ = GfxDroppedFilesEx(lWindowNumber&)
```

Parameters

None.

Return Value

If one or more files have been dropped on a graphics window since the last time this function was used, it will return a string containing the name(s) of the file(s). If the string contains two or more file names they will be separated by Carriage Return characters (`CHR$(13)`).

Remarks

If a graphics windows has the `GFX_STYLE_DROP_FILES` style (or the VB AcceptFiles property is True) and if a user drags and drops one or more files onto that window (for example, from Windows Explorer), Graphics Tools will generate a window message:

```
WM_GFX_EVENT + WM_DROPFILES
```

PowerBASIC programs will receive that message in a callback function. Your PB program should respond by using the GfxDroppedFiles function to obtain the name(s) of the file(s). Your program can then display the file(s), or take some other appropriate action.

Visual Basic programs will receive a DropFiles event, which passes the file name(s) to your program as a parameter of the `_DropFiles` event handler. It is not necessary for VB programs to use the GfxDroppedFiles function.

Note that both the window message and VB event both contain information about which window received the files, and the exact point in the graphics window where the drop took place. See **Example** below.

Note also that using this function *clears* the file name(s). If no files have been dropped on a graphics window since the last time this function was used (or since the program was started) it will return an empty string.

Example

```
'PowerBASIC DDT code:
'This code must be placed in a DDT CALLBACK function.
If CbMsg = %WM_GFX_EVENT + %WM_DROPFILES Then
      MsgBox "The following file(s) were dropped on" + $CR+ _
             "graphics window #" + _
             Format(HiWrd(CbLParam)) + _
             " at location " + _
             Format$(LoWrd(CbLParam)) + "," + _
             Format$(HiWrd(CbLParam)) + ":" + $CR + $CR + _
             GfxDroppedFiles
End If

'PowerBASIC SDK-style code:
'This code must be placed in a window callback function.
If wMsg = %WM_GFX_EVENT + %WM_DROPFILES Then
      MsgBox "The following file(s) were dropped on" + $CR+ _
             "graphics window #" + _
             Format$(HiWrd(wParam)) + _
             " at location " + _
             Format$(LoWrd(lParam)) + "," + _
             Format$(HiWrd(lParam)) + ":" + $CR + $CR + _
             GfxDroppedFiles
End If

Visual Basic code:
Private Sub GfxWindow*_DropFiles(X, Y, FileList)
      MsgBox "The following files were dropped" + vbCr + _
             "at location " + _
             Format$(X) + "," + _
             Format$(Y) + ":" + vbCr + vbCr + _
             FileList
End Sub
```

See Also

Window Styles
Window Messages

GfxFont (Font properties)

Purpose

Creates a font that can be used by the DrawTextBox and DrawTextRow functions.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFont(sFontName$, _
                   lWidth&, _
                   lHeight&, _
                   lWeight&, _
                   lColor&, _
                   lEffects&, _
                   dpAngle#)
```

VB: Use the various Font properties instead of the GfxFont function.

```
lResult& = GfxFontEx(lWindowNumber&, _
                     sFontName$, _
                     lWidth&, _
                     lHeight&, _
                     lWeight&, _
                     lColor&, _
                     lEffects&, _
                     dpAngle#)
```

(See Syntax Options)

Parameters

sFontName$

The "face name" of the font, such as "Arial", "Courier New", "System", or "Wingdings". (The font name parameter is not case-sensitive.) You must use the *exact* name of a font that is currently available on the computer, i.e. installed on the computer at *runtime*. When using certain fonts you may be required to append a "character set number" to the font name parameter. See **Remarks** below for details.

lWidth& and *lHeight&*

The width and height of the font. Keep in mind that these values are *scaled* (see **Remarks** below) and that Windows will not accept all width/height combinations. If you use zero (0) for these parameters, Windows will use the font's default size.

lWeight&

The "boldness" of the font, from zero (0) to nine (9). You may also use the standard Windows numbering system, from 100 to 900. The larger the number, the bolder the font. If you use zero, Windows will use the font's default weight. *Note that not all fonts support nine different weights.* See **Remarks** for additional information.

lColor&

The color in which the font will be displayed, between zero (black) and

MAXCOLOR (white). You can optionally use the HSW function for this parameter, as shown in the second **Example** below.

lEffects&

This parameter is used to specify certain font-based special effects like italics and underlining. See **Remarks** below for a list of valid values. (For effects like shadows, borders, and edges, see the DrawTextRow function.)

dpAngle#

The angle at which the text will be displayed, between zero (0) and plus-or-minus 360 degrees. (If you use the GfxOption GFX_USE_RADIANS function, you must specify this value in radians instead of degrees. This function is *not* affected by the GfxOption GFX_BASE_ANGLE function.) Positive *dpAngle#* values are used to specify a *counter-clockwise* rotation of the text, and negative numbers rotate the text clockwise. (Actually, values up to 999,999 degrees can be used, but they are automatically normalized into the 0-360 range.) Note that this is a floating-point parameter, so fractional values can be specified. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font was created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested. See **Remarks** below.

Remarks

The first thing you need to know about "Windows fonts" is that *Windows* controls them. You can *request* that Windows create a font using certain parameters, but it will not always comply. (All Windows applications must deal with this fact, and it does not usually cause serious problems for a program.)

We recommend that, if possible, you use TrueType fonts with the GfxFont and GfxFontName functions. TrueType fonts can be scaled to virtually any size, so if you use them Windows is more likely to create exactly the font that you request. If you use *non*-TrueType fonts, your selection of height/width/weight combinations will be relatively limited. If you use values that Windows doesn't support, it will substitute default values. *And it will not report an error when it does this.*

Common TrueType fonts include Arial, Courier New (*not* Courier), Symbol, and Wingdings. Those fonts *usually* exists on all Windows computers, but it *is* possible for them to be deleted or replaced. Beyond those four fonts, Windows 95, 98, ME, NT, and 2000 all provide different selections of TrueType fonts, so you must be careful to use a font that you are certain will be available at runtime. If you use a font that is not available, Windows will substitute a default font. If you want to make absolutely sure that a particular font is used, you should secure the necessary copyright and distribution rights, and distribute the font with your application.

To be clear, the font must be installed only if text is to be *created* at runtime. It is possible, for example, to draw and save a bitmap using a font, and then to display that bitmap on *any* computer, even if the font is not installed on that computer. So if you want to display text in a font that may or may not be available at runtime, you might consider using a bitmap instead of drawing the text at runtime.

Font Scaling

It is important to keep in mind that Graphics Tools scales the font-width and font-height values that you use, just as it scales virtually all drawing operations.

Because of this, Graphics Tools Font Size numbers are *not* "point sizes" in the normal sense. You should think of the *lWidth&* and *lHeight&* parameters as using Drawing Units, although this is not exactly correct.

Graphics Tools Font Size numbers are scaled so that a "square" font size of 64x64 (for example) will produce a normally proportioned font. The actual characters will not look square. The height and width of the font will be determined by Windows on a *character by character* basis. Note the difference in the widths of the letters I, T, and W.

A 64x64 font will not usually produce characters that are 64 Drawing Units tall. You must take into account the space that is reserved above and below each character, such as the space that is normally placed between rows of text.

Font Sizes

If you create a font using 50 and 50 for the width and height parameters, that might allow your program to display ten rows of text in a graphics window of a certain size. If you were to then run your program on a computer with a different screen resolution (say, 640x480 instead of 1024x768) Graphics Tools would create a font that was scaled so that it would still allow ten rows of text to be displayed in the much-smaller graphics window. *But...*

Remember, Windows does not always create fonts exactly as requested, we recommend that you test your programs using a variety of screen resolutions.

If you use zero for the *lHeight&* parameter, Windows will use the default font height.

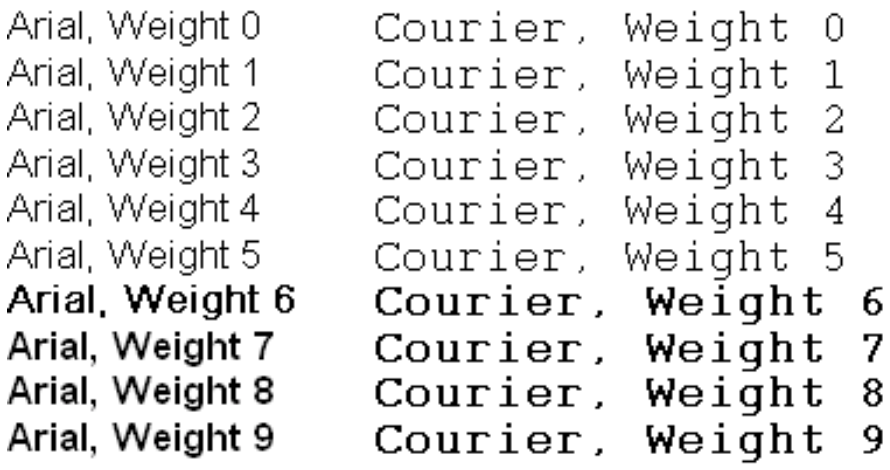
If you use a positive value for the *lHeight&* parameter, Windows will create a font using the value that you specify for the *cell* height. If you use a negative *lHeight&* value, Windows will create a font with that *character* height. (This is a subtle difference that rarely matters, but Windows does provide this functionality if you need it.)

If you use zero for the *lWidth&* parameter, Windows will use the width that best matches the font height.

You must use zero or a positive value for the *lWidth&* parameter.

Font Weights

Windows allows you to specify nine (9) different weight values, from 1 to 9, but not all fonts support nine different sizes. In fact, most fonts support only two or three. Here are a couple of common examples...



Note that the Arial font is actually available in three different weights, and Courier in only two. *This is a limitation of Windows* and it is the primary reason that most word processing programs support only "normal" and "bold" weights. Graphics Tools gives you access to all of the weights that each font supports, but we are not aware of any Windows fonts that actually provide nine different weights.

Note also that the Arial, Weight 6 line is actually *longer* than either Weight 5 or 7. This is perfectly normal, and a side-effect of the way in which Windows makes fonts bolder.

Special Effects

The valid values for the *lEffects&* parameter are...

> `FONT_ITALIC` produces an italicized (i.e. slightly slanted) font.
>
> `FONT_UNDERLINE` produces an underlined font.
>
> `FONT_STRIKEOUT` produces a font with a "strikeout" line drawn horizontally through the characters.

It is possible to combine these values. For example, using `FONT_ITALIC OR FONT_UNDERLINE` would produce a font that was both italicized and underlined.

Displaying Text At An Angle

If you specify a *dpAngle#* value of zero (0), text will be drawn along an imaginary line that starts at the LPR and extends to the right, i.e. toward 3 o'clock or "east". (Notice that the *default* text angle is therefore 180 degrees, not zero degrees.)

If you specify a nonzero *dpAngle#* value, the imaginary line will still start at the LPR, but the line will be rotated counter-clockwise by the number of degrees that you specify. For example, if you use a value of 45 for *dpAngle#*, the right end of the imaginary line would be "lifted" until the angle was 45 degrees from the horizontal. If you were to use negative 45 degrees (`-45`) for this parameter, the line would be rotated clockwise by the same amount, and the right end of the line would be "lowered" 45 degrees.

As you'll see, one of the most complex operations that Graphics Tools must perform is drawing text at an angle. If you are using a graphics window that is proportioned normally (i.e. that is not greatly compressed or stretched either horizontally or vertically) you won't have many problems at all. But if you aren't, some difficulties will probably be encountered.

For example, let's say that you are using a font that was created with a width of 50 and a height of 50 Drawing Units. In a normally-proportioned graphics window that would create a relatively normal-looking font with typical characters like "T" that are roughly twice as tall, on average, as they are wide. If you were to then create a graphics window that was very narrow -- compressed or "squished" *horizontally* -- Graphics Tools would compensate by changing the font so that it, too, was compressed horizontally. Even though you used 50 and 50, the characters would appear to be very thin and tall, just like the window, so roughly the same number of characters could be displayed in the window.

If you were to then rotate the text by 45 degrees, the text would *not* be scaled correctly for the graphics window. The very tall, thin letters would be drawn along a 45-degree line, but their height and width would be the same as if they were printed without rotation. (Windows can only compress or stretch fonts horizontally or vertically. It can't do "diagonal compression".) And if the same text was then printed at a 90 degree angle, they would appear to be very badly scaled. They would, in effect, be stretched *horizontally* while everything else in the graphics window would be stretched vertically.

Again, this only becomes an issue if **1)** the horizontal and vertical scaling values of the graphics window are very different from each other, and **2)** you need to print text at an angle. If the window is greatly stretched or compressed along one axis or the other, problems are likely and you may need to write additional code to compensate for this type of effect.

If you have to use a graphics window that is compressed or stretched, we recommend that you stick with fonts that are relatively "square" -- which have equal width and height values -- to minimize the unpleasant side effects.

Character Sets

Most fonts do not require them, but when you use *certain* fonts you will be required to append a "character set number" to the face name. For example, the standard Windows Wingdings font must be specified like this:

```
Wingdings/2
```

The `/2` tells Windows that Wingdings is a "symbol font". Here are the character set numbers that are common to most versions of Windows:

```
ANSI          0
DEFAULT       1
SYMBOL        2
SHIFTJIS    128
HANGEUL     129
HANGUL      129
CHINESEBIG5 136
OEM         255
```

Other numbers may also be recognized, depending on the version of Windows that is being used. If you do not get the results that you expect when you use a particular font with Graphics Tools, you should try using a different character set number.

Generally speaking, you should try to avoid the `DEFAULT` and `OEM` character sets because they depend on the system's runtime configuration, and as a result they are likely to produce different results on different systems.

If you do not specify a character set by appending a number to the face name, Graphics Tools will automatically use the ANSI character set (0), which is the most common. Other than that, Graphics Tools does not automatically assign character set numbers because fonts can exist in two or more character sets, and the use of the correct number is *necessary* to distinguish between them.

Example

```
'Create an Arial font that is approximately
'50 drawing units tall, very bold, bright blue,
'underlined, and printed at a 45 degree angle
'from the horizontal...

GfxFont "Arial", 50, 50, 9, HIBLUE, FONT_UNDERLINE, 45

'Create a Courier New font using the Windows default
'parameters, and using the HSW function to create a
'custom color...

GfxFont "Courier New" 0, 0, 0, HSW(100,255,0), 0, 0
```

See Also

GfxFontAngle, GfxFontColor, GfxFontEffects, GfxFontHeight, GfxFontName, GfxFontWeight, GfxFontWidth

GfxFontAngle (FontAngle property)

Purpose

Changes the angle at which the graphics window font is drawn, without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function or property.)

Availability

Graphics Tools Standard and Pro

Warning

Windows allows only TrueType fonts to be displayed at an angle. Using this function with non-TrueType fonts will have no effect.

Syntax

```
lResult& = GfxFontAngle(dpAngle#)
```

VB property: *GfxWindow**.FontAngle = dpAngle#

```
lResult& = GfxFontAngleEx(lWindowNumber&, _
                          dpAngle#)
```

(See Syntax Options)

Parameters

dpAngle#

The angle at which the text will be displayed, between zero (0) and plus-or-minus 360 degrees. (If you use the GfxOption GFX_USE_RADIANS function, you must specify this value in radians instead of degrees. This function is *not* affected by the GfxOption GFX_BASE_ANGLE function.) Positive *dpAngle#* values are used to specify a *counter-clockwise* rotation of the text, and negative values rotate the text clockwise. (Actually, values up to 999,999 degrees can be used, but they are automatically normalized into the 0-360 range.) Note that this is a floating-point parameter, so fractional values can be specified.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested. See GfxFont for more information about why Windows may not create the font that you request.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontAngle function, we strongly recommend that you read the GfxFont entry.

Example

```
'change the font angle to 45 degrees...
GfxFontAngle  45
```

See Also

GfxFontColor, GfxFontEffects, GfxFontHeight, GfxFontName, GfxFontWeight, GfxFontWidth

GfxFontColor (FontColor property)

Purpose

Changes the color in which the graphics font is displayed, without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

NOTE: The GfxFontColor value is ignored if a Gradient Brush is being used to fill text.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontColor(lColor&)
```

VB property: ```GfxWindow*.FontColor = lColor```

```
lResult& = GfxFontColorEx(lWindowNumber&, _
                          lColor&)
```

(See Syntax Options)

Parameters

lColor&

The color in which the font will be displayed, between zero (black) and MAXCOLOR (white). You can optionally use the HSW function for this parameter.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontColor function, we strongly recommend that you read the GfxFont entry.

Example

```
GfxFontColor  HIMAGENTA
```

See Also

GfxFontAngle, GfxFontEffects, GfxFontHeight, GfxFontName, GfxFontWeight, GfxFontWidth

GfxFontEffects (FontUnderline, FontItalic, FontStrikeout properties)

Purpose

Changes the effects that are used when displaying the current font, without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontEffects(lEffects&)
```

VB properties:

```
GfxWindow*.FontItalic
GfxWindow*.FontUnderline
GfxWindow*.FontStrikeout
```

```
lResult& = GfxFontEffects(lEffects&)
```

(See Syntax Options)

Parameters

lEffects&

This parameter is used to specify certain font-based special effects like italics and underlining. See **Remarks** below for a list of valid values. (For effects like shadows, borders, and edges, see the DrawTextRow function.)

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The valid values for the *lEffects&* parameter are...

`FONT_ITALIC` produces an italicized (i.e. slightly slanted) font.

`FONT_UNDERLINE` produces an underlined font.

`FONT_STRIKEOUT` produces a font with a "strikeout" line drawn horizontally through the characters.

It is possible to combine these values. For example, using `FONT_ITALIC OR FONT_UNDERLINE` would produce a font that was both italicized and underlined.

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontEffects function, we strongly recommend that you read the GfxFont entry.

Example

```
GfxFontEffects  FONT_ITALIC
```

See Also

GfxFontAngle, GfxFontColor, GfxFontHeight, GfxFontName, GfxFontWeight, GfxFontWidth

GfxFontHeight (FontHeight property)

Purpose

Changes the height of the graphics font, usually without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontHeight(lHeight&)

VB property: GfxWindow*.FontWeight = lHeight&

lResult& = GfxFontHeightEx(lWindowNumber&, _
                           lHeight&)
```

(See Syntax Options)

Parameters

lHeight&

The height of the font. Keep in mind that this values is *scaled* and that Windows will not accept all width/height combinations. If you use zero (0), Windows will use the font's default size.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontHeight function, we strongly recommend that you read the GfxFont entry.

Example

```
GfxFontHeight  20
```

See Also

GfxFontAngle, GfxFontColor, GfxFontEffects, GfxFontName, GfxFontWeight, GfxFontWidth

GfxFontName (FontFaceName property)

Purpose

Changes the "face name" of the graphics font, usually without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

This function can also be used to obtain the current face name, without changing it.

Availability

Graphics Tools Standard and Pro

Warning

Visual Basic programmers who use the "Direct To DLL" method (i.e. who do *not* use the Graphics Tools OCX control) should be sure to read the section of this document titled "Functions Which Return Strings" in Four Critical Steps For Visual Basic Programmers (DLL).

Syntax

```
sResult$ = GfxFontName(sName$)
```

VB property: `GfxWindow*.FontFaceName = sName$`

```
sResult$ = GfxFontNameEx(lWindowNumber&, _
                         sName$)
```

(See Syntax Options)

Parameters

sName$

The "face name" of the font you want to use, such as "Arial", "Courier New", "System", or "Wingdings". (The font name parameter is not case-sensitive.) You must use the *exact* name of a font that is currently available on the computer, i.e. installed on the computer at *runtime*. When using certain fonts you may be required to append a "character set number" to the font name parameter. See GfxFont for details.

If you use a question mark ("?") for this parameter, this function will return the face name of the font that your program most recently told Graphics Tools to use, without changing the font.

If you use an empty string ("") for this parameter, this function will return the *actual* current face name, without changing the font. Because of the way Windows creates fonts, this may or may not be the name of the font that you told Windows to use.

Return Value

Unless "" or "?" is used for the *sName$* parameter (see just above), the return value of this function will be an empty string.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using this function, we strongly recommend that you read the GfxFont entry.

Examples

```
GfxFontName  "Arial Bold"

GfxFontName  "Wingdings/2"
```

See Also

GfxFontAngle, GfxFontColor, GfxFontEffects, GfxFontHeight, GfxFontWeight, GfxFontWidth

GfxFontSize

Purpose

Simultaneously changes the current font's width and height.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontSize(lSize&)
```

VB: Use the FontWidth and FontHeight properties instead of the GfxFontSize function.

```
lResult& = GfxFontSizeEx(lWindowNumber&, _
                         lSize&)
```

Parameters

lSize&

The width and height of the font you wish to create.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontHeight function, we strongly recommend that you read the GfxFont entry.

Example

```
GfxFontSize 100
```

See Also

GfxFontWidth, GfxFontHeight

GfxFontWeight (FontWeight property)

Purpose

Changes the weight (i.e. boldness) of the graphics font, without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontWeight(lWeight&)

VB property: GfxWindow*.FontWeight = lWeight&

lResult& = GfxFontWeightEx(lWindowNumber&, _
                           lWeight&)
```

(See Syntax Options)

Parameters

lWeight&

The "boldness" of the font, from zero (0) to nine (9). You may also use the standard Windows numbering system, from 100 to 900. The larger the number, the bolder the font. If you use zero, Windows will use the font's default weight. *Note that not all fonts support nine different weights.* See **Remarks** and GfxFont for additional information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font, including a detailed description of Font Weight. Because the various font parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontWeight function, we strongly recommend that you read the GfxFont entry.

Example

```
'Make the current font as bold as possible
GfxFontWeight  9
```

See Also

GfxFontAngle, GfxFontColor, GfxFontEffects, GfxFontHeight, GfxFontName, GfxFontWidth

GfxFontWidth (FontWidth property)

Purpose

Changes the width of the graphics font, usually without otherwise affecting the font. (If you are going to change two or more font parameters at the same time, it is much more efficient to use the GfxFont function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxFontWidth(lWidth&)
```

VB property: *`GfxWindow*`*`.FontWidth`

```
lResult& = GfxFontWidthEx(lWindowNumber&,_
                          lWidth&)
```

(See Syntax Options)

Parameters

lWidth&

The width of the font. If you use zero (0), Windows will use the font's default size. Keep in mind that these values are *scaled* and that Windows will not accept all width/height combinations.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested font is created without errors. Otherwise, a Graphics Tools Error Code will be returned. Please note that SUCCESS does not *necessarily* mean that the font was created exactly as you requested.

Remarks

This document's GfxFont entry contains a complete discussion of the various factors that must be considered when creating a font. Because the various parameters may interact, and because a general understanding of fonts is very useful when you are using the GfxFontWidth function, we strongly recommend that you read the GfxFont entry.

Example

```
GfxFontWidth  75
```

See Also

GfxFontAngle, GfxFontColor, GfxFontEffects, GfxFontHeight, GfxFontName, GfxFontWeight

GfxIsOpen

Purpose

Returns a True or False value indicating whether or not a graphics window is currently open.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxIsOpen(lWindowNumber&)
```

VB: `lResult& = GfxIsOpen(`*`GfxWindow*`*`.ID)`

Parameters

lWindowNumber&

The number of a graphics window. (VB users: see ID.)

Return Value

The value of *lResult&* will be True (negative one) if the graphics window is open, or False (zero) if it is not.

Remarks

This function will return True (negative one) if the graphics window is open, or False (zero) if it is not. The *visibility* of the graphics window is not considered, just its availability for drawing operations.

Example

```
If GfxIsOpen(2) Then
      'It is possible to draw in graphics window #2
End If
```

See Also

The Graphics Window

GfxLPR

Purpose

Returns the current X and Y location of the LPR.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxLPR(lXPos&, _
                  lYPos&)
```

VB: use `GfxLPREx`

```
lResult& = GfxLPREx(lWindowNumber&, _
                    lXPos&, _
                    lYPos&)
```

(See Syntax Options)

Parameters

lXPos& and *lYPos&*

These are output parameters. You should usually use *variables* for these parameters. The values to these variables will have no effect on the operation of this function, or on the LPR. The GfxLPR function will *assign* values to the variables which correspond to the X and Y locations of the LPR.

Return Value

The value of *lResult&* will be SUCCESS (zero) if Windows is able to determine the location of the LPR, or a Graphics Tools Error Code if it can't. The useful values that are provided by this function can be obtained by examining the *lXPos&* and *lYPos&* parameters after the function returns SUCCESS.

Remarks

This function obtains the current location of the LPR directly from Windows, and resets the Graphics Tools internal values to match Windows.

You can examine the Graphics Tools internal values by using the GfxX and GfxY functions. Unless you have used Windows graphics API functions to change the LPR, GfxLPR should always return exactly the same values as GfxX and GfxY, but using GfxLPR is significantly *slower*.

If you use the UserDraw function to tell Graphics Tools than its internal LPR values are no longer valid (because you have used an API function), you can (later) use the GfxLPR function to re-activate it. You could simply use this code...

```
GfxLPR 0,0
```

...to tell Graphics Tools to calculate the current location of the LPR and reset its internal tracking system. Since you don't really care about the values, in this case it is not necessary to use variables for the GfxLPR parameters.

Example

```
If GfxLPR(lXPos&, lYPos&) = SUCCESS Then
      'The lXPos& and lYPos& variables now
      'contain the location of the LPR.
End If
```

See Also

The LPR

GfxLoc

Purpose

Saves the current LPR, or restores the LPR to a previously saved location, or restores only the X or Y component of a saved location, or provides the X or Y value of a saved location without changing the LPR, or clears a range of saved locations.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxLoc(lAction&, _
                  lWhich&)
```

VB method: *GfxWindow**.GfxLoc

```
lResult& = GfxLocEx(lWindowNumber& _
                    lAction&, _
                    lWhich&)
```

(See Syntax Options)

Parameters

lAction&

Determines which action the GfxLoc function will perform. See **Remarks** below for a list of valid values.

lWhich&

Determines the location number(s) upon which the action will be performed, between one (1) and 256 (or the number that you specified with GfxOption GFX_MAX_SAVED_LOCS, up to 32,767).

Return Value

The value of *lResult&* will be SUCCESS if the operation is performed or, in the cases of LOC_X_VALUE and LOC_Y_VALUE (see below) the return value of the function will be the requested X or Y value.

ERROR_BAD_PARAMETER_VALUE will be returned if an invalid *lWhich&* or *lAction&* value is used.

Remarks

Normally, you must use a value between one (1) and 256 for the *lWhich&* parameter. (It is possible to increase the location-storage capacity from 256 up to a value as large as 32,767 by using the GfxOption GFX_MAX_SAVED_LOCS function. The storage of each location requires four (4) bytes of memory.)

If *lAction&* is LOC_SAVE the current LPR is saved as location number *lWhich&*. For example...

```
GfxLoc LOC_SAVE, 10
```

...would save the current LPR as location number 10.

If *lAction&* is LOC_RESTORE the LPR is changed back to saved location number *lWhich&*. For example...

```
GfxLoc LOC_RESTORE, 10
```

...would return the LPR to the location that was saved by the first example.

If *lAction&* is LOC_X_RESTORE or LOC_Y_RESTORE, only the X or Y component of the LPR will be changed. The other component of the LPR will not be affected.

If *lAction&* is LOC_X_VALUE or LOC_Y_VALUE, the LPR will not be changed, and the requested saved value will be returned as *lResult&* (see **Syntax** above).

If *lAction&* is LOC_CLEAR all of the previously saved locations (starting with *lWhich&*) will be reset to the "home" location of 0,0.

Example

See **Remarks** above.

See Also

The LPR

GfxMetrics

Purpose

Provides "metric" or "measurement" information about the graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxMetrics(lWhich&)
```

VB method: *GfxWindow**.GfxMetrics

```
lResult& = GfxMetricsEx(lWindowNumber&, _
                        lWhich&)
```

Parameters

lWhich&

Determines which measurement will be returned by the function. See **Remarks** below for a list of valid values.

Return Value

If a valid *lWhich&* value is used, the value of *lResult&* will be the requested measurement. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can return the following graphics windows "metrics"...

`GFX_AUTOSHOW`

If the graphics window is currently "frozen" (via the GfxWindow `GFX_FREEZE` option), this value will be zero (0). Otherwise it will be True (`-1`).

`GFX_BKGD_COLOR`

The current background color, as set with the GfxBkgdColor function.

`GFX_BKGD_MODE`

The current background mode, as set with the GfxBkgdMode function.

`GFX_BRUSH_COLOR`, `GFX_BRUSH_HATCH`, and `GFX_BRUSH_STYLE`

The color, hatch, and style settings of the current Brush. If the current brush is a BrushBitmap, the style value will be three (3) and the other two values will not be useful.

`GFX_DRAWING_HEIGHT` and `GFX_DRAWING_WIDTH`

The current vertical (Y) and horizontal (Y) sizes of the graphics window, in *pixels*. See The Graphics Window. Compare `GFX_VISIBLE_HEIGHT` and `GFX_VISIBLE_WIDTH`.

GFX_FONT_ANGLE

The current font's angle, times 1000. In other words, the return value of this function can be divided by 1000 to obtain a floating point value that corresponds to the angle at which the DrawTextBox and DrawTextRow function are currently displaying text.

GFX_FONT_COLOR, GFX_FONT_EFFECTS, GFX_FONT_HEIGHT, GFX_FONT_WEIGHT, and GFX_FONT_WIDTH

The color, effects, height, weight, and width values that were used to *create* the current font. By comparing these values to the values that are returned by the rest of the GFX_FONT_ constants (see just below) you can determine whether or not Windows used exactly the values that you requested. (See GfxFont for more information about creating Windows fonts, and why Windows does not always use the values that you supply.)

GFX_FONT_ACTUAL_HEIGHT

The height of the graphics font that was actually created by Windows. Compare this to GFX_FONT_HEIGHT above.

GFX_FONT_AVE_WIDTH

The *average* width of the characters that are produced by the current graphics font. For fixed-pitch ("monospaced") fonts such as Courier New, this will be the width of all of the characters. Compare this value to the value returned by GFX_FONT_WIDTH (see above) to determine whether or not Windows used exactly the width value that you requested.

GFX_FONT_MAX_WIDTH

The width of the widest character (typically the capital letter W) that is produced by the current font.

GFX_FONT_ACTUAL_WEIGHT

The weight of the current font, as reported by Windows. Compare this value to the value returned by GFX_FONT_WEIGHT (see above) to determine whether or not Windows used the weight value that you requested. (If you used a value between one (1) and nine (9) it is *normal* for windows to return a value between 100 and 900, since that is how Windows defines font weights internally.)

GFX_FONT_ASCENT

The "ascent" of the characters of the current graphics font, i.e. the number of Drawing Units above the base line.

GFX_FONT_DESCENT

The "descent" of the characters of the current graphics font, i.e. the number of Drawing Units below the base line.

GFX_FONT_INT_LEADING

The "internal leading" value of the current graphics font, i.e. the amount of "blank space" between the tops of most characters and the top edge of the font. Accent marks and other diacritical characters can be displayed in this area.

GFX_FONT_EXT_LEADING

The "external leading" value of the current graphics font, i.e. the amount of blank space that Windows displays between rows of text. Nothing is printed in this area.

GFX_FONT_ASPECT_X and GFX_FONT_ASPECT_Y

The ratio of these two values is used to indicate the aspect ratio of the device for which the font face was originally designed.

GFX_FONT_CHAR_FIRST

The ASCII value of the first character that is defined by the font. Not all fonts support all characters, so this value will not always be zero (meaning CHR$(0)).

GFX_FONT_CHAR_LAST

The ASCII value of the last character that is defined by the font. Not all fonts support all characters, so this value will not always be 255 (meaning CHR$(255)).

GFX_FONT_CHAR_DEFAULT

The ASCII value of the character that will be displayed if you attempt to display a character that is not defined by the font.

GFX_FONT_CHAR_BREAK

The ASCII value of the character that will be used to define word breaks for text justification.

GFX_FONT_ITALIC, GFX_FONT_UNDERLINED, and GFX_FONT_STRIKEOUT

If one or more of these values are nonzero, the current graphics font has the corresponding properties.

GFX_FONT_PITCH_FAMILY

The *first* four (4) bits of this value describe the most basic characteristics of the current graphics font. If the first bit is set, the font is a Fixed Pitch font. If the second bit is set, it is a Vector font. If the third bit is set, it is a TrueType Font. If the fourth bit is set, it is a Device Font. Note that it is possible for two or more of these types to be combined. For example, a TrueType Font can also be a Fixed Pitch font.

The *last* four (4) bits of this value tell you the Font Family to which the font belongs. Use a bitmask (AND &F0) to eliminate the four low-order bits, and

the result will be a numeric value that corresponds to the Font Family.

GFX_FONT_OVERHANG

The extra width (per string) that is added to "synthesized" fonts, such as when a plain font is modified by Windows to create an italic or bold version of the font. (Some fonts are *designed* as italic and/or bold. Overhang only applies to bold and italic fonts that are synthesized by Windows.)

When synthesizing fonts, Windows may be required to add a small amount of width to a string. For example, Windows makes a font "bold" by expanding the spacing of each character, and then "overstriking" by a certain offset value. And Windows italicizes a font by "shearing" the original font. In both cases, there will be an overhang past the end of the original string. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the distance that the top of the font is sheared past the bottom.

GFX_FONT_CHARSET

The character set of the current graphics font.

GFX_FRAME_OPTIONS and GFX_FRAME_TYPE

These Graphics Tools Version 1 values are no longer used. See the DrawFrame function.

GFX_HORIZ_LOC

The current horizontal (X) location of the LPR.

GFX_HORIZ_SCALE

The current number of horizontal (X) Drawing Units. See The Graphics Window.

GFX_NONCLIENT_HEIGHT and GFX_NONCLIENT_WIDTH

The height and width, in pixels, of the "client area" of the graphics window, i.e. the area in which drawing can take place. Unless a graphics window has borders (see Window Styles) these values will be the same as GFX_VISIBLE_HEIGHT and GFX_VISIBLE_WIDTH.

GFX_NONCLIENT_LEFT and GFX_NONCLIENT_TOP

The left and top coordinates, in pixels, of the "client area" of the graphics window, i.e. the area in which drawing can take place, relative to the top-left corner of the graphics window. Unless a graphics window has borders (see Window Styles) these values will both be zero.

GFX_PEN_COLOR, GFX_PEN_WIDTH, and GFX_PEN_STYLE

The color, width, and style settings of the current Pen.

GFX_SCALE_MODE

This Graphics Tools Version 1 value is no longer supported. Graphics Tools version 2 always uses scaling to provide a drawing World.

GFX_SCROLL_LIMIT_H and GFX_SCROLL_LIMIT_V

The horizontal and vertical scroll limits of the graphics window. Unless the window has been made smaller than its original size, these values will be zero.

GFX_SCROLL_POS_H and GFX_SCROLL_POS_V THEN

The current horizontal and vertical scroll bar positions. If a graphics window does not have scroll bars, or if the scroll bars are at a minimum value, zero (0) will be returned.

GFX_VERT_LOC

The current vertical (Y) location of the LPR.

GFX_VERT_SCALE

The current number of vertical (Y) Drawing Units. See The Graphics Window.

GFX_VISIBLE_HEIGHT and GFX_VISIBLE_WIDTH

The current visible height and width of the graphics window, in pixels. Unless the graphics window has been resized, these values will be identical to GFX_DRAWING_HEIGHT and GFX_DRAWING_WIDTH. Note that the GFX_VISIBLE_ values do *not* include a graphics window's borders and caption, if any. To calculate the actual "physical" size of a graphics window, add the GFX_VISIBLE_WIDTH and _HEIGHT values to the GFX_NONCLIENT_WIDTH and _HEIGHT values.

GFX_WINDOW_EX_STYLE

The graphics window's Extended Window Styles. Also see GFX_WINDOW_STYLE below.

GFX_WINDOW_NUMBER

The graphics window's window number, between zero (0) and the maximum number of windows that Graphics Tools can create.

GFX_WINDOW_STATE

Either GFX_SHOW if the graphics window is visible, or GFX_HIDE (zero) if it is not.

GFX_WINDOW_STYLE

The graphics window's basic Window Styles. Also see GFX_WINDOW_EX_STYLE above.

Example

```
lResult& = GfxMetrics(GFX_DRAWING_HEIGHT)
'The lResult& variable now contains the height of
'the current graphics window, in pixels.
```

See Also

GfxOption

GfxMove

Purpose

Moves the graphics window relative to the top-left corner of its parent window (but *only* if the graphics window has the GFX_STYLE_MOVABLE window style.)

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxMove(lLeft&, _
                   lTop&)
```

VB: Use the standard GfxWindow*.Left and GfxWindow*.Top properties to move a graphics window, not these functions.

```
lResult& = GfxMoveEx(lWindowNumber&, _
                     lLeft&, _
                     lTop&)
```

(See Syntax Options)

Parameters

lLeft& and *lTop&*

The distance from the left and top of the parent window, in pixels, where the graphics window should be moved. You can use %GFX_SAME for one of these parameters if you want to move the window along one axis without affecting the other.

Return Value

This function will return SUCCESS if the window is moved without errors, or a Graphics Tools Error Code if it is not.

Remarks

NOTE: Unless you use the GFX_STYLE_MOVABLE window style, graphics windows cannot be moved. This style can be used when a graphics window is first created with OpenGfx, or it can be added (or removed) at any time by using the GfxWindow function.

Example

```
GfxMove 100,100
```

See Also

GfxResize

GfxOption

Purpose

Changes the value of a graphics window option. Options are used to control a wide variety of functions.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxOption(lOption&, _
                     lValue&)
```

VB method: *GfxWindow**.GfxOption

```
lResult& = GfxOptionEx(lWindowNumber&, _
                       lOption&, _
                       lValue&)
```

(See Syntax Options)

Parameters

lOption&

The option that you wish to change. See **Remarks** below for a list of valid values.

lValue&

The numeric value that the option should be given. See **Remarks** below for a list of valid values for each *lOption&*.

Return Value

The value of *lResult&* will be SUCCESS if the option is changed to the value that you specified, or a Graphics Tools Error Code if it is not.

Remarks

IMPORTANT NOTE: Most GfxOption settings can be changed on a window-by-window basis, but some settings affect *all of the graphics windows in a program* at the same time. values that say (GLOBAL SETTING) below affect all windows.

The following values can be used for *lOption&*:

```
GFX_3D_HIGHLIGHT                (GLOBAL SETTING)
```

This option controls the amount of highlighting that Graphics Tools adds when a clear pen is used to draw a Three-Dimensional Figure. The default value is `+200`.

```
GFX_ANIMATE_BKGD                (per window)
```

This option affects the way in which the graphics window's background is re-painted between the frames of animated icons and animated cursors. See AnimateCursor or AnimateIcon for more information.

GFX_BASE_ANGLE (GLOBAL SETTING)

This option controls the Graphics Tools definition of "zero degrees" when dealing with most angles. The default value is nine (9), which means that zero degrees is defined as "west" or "nine o'clock". If you change this option's value to twelve (12), zero degrees will be defined as "north" or "12 o'clock". A value of three (3) corresponds to "east" or "3 o'clock", and six (6) corresponds to "south" or "6 o'clock". It is important to keep in mind that most but *not all* Graphics Tools angle values are affected by this setting. For example, the angle at which text is printed (see the GfxFontAngle function) is always based on zero degrees meaning "normal, horizontal text", regardless of the value of the GFX_BASE_ANGLE option.

GFX_CONFINE_TO_PARENT

If this option is zero (the default) graphics windows that have the GFX_STYLE_MOVABLE style can be moved anywhere, even if part or all of the graphics window is outside the parent window and therefore not visible. If this option is changed to one (1) it will not be possible to move a graphics window so that any part of the window is outside the parent window.

GFX_CUBE_CAP_COLOR (GLOBAL SETTING)

This option is used to change the color of the "cap" (the top or bottom) of cubes and boxes that are drawn with the DrawCube function. The default value is GFX_AUTO, which tells Graphics Tools to use the current Brush or Gradient.

GFX_CUBE_DEPTH_SCALING (GLOBAL SETTING)

This option is used to change the "depth scaling" that the DrawCube function uses to produce square-looking cubes. The default value is 0.62, which produces good results under most circumstances.

GFX_CUBE_FRONT_COLOR (GLOBAL SETTING)

This option is used to change the color of the "front" (the side closest to the viewer) of cubes and boxes that are drawn with the DrawCube function. The default value is GFX_AUTO, which tells Graphics Tools to use the current Brush or Gradient.

GFX_CUBE_SHADING_LEVEL (GLOBAL SETTING)

This option is used to change the amount by which the DrawCube function shades the sides and bottom of cubes and boxes. The default value is -200.

GFX_CUBE_SIDE_COLOR (GLOBAL SETTING)

This option is used to change the color of the "side" of cubes and boxes that are drawn with the DrawCube function. The default value is GFX_AUTO, which tells Graphics Tools to use the current Brush or Gradient.

GFX_CURSOR_TYPE (per window)

This option does exactly the same thing as the GfxCursor function.

GFX_CYLINDER_CAP_COLOR (GLOBAL SETTING)

This option is used to change the color of the "cap" (the top or bottom) of cylinders that are drawn with the DrawCylinder function. The default value is GFX_AUTO, which tells Graphics Tools to use the current Brush or Gradient.

GFX_CYLINDER_SHADING_LEVEL (GLOBAL SETTING)

This option is used to change the amount by which the DrawCylinder function shades the sides of cylinders. The default value is -200.

GFX_EVENT_MESSAGE (GLOBAL SETTING)

See Window Messages.

GFX_FAST_POLYDRAW

This Graphics Tools Version 1 option is no longer supported. All Graphic Tools polygons are now drawn using the Fast mode.

GFX_FOCUS_RECT_COLOR (GLOBAL SETTING)

This option can be used to change the appearance of the focus rectangle that is drawn around graphics windows that have both a border and the GFX_STYLE_TABSTOP window style. The default value is GFX_AUTO, which tells Graphics Tools to draw focus rectangles in the same way that Windows normally draws them, using a dotted gray line. If you change this option to a Windows color value between zero (0) and MAXCOLOR, Graphics Tools will use a solid line of that color for graphics window focus rectangles. Also see GFX_NO_FOCUS_RECT_COLOR below.

GFX_FONT_ANIMATE_SPEED (per window)

This option controls the speed at which characters are displayed by the DrawTextRow function when the TEXT_ANIMATE effect is used. The default value is 100 milliseconds (1/10 of a second). You can use any value from zero (0) to 5000 milliseconds (5 seconds).

GFX_FONT_SCALE_DISABLE (GLOBAL SETTING)

This option can be used to disable automatic font scaling. If you set this value to a nonzero value, Graphics Tools will ignore the current World and use the literal font size (width and/or height) that you specify.

It is important to note that this setting will not affect the *current* font in any way. This setting has no effect until you tell Graphics Tools to *change* the font size.

GFX_FRAME_OPTIONS and GFX_FRAME_TYPE

These Graphics Tools Version 1 options are no longer needed or supported. See the DrawFrame function.

GFX_HOLE_MARGIN (window zero only)

This option can be used to automatically add one or more pixels to the four edges of the holes that are produced by the DrawHole function. Changing the value of this option affects only the holes that are created *after* the value is changed.

GFX_HORIZ_SCALE (per window)

This option can be used to change the number of horizontal Drawing Units that Graphics Tools uses when scaling all of the various drawing operations, but *only* if the GfxWorld function has not already been used to define the World.

The default value of this option is 1024.

The value of this option can be changed at any time, and will affect all future drawing operations. It will not change the current appearance of the graphics window.

Also see GFX_VERT_SCALE below.

GFX_HSW_POINTS (GLOBAL SETTING)

See The HSW Point System.

GFX_H_SCROLL_BAR_SPEED (per window)

If a graphics window has a horizontal scroll bar, this option controls the distance by which the window is scrolled when the right- or left-arrow key is pressed or the scroll-right or scroll-right button is clicked. It thereby controls the speed at which the window scrolls. The default value is 4 pixels.

GFX_IMAGE_DRAW_MODE (per window)

This option controls the way in which Graphics Tools displays bitmaps and JPEG files that are loaded with the DisplayBitmap, DisplayJpeg, StretchBitmap, StretchJpeg, CropBitmap and CropJpeg functions.

The default value is called SRCCOPY, and it causes the source image to be copied directly into the graphics window. The original contents of the graphics window are "covered up" by the bitmap or JPEG image.

Perhaps the most useful option is the SRCINVERT mode, which uses XOR ("exclusive OR") drawing. If you display an image using the SRCINVERT mode, the pixels of the image will be combined with the pixels of the graphics window using a technique that is *reversible*. The image and the original contents of the graphics window will both be visible, and if you use the SRCINVERT mode to display exactly the same image a second time, in exactly the same location, the image will be "un-drawn" and the original contents of the graphics window will be restored.

You can use any of the following drawing modes to obtain different effects. The necessary values are listed in the PowerBASIC WIN32API.INC file and the Visual Basic WIN32API.TXT file. . 'Source" refers to the image, and

"destination" refers to the graphics window. The NOT, AND, OR, and XOR operators indicate binary operations.

Value	Result
SRCCOPY	source image copied directly to destination
NOTSRCCOPY	NOT source
SRCPAINT	source OR destination
MERGEPAINT	NOT source OR destination
SRCAND	source AND destination
SRCINVERT	source XOR destination
SRCERASE	source AND NOT destination
NOTSRCERASE	NOT source AND NOT destination

The following modes use only the *size* of the source image. The contents of the image (i.e. the colors that it contains) are ignored.

DSTINVERT	NOT destination
BLACKNESS	Black
WHITENESS	White

GFX_IMAGE_FLIP_H (GLOBAL SETTING)

This option controls the horizontal "flipping" of bitmaps and JPEG images by the DisplayBitmap, DisplayJpeg, StretchBitmap, StretchJpeg, CropBitmap and CropJpeg functions.
If you change this option to a nonzero (True) value, the images that are drawn by those functions will be drawn from right-to-left instead of left-to-right, resulting in a "mirror image".

GFX_IMAGE_FLIP_V (GLOBAL SETTING)

This option controls the vertical "flipping" of bitmaps and JPEG images by the DisplayBitmap, DisplayJpeg, StretchBitmap, StretchJpeg, CropBitmap and CropJpeg functions.
If you change this option to a nonzero (True) value, the images that are drawn by those functions will be drawn from bottom to top instead of top to bottom, resulting in an upside-down image.

GFX_IMAGE_LOAD_MODE (GLOBAL SETTING)

This option controls the way Graphics Tools loads bitmaps, cursors, and icons. You can use the following values to produce different effects:

LOAD_MONOCHROME

Converts the bitmap, cursor, or icon to black-and-white.

LOAD_TRANSPARENT

(Please note: The original Microsoft Windows name for this option is LR_TRANSPARENT and while the term "transparent" has been maintained here, it can be misleading because this option provides only the *illusion* of transparency, and only under certain circumstances.) This option tells Windows to determine the color of the first pixel of the bitmap, cursor, or icon, and to replace that color

with the current COLOR_WINDOW color, which is the color that Windows is currently using for the background color of most windows. The COLOR_WINDOW value can be changed by using the Windows Desktop Properties/Appearance dialog. This option only works if the bitmap, cursor, or icon contains a Color Table, which not all do.

The SystemColor function can be used to determine the current value of the COLOR_WINDOW setting.

LOAD_MAP3DCOLORS

This option tells Windows to replace three particular shades of gray (in a bitmap or icon) with colors that are determined by the Windows Desktop Properties/Appearance dialog. Specifically...

"Dark Gray", which is RGB(128,128,128), is replaced with COLOR_3DSHADOW.

"Gray, which is RGB(192,192,192), is replaced with COLOR_3DFACE.

'Light Gray", which is RGB(223,223,223), is replaced with COLOR_3DLIGHT.

The SystemColor function can be used to determine the current values of the various COLOR_ settings.

GFX_IMAGE_SAVE_M0DE

This Graphics Tools Version 1 option is no longer needed or supported. See the SaveGfxMode function.

GFX_JPEG_LIB_VERSION (GLOBAL SETTING)

This option can be used to tell Graphics Tools which JPEG Library it should use to save and load JPEG images. The default value of the GFX_JPEG_LIB_VERSION option is GFXT_JPEG (zero), which tells Graphics Tools to use the Graphics Tools JPEG Library. You can also use INTEL_JPEG, which tells it to use the Intel JPEG Library.

GFX_JPEG_SUBSAMPLING (GLOBAL SETTING)

This options can be used to control the way Graphics Tools saves JPEG images with the Intel JPEG Library. The default value of the GFX_JPEG_SUBSAMPLING option is 411, which tells the Intel Library to use 4-1-1 subsampling. Changing this option to 422 tells it to use 4-2-2 subsampling. Not all JPEG viewers can use 4-2-2 images. This option has no effect on the Graphics Tools JPEG Library.

GFX_LARGEST_ICON

This Graphics Tools Version 1 option is no longer needed or supported. Graphics Tools version 2 can use icons of any valid size.

GFX_MAX_HOLES (GLOBAL SETTING)

This option determines the maximum number of "holes" that can be created with the DrawHole function. The default value is 32, and you may change it to any value between 1 and 1,024. Using a value that is larger than you need results in a small speed penalty. IMPORTANT NOTE: This option cannot be changed while a graphics window is being displayed. You must set this value *before* a graphics window is created.

GFX_MAX_SAVED_LOCS (GLOBAL SETTING)

This option determines the maximum number of locations that can be saved with the GfxLoc function. The default value is 256, and you can change it to any value between 100 and 32,768. Each location that is specified with this option requires 4 bytes of memory, regardless of whether or not it is actually used.

GFX_MODULE_INSTANCE (GLOBAL SETTING)

This option is used to change the "instance handle" value that Graphics Tools uses when accessing resources (such as icons, bitmaps, and cursors) that are embedded in your EXE and DLL files. If your program needs to access resources that are embedded in one or more different executable modules (*not* counting the Graphics Tools Runtime Files), you will need to set this option to the appropriate "instance" value, as provided by the module's WinMain or LibMain function, *before* accessing the resource.

GFX_NO_FOCUS_RECT_COLOR (GLOBAL SETTING)

If a graphics window which has both a border and the GFX_STYLE_TABSTOP window style is given the Windows focus by the user, a focus rectangle will be drawn. When the graphics window does *not* have the focus, *no* rectangle is normally drawn. The GFX_NO_FOCUS_RECT_COLOR option can be used to tell Graphics Tools to draw a rectangle around a graphics window when it does *not* have the focus. The default value is GFX_AUTO, which tells Graphics Tools not to draw a rectangle when the window does not have the focus. If you change this option to a Windows color value between zero (0) and MAXCOLOR, Graphics Tools will use a solid line of that color. Also see GFX_FOCUS_RECT_COLOR above.

GFX_OLE_STRINGS (GLOBAL SETTING)

Graphics Tools functions that require string (non-numeric) parameters normally accept ASCIIZ strings, i.e. nul-terminated ANSI strings. If you are using a computer language such as PowerBASIC that uses OLE (BSTR) strings, Graphics Tools must perform certain extra "string housekeeping" operations.

The normal value of this option is False (zero), meaning that Graphics Tools is expecting ASCIIZ strings. You can set this option to True (any nonzero value) to tell Graphics Tools that you are using a language that supports OLE strings.

VISUAL BASIC USERS: It is *never* necessary for you to change the value of this option. Graphics Tools automatically detects Visual Basic and sets this option's value.

POWERBASIC USERS: It is not *normally* necessary for you to change the value of this option. The GfxT_Std.INC and GfxT_Pro.INC files both include a tiny function called Gfx_OleStrings. When you include one of the INC files in your program, that function will be automatically added to your program. Graphics Tools will see that function as a signal that PowerBASIC is being used, and automatically set the value of this option. If you want to remove that function from the INC file (to reduce your program's size by a few bytes) you should add this line of code to your program:

```
GfxOption GFX_OLE_STRINGS, 1
```

GFX_OVERLAY_MODE (GLOBAL SETTING)

This setting can be used to modify the non-transparent portion of an image that is displayed with the OverlayWindow function. You can use OVERLAY_BLEND25 or OVERLAY_BLEND50 to produce somewhat different effects, or use OVERLAY_TRANSPARENT (the default value) to produce a normal overlay.

GFX_PEN_SCALE_DISABLE (GLOBAL SETTING)

This option can be used to disable automatic pen scaling. If you set this value to a nonzero value, Graphics Tools will ignore the current World and use the literal pen size that you specify.

This option is provided primarily for programs that were developed with Graphics Tools Version 1, which did not support pen scaling.

GFX_PIE_EXPLOSION_DISTANCE (per window)

This option is used to tell Graphics Tools to "explode" pies and wedges, i.e. to move them away from the center of the surrounding ellipse by a certain amount. See DrawWedge and DrawPie.

GFX_PIE_EXPLOSION_OFFSET (per window)

Under certain circumstances, pie and wedge charts that are "exploded" can look somewhat unrealistic. This option can be used to add a vertical offset to the explosion (as opposed to the "angled" offset that GFX_PIE_EXPLOSION_DISTANCE provides). This can help you create a more realistic-looking pie or wedge chart. See DrawWedge and DrawPie.

GFX_QUICK_DRAW (per window)

This option tells Graphics Tools to draw as rapidly as possible, without waiting for each new graphics element to be displayed. This can produce greatly increased drawing speeds, but usually results in the problems described in Refreshing The Display. While the GFX_QUICK_DRAW mode may be useful under certain circumstances, it is usually better to use the GfxWindow freeze/unfreeze method to speed up drawing operations.

GFX_REVERSE_ANGLES (GLOBAL SETTING)

This option reverses the way in which Graphics Tools interprets angles. If you set this option to a nonzero value, angles will be measured *counter-*

clockwise from the zero-degree mark, instead of clockwise. This option will affect all angle measurements, with the exception of **1)** font angles, which already use counter-clockwise angles by default, and **2)** HSW "hue" angles, which always use clockwise angles.

GFX_SET_CALLBACK

This option is used to specify a Callback function in an external module. If this option is set, when Graphics Tools needs to send a Window Message to your program it will call the specified function directly instead of sending a message. This option is provided primarily for Direct To DLL users.

GFX_TEXT_ANIMATE_SPEED (per window)

This Graphics Tools version 1 option has been renamed GFX_FONT_ANIMATE_SPEED. See above.

GFX_TEXT_EDGE_COLOR (per window)

This option is used to specify the color that is used to draw edges when the TEXT_C_EDGES option is used. (The C stands for Color.) The default color is Red. See DrawTextRow for more information.

GFX_TEXT_OUTLINE_COLOR (per window)

This option controls the color of the outline that is produced by the DrawTextRow function when the TEXT_OUTLINE effect is used. The default value is zero (black). You may use any valid Windows color value.

GFX_TEXT_OUTLINE_SIZE (per window)

This option controls the width of the outline that is produced by the DrawTextRow function when the TEXT_OUTLINE effect is used. The default value is six (6), which tells Graphics Tools to use an outline that is 6% of the height of the current font. (Using a percentage in this way produces a reasonably consistent outline for a wide variety of font sizes.) You can use any value between 1% and 100%.

GFX_TEXT_ROW_ORIGIN (per window)

This option can be used to tell the DrawTextRow function where to draw text, relative to the LPR. See that function for more information.

GFX_TEXT_SHADOW_ANGLE (per window)

This option controls the angle of the shadow that is produced by the DrawTextRow function when the TEXT_SHADOW effect is used. The default value is 225 degrees, which produces shadows below and to the right of the body of the text. This value must be specified in degrees even if the GfxOption GFX_USE_RADIANS option is being used.

GFX_TEXT_SHADOW_COLOR (per window)

This option controls the color of the shadow that is produced by the DrawTextRow function when the TEXT_SHADOW effect is used. The default value is zero (black). You may use any valid Windows color value.

GFX_TEXT_SHADOW_SIZE (per window)

This option controls the depth of the shadow that is produced by the DrawTextRow function when the TEXT_SHADOW effect is used. The default value is eight (8), which tells Graphics Tools to use a shadow that is 8% of the height of the current font. (Using a percentage in this way produces a reasonably consistent shadow for a wide variety of font sizes.) You can use any value between 1% and 100%.

GFX_TRANSFER_MODE (per window)

This options is virtually identical to the GFX_IMAGE_DRAW_MODE option (see above), except that it affects all drawing functions, not just bitmaps. It is provided for very unusual circumstances, and is not generally useful. It should be used only under instructions from Perfect Sync Technical Support.

GFX_TUBE_SHADING_LEVEL (GLOBAL SETTING)

This option can be used to change the amount of shading that the DrawTube function uses when drawing the "inside" of a tube. The default value is -400.

GFX_USE_DLG_UNITS (GLOBAL SETTING)

This option tells the OpenGfx function that the numbers that you will provide for the size and location of the graphics window will be in Dialog Units instead of pixels.

GFX_USE_RADIANS (GLOBAL SETTING)

This option tells Graphics Tools to use Radians instead of Degrees for most angle measurements. Use any nonzero value to activate the "Use Radians" mode. Example: GfxOption GFX_USE_RADIANS, 1.

GFX_VERT_SCALE (per window)

This option can be used to change the number of vertical Drawing Units that Graphics Tools uses when scaling all of the various drawing operations, but *only* if the GfxWorld function has not already been used to define the World.

If you are using Graphics Tools, the default value of this option is 1024. If you are using Console Tools Plus Graphics, the default value of this option is 512.

The value of this option can be changed at any time, and will affect all future drawing operations. (It will not change the current appearance of the graphics window.)

Also see GFX_HORIZ_SCALE above.

GFX_V_SCROLL_BAR_SPEED (per window)

If a graphics window has a vertical scroll bar, this option controls the distance by which the window is scrolled when the up- or down-arrow key is pressed or the scroll-down or scroll-up button is clicked. It thereby controls the speed at which the window scrolls. The default value is 4 pixels.

GFX_WEDGE_CAP_COLOR (GLOBAL SETTING)

This option is used to change the color of the "cap" (the top or bottom) of wedges that are drawn with the DrawWedge function. The default value is GFX_AUTO, which tells Graphics Tools to use the current Brush or Gradient.

GFX_WEDGE_SHADING_LEVEL (GLOBAL SETTING)

This option can be used to change the amount of shading that the DrawWedge function uses when drawing the sides of a wedge. The default value is -200.

GFX_WINDOW_DEFAULT_STYLE (GLOBAL SETTING)

If the value GFX_DEFAULT is used for the *lStyle&* parameter of the OpenGfx function, this style is used.

GFX_WINDOW_NUMBER_OFFSET (GLOBAL SETTING)

The default value of this option is 5000. That means that if you use a resource script, PowerBASIC's DDT, or the CreateWindowEx API function create a graphics window, you should use a window ID number that is 5000 greater than the graphics window number. For example, to create graphics window number 10, use a window ID number of 5010. If this numbering system is not convenient for you, you can change the value of this option but you *must change this value before any graphics window have been created* or serious malfunctions will occur.

Example

```
GfxOption  GFX_V_SCROLL_BAR_SPEED, 10
```

See Also

GfxMetrics

GfxPrintArea

Purpose

Sends a rectangular portion of the graphics window to the current printer.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintArea(lLeft&, _
                        lTop&, _
                        lRight&, _
                        lBottom&, _
                        sDocName$, _
                        sCaption$)
```

VB method: *GfxWindow**.GfxPrintArea

```
lResult& = GfxPrintAreaEx(lWindowNumber&, _
                          lLeft&, _
                          lTop&, _
                          lRight&, _
                          lBottom&, _
                          sDocName$, _
                          sCaption$)
```

(See Syntax Options)

Parameters

lLeft&, lTop&, lRight&, and *lBottom&*

These parameters define the portion of the graphics window that will be printed, in Drawing Units. You can use either zero (0) or GFX_ALL for *lRight&* and/or *lBottom&* to specify the right and/or bottom edge of the window.

sDocName$

The name of the document. This string will be displayed by various Windows dialogs, such as the control panel's Printers applet, to label the print job. If you use the GfxPrintParam PRINTER_SETUP_OPTIONS, PRINT HEADER option, it will also be printed at the top of the page.

sCaption$

The text that should be displayed in the Progress Box's caption while the graphics image is being prepared for printing. If you use an empty string ("") for this parameter, no Progress Box will be displayed.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned. The Error Code will usually be between ERROR_FIRST_PRINTER_ERROR and ERROR_LAST_PRINTER_ERROR but it is possible for other Error Codes to be returned.

If this function returns ERROR_FIRST_PRINTER_ERROR + 800, the computer does not have sufficient memory to print an image in the specified size. This should be

relatively rare.

If `ERROR_FIRST_PRINTER_ERROR + 900` is returned it means that the target printer does not support the printing of images. You will find that if you attempt to use Microsoft Paint to print an image, it will also fail.

These problems can *sometimes* be remedied by installing more memory or an updated printer driver.

Remarks

This function is identical to the GfxPrintWindow function, except that it can be used to print a *portion* of a graphics window.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
'print the top-left quarter of
'a 1024x1024 graphics window
GfxPrintArea 0, 0, 512, 512, "My print job"
```

Sample Program

Printing An Image

See Also

SaveGfxArea

GfxPrintDefaults

Purpose

Resets all of Graphics Tools printing-related settings to their default values.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintDefaults
```

VB method: *GfxWindow**.GfxPrintDefaults

```
lResult& = GfxPrintDefaultsEx(lWindowNumber&)
```

(See Syntax Options)

Parameters

None.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function returns all of the Graphics Tools printing-related settings to their default values. This usually selects the system's default printer, if it has one.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
GfxPrintDefaults
```

Sample Program

Printing An Image

See Also

GfxPrintArea, GfxPrintWindow

GfxPrintPageSetup

Purpose

Displays a standard Windows "Page Setup" dialog, in preparation for printing an image.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintPageSetup(lLeft&, _
                             lTop&, _
                             lRight&, _
                             lBottom&)
```

VB method: *GfxWindow**.GfxPrintPageSetup

```
lResult& = GfxPrintPageSetupEx(lWindowNumber&, _
                               lLeft&, _
                               lTop&, _
                               lRight&, _
                               lBottom&)
```

(See Syntax Options)

Parameters

lLeft&, lTop&, lRight&, and *lBottom&*

These parameters are used to specify, in Drawing Units, the area of the graphics window that will be printed. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the user selects the Ok button, or ERROR_USER_CANCEL if they select the Cancel button or close the dialog in some other way. Under certain circumstances a Graphics Tools Error Code will be returned.

Remarks

This function is used to display a standard Windows "Page Setup" dialog, in preparation for printing an image.

The *lLeft&*, *lTop&*, *lRight&*, and *lBottom&* parameters are used to tell this function which portion of a graphics window to display in the dialog's "thumbnail" image. *Those parameters do not actually tell Graphics Tools which area to print.* It is assumed that you will pass the same parameter four values to the GfxPrintArea function so that the thumbnail and the printed area will match.

If you use 0,0,0,0 for the parameters of this function, the entire graphics window will be displayed as the thumbnail image. You can also use GFX_ALL for the *lRight&* and *lBottom&* parameters, to accomplish the same thing.

PLEASE NOTE: Most systems have a printer. If this function is used on a system that does not have a printer, in *most* cases Windows will, depending on the version of

Windows, either 1) return an error code which this function will return to your program, or 2) display a message box instructing the user to install a printer before attempting to print. Under certain *rare* circumstances Windows will fail to do either of those things and this function will exit with a return value of ERROR_USER_CANCEL as if the user had selected the cancel button. In reality it is *Windows* that canceled the operation, and Windows does not provide a method of telling the difference.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
'Set up to print the entire
'graphics window image
GfxPrintPageSetup 0,0,0,0
```

Sample Program

Printing An Image

See Also

GfxPrintArea, GfxPrintWindow

GfxPrintParam

Purpose

Sets the value of a printing-related setting.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintParam(lParameter&, _
                         lValue&)
```

VB method: *GfxWindow**.GfxPrintParam

```
lResult& = GfxPrintParamEx(lWindowNumber&, _
                           lParameter&, _
                           lValue&)
```

(See Syntax Options)

Parameters

lParameter&

The parameter that should be set. See **Remarks** below for a list of valid values.

lValue&

The value that the parameter should be given.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Even though the GfxPrintParam**Ex** function has a parameter called *lWindowNumber&*, this function affects the printing of images from *all* windows. The window number is required so that Graphics Tools can make sure that you are not attempting to print the contents of a graphics window that does not exist.

lParameter& must be one of the following values:

```
PRINTER_COPIES
```

The number of copies of the image that will be printed. *lValue&* must be greater than zero (0).

```
PRINTER_MARGIN_LEFT
PRINTER_MARGIN_TOP
PRINTER_MARGIN_RIGHT
PRINTER_MARGIN_BOTTOM
```

The left, top, right, and bottom margin that will be used. The unit of measurement is determined by the system at runtime, based on the system's default values and certain user settings that affect all programs. See PRINTER_SETUP_UNITS below.

Using negative values for these settings will produce unpredictable results.

PRINTER_ORIENTATION

Use zero (0) for the Portrait orientation, or one (1) for Landscape.

PRINTER_PAPER_SIZE

lValue& must be one of the DMPAPER_ values from the PowerBASIC WIN32API.INC file or the Visual Basic WIN32API.TXT file.

PRINTER_SETUP_OPTIONS

Use one or more of the following values, combined with the OR operator.

PRINT_ACTUAL_SIZE

This option tells Graphics Tools to ignore the Page Setup margin settings and to print the image in its "natural" size, pixel for pixel. The size of the printed image will depend on the printer's "dots per inch" rating. This setting produces very clean, accurate images because no "stretching" takes place.

PRINT_FILL_SPECIFIED_AREA

Graphics Tools normally maintains the aspect ratio of the graphics window when printing an image, so that round circles print round, squares print square, and so on. Using this option tells Graphics Tools to stretch the image to fit the specified margins, regardless of how the image is affected.

PRINT_HEADER

This option tells Graphics Tools to print the *sDocName$* string (from the GfxPrintWindow or GfxPrintArea function) at the top-left corner of the document, using the printer's default font.

PAGE_SETUP_DISABLE_MARGINS
PAGE_SETUP_DISABLE_PRINTER
PAGE_SETUP_DISABLE_ORIENTATION
PAGE_SETUP_DISABLE_PAPER
PAGE_SETUP_NO_NETWORK_BUTTON

These options are used to disable elements of the Page Setup dialog so that the user does not have access to the corresponding functions.

PRINTER_SETUP_UNITS

See the PRINTER_MARGIN_ settings above for background. This value can be set to either zero (0) for "thousandths of inches", or one (1) for

"hundredths of millimeters". The Page Setup dialog will display either English or Metric units, accordingly.

PRINTER_TRANSFER_MODE

This value can be used to tell Graphics Tools which method it should use to transfer an image from the screen to the printer. If this value is zero (the default) this selection is made automatically, based on the capabilities of the printer. You can set this value to one of the following modes, which are defined in the Visual Basic Win32API.TXT file and the PowerBASIC Win32API.INC file:

```
RC_BITBLT
RC_STRETCHBLT
RC_STRETCHDIB
```

Using a different transfer mode may, depending on the printer and driver, produce different results. For example if a non-color printer is being used, screen colors may be translated into shades of gray differently.

Not all printers support all three transfer modes.

You can also use GFX_QUERY for this value. Doing that does not *change* the transfer mode, it *returns* a bitmasked value which indicates the capabilities of the most-recently-used printer. You must actually print something before GFX_QUERY will return a value.

Example

```
'print 10 copies
GfxPrintParam PRINTER_COPIES, 10

'Disable the Margins and Printer buttons:
GfxPrintParam PRINTER_SETUP_OPTIONS, _
              PAGE_SETUP_DISABLE_MARGINS OR _
              PAGE_SETUP_DISABLE_PRINTER
```

Sample Program

Printing An Image

See Also

GfxPrintArea, GfxPrintWindow

GfxPrintSetup

Purpose

Displays a standard Windows "Print Setup" dialog, in preparation for printing an image.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintSetup
```

VB method: *GfxWindow**.GfxPrintSetup

```
lResult& = GfxPrintSetupEx(lWindowNumber&)
```

(See Syntax Options)

Parameters

None.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the user selects the dialog's Ok button, or ERROR_USER_CANCEL if they select the Cancel button or close the dialog in some other way. Under certain circumstances a Graphics Tools Error Code will be returned.

Remarks

Note: Microsoft now recommends the use of the GfxPrintPageSetup dialog instead of this dialog.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
If GfxPrintSetup = SUCCESS Then
    'print the image
End If
```

Sample Program

Printing An Image

See Also

GfxPrintPageSetup

GfxPrintStatus

Purpose

Returns the current status of the most recent print job that was created by the GfxPrintArea or GfxPrintWindow function.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
sResult$ = GfxPrintStatus(lWhich&)
```

VB method: *`GfxWindow*`*`.GfxPrintStatus`

```
sResult$ = GfxPrintStatusEx(lWindowNumber&, _
                            lWhich&)
```

(See Syntax Options)

Parameters

lWhich&

One of the constants described in **Remarks** below.

Return Value

The value of *sResult&* will be a string that contains the requested status value, or an empty string ("") if the print job has finished.

Remarks

This function can be used to obtain information about the *most recent* print job that was started by the GfxPrintArea or GfxPrintWindow function. If more than one print job is pending for a graphics window, it is not possible to obtain information about any job except the most recent one.

Use one of the following values for the *lWhich&* parameter.

Not all printer drivers support all of these values.

`PRINTER_JOB_NUMBER`

The Job Number that Windows assigned to the print job.

`PRINTER_NAME`

The name of the printer where the job was sent.

`PRINTER_MACHINE`

The name of the machine (i.e. computer) to which the printer is connected.

`PRINTER_USER`

The name of the user who is logged into the machine to which the printer is connected.

`PRINTER_JOB_NAME`

The name that was given to the print job by the *sDocName$* parameter of GfxPrintWindow or GfxPrintArea.

`PRINTER_DATATYPE`

The data type used for the print job.

`PRINTER_JOB_STATUS`

Depending on the printer driver, either a plain-English status message or a numeric value (in string form) that corresponds to one of the `JOB_STATUS_` values in the Visual Basic Win32API.TXT file or the PowerBASIC Win32API.INC file.

`PRINTER_JOB_PRIORITY`

The priority of the print job as a numeric value (in string form) that corresponds to one of the values `MIN_PRIORITY`, `MAX_PRIORITY`, or `DEF_PRIORITY` in the Visual Basic Win32API.TXT file or the PowerBASIC Win32API.INC file.

`PRINTER_JOB_POSITION`

The job's position in the print que, as a numeric value in string form.

`PRINTER_PAGES_TOTAL`

The number of pages in the job, .as a numeric value in string form.

`PRINTER_PAGES_DONE`

The number of pages that have been printed, as a numeric value in string form.

`PRINTER_JOB_START_TIME`

The date/time that the job was submitted to the que, as a string in the form:

`YYMMwwDDHHmmSS`

...where YY=Year, MM=Month, ww=Weekday (0-6), DD=Day, HH=Hour, mm=Minute, and SS=Second. Depending on the version of Windows that you are using, this date/time may or may not be adjusted for your local time zone.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
sResult$ = GfxPrintStatus(PRINTER_JOB_STATUS)
'sResult$ is now an empty string if the
'print job is finished, or a string as
'described above.
```

Sample Program

Printing An Image

See Also

SaveGfxWindow

GfxPrintWindow

Purpose

Prints the contents of the entire graphics window.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxPrintWindow(sDocName$, _
                          sCaption$)
```

VB method: *GfxWindow**.GfxPrintWindow

```
lResult& = GfxPrintWindow(lWindowNumber&, _
                          sDocName$, _
                          sCaption$)
```

(See Syntax Options)

Parameters

sDocName$

The name of the document. This string will be displayed by various Windows dialogs, such as the control panel's Printers applet, to label the print job. If you use the GfxPrintParam `PRINTER_SETUP_OPTIONS, PRINT HEADER` option, it will also be printed at the top of the page.

sCaption$

The text that should be displayed in the Progress Box's caption while the graphics image is being prepared for printing. If you use an empty string ("") for this parameter, no Progress Box will be displayed.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned. The Error Code will usually be between `ERROR_FIRST_PRINTER_ERROR` and `ERROR_LAST_PRINTER_ERROR` but it is possible for other Error Codes to be returned.

If this function returns `ERROR_FIRST_PRINTER_ERROR + 800`, the computer does not have sufficient memory to print an image in the specified size. This should be relatively rare.

If `ERROR_FIRST_PRINTER_ERROR + 900` is returned it means that the target printer does not support the printing of images. You will find that if you attempt to use Microsoft Paint to print an image, it will also fail.

These problems can *sometimes* be remedied by installing more memory or an updated printer driver.

Remarks

This function is identical to the GfxPrintArea function except that it always prints the *entire* graphics window.

Please see Printing Graphics Tools Images for a complete overview of all of the printing-related functions, and how they interact.

Example

```
GfxPrintArea "My Image"
```

Sample Program

Printing An Image

See Also

SaveGfxWindow

GfxRefresh

Purpose

Causes the entire graphics window to be re-displayed by Windows, to insure that the results of all drawing operations are visible and that there are no "gaps" in the display.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxRefresh(lForce&)
```

VB method: *GfxWindow**.GfxRefresh

```
lResult& = GfxRefreshEx(lWindowNumber&, _
                        lForce&)
```

(See Syntax Options)

Parameters

lForce&

If you use zero (0) for this parameter, the graphics window will be updated only if it is not currently "frozen". (See GfxWindow GFX_FREEZE.) This function will have no effect if you use zero for the *lForce&* parameter when the window is frozen. If you use a *nonzero* value for this parameter, the graphics window will be updated regardless of whether or not it is currently frozen.

Return Value

The value of `lResult&` will be True (`-1`) if the graphics window is refreshed, or False (zero) if it is not.

Remarks

Under certain circumstances, such as the use of Console Tools Plus Graphics on a Windows 95/98/ME computer, the graphics window should be periodically refreshed. (See Refreshing The Display for more information about the reasons for this.)

This is usually accomplished by using a time-based function (such as the Console Tools OnTimer function) to periodically refresh the display. The following code...

```
GfxRefresh  0
```

...should be executed once per second or so, to make sure that the display is periodically refreshed. The zero (0) parameter tells Graphics Tools "if the graphics window is currently in the GfxWindow GFX_FREEZE mode, don't refresh the display".

The use of...

```
GfxRefresh  1
```

...with a nonzero parameter can be used if you want to force the display to be refreshed regardless of the "freeze" state of the graphics window.

Refreshing Other Windows

It is sometimes necessary for Graphics Tools programs to refresh *other* windows. For example, when using Console Tools Plus Graphics it is fairly common to need to refresh the graphics window's *parent* window (the console).

To do this, use the GfxRefreshEx function and use the window's handle for the *lWindowNumber&* parameter. The *lForce&* parameter is ignored when this is done.

Console Tools Plus Graphics programmers can use the Console Tools hConsoleWindow function to obtain the handle of the console window. Refreshing the console window in this way does not *always* work (because of the internal workings of the Microsoft console) but it will help under most circumstances.

Example

See **Remarks** above.

See Also

Refreshing The Display

GfxResize

Purpose

Changes **1)** the visible size of the graphics window and/or **2)** the size of the drawing area

Availability

Graphics Tools Pro Only

Warning

Visual Basic programs cannot use this function's GFX_VISIBLE_SIZE option, they must use the GfxWindow*.Width and .Height properties

Syntax

```
lResult& = GfxResize(lWhich&, _
                     lWidth&, _
                     lHeight&)
```

Visual Basic programs must use the *GfxWindow*.Width* and .Height properties to change the GFX_VISIBLE_SIZE property of a graphics window. VB programs can, however, use this function to change the GFX_DRAWING_SIZE.

```
lResult& = GfxResizeEx(lWindowNumber&, _
                       lWhich&, _
                       lWidth&, _
                       lHeight&)
```

(See Syntax Options)

Parameters

lWhich&

Use GFX_VISIBLE_SIZE to change the visible size of the graphics window, or GFX_DRAWING_SIZE to change the drawing size, or GFX_ALL to change both at the same time. (See **Remarks** below for an explanation of drawing size and visible size.)

lWidth& and *lHeigh&t*

The width and height (in *pixels*) to which the graphics window will be changed. Do not use positive values less than 32, which is the minimum size.

To change only the width *or* height use GFX_SAME for the other parameter.

Negative numbers and the special value GFX_AUTO can also be used for *lWidth&* and *lHeight&*. See **Remarks** below for more information.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

If you attempt to use this function with a graphics window that does not have the GFX_STYLE_SIZABLE style, ERROR_CANT_BE_DONE will be returned.

If you attempt to use **1)** a positive value less than 32 for either *lWidth&* or *lHeight&*, or **2)** a value which would make the visible size larger than the drawing size, this

function will return ERROR_SIZE_LIMIT but the resizing operation *will* be performed, using the minimum or maximum legal value, as appropriate.

Remarks

Only graphics windows which have the GFX_STYLE_SIZEABLE window style can be resized. This style can be specified when the window is created, or it can be added (or removed) with the GfxWindow function.

Visible Size vs. Drawing Size

Each graphics window has two different size adjustments, the *visible size*, which is the apparent "physical" size of the graphics window on the computer screen, and the *drawing size*, which is the size of the drawing area.

For example, if you create a graphics window that is 200x200 pixels, both the drawing size and visible size will be 200x200 pixels. If you then use the GfxResize function to change the *visible* size to 100x100, the graphics window (i.e. the thing that you see) will become smaller and scroll bars will be added. The graphics image will still be 200x200, and the user will be able to use the scroll bars view the entire 200x200 image.

If you start with a 200x200 graphics window and change the *drawing* size to 400x400, the window will not change size but scroll bars will be added and the user will be able to use them to see the entire 400x400 image.

The visible size can never be greater than the drawing size, unless the GFX_STYLE_STRETCHABLE window style is used.

If the visible size is less than the drawing size, scroll bars will be displayed to allow the user to scroll the image unless the window was created with the GFX_STYLE_SCROLLBARS_NONE or GFX_STYLE_STRETCHABLE window style.

Changing the Visible Size

Visual Basic programs cannot use the GFX_VISIBLE_SIZE option, they must use the GfxWindow*.Width and .Height properties to change the visible size.

If the *lWhich&* parameter is GFX_VISIBLE_SIZE, the GfxResize function will change the visible size of the graphics window.

If you attempt to make the graphics window larger than the drawing size, it will be made the same size as the drawing size (unless the window has the GFX_STYLE_STRETCHABLE style).

If you use a negative number for *lWidth&* and/or *lHeight&*, Graphics Tools will use a "percentage" of the current drawing size. For example, using negative twenty-five (-25) would tell Graphics Tools to change the visible size so that the width or height was equal to 25% of the drawing area's width or height. Negative values between -1 and -100 can be used, but small negative values like -1 will usually produce the minimum window size of 32 pixels.

If you use the special value GFX_AUTO for *lWidth&* and/or *lHeight&* Graphics Tools will match the visible size to the current drawing size.

Changing the Drawing Size

If the *lWhich&* parameter is `GFX_DRAWING_SIZE`, the graphics window will be destroyed and re-created using the new size that you specify. Note that all previous drawing operations will be erased whenever this is done, so you may wish to save the contents of the window, change the drawing size, and then restore the contents.

If you use a negative number for *lWidth&* and/or *lHeight&*, Graphics Tools will use a "percentage" of the current *visible* size. For example, using negative twenty-five (`-25`) would tell Graphics Tools to change the drawing width or height so that 25% of the drawing area was visible. Negative values between `-1` and `-100` can be used, but small negative values like `-1` may produce results that are larger than Windows will allow. Windows may refuse to create an *extremely* large graphics window.

You can also use an *lWidth&* and/or *lHeight&* value of `GFX_AUTO` to tell Graphics Tools to fill the graphics window's parent window. (In the case of the Graphics Tools OCX, the parent window is the OCX "container", not the VB form.)

Because it destroys and re-creates the graphics window, the use of `GFX_DRAWING_SIZE` effectively performs a "reset' of many different graphics window settings. While the pen, brush, font, gradient, and world will *not* be affected, you will find that certain settings such as the LPR, scroll position, scroll speed, and drawing mode will be reset to their default values. For this reason it is a good idea to use `GFX_DRAWING_SIZE` as early as possible in your program, *then* perform additional configuration of the graphics window using the GfxOption function.

Changing the Drawing and Visible Size Together

If *lWhich&* is `GFX_ALL` you can specify a positive number for *lWidth&* and *lHeight&*, or you can use `GFX_AUTO` to tell Graphics Tools to fill the graphics window's parent window. (In the case of the Graphics Tools OCX, the parent window is the OCX "container", not the VB form.)

Example

```
GfxWindow GFX_STYLE_SIZABLE
GfxResize GFX_VISIBLE_SIZE, -25, -25
```

Sample Program

Resizing a Graphics Window

See Also

Graphics Window Styles

GfxResponse

Purpose

This function is used to respond to certain window messages from a graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxResponse(hWnd&, _
                       lResponse&)
```

VB: Visual Basic programs cannot respond to window messages.

(No Ex *function is necessary.)*

Parameters

hWnd&

The handle of the graphics window that sent the window message that requires a response.

lResponse&

The desired response. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For background information, see Graphics Window Messages.

A small number of window messages allow your program to "respond" and influence the way Graphics Tools works. For example, if your program receives the message...

```
WM_GFX_EVENT + WM_WINDOWPOSCHANGING
```

...that means that the window's position -- its location or size -- is about to change. If you use the GfxResponse function in your program's callback function to respond to that message with a nonzero value like this...

```
FUNCTION = GfxResponse(hWnd, 1)
```

...Graphics Tools will refuse to allow the window's location and/or size to be changed. If you respond with a zero value, or if you do not respond at all, the window will be allowed to change.

Similarly, if the user presses a key while the graphics window has the focus, both of these messages will be received by your program...

```
WM_GFX_EVENT + WM_KEYDOWN
WM_GFX_EVENT + WM_CHAR
```

If you respond to those messages with a nonzero value, that tells Graphics Tools that your program has handled the keypress. For example, if your program wants to handle the PgUp and PgDn keys itself instead of allowing Graphics Tools to scroll the graphics window as it normally does, respond to those specific keystroke messages with nonzero values.

Most mouse events cannot be "refused", but there is one exception. If your program receives...

```
WM_GFX_EVENT + WM_LBUTTONDOWN
```

...with a window message *lParam* (or `CBLPARAM`) value of `&hFFFFFFFF` it means that the user has clicked the window's caption. If the graphics window has the `GFX_STYLE_MOVABLE` window style, this usually indicates that the user is about to begin dragging the window. To refuse to allow this, use GfxResponse to respond to the message with a nonzero value.

NOTE: The `FUNCTION = GfxResponse` line should always be the *last* line of code that is executed by your callback function before it exits.

TECHNICAL DETAILS: In many circumstances it is possible to simply respond to a window message with `FUNCTION = 1`. But under some circumstances, especially when dialog windows are used, this is not sufficient. Using GfxResponse performs an additional step that insures that your program's response will be received by Graphics Tools.

Example

See **Remarks** above.

See Also

Graphics Window Messages

GfxScroll

Purpose

Scrolls the graphics window vertically or horizontally, or both.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GfxScroll(lDirection&, _
                     lDistance&)
```

VB method: *GfxWindow**.GfxScroll

```
lResult& = GfxScrollEx(lWindowNumber&, _
                       lDirection&, _
                       lDistance&)
```

(See Syntax Options)

Parameters

lDirection&

Either SCROLL_HORIZ or SCROLL_HORIZ_POS to scroll the window horizontally, or SCROLL_VERT or SCROLL_VERT_POS to scroll it vertically, or SCROLL_HOME or SCROLL_END to scroll it in both directions at the same time.

lDistance&

If *lDirection&* is SCROLL_HORIZ or SCROLL_VERT, *lDistance&* is the distance by which the window should be scrolled, in pixels. Use a positive integer to scroll in one direction or a negative integer to scroll in the other direction, or (if *lDirection&* is SCROLL_VERT) the special values SCROLL_PAGE_UP and SCROLL_PAGE_DOWN can be used.

If *lDirection&* is SCROLL_HORIZ_POS or SCROLL_VERT_POS, *lDistance&* is the position (in Drawing Units) to which the window should be scrolled. See **Remarks** below for more information.

If *lDirection&* is SCROLL_HOME or SCROLL_END the *lDistance&* parameter is ignored.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

If the graphics window's *visible size* is the same as its *drawing size*, this function will have no effect. See GfxResize for more information about these values.

If you use an *lDirection&* value of SCROLL_HORIZ the *lDistance&* parameter specifies the horizontal distance, in pixels.

If you use an *lDirection&* value of SCROLL_VERT the *lDistance&* parameter specifies the vertical distance, in pixels. You can also use the values SCROLL_PAGE_UP and SCROLL_PAGE_DOWN to scroll the window by an amount equal to the current visible window height.

If you use an *lDirection&* of SCROLL_HORIZ_POS or SCROLL_VERT_POS, the *lDistance&* parameter specifies the position (in Drawing Units) to which the window should be scrolled. Graphics Tools will place the specified location in the *center* of the visible portion of the graphics window, within the limits of the window's ability to scroll.

If you use an *lDirection&* value of SCROLL_HOME or SCROLL_END, the *lDistance&* parameter is ignored and the window is scrolled to the home (far top-left) or end (far bottom-right) position.

If any scrolling action would result in the graphics window scrolling past its limits, the window will be scrolled to its limit.

Example

```
GfxScroll SCROLL_VERT, SCROLL_PAGE_DOWN
```

See Also

GfxResize

GfxSquareness

Purpose

Returns a value that indicates the "squareness" of the current World.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
dpResult# = GfxSquareness
```

VB property: *`GfxWindow*`*`.WorldAspect` (See **Remarks** below.)

```
dpResult# = GfxSquarenessEx(lWindowNumber&)
```

(See Syntax Options)

Parameters

None.

Return Value

The value of *dpResult#* will be a floating point value that indicates how "square" the graphics window's World is. A return value of zero (0) indicates an error condition, such as attempting to obtain the squareness of a window number that does not exist.

Remarks

For background information, see Drawing Units and Using Different "Worlds", with particular attention to the meaning of the term "square world". It does *not* refer to the physical proportions of the graphics window, it refers to the window's ability to draw round circles and square squares.

This function will return a value of exactly one (1) if the current world is perfectly square. It will return a value less than one if the window's World is too narrow, or a value greater than one if it is too tall.

For example, a return value of `0.5` would indicate that the world is one-half (1/2) as wide as it is tall. A return value of `2.0` would indicate that the world is twice as wide as it is tall.

It is important to note that it may be difficult to achieve a squareness value of exactly one (1). Generally speaking, a value between 0.995 and 1.005 should be considered acceptable. (It is very unlikely that any given computer monitor's width and height will be adjusted within 1/2 of one percent of perfection. See Using Different "Worlds" for more information about this.)

Visual Basic programmers should note that this value's VB property name is WorldAspect, not Squareness. This was done so that this value will be grouped with the other world-related values (WorldLeft, etc.) when the properties are displayed alphabetically. You will usually want to monitor the value of the WorldAspect property while you are changing the other world-related properties. See WorldAspect.

Example

```
If Abs(GfxSquareness-1) < 0.005 Then
      'The window is acceptably square.
End If
```

See Also

Drawing Units

GfxSyncCursor

Purpose

Moves the Windows mouse cursor to the LPR.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxSyncCursor
```

VB method: *GfxWindow**.GfxSyncCursor

```
lResult& = GfxSyncCursorEx(lWindowNumber&)
```

Parameters

None.

Return Value

The return value of this function will always be SUCCESS (zero) unless you attempt to sync the cursor to the LPR of a graphics window that does not exist, so it is generally safe to ignore the return value.

Remarks

This function causes the location of the Windows mouse cursor to be moved so that it points at the LPR.

IMPORTANT NOTE: This function will move the mouse cursor even if the graphics window does not currently have the Windows focus, and even if the graphics window is currently covered by another window. That means that if your program is covered by another application when this function is used, the mouse cursor will mysteriously move to a screen location that is meaningless to the user. And if an operation such as a drag-and-drop is being performed, the user will probably not be very happy about it.

Unless you are certain that your program has the "right" to move the cursor (such as immediately after a key has been pressed), you should make sure that you really *want* to move the cursor before you use this function. For example, the Windows GetForegroundWindow API function could be used to confirm that your application has the Windows focus. (Console Tools Plus Graphics users can use the ConsoleHasForeground function.)

Example

```
GfxSyncCursor
```

See Also

Please refer to the Windows API documentation (such as the WIN32.HLP Help File) for information about GetForegroundWindow, GetFocus, and other functions that might be useful when determining whether or not your program should move the cursor.

GfxTextHole

See the Console Tools Documentation for information about the GfxTextHole function.

GfxTitle

This Graphics Tools Version 1 function has been replaced by GfxCaption.

GfxToolsVersion

Purpose

Returns the version number of the currently-loaded Graphics Tools Runtime Files.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxToolsVersion
```

Parameters

None.

Return Value

The return value of this function will be `200` (two hundred) if version `2.00` of the primary Graphics Tools Runtime File is currently installed, or `201` if version `2.01` is installed, and so on.

Remarks

If your application relies on a certain version of the Graphics Tools Runtime Files being installed, it should check the value of this function when the program initializes. Then, if the detected version is too old, your program can either avoid using certain functions, or display an error message and exit gracefully.

GfxUpdate

Purpose

Causes a rectangular area of the graphics window to be re-displayed, to insure that the results of all drawing operations are visible and that there are no "gaps" in the display.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxUpdate(lLeft&, _
                     lTop&, _
                     lRight&, _
                     lBottom&)
```

VB method: *GfxWindow**.GfxUpdate

```
lResult& = GfxUpdateEx(lWindowNumber&, _
                       lLeft&, _
                       lTop&, _
                       lRight&, _
                       lBottom&
```

Parameters

lLeft& and *lTop&*

The X and Y locations of the top-left corner of the rectangle that is to be updated, in Drawing Units.

lRight& and *lBottom&*

The X and Y locations of the bottom-right corner of the rectangle, in Drawing Units.

Return Value

The return value of this function will always be SUCCESS unless you attempt to update a graphics window that does not exist, so it is generally safe to ignore the return value.

Remarks

If the graphics window is currently frozen (see GfxWindow GFX_FREEZE) this function will have no effect on the display. Otherwise, the rectangle that you specify will be updated by Windows.

It is not usually necessary to use this function unless you are using the Windows API to perform drawing functions directly and you want Graphics Tools to perform the final step of displaying the results of your drawing operations.

To update the *entire* graphics window (which is a much more common operation), see the GfxRefresh function.

Example

```
GfxUpdate 100,100,200,200
```

See Also: Refreshing The Display

GfxWindow

Purpose

Shows, hides, freezes, un-freezes, or destroys the graphics window. Can also be used to add or Remove certain window styles, and to capture or release the mouse.

Availability

Graphics Tools Standard and Pro

Warning

The mouse capture/release functions cannot be used by Console Tools Plus Graphics programs.

Syntax

```
lResult& = GfxWindow(lAction&)
```

VB method: *GfxWindow**.GfxWindow

```
lResult& = GfxWindowEx(lWindowNumber&, _
                       lAction&)
```

(See Syntax Options)

Parameters

lAction&

See **Remarks** below for a list of valid values.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned. If you are certain that a valid *lAction&* value is being used, it is usually safe to ignore the return value of this function.

Remarks

Visual Basic programmers: Instead of using GfxWindow GFX_SHOW and GFX_HIDE you should use the graphics window's `GfxWindow*.Visible` *property.*

If *lAction&* is `GFX_SHOW`, it causes the graphics window to become visible.

If *lAction&* is `GFX_HIDE`, it causes a currently-visible graphics window to be hidden but not destroyed. Your program can continue to use the various Graphics Tools functions while the window is hidden, and then reveal the results by using `GFX_SHOW`.

If *lAction&* is `GFX_FREEZE`, it causes the current contents of the graphics window to remain visible, but any new drawing operations that you perform will not be made visible until you use `GFX_UNFREEZE` This option is particularly useful when you want to perform a complex drawing operation but do not want the individual steps to be visible to the user. Also, because the graphics window is not constantly re-displayed every time each individual drawing operation is performed, using `GFX_FREEZE` before and `GFX_UNFREEZE` after a complex drawing operation can *greatly* increase the execution speed of your program.

See The Graphics Window for more information about `GFX_SHOW`, `GFX_HIDE`, `GFX_FREEZE`, and `GFX_UNFREEZE`.

If *lAction&* is GFX_CLOSE, the graphics window is hidden and then destroyed. It is usually not necessary to perform this step manually unless your program wants to "shut down" Graphics Tools for a period of time, in order to free up memory (or for some other reason). Graphics Tools *automatically* closes the graphics window each time you create a new one using the same window number, and when your program closes.

If you are using Graphics Tools Pro and *lAction&* is GFX_STYLE_SIZABLE, GFX_STYLE_MOVABLE, or GFX_STYLE_STRETCHABLE, that window style will be given to the graphics window. (These are the only window styles that can be changed without destroying and re-creating the window.) See the NOT operator below for more information.

If *lAction&* is SET_FOCUS the graphics window will be given the keyboard focus.

If *lAction&* is GFX_MOUSE_CAPTURE all future mouse events will be directed to the graphics window, even when the mouse cursor is not *over* the graphics window. Use this option with caution! You will not be able to use the mouse for *anything else* until the mouse is returned to normal with GFX_MOUSE_RELEASE. That means that you will not, for example, be able to use the mouse to click your application's Close button.

If *lAction&* is GFX_MOUSE_RELEASE the mouse will be returned to normal operation.

Please note that the mouse capture/release functions cannot be used by Console Tools Plus Graphics programs.

Graphics Tools Version 1 also supported FOREGROUND as a parameter for the GfxWindow function, but because of recent changes in the Windows operating system, this function is no longer supported.

Using the NOT Operator

Graphics Tools understands the NOT operator when it is used with the GfxWindow function. For example, these two lines of code will perform exactly the same operation:

```
GfxWindow  GFX_UNFREEZE
GfxWindow  NOT GFX_FREEZE
```

This option is most useful when you want to remove one of the three GFX_TYLE_ properties from a window (see above) but most of the other GfxWindow values can also be set in this way.

Example

```
GfxWindow GFX_FREEZE

'perform many different drawing operations here

GfxWindow GFX_UNFREEZE
```

See Also

The Graphics Window

GfxWorld

Purpose

Defines the Drawing World for a graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxWorld(lLeft&, _
                    lTop&, _
                    lRight&, _
                    lBottom&)
```

VB: Use the various World properties.

```
lResult& = GfxWorldEx(lWindowNumber&, _
                      lLeft&, _
                      lTop&, _
                      lRight&, _
                      lBottom&)
```

(See Syntax Options)

Parameters

lLeft&, lTop&, lRight&, and *lBottom&*

These parameters define the left, top, right, and bottom coordinates of the drawing world. Use `GFX_SAME` for one or more parameters if you want to leave those values unchanged.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

For a complete discussion of this function, see Drawing Units and Using Different "Worlds".

Example

See Using Different "Worlds".

Sample Programs

A World With Zero In The Center
A Rectangular Graphics Window

See Also

The Graphics Window

GfxX

Purpose

Returns the current X (left-right) location of the LPR.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxX
```

VB read-only property: *`GfxWindow*`*`.GfxX`

```
lResult& = GfxXEx(lWindowNumber&)
```

Parameters

None.

Return Value

The value of *lResult&* will be the X component of the LPR, in Drawing Units.

Remarks

The GfxX and GfxY functions can be used to determine the current location of the LPR, if your program does not keep track of it internally.

Example

```
lResult& = GfxX
'The lResult& variable now contains
'the X location of the LPR.
```

See Also

GfxLPR

GfxY

Purpose

Returns the current Y (up-down) location of the LPR.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfxY
```

VB read-only property: *GfxWindow**.GfxY

```
lResult& = GfxYEx(lWindowNumber&)
```

Parameters

None.

Return Value

The value of `lResult&` will be the Y component of the LPR, in Drawing Units.

Remarks

The GfxX and GfxY functions can be used to determine the current location of the LPR, if your program does not keep track of it internally.

Example

```
lResult& = GfxY
'The lResult& variable now contains
'the Y location of the LPR.
```

See Also

GfxLPR

GradientBrush

Purpose

Tells Graphics Tools to fill certain figures with a Gradient instead of the current Brush. (Can also return the current fill setting.)

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GradientBrush(lFigures&)
```

VB method: *GfxWindow**.GradientBrush

```
lResult& = GradientBrushEx(lWindowNumber&, _
                           lFigures&)
```

(See Syntax Options)

Parameters

lFigures&

One or more of the following values: GRADIENT_FILL_ROUND_FIGURES, GRADIENT_FILL_SQUARE_FIGURES, GRADIENT_FILL_TEXT_BODY, and GRADIENT_FILL_TEXT_EFFECTS. To specify more than one, use the OR operator. To specify all four at the same time, use GFX_ALL. To disable the Gradient Brush, use zero (0) or GFX_NONE. To retrieve the current GradientBrush setting (without changing it) use GFX_QUERY.

Return Value

If *lFigures&* is GFX_QUERY, the value of *lResult&* will be the current GradientBrush setting.

Otherwise, the value of *lResult&* will be SUCCESS (zero) if the Gradient Brush is changed as requrested, or a Graphics Tools Error Code if it is not.

Remarks

See Using Gradients To Fill Specific Figures for a complete discussion of this function.

Example

See Using Gradients To Fill Specific Figures.

See Also

Gradients

GradientColor

Purpose

Returns one color of a Gradient, based on an index number.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GradientColor(lIndex&)
```

VB read-only property: *GfxWindow**.GradientColor

```
lResult& = GradientColorEx(lWindowNumber&, _
                           lIndex&)
```

(See Syntax Options)

Parameters

lIndex&

If you picture a Gradient as being "an array of colors", this is the "array element number" of the color you wish to obtain.

Return Value

A Windows color value.

Remarks

This function can be used to obtain the colors of a gradient one by one, for various purposes. For example, if you want to draw an unusual figure with a gradient you could build it up, line by line, by drawing first with GradientColor(1), then GradientColor(2), and so on.

See Also

Gradients

GradientLoad

Purpose

Loads a predefined Gradient from a disk file.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GradientLoad(sFileName$)
```

VB method: *GfxWindow**.GradientLoad

```
lResult& = GradientLoadEx(lWindowNumber&, _
                          sFileName$)
```

(See Syntax Options)

Parameters

sFileName$

The name (and optional drive/path) of a file that contains a gradient. These files are almost always created with the GradientSave function.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function provides an easy way to load a predefined gradient.

We recommend the use of the extension `.GTG` (for Graphics Tools Gradient) but it is not required. A number of `.GTG` files are provided with Graphics Tools, and you can create your own with the GradientSave function.

Example

```
GradientLoad "MyGradient.GTG"
```

See Also

Gradients

GradientSave

Purpose

Saves a graphics window's gradient in a disk file, for later re-use.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GradientSave(sFileName$)
```

VB method: *GfxWindow**.GradientSave

```
lResult& = GradientSaveEx(lWindowNumber&, _
                          sFileName$)
```

(See Syntax Options)

Parameters

sFileName$

The name (and optional drive/path) of the file where the gradient should be saved.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function provides an easy way to save a gradient for later use. Gradients that are saved with this function can be loaded with the GradientLoad function.

We recommend the use of the extension `.GTG` (for Graphics Tools Gradient) but it is not required.

Example

```
GradientSave "MyGradient.GTG"
```

See Also

Gradients

GradientValue

Purpose

Sets or returns a value that is part of the current gradient.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = GradientValue(lWhich&, _
                         lValue&)
```

VB method: *GfxWindow**.GradientValue

```
lResult& = GradientValueEx(lWindowNumber&, _
                           lWhich&, _
                           lValue&)
```

(See Syntax Options)

Parameters

lWhich&

A value (see **Remarks** below) that indicates which one of the gradient's values should be changed or retrieved.

lValue&

The new value. Use GFX_QUERY to obtain a current value without changing it.

Return Value

If GFX_QUERY is used, this function will return the requested value.

If any other value is used, this function will return SUCCESS unless an error is encountered, such as trying to change the gradient of a non-existent graphics window.

Remarks

For a complete discussion of this function, see Designing Your Own Gradients.

Example

See Designing Your Own Gradients.

See Also

Gradients

GraphicsToolsAuthorize

Purpose

Tells the Graphics Tools runtime files that your program is authorized to use them.

Availability

Graphics Tools Standard and Pro

Warning

See **Remarks** below.

Syntax

```
lResult& = GraphicsToolsAuthorize(lAuthCode&)
```

(See Syntax Options)

Parameters

lAuthCode&

The Authorization Code that is embedded in your copy of the Graphics Tools runtime files.

Return Value

If an invalid *lAuthCode&* value is used, this function will return `ERROR_CANT_BE_DONE` to indicate that the runtime files cannot be authorized with that code.

If **1)** the correct code or **2)** if a "dummy code" is used, this function will return `SUCCESS` the first time it is called.

After `SUCCESS` has been returned, subsequent calls to this function will return True (negative one) regardless of the value of *lAuthCode&*.

Remarks

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 Graphics Tools Runtime File(s) illegally. YOU are legally responsible for preventing that from happening.

See Authorization Codes for a complete discussion of this function.

Example

```
GraphicsToolsAuthorize &h12345678
```

Example only. Please note that &h12345678 is not a valid Authorization Code.

See Also

Software License Agreement

GreenValue

Purpose

Returns the amount of green in a Windows Color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GreenValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and MAXCOLOR.

Return Value

This function will always return a value between zero (0) and 255 which indicates how much green the specified color contains.

Remarks

The RedValue, GreenValue, and BlueValue functions can be used to obtain the amounts of Red, Green, and Blue that a Windows Color contains.

Example

```
If GreenValue(lColor&) = 0 Then
      'The color contains no green
End If
```

See Also

Windows Colors

hGfx

Purpose

Returns the value of a Windows handle that is being used by Graphics Tools, for use with Windows API functions.

Availability

Graphics Tools Standard and Pro

Warning

The incorrect use of a Windows Handle can cause Application Errors (General Protection Faults).

Syntax

```
lResult& = hGfx(lWhich&)
```

VB programs must use the hGfxEx function, using the graphics window's ID property for *lWindowNumber&*.

```
lResult& = hGfxEx(lWindowNumber&, _
                  lWhich&)
```

Parameters

lWhich&

Determines which handle will be returned. See **Remarks** below for a list of valid values.

Return Value

If a valid *lWhich&* value is used, the value of *lResult&* will be the current value of the requested handle. If an invalid *lWhich&* value is used, or if the specified graphics window does not exist, zero (0) will be returned.

Remarks

Handle values are not static, so you should use the hGfx function every time you need a handle value, rather than using it once and storing the value in a variable.

The following *lWhich&* values can be used to obtain the corresponding Windows Handles: (An explanation of the possible *uses* of these handles, and exactly what they are, is well beyond the scope of this document. Please consult the Microsoft API documentation.)

`GFX_BITMAP_HANDLE`

The handle of the graphics window's bitmap.

`GFX_BRUSH_HANDLE`

The handle of the current Brush.

`GFX_COMPAT_DC_HANDLE`

The handle of the Compatible Device Context of the graphics window.

`GFX_DC_HANDLE`

The handle of the Device Context of the graphics window.

`GFX_FONT_HANDLE`

The handle of the current Font.

`GFX_INSTANCE_HANDLE`

The "instance handle" that Graphics Tools is currently using. This value is used when an embedded bitmap or icon is retrieved from an EXE or DLL module.

`GFX_PARENT_HANDLE`

The handle of the graphics window's parent window.

`GFX_PEN_HANDLE`

The handle of the current Pen.

`GFX_WINDOW_HANDLE`

The handle of the actual graphics window.

Example

This function is used to provide handle values to external functions, so a meaningful example is not possible.

See Also

hGfxFont

hGfxFont

Purpose

Returns the handle of the font that is being used by Graphics Tools.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = hGfxFont
```

VB programs must use the hGfxFontEx function, using the graphics window's ID property for *lWindowNumber&*.

```
lResult& = hGfxFontEx(lWindowNumber&)
```

Parameters

None.

Return Value

The value of *lResult&* will be the handle of the graphics window's Graphics Tools font.

Remarks

This function returns a value that is identical to the results of hGfx `GFX_FONT_HANDLE`, except that if the font does not currently exist, this function will create a default font.

Example

This function is used to provide a handle value for external functions, so a meaningful example is not possible.

See Also

hGfx

HighlightColor

Purpose

Returns a color value that is the "highlighted" or "shaded" version of the specified color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = HighlightColor(lColor&, _
                          lAmount&)
```

Parameters

lColor&

The color for which a highlighted or shaded color is needed.

lAmount&

The amount by which the color should be brightened or darkened.

Return Value

A Windows color value that is (if possible) brighter or darker than *lColor&*.

Remarks

Graphics Tools uses this function internally when creating highlights for Three-Dimensional Figures which are drawn with clear pens, and when creating "shaded" areas of 3D figures.

Your programs can use this function whenever you need a color that is brighter or darker that the specified color.

This function works by starting with the specified color and adding or subtracting a small amount of color and/or white. Using a relatively small positive value for *lAmount&* (such as `100`) produces a subtle brightening effect. Using a small negative value (like `-100`) produces a subtle darkening. Using larger values, of course, produces more intense effects. If the *lAmount&* value is too large, white or black will be produces.

Because Windows provides only a limited range of colors, using certain colors with HighlightColor may not produce useful highlights, i.e. there may be little or no difference between the specified color and the highlight color. For example, if you attempt to brighten the brightest color that Windows can display (`MAXCOLOR` or "high white") the return value of this function will be `MAXCOLOR`. Similarly, black can't be made darker than "zero".

And if you attempt to shade a very intense, bright color such as Yellow (color value `&h00FFFF`) the HighlightColor function won't be able to add more Yellow because the color is already "saturated". It will add white instead, in an attempt to create a slightly brighter Yellow, but this does not always produce the same *amount* of highlighting as adding more color.

For this reason, the HighlightColor function works best when you give it some

"breathing room". Try to use colors that are not fully saturated or too close to white or black. Even a small reduction in a color's saturation -- say from 255 to 248 -- can produce a color/highlight combination that is much more pleasing.

Also keep in mind that not all computer systems are configured to use TrueColor. If a system is using limited number of colors at runtime (16, 256, 32k, or 64k colors) it may not be able to reproduce *subtle* color differences.

See HSW Colors for more information about "fine tuning" colors to have certain characteristics.

Example

```
BrushColor LOBLUE

lResult& = HighlightColor(LOBLUE, 200)

PenColor lResult&
```

See Also

Pens and Brushes

HSW

Purpose

Produces the Windows Color value that corresponds to the specified Hue, Saturation, and Whiteness values.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = HSW(lHue&, _
               lSaturation&, _
               lWhiteness&)
```

Parameters

lHue&, lSaturation&, and *lWhiteness&*

The Hue, Saturation, and Whiteness values of the desired color.

Return Value

The value of `lResult&` will be a Windows Color value, between zero (0) and `MAXCOLOR` (white).

Remarks

The HSW function can be used *anyplace* that a Windows color value is required. For example, the PenColor function can be used like this...

```
PenColor HIRED
```

...or like this...

```
PenColor HSW(0,255,0)
```

For a complete discussion of this function, please refer to HSW Colors.

Example

See HSW Colors for many different examples.

See Also

Windows Colors

HSWtoRGB

Purpose

Converts Hue, Saturation and Whiteness values to the corresponding Red, Green, and Blue values.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
HSWtoRGB  X&, Y&, Z&
```

Parameters

X&, Y&, and *Z&*

These are input/output values, so you should always use *variables* for these parameters. If you use these parameters to pass Hue, Saturation, and Whiteness values to this function, it will return Red, Green, and Blue values in their place.

Return Value

The return value of this function will always be SUCCESS (zero) so it is safe to ignore it. The useful values that this function returns are the X&, Y&, and Z& parameters.

Remarks

This function can be used to obtain the Red, Green, and Blue values that correspond to the Hue, Saturation, and Whiteness values that you provide.

Compare this function to the RGBtoHSW function.

Please refer to HSW Colors for a complete discussion of this topic.

Example

```
'specify the H, S, and W values...
X& = 90
Y& = 30
Z& = 40

HSWtoRGB X&,Y&,Z&

'The X&, Y&, and Z& variables now contain
'the R, G, and B values that correspond to
'the specified HSW color.
```

See Also

HSW Colors

HueValue

Purpose

Returns the hue (the "color angle") of a Windows Color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = HueValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and `MAXCOLOR`.

Return Value

This function will always return a value between zero (0) and 359 which indicates the hue of the specified color, expressed as an angle, in degrees. (If you use the GfxOption `GFX_USE_RADIANS` function, the return value of this function will be expressed in radians instead of degrees. This value is *not* affected by the GfxOption `GFX_BASE_ANGLE` function.)

Remarks

The HueValue, SaturationValue, and WhitenessValue functions can be used to determine the Hue, Saturation, and Whiteness of a Windows Color.

Example

```
If HueValue(lColor&) = 120 Then
      'The color is pure green.
End If
```

See Also

HSW Colors

IconParam

Purpose

Provides information about an icon.

Availability:

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = IconParam(sIcon$, _
                     lInfoType&)
```

Parameters

sIcon$

Use either **1)** the name of an icon file, or **2)** an empty string, for the default Graphics Tools icon, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools icon or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows icon (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of an icon that is embedded in an EXE or DLL file.

lInfoType&

Use one of the values listed in **Remarks** below.

Return Value

The value of *lResult&* will be the requested value, or a Graphics Tools Error Code.

Remarks

By using different values for *lInfoType&*, this function can be used to obtain five different pieces of information about an icon.

`IMAGE_WIDTH` and `IMAGE_HEIGHT` can be used to obtain the width and height of the icon, in pixels. (These values indicate the *native* size of the icon, without regard to the size of the graphics window, the screen resolution, the Graphics Tools World, or the displayed size of the icon.)

You can also obtain the width and height of an icon in a single call to IconParam by using `IMAGE_WIDTH_HEIGHT`. This is significantly more efficient than using IconParam twice. The return value *lResult&* will contain the width of the icon in the Low Word and the height of the icon in the High Word. See High Words and Low Words.

The following values are "synthesized" values that are derived from the bitmap that is internally associated with each Windows icon.

`IMAGE_BITS_PER_PIXEL` indicates the number of bits that are required to store the color value of one pixel. Common values include one (monochrome), four (16 colors), eight (256 colors), sixteen (32k or 64k colors), twenty-four ("Windows Color" or "TrueColor") and thirty-two (TrueColor32).

`IMAGE_PLANES` is the number of color planes in the icon. Monochrome icons will

always return one (1).

`IMAGE_BYTES_PER_LINE` is the number of bytes that are used for each scan line. Microsoft's documentation provides conflicting information about this value, saying that "*This value must be divisible by 2, because Windows assumes that the bit values of a bitmap form an array that is word aligned*" but also saying that "*Each scan is a multiple of 32 bits*" which would imply that the value will always be divisible by *four*.

Example

```
lResult& = IconParam("\GfxTools\Images\Icon32.ico", _
                     IMAGE_HEIGHT)
```

See Also

BitmapParam, CursorParam, JpegParam

ID

Purpose

This read-only Visual Basic property returns the internal ID number that Graphics Tools uses to identify a graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = GfXWindow*.ID
```

Parameters

None.

Return Value

The internal ID number that Graphics Tools uses to identify *GfxWindow**.

Remarks

Many different Graphics Tools functions require you to identify which graphics window you want to affect. In Visual Basic programs this is usually done by specifying the *name* of the graphics window, like `GfxWindow1.DrawCircle`. In some cases however, such as when the Ex functions are used, you must specify a graphics window *number*.

This is *not* the same number that Visual Basic uses in the control's name. For example, `GfxWindow1` will usually have an ID value of zero (0). Changing a control's name has no effect on the ID value.

See Two Of Everything: The Ex Functions for a complete discussion of this topic.

Example

```
DrawCircleEx  GfxWindow1.ID, 100
```

See Also

Using Graphics Tools with Visual Basic

ImageParam

Purpose

Provides information about an image file.

Availability:

Graphics Tools Standard and Pro

Warning

This function's ability to obtain information about various image formats is dependent on the version of Windows that is installed on the runtime system, and other factors. See Displaying Images From Files for more information. Also see BitmapParam, IconParam, CursorParam, and JpegParam for functions that are *not* system-dependent.

Syntax

```
lResult& = ImageParam(sFileName$, _
                      lInfoType&)
```

Parameters

sFileName$

The name of an image file.

lInfoType&

Use one of the values listed in **Remarks** below.

Return Value

The value of *lResult&* will be the requested value, or a Graphics Tools Error Code.

Remarks

IMORTANT NOTE: If you need information about a bitmap, cursor, icon, or JPEG file it is usually more efficient to use the BitmapParam, CursorParam, IconParam, or JpegParam function than to use this function.

By using different values for *lInfoType&*, this function can be used to obtain two different pieces of information about an image.

`IMAGE_WIDTH` and `IMAGE_HEIGHT` can be used to obtain the width and height of the image, in pixels. (These values indicate the *native* size of the image, without regard to the size of the graphics window, the screen resolution, the Graphics Tools scaling mode, or the displayed size of the image.)

You can also obtain both the width and height of an image in a single call to ImageParam by using `IMAGE_WIDTH_HEIGHT`. This is significantly more efficient than using ImageParam twice. The return value *lResult&* will contain the width of the image in the Low Word and the height of the image in the High Word. See High Words and Low Words.

Example

```
lResult& = ImageParam("\GfxTools\Images\Locator.jpg", _
                      IMAGE_HEIGHT)
```

See Also

BitmapParam, CursorParam, IconParam

InitGfx

This function was used by Graphics Tools Version 1.x programs to both initialize Graphics Tools and to create a graphics window.

Graphics Tools Version 2.x requires the use of the InitGraphicsTools function to initialize Graphics Tools, and the OpenGfx function to create one or more graphics windows. Please note that it is also necessary to use the GraphicsToolsAuthorize function before InitGraphicsTools can be used.

InitGradientHSW

Purpose

Initializes a GT_HSW_GRADIENT structure so that it can be used as the basis for a new HSW gradient.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = InitGradientHSW(tGradient)
```

(See Syntax Options)

Parameters

tGradient

A variable that was dimensioned as a User Defined Type of type GT_HSW_GRADIENT.

Return Value

The value of *lResult&* will always be SUCCESS (zero).

Remarks

See Designing Your Own Gradients for more information about using this function.'

Example

See Designing Your Own Gradients.

See Also

Gradients

InitGradientRGB

Purpose

Initializes a GT_RGB_GRADIENT structure so that it can be used as the basis for a new HSW gradient.

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = InitGradientRGB(tGradient)
```

(See Syntax Options)

Parameters

tGradient

A variable that was dimensioned as a User Defined Type of type GT_RGB_GRADIENT.

Return Value

The value of *lResult&* will always be SUCCESS (zero).

Remarks

See Designing Your Own Gradients for more information about using this function.'

Example

See Designing Your Own Gradients.

See Also

Gradients

InitGraphicsTools

Purpose

Initializes certain Graphics Tools internal parameters, in preparation for the creation of graphics windows.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = InitGraphicsTools(lMaxWindow&, _
                             lToolset&)
```

VB: This function is not needed by VB programs.

(See Syntax Options)

Parameters

lMaxWindow&

The maximum window number that your program will use, between zero (0) and the maximum window number that Graphics Tools can create. Graphics Tools Standard can create up to four windows (numbers 0-3) and Graphics Tools Pro can create up to 256 (numbers 0-255).

lToolset&

The default toolset, either `GFX_TOOLSET_WINDOWS`, `GFX_TOOLSET_CONSOLE`, or `GFX_TOOLSET_PRINTER`. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

It is not necessary for your program to use this function unless you create graphics using a resource script, PowerBASIC's DDT system, or the CreateWindowEx API function. *Visual Basic programs and programs that use the OpenGfx function do not need to use this function.*

The primary function of InitGraphicsTools is to tell the runtime files how many graphics windows your program will be creating. If you attempt to create a graphics window with a number larger than *lMaxWindow&*, or if you attempt to create a graphics window before InitGraphicsTools has been called, the window creation will fail. Application Errors (General Protection Faults) are possible.

This function is also used to tell Graphics Tools which "toolset" to use when a new graphics window is created.

`GFX_TOOLSET_CONSOLE` creates a black brush, white pen, and white font, for use with a console window (which has a black background and uses a white console font by default).

`GFX_TOOLSET_PRINTER` uses a white brush, black pen, and black font, for use with non-color printers.

`GFX_TOOLSET_WINDOWS` uses a brush, pen, and font color that are determined at runtime, based on the system's user preferences. If the system is using the default Windows color scheme, this results in a gray brush, a black pen, and a black font.

Keep in mind that the toolset defines only the *initial* tools. Your program is, of course, free to create new tools at any time.

Example

```
'Create up to 2 graphics windows (numbers 0 and 1)
'and start with the "printer" toolset.
InitGraphicsTools 1, GFX_TOOLSET_PRINTER
```

See Also

The Graphics Window

Initialize

Purpose

This Visual Basic Event is fired when a graphics window is first created. (There is no corresponding Graphics Window Message for non-VB programs.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

None.

Remarks

The Initialize Event is fired when a graphics window is first created, immediately before the Load Event.

See Also

Using Graphics Tools With Visual Basic

JpegParam

Purpose

Provides information about a JPEG image file.

Availability:

Graphics Tools Pro Only

Warning

This function is not thread-safe. If you are writing a multi-threaded application you are responsible for protecting this function with an appropriate synchronization object (Critical Section, mutex, etc.)

Syntax

```
lResult& = JpegParam(sFileName$, _
                     lInfoType&)
```

Parameters

sFileName$

The name of a JPEG image file.

lInfoType&

Use one of the values listed in **Remarks** below.

Return Value

The value of *lResult&* will be the requested value, or a Graphics Tools Error Code.

Remarks

By using different values for *lInfoType&*, this function can be used to obtain two different pieces of information about a JPEG image.

`IMAGE_WIDTH` and `IMAGE_HEIGHT` can be used to obtain the width and height of the image, in pixels. (These values indicate the *native* size of the image, without regard to the size of the graphics window, the screen resolution, the Graphics Tools scaling mode, or the displayed size of the image.)

You can also obtain both the width and height of a JPEG image in a single call to JpegParam by using `IMAGE_WIDTH_HEIGHT`. This is significantly more efficient than using JpegParam twice. The return value *lResult&* will contain the width of the image in the Low Word and the height of the image in the High Word. See High Words and Low Words.

Example

```
lResult& = JpegParam("\GfxTools\Images\Locator.jpg", _
                     IMAGE_HEIGHT)
```

See Also

BitmapParam, CursorParam, IconParam

Keypress

Purpose

This Visual Basic Event is fired when the user presses a key while the graphics window has keyboard focus. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

Keycode and *Shift*

These are standard Visual Basic keypress values. Please consult the VB documentation for complete information.

Remarks

The Keypress Event is fired when the user presses a key while the graphics window has the keyboard focus.

If your program changes the value of *KeyCode* to zero (0) the keystroke will be ignored. Otherwise Graphics Tools will process the keystroke normally.

For example, if the user presses PgDn the graphics window (if it has scroll bars) would be scrolled down. If your program changes the value of *KeyCode* to zero, the window will not be scrolled.

See Also

Using Graphics Tools With Visual Basic

Load

Purpose

This Visual Basic Event is fired when a graphics window is first created. (There is no corresponding Graphics Window Message for non-VB programs.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

None.

Remarks

The Load Event is fired when a graphics window is first created, immediately after the Initialize Event.

See Also

Using Graphics Tools With Visual Basic

MatchConsoleFont

See the Console Tools Documentation for information about the MatchConsoleFont function.

MouseDown, MouseMove, MouseRightDown, MouseRightUp, and MouseUp

Purpose

These Visual Basic Events are fired when the user presses or releases a mouse button or moves the mouse over the graphics window. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

X and *Y*

The window location, in Drawing Units, where the mouse event occurred.

Shifts

See **Remarks** below.

Remarks

These Events are fired when the user presses or releases a mouse button or moves the mouse over the graphics window.

The *Shifts* parameter may contain one or more of the following values, indicating that one or more shift-keys were being held down when the mouse event occurred.

```
MK_SHIFT
MK_CONTROL
```

See Also

Using Graphics Tools With Visual Basic

MouseDownCaption and MouseUpCaption

Purpose

These Visual Basic Events are fired when the user presses or releases the left mouse button while the mouse cursor is over a graphics window's caption. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Pro Only

Warning

None.

Parameters

None.

Remarks

These Events are fired when the user presses or releases the left mouse button while the mouse cursor is over a graphics window's caption. Also see CaptionClick.

See Also

Using Graphics Tools With Visual Basic

MouseOver

Purpose

Returns the current location of the Windows mouse cursor, or the number of the graphics window that is located below the mouse cursor.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = MouseOver(lWhich&)
```

VB read-only property: *GfxWindow**.MouseOver

```
lResult& = MouseOverEx(lWindowNumber&, _
                       lWhich&)
```

Parameters

lWhich&

One of the following: WINDOW_NUMBER, HORIZONTAL_AXIS, VERTICAL_AXIS, MOUSE_LOCATION, RELATIVE_TO_WINDOW, RELATIVE_TO_PARENT, or RELATIVE_TO_SCREEN. See **Remarks** below.

Return Value

See **Remarks** below.

Remarks

If *lWhich&* is WINDOW_NUMBER, this function will return the window number of the graphics window over which the mouse cursor is located. If the mouse cursor is not located over a graphics window that was created by your program, this function will return GFX_NONE.

If *lWhich&* is HORIZONTAL_AXIS or VERTICAL_AXIS this function will return the X or Y drawing location of the mouse cursor, in Drawing Units. (The MouseOverX and MouseOverY functions can also be used to obtain these values.) If the mouse cursor is located over a graphics window but outside the drawing area, one of the following special values will be returned:

If the cursor is over the window's border...

```
GFX_BORDER_LEFT
GFX_BORDER_RIGHT
GFX_BORDER_TOP  (includes caption)
GFX_BORDER_BOTTOM
```

If the cursor is over a scrollbar button...

```
GFX_SCROLLBUTTON_UP
GFX_SCROLLBUTTON_DOWN
GFX_SCROLLBUTTON_LEFT
GFX_SCROLLBUTTON_RIGHT
```

If the cursor is over a scrollbar but not over a button...

```
GFX_SCROLLBAR_RIGHT
GFX_SCROLLBAR_BOTTOM
```

(It is important to note that whenever HORIZONTAL_AXIS returns GFX_SCROLLBAR_RIGHT, VERTICAL_AXIS will return a value that indicates the vertical position of the mouse *along the scroll bar*. The value is calibrated in Drawing Units and can be used with the GfxScroll(SCROLL_VERT_POS) function. The same is true for VERTICAL_AXIS, GFX_SCROLLBAR_BOTTOM, and SCROLL_HORIZ_POS. See the PB/CC sample program Resize_ProOnly.BAS for an example.)

If the cursor is over the area to the right of the horizontal scroll bar and below the vertical scroll bar...
GFX_SIZE_HANDLE

(Note that GFX_SIZE_HANDLE will be returned even if the graphics window is not sizable and therefore does not have an actual sizing handle. This value refers to the square *area* where a sizing handle is normally located.)

If the mouse cursor is not located over the graphics window, this function will return GFX_NONE.

If *lWhich&* is MOUSE_LOCATION the return value will contain the HORIZONTAL_AXIS value in the low word and the VERTICAL_AXIS value in the high word. (See High Words and Low Words.) Using this option is more efficient than using the MouseOver function twice. Note that because these horizontal and vertical values are limited to the WORD range (0-64k), the GFX_BORDER_ GFX_SCROLL..., and GFX_NONE values cannot be returned "directly" when MOUSE_LOCATION is used. For example, instead of GFX_BORDER_LEFT (&h7FFFFF21&) this function will return &hFF21.

If *lWhich&* is RELATIVE_TO_WINDOW this function will return a HiWord/LoWord value similar to MOUSE_LOCATION but in *pixels* (not drawing units) relative to the top-left corner of the drawing window (which is not necessarily the same thing as the graphics window's 0,0, location). Note that RELATIVE_TO_WINDOW will never return a GFX_BORDER_ value or GFX_NONE (see above) so it can be used to determine the location of the mouse even when it is not located over a graphics window.

If *lWhich&* is RELATIVE_TO_PARENT this function will return a value (in pixels) similar to RELATIVE_TO_WINDOW but the HiWord/LoWord values will indicate the mouse's location relative to the top-left corner of the graphics window's *parent* window, not the graphics window itself.

If *lWhich&* is RELATIVE_TO_SCREEN this function will return a value (in pixels) similar to RELATIVE_TO_WINDOW but the HiWord/LoWord values will indicate the mouse's location relative to the top-left corner of the Windows desktop.

Example

```
lResult& = MouseOver(WINDOW_NUMBER)
'lResult& now contains either a graphics
'window number or the value GFX_NONE.
```

See Also

MouseOverX, MouseOverY

MouseOverX

Purpose

Returns the current X (left-right) location of the Windows mouse cursor.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = MouseOverX
```

VB real-only property: *GfxWindow**.MouseOverX

```
lResult& = MouseOverXEx(lWindowNumber&)
```

Parameters

None.

Return Value

See **Remarks** below.

Remarks

The value of *lResult&* will be the graphics window X location that corresponds to the current mouse cursor location, in Drawing Units, unless it is one of the following special values.

If the cursor is over the window's border...
GFX_BORDER_LEFT
GFX_BORDER_RIGHT

If the cursor is over a scrollbar button...
GFX_SCROLLBUTTON_UP
GFX_SCROLLBUTTON_DOWN
GFX_SCROLLBUTTON_LEFT
GFX_SCROLLBUTTON_RIGHT

If the cursor is over the vertical scrollbar but not over a button...
GFX_SCROLLBAR_RIGHT

(It is important to note that whenever MouseOverX returns GFX_SCROLLBAR_RIGHT, MouseOverY will return a value that indicates the vertical position of the mouse *along the scroll bar*. The MouseOverY value is calibrated in Drawing Units and can be used with the GfxScroll(SCROLL_VERT_POS) function.)

If the cursor is over the area to the right of the horizontal scroll bar and below the vertical scroll bar...
GFX_SIZE_HANDLE

(Note that GFX_SIZE_HANDLE will be returned even if the graphics window is not sizable and therefore does not have an actual sizing handle. This value refers to the square *area* where a sizing handle is normally located.)

If the mouse cursor is not currently located over the graphics window, GFX_NONE will

be returned.

It is important to note that this function will return a value even if the graphics window is not currently visible. For example, imagine that your program occupies the middle of the desktop, but that another program is covering it. If the mouse cursor is positioned over the middle of the application that is covering yours, and you use the MouseOverX function to obtain the mouse cursor location, it *will* return a value.

Example

```
If MouseOverX = GFX_NONE Then
      'The cursor is NOT over
      'the graphics window
Else
      'The cursor is somewhere
      'over the graphics window
End If
```

See Also

MouseOverY

MouseOverY

Purpose

Returns the current Y (up-down) location of the Windows mouse cursor.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = MouseOverY
```

VB read-only property: *GfxWindow**.MouseOverY

```
lResult& = MouseOverYEx(lWindowNumber&)
```

Parameters

None.

Return Value

See **Remarks** below.

Remarks

The value of lResult& will be the graphics window Y location that corresponds to the current mouse cursor location, in Drawing Units, unless it is one of the following special values.

If the cursor is over the window's border...
- GFX_BORDER_TOP (includes caption)
- GFX_BORDER_BOTTOM

If the cursor is over a scrollbar button...
- GFX_SCROLLBUTTON_UP
- GFX_SCROLLBUTTON_DOWN
- GFX_SCROLLBUTTON_LEFT
- GFX_SCROLLBUTTON_RIGHT

If the cursor is over the horizontal scrollbar but not over a button...
- GFX_SCROLLBAR_BOTTOM

(It is important to note that whenever MouseOverY returns GFX_SCROLLBAR_BOTTOM, MouseOverX will return a value that indicates the horizontal position of the mouse *along the scroll bar*. The MouseOverX value is calibrated in Drawing Units and can be used with the GfxScroll(SCROLL_HORIZ_POS) function.)

If the cursor is over the area to the right of the horizontal scroll bar and below the vertical scroll bar...
- GFX_SIZE_HANDLE

(Note that GFX_SIZE_HANDLE will be returned even if the graphics window is not sizable and therefore does not have an actual sizing handle. This value refers to the square *area* where a sizing handle is normally located.)

If the mouse cursor is not currently located over the graphics window, GFX_NONE will be returned.

It is important to note that this function will return a value even if the graphics window is not currently visible. For example, imagine that your program occupies the middle of the desktop, but that another program is covering it. If the mouse cursor is positioned over the middle of the application that is covering yours, and you use the MouseOverY function to obtain the mouse cursor location, it *will* return a value.

Example

```
If MouseOverY = GFX_NONE Then
      'The cursor is NOT over
      'the graphics window
Else
      'The cursor is somewhere
      'over the graphics window
End If
```

See Also

MouseOverX

Moved

The current OCX version of Graphics Tools does not support this Event. It is reserved for possible future use.

If you want to move a graphics window, use the standard Visual Basic `GfxWindow*.Left` and `.Top` properties. The window will be moved, but no event will be generated.

MoveLPR

Purpose

Moves the LPR by the specified distance(s).

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = MoveLPR(lDistanceX&, _
                   lDistanceY&)
```

VB method: *GfxWindow**.MoveLPR

```
lResult& = MoveLPREx(lWindowNumber&, _
                     lDistanceX&,
                     lDistanceY&)
```

Parameters

lDistanceX& and *lDistanceY&*

The distance(s) that the LPR should be moved.

Return Value

Unless you attempt to use it with a graphics window number that does not exist, the return value of this function will always be SUCCESS (zero) so it is usually safe to ignore it.

Remarks

If you use a nonzero value for *lDistanceX&* and/or *lDistanceY&*, the LPR will be moved by that amount.

Example

```
'Move the LPR one drawing unit down
'and one to the right...
MoveLPR 1,1
```

See Also

DrawFrom

OpenGfx

Purpose

Creates a graphics window.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

(VB programs do not use these functions unless they are using the Direct To DLL method.)

```
lResult& = OpenGfx(hParentWindow&, _
                   lLeft&, _
                   lTop&, _
                   lWidth&, _
                   lHeight&, _
                   lStyles&)

lResult& = OpenGfxEx(lWindowNumber&, _
                     hParentWindow&, _
                     lLeft&, _
                     lTop&, _
                     lWidth&, _
                     lHeight&, _
                     lStyles&) AS LONG
```

(See Syntax Options)

Parameters

hParentWindow&

The window that should be used for the graphics window's parent.

lLeft& and lTop&

The location of the graphics window, relative to the top-left corner of the parent window, in pixels. (These values can be expressed in Dialog Units instead of pixels if you use the GfxOption(`GFX_USE_DLG_UNITS`) setting.)

lWidth& and lHeight&

The size of the graphics window, in pixels. (These values can be expressed in Dialog Units instead of pixels if you use the GfxOption(`GFX_USE_DLG_UNITS`) setting.)

lStyles&

The window styles that the graphics window should have. You can also use the value `GFX_DEFAULT` to specify a style that has been defined with the GfxOption `GFX_WINDOW_DEFAULT_STYLE` option.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

Graphics Tools provides several different method for creating a graphics window. The OpenGfx function is the most commonly used method.

If you use the OpenGfx function to open a graphics window number that is already open, the existing window will be automatically closed before the new window is created. See Using Multiple Graphics Windows for more information.

The first parameter of the OpenGfx function is used to specify which window or dialog should be used as the parent window of the graphics window. The parent window is the one to which the graphics window is "attached". In most cases this will be your program's main window or dialog.

For example, if you are using the PowerBASIC DDT system to create a 500x300 main dialog like this...

```
Dialog New 0, "My Application",,, 500, 300 To hDlg&
```

...then the main dialog's handle would be returned in the *hDlg&* variable, and you would use *hDlg&* as the first parameter of the OpenGfx function. If you create a main window or dialog using another technique (such as an API function), the principle is exactly the same.

The *lLeft&* and *lTop&* parameters are used to specify where on the parent window the graphics window should be located. The *lWidth&* and *lHeight&* parameters specify how large the graphics window should be. It is important to note that all four of these values must normally be specified in *pixels*. The values can be expressed in Dialog Units instead of pixels if you use the GfxOption(`GFX_USE_DLG_UNITS`) setting.

Continuing the 500x300 example from above, keep in mind that the DDT system (and several of the Windows API functions that can be used to create windows) use Dialog Units, not pixels. For more information about Dialog Units, consult your programming language's documentation or any Windows API reference.

If you were to create a graphics window that was 500x300 pixels, it would *not* be the same size as the dialog. You must convert the Dialog Units to pixels in order to obtain useful values. PowerBASIC DDT programs can use the `DIALOG UNITS TO PIXELS` function, like this:

```
Dialog Units hDlg&, 500, 300 To Pixles lWidth&, lHeight&
```

That code would convert 500 horizontal Dialog Units into pixels and place the result in the *lWidth&* variable, and convert 300 vertical Dialog Units into pixels and place the result in the *lHeight&* variable. You could then use *lWidth&* and *lHeight&* with the OpenGfx function. Windows API functions are also available to perform this type of conversion.

In a real-world program you probably would not want to fill the entire dialog, you would want to leave room for buttons and other controls. So you would probably not use 500 and 300, you would use somewhat smaller numbers.

If you do not want your graphics window to be located at the top-left corner of the parent window, you would need to perform similar calculations for the *lLeft&* and *lTop&* values.

By the way, for technical reasons, Graphics Tools can only be used to create graphics windows with widths and heights that are both *even numbers*. If you specify an odd number of pixels for the width or height when creating a graphics window, Graphics Tools will automatically subtract one (1) pixel to insure that the final value is an even number.

For the best possible graphics performance, you should (if possible) choose graphics window *width* measurements that are multiples of *four* (4) pixels. Windows is a 32-bit operating system, and it works best when using multiples of 32 bits (4 bytes). The *height* can be any even number.

If you use zero (0) for the *lWidth&* and/or *lHeight&* parameter, the graphics window will automatically fill the parent window in the corresponding direction(s). If you use a negative value for *lWidth&* and/or *lHeight&*, that amount will be subtracted from the width/height of the parent window. For example, using negative ten (`-10`) for the *lWidth&* parameter would create a graphics window with a width that was ten pixels less than the width of the parent window.

You can therefore create a graphics window that fills the parent window except for a 10-pixel "border" by using code like this...

```
OpenGfx hDlg&, 10, 10, -10, -10, 0
```

For a detailed discussion of the *lStyle&* parameter, see Graphics Window Styles.

Creating Oversized Graphics Windows

It is possible to use the OpenGfx function to create a graphics window that is *larger* than its parent window. In most cases you will then want to use the Pro version's GfxResize function to reduce the "visible" size of the window so that it fits on the parent window. When this is done, scroll bars will be automatically added to the graphics window. See Window Styles for more about scroll bars.

Example

```
'Create a 200x200-pixel graphics window that is
'located 10 pixels from the top and left edges of
'the parent window (hWnd&), using the "raised" style.

OpenGfx hWnd&, 10, 10, 200, 200, GFX_STYLE_RAISED
```

See Also

The Graphics Window

OverlayWindow

Purpose

Copies an image from one drawing window into another, ignoring a certain color. This results in certain areas of the copied image appearing to be transparent because the target window's image "shows through".

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = OverlayWindow(lSourceWindow&, _
                         lTransparentColor&)
```

VB method: `GfxWindow*.OverlayWindow`

```
lResult& = OverlayWindowEx(lTargetWindow, _
                           lSourceWindow&, _
                           lTransparentColor&)
```

(See Syntax Options)

Parameters

lSourceWindow&

The Graphics Window from which the image should be copied.

lTransparentColor&

The color which, when it appears in the source window's image, should be ignored. You can specify **1)** a color value or **2)** `GFX_AUTO`, which tells the OverlayWindow function to automatically use the color of the top-left pixel (1,1) of the source window as the transparent color.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested window's image is copied without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function requires the use of two graphics windows. The first window, called the source window, is (usually) hidden from the user. The second window, called the target window, is (usually) visible to the user. You should familiarize yourself with Using Multiple Graphics Windows before attempting to use this function.

When the OverlayWindow function is used, the image from the source window is copied to the target window just as if the DisplayWindow function had been used, except that *one specific color* is ignored. The areas of the source window which contain that color are *not* copied, resulting in areas that appear to be transparent.

You can visualize the process this way...

1) Imagine that your program creates a normal graphics window and draws something in it. It doesn't matter what it draws. (The Transparent Areas example program draws a number of random ellipses.)

2) Your program then creates a second, hidden graphics window by using the OpenGfxEx function.

3) In the hidden window, your program creates a Blue brush and uses GfxClsEx to fill the window with Blue. Then it uses a Red pen to draw several lines, and White pen to draw come circles.

4) Finally, your program uses the OverlayWindow function to overlay the image from the hidden window onto the main graphics window. If you use Blue for the *lTransparentColor&* parameter, the blue portions of the hidden window will *not* be copied. Only the Red lines and White circles will be copied. The Blue background will be treated as "transparent".

The GfxOption `GFX_OVERLAY_MODE` setting can be used to modify the non-transparent portion of the image. You can use `OVERLAY_BLEND25` or `OVERLAY_BLEND50` to produce somewhat different effects, or use `OVERLAY_TRANSPARENT` (the default value) to produce a normal overlay.

Example

See the Drawing An Image With Transparent Areas sample programs.

See Also

DrawHole

Pen

Purpose

Creates a Pen that can be used by various other functions.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = Pen(lColor&, _
               lStyle&, _
               lWidth&)
```

VB: Use the PenColor, PenStyle, and PenWidth properties instead of the Pen function.

```
lResult& = PenEx(lWindowNumber&, _
                 lColor&, _
                 lStyle&, _
                 lWidth&)
```

(See Syntax Options)

Parameters

lColor&

The color of the Pen, from zero (0) to `MAXCOLOR` (white).

lStyle&

The style of the Pen. See **Remarks** below for a list of valid values.

lWidth&

The width of the Pen. See **Remarks** below.

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the requested pen is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See Pens and Brushes for an overview of Pens.

The *lColor&* parameter must be a valid Windows Color value. You may, of course, use the HSW function to create a color value.

The *lStyle&* parameter must be one of the following values:

`GFX_SOLID` for a solid-color pen.
`GFX_CLEAR` for a "hollow" or "null" pen that is not visible when used.
`GFX_DASHED` for a line made up of dashes.
`GFX_DOTTED` for a line made up of dots.
`GFX_DASHDOT` for a repeating dash-dot pattern.
`GFX_DASHDOTDOT` for a repeating dash-dot-dot pattern.

In the case of GFX_DASHED, GFX_DOTTED, GFX_DASHDOT, and GFX_DASHDOTDOT, the "drawn" part of the line will be the color specified by *lColor&*, and the "in between" color will be determined by the GfxBkgdMode and (possibly) the GfxBkgdColor functions.

It is important to note that Microsoft Windows requires the use of a certain *size* Pen when styles other than GFX_SOLID and GFX_CLEAR are used, so Graphics Tools automatically uses a Pen *lWidth&* of one (1) pixel when a "pattern" Pen is created. If you use a pattern Pen and attempt to change the Pen width, a solid pen will be automatically selected.

Pen Width

A Pen's *lWidth&* value is normally "scaled" in the same way that all Graphics Tools drawing operations are scaled. When you tell Graphics Tools to create a Pen of width *lWidth&*, the size of the graphics window will be taken into account. If the graphics window is small, a small Pen will be created. If the graphics window is larger, a somewhat larger Pen will be created. This is done to produce consistent results, regardless of the size of the graphics window, when different size pens are used.

The *lWidth&* value is not, however, calibrated in Drawing Units. The scale is much "finer" than that.

Creating a Pen of *lWidth&* zero (0) tells Windows to use a default Pen size, which is exactly one (1) pixel wide.

Using an *lWidth&* of one (1) will usually create a Pen that is one pixel wide. (To be precise, it will be one "logical unit" wide. More about this in a moment.)

Using an *lWidth&* value of two (2), however, will *not* usually create a two-pixel Pen. You will usually (depending on the size of the graphics window) need to use a value of ten (10) or larger to create a two-pixel-wide pen.

It should be noted that because Pens can be used to draw vertical, horizontal, or angled lines, the Pen width must be scaled based on a *combination* of the graphics window's horizontal and vertical scales. If you are using a graphics window that is not a square world, this may produce unexpected results.

You can use a negative number for *lWidth&* to specify an exact Pen width. For example, using negative two (-2) will *usually* produce a two-pixel pen. The negative *lWidth&* value must be stated in units that Windows calls "logical units" which *usually* correspond to one pixel per unit. But, depending on the graphics hardware and software that is installed, Windows can perform its own scaling operations on the value that you submit. For example, if an extremely high screen resolution is being used, a one-pixel-wide line might not be easily visible, so Windows might automatically increase the size of the Pen to compensate. This type of scaling is beyond the control of your program, except that using zero (0) for the *lWidth&* parameter tells Windows to create a Brush that is one pixel wide, regardless of the current hardware configuration.

If is also possible to disable Graphic's Tools Pen scaling by using this code:

```
GfxOption GFX_PEN_SCALING, FALSE
```

This options is provided primarily for compatibility with Graphics Tools Version 1, which did not perform Pen scaling.

Example

```
Pen HIGREEN, SOLID, 1
```

See Also

PenColor, PenWidth, PenStyle

PenColor

Purpose

Changes the color of the current Pen, without changing the pen's style or width. (If you are changing two or more Pen parameters, it is more efficient to use the Pen function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = PenColor(lColor&)
```

VB property: *`GfxWindow*`*`.PenColor`

```
lResult& = PenColorEx(lWindowNumber&, _
                      lColor&)
```

(See Syntax Options)

Parameters

lColor&

The color of the Pen, from zero (0) to `MAXCOLOR` (white).

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the requested pen is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The *lColor&* parameter must be a valid Windows Color value. You may, of course, use the HSW function to create a color value, as shown in the second **Example** below.

See Pens and Brushes for an overview of Pens.

Example

```
PenColor HIRED
```

or...

```
PenColor HSW(0,128,0)
```

See Also

PenStyle, PenWidth

PenStyle

Purpose

Changes the style of the current Pen, usually without changing the pen's color or width. (If you are changing two or more Pen parameters, it is more efficient to use the Pen function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = PenStyle(lStyle&)
```

VB property: *GfxWindow**.PenStyle

```
lResult& = PenStyleEx(lWindowNumber&, _
                      lStyle&)
```

(See Syntax Options)

Parameters

lStyle&

The style of the Pen. See **Remarks** below for a list of valid values.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested pen is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The *lStyle&* parameter must be one of the following values:

GFX_SOLID for a solid-color pen.

GFX_CLEAR for a "hollow" or "null" pen that is not visible when used.

GFX_DASHED for a line made up of dashes.

GFX_DOTTED for a line made up of dots.

GFX_DASHDOT for a repeating dash-dot pattern.

GFX_DASHDOTDOT for a repeating dash-dot-dot pattern.

In the case of GFX_DASHED, GFX_DOTTED, GFX_DASHDOT, and GFX_DASHDOTDOT, the "drawn" part of the line will be the color specified by *lColor&*, and the "in between" color will be determined by the GfxBkgdMode and (possibly) the GfxBkgdColor functions.

It is important to note that Microsoft Windows requires the use of a certain *size* Pen when styles other than GFX_SOLID and GFX_CLEAR are used, so Graphics Tools automatically uses a Pen width of one (1) pixel when a "pattern" Pen is created. If you use a pattern Pen and attempt to change the Pen width, a solid pen will be automatically selected.

See Pens and Brushes for an overview of Pens.

Example

```
PenStyle GFX_DOTTED
```

See Also

PenWidth, PenColor

PenWidth

Purpose

Changes the width of the current Pen, usually without changing the pen's color or style. (If you are changing two or more Pen parameters, it is more efficient to use the Pen function than this function.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = PenWidth(lWidth&)
```

VB property: *`GfxWindow*`*`.PenWidth`

```
lResult& = PenWidthEx(lWindowNumber&, _
                      lWidth&)
```

(See Syntax Options)

Parameters

lWidth&

The width of the Pen. See **Remarks** below.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested pen is created without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

A Pen's *lWidth&* value is normally "scaled" in the same way that all Graphics Tools drawing operations are scaled. When you tell Graphics Tools to create a Pen of width *lWidth&*, the size of the graphics window will be taken into account. If the graphics window is small, a small Pen will be created. If the graphics window is larger, a somewhat larger Pen will be created. This is done to produce consistent results, regardless of the size of the graphics window, when different size pens are used.

The *lWidth&* value is not, however, calibrated in Drawing Units. The scale is much "finer" than that.

Creating a Pen of *lWidth&* zero (0) tells Windows to use a default Pen size, which is exactly one (1) pixel wide.

Using an *lWidth&* of one (1) will usually create a Pen that is one pixel wide. (To be precise, it will be one "logical unit" wide. More about this in a moment.)

Using an *lWidth&* value of two (2), however, will *not* usually create a two-pixel Pen. You will usually (depending on the size of the graphics window) need to use a value of ten (10) or larger to create a two-pixel-wide pen.

It should be noted that because Pens can be used to draw vertical, horizontal, or angled lines, the Pen width must be scaled based on a *combination* of the graphics

window's horizontal and vertical scales. If you are using a graphics window that is not a square world, this may produce unexpected results.

You can use a negative number for *lWidth&* to specify an exact Pen width. For example, using negative two (-2) will *usually* produce a two-pixel pen. The negative *lWidth&* value must be stated in units that Windows calls "logical units" which *usually* correspond to one pixel per unit. But, depending on the graphics hardware and software that is installed, Windows can perform its own scaling operations on the value that you submit. For example, if an extremely high screen resolution is being used, a one-pixel-wide line might not be easily visible, so Windows might automatically increase the size of the Pen to compensate. This type of scaling is beyond the control of your program, except that using zero (0) for the *lWidth&* parameter tells Windows to create a Brush that is one pixel wide, regardless of the current hardware configuration.

If is also possible to disable Graphic's Tools Pen scaling by using this code:

```
GfxOption GFX_PEN_SCALING, FALSE
```

This options is provided primarily for compatibility with Graphics Tools Version 1, which did not perform Pen scaling.

See Pens and Brushes for an overview of Pens.

Example

```
PenWidth  20
```

See Also

PenColor, PenStyle

PixelColor

Purpose

Returns the current color of the specified pixel.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = PixelColor(lXPos&, _
                      lYPos&)
```

VB method: *`GfxWindow*`*`.PixelColor`
(Also see DrawPixel.)

```
lResult& = PixelColorEx(lWindowNumber&, _
                        lXPos&, _
                        lYPos&)
```

Parameters

lXPos& and *lYPos&*

The X (left-right) and Y (up-down) location of the pixel, in Drawing Units.

Return Value

The value of *lResult&* will be the Windows Color of the specified pixel.

If you specify a location that is outside the limits of the graphics window, or if another error occurs (such as using a graphics window that does not exist) negative one (`-1`) will be returned.

Remarks

It is sometimes useful to compare the color that your program attempts to use to draw a figure to the *actual* color of the pixel. See Windows Colors for more information about why those two values might be different.

Example

```
lResult& = PixelColor(100,100)
```

See Also

DrawPixel

PolyFillMode

Purpose

Determines how Graphics Tools fills polygons which have sides (lines) that intersect each other.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = PolyFillMode(lMode&)
```

VB method: *GfxWindow*.PolyFillMode*

```
lResult& = PolyFillModeEx(lWindowNumber&, _
                          lMode&)
```

Parameters

lMode&

Either WINDING_FILL (the default mode) or ALTERNATE_FILL. See **Remarks** below for more information.

Return Value

Unless the graphics window does not exist, the return value of this function will always be SUCCESS (zero) so it is safe to ignore it. (If you use an invalid value for *lMode&*, the default fill mode will be used.)

Remarks

This function affects only polygons that are created with the DrawPolygon and DrawXagon function, and only polygons with lines that cross each other are affected.

If you use the default mode, Graphics Tools will completely fill any polygon, using the current Brush.

If you use the "alternate" mode (that's the Windows terminology), polygons with crossed lines will be filled so that only the areas between odd-numbered and even-numbered sides are filled.

In practical terms, you can easily see the difference between the two modes if you use the DrawXagon function to draw a pentagram, which is a five-pointed "star" that is created with five straight lines. If you use the default mode, the entire star will be filled using the current Brush. If you use the alternate mode, only the "points" of the star will be filled, and the central pentagon will not be filled.

Example

See DrawXagon.

See Also

DrawPolygon

RedValue

Purpose

Returns the amount of red in a Windows Color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = RedValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and MAXCOLOR.

Return Value

This function will always return a value between zero (0) and 255 which indicates how much red the specified color contains.

Remarks

The RedValue, GreenValue, and BlueValue functions can be used to obtain the amounts of Red, Green, and Blue that a Windows Color contains.

Example

```
If RedValue(lColor&) > GreenValue(lColor&) Then
      'The color contains more red than green
End If
```

See Also

Windows Colors

Resized

Purpose

This Visual Basic Event is fired when the graphics window is resized. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Parameters

X and *Y*

The new size of the graphics window.

Remarks

This Event is fired when the graphics window is sized.

See Also

Using Graphics Tools With Visual Basic

RGBtoHSW

Purpose

Converts Red, Green, and Blue values to the corresponding Hue, Saturation, and Whiteness values.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
RGBtoHSW  X&, Y&, Z&
```

X&, Y&, and *Z&*

These are input/output values, so you should always use *variables* for these parameters. If you use these parameters to pass Red, Green, and Blue values to this function, it will return Hue, Saturation, and Whiteness values in their place.

Return Value

The return value of this function will always be SUCCESS (zero) so it is always safe to ignore it. The useful values that this function returns are the *X&*, *Y&*, and *Z&* parameters.

Remarks

This function can be used to obtain the Hue, Saturation, and Whiteness values that correspond to Red, Green, and Blue values that you provide.

Compare this function to the HSWtoRGB function.

Please refer to HSW Colors for a complete discussion of this topic.

Example

```
'specify the R, G, and B values...
X& = 90
Y& = 30
Z& = 40

RGBtoHSW  X&,Y&,Z&

'The X&, Y&, and Z& variables now contain
'the H, S, and W values that correspond to
'the specified RGB color.
```

See Also

HSW Colors

SampleText

Purpose

This Visual Basic Property specifies the sample text that is displayed in the graphics window at design time. It has no effect at runtime.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

The default value of this property is "Sample Text".

The sample text is automatically displayed in the graphics window at design time to make it easier to set up the various Font properties, such as FontWidth and FontHeight. The sample text is not displayed at runtime.

The SampleText property can be used as a convenient way to label multiple graphics windows at design time, by changing the property to "Graphics Window #1", "Graphics Window #2", and so on.

See Also

Using Graphics Tools With Visual Basic

SaturationValue

Purpose

Returns the saturation (the "color intensity") of a Windows Color.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = SaturationValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and `MAXCOLOR`.

Return Value

This function will always return a value between zero (0) and 255 which indicates how saturated the specified color is.

Remarks

The HueValue, SaturationValue, and WhitenessValue functions can be used to determine the Hue, Saturation, and Whiteness of a Windows Color.

Example

```
If SaturationValue(lColor&) = 0 Then
      'The color is black, gray, or white.
End If
```

See Also

HSW Colors

SaveGfxArea

Purpose

Saves a rectangular area of the graphics window to a standard Windows bitmap file, or to the Windows clipboard. If Graphics Tools Pro is being used, this function can also save JPEG files.

Availability

Bitmaps: Graphics Tools Standard and Pro
JPEG: **Graphics Tools Pro Only**

Warning

None.

Syntax

```
lResult& = SaveGfxArea(sFileName$, _
                       lLeft&, _
                       lTop&, _
                       lRight&, _
                       lBottom&)
```

VB method: *GfxWindow*.SaveGfxArea*

```
lResult& = SaveGfxAreaEx(lWindowNumber&, _
                         sFileName$, _
                         lLeft&, _
                         lTop&, _
                         lRight&, _
                         lBottom&)
```

(See Syntax Options)

Parameters

sFileName$

Either **1)** The name of the disk file that should be created, or **2)** the word "CLIPBOARD" in uppercase letters. (Recent versions of PowerBASIC can use the string equate $CLIPBOARD.) An empty string can also be used (see **Return Value** below).

lLeft& and *lTop&*

The X (left-right) and Y (up-down) locations of the top-left corner of the rectangle of the graphics window that you want to save in the file.

lRight& and *lBottom&*

The X (left-right) and Y (up-down) locations of the bottom-right corner of the rectangle.

Return Value

If an error occurs, this function will return a Graphics Tools Error Code.

If an empty string ("") is used for *sFileName$*, this function will not create a disk file, it will simply return the current screen "color depth" (1, 4, 8, 16, 24, or 32). If *sFileName$* is not an empty string...

Bitmaps: If the specified disk file is created without errors, the value of *lResult&* will

be greater than zero and less than or equal to 32. The number that is returned indicates the "color depth" of the bitmap that was created, either 1, 4, 8, 16, 24, or 32, as specified by the SaveGfxMode function.

JPEG Files: If the specified disk file is created without errors, the value of *lResult&* will be the "quality" of the JPEG image, between `JPEG_QUALITY` and `JPEG_BEST`, as specified by the SaveGfxMode function.

Remarks

This function can be used to save a portion of the current graphics window to a standard Windows bitmap file or to the Windows clipboard. If you are using Graphics Tools Pro it can also be used to save a JPEG file containing the image.

With the exception of the left/right/top/bottom parameters, this function is identical to SaveGfxWindow. Please refer to that function's entry in this document for additional details.

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL and you attempt to save a JPEG file, this function will return `ERROR_LIBRARY_NOT_FOUND`.

Example

```
SaveGfxArea "MyImage.BMP", 100,100,200,200
```

See Also

SaveGfxWindow

SaveGfxMode

Purpose

Determines the type of disk files that will be created by subsequent calls to SaveGfxArea and/or SaveGfxWindow.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = SaveGfxMode(lMode&)
```

VB method: *`GfxWindow*`*`.SaveGfxMode`

```
lResult& = SaveGfxModeEx(lWindowNumber&, _
                         lMode&)
```

(See Syntax Options)

Parameters

lMode&

See **Remarks** below.

Return Value

The value of *lResult&* will be always be SUCCESS (zero), so it is safe to ignore the return value of this function. (Using an invalid value for *lMode&* can cause future calls to SaveGfxArea and SaveGfxMode to fail and return an error code.)

Remarks

Using an *lMode&* value of zero (0) tells Graphics Tools to create bitmaps using the color depth of the runtime system, i.e. the current monitor color settings. This is the default *lMode&* value.

To tell Graphics Tools to save bitmap files of other types, use one of the following values to indicate the "color depth" of the bitmap files that will be created:

```
BMP_MONOCHROME (Black and White)
BMP_16_COLORS
BMP_256_COLORS
BMP_32K_COLORS
BMP_TRUECOLOR
BMP_TRUECOLOR32
```

Other monochrome bitmaps (besides Black And White) can be created by using negative values. For example, using an *lMode&* value of `0-HIGREEN` (zero minus the color value `HIGREEN`) would produce a black-and-green monochrome bitmap.

Graphics Tools Pro Only

To tell Graphics Tools to save JPEG files instead of bitmaps, use an *lMode&* value that is equal to `JPEG_QUALITY` plus the "quality" that you want to produces, from one

(1) to one hundred (100). For example, to create a "75 quality" JPEG image, use `JPEG_QUALITY+75`.

You can also use the following values which correspond to commonly-used qualities:

```
75:  JPEG_TYPICAL
100: JPEG_BEST
```

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL and you attempt to save a JPEG file, the SaveGfxArea and SaveGfxWindow functions will return `ERROR_LIBRARY_NOT_FOUND`.

Example

```
SaveGfxMode BMP_TRUECOLOR
```

See Also

Bitmaps and JPEG Files

SaveGfxScreen

This Graphics Tools Version 1 function has been renamed SaveGfxWindow.

SaveGfxWindow

Purpose

Saves the entire current graphics window to a standard Windows bitmap file, or to the Windows clipboard. If Graphics Tools Pro is being used, this function can also save JPEG files.

Availability

Bitmaps: Graphics Tools Standard and Pro
JPEG: **Graphics Tools Pro Only**

Warning

None.

Syntax

```
lResult& = SaveGfxWindow(sFileName$)
```

VB method: *GfxWindow*.SaveGfxWindow*

```
lResult& = SaveGfxWindowEx(lWindowNumber&, _
                           sFileName$)
```

(See Syntax Options)

Parameters

sFileName$

Either **1)** The name of the disk file that should be created, or **2)** the word "CLIPBOARD" in uppercase letters. (Recent versions of PowerBASIC can use the string equate $CLIPBOARD.) An empty string can also be used (see **Return Value** below).

Return Value

If an error occurs, this function will return a Graphics Tools Error Code.

If an empty string ("") is used for *sFileName$*, this function will not create a disk file, it will simply return the current screen "color depth" (1, 4, 8, 16, 24, or 32). If *sFileName$* is not an empty string...

Bitmaps: If the specified disk file is created without errors, the value of *lResult&* will be greater than zero and less than or equal to 32. The number that is returned indicates the "color depth" of the bitmap that was created, either 1, 4, 8, 16, 24, or 32, as specified by the SaveGfxMode function.

JPEG Files: If the specified disk file is created without errors, the value of *lResult&* will be the "quality" of the JPEG image, between JPEG_QUALITY and JPEG_BEST, as specified by the SaveGfxMode function.

Remarks

This function is used to save the entire current graphics window to a standard Windows bitmap file or to the Windows clipboard. (To save just a *portion* of the graphics window, use SaveGfxArea. Most of the information in this section of this document applies to that function as well.)

If you specify the word "CLIPBOARD" (in uppercase letters) for the *sFileName$* parameter, a copy of the current graphics window will be placed into the Windows

clipboard. The image can then be retrieved from the clipboard with the DisplayBitmap, StretchBitmap, and/or CropBitmap function. (Other programs such as Microsoft Paint and Microsoft Word can also retrieve or "Paste" bitmap images from the clipboard.)

If you specify a filename, you should usually specify the *entire* bitmap file name. If you do not specify an extension, `.BMP` will be automatically added to the names of bitmap files, and `.JPG` to JPEG files. You may also choose to include a drive and path spec. If you include any of the following four special codes in your file name, Graphics Tools will insert the appropriate information:

`=w`	The Width of the bitmap, in pixels.
`=h`	The Height of the bitmap, in pixels.
`=c`	The number of "color bits" that the bitmap uses.
`=0c`	Same as `=c`, but always two digits long (like 08 instead of 8).

For example, using...

```
SaveGfxWindow "BITMAP=c"
```

...to save a 24-bit bitmap would produce a disk file called "`BITMAP24.BMP`". *Please note that lower-case letters must be used for these special codes.*

Since the SaveGfxWindow and SaveGfxArea functions create a "disk image" of a screen image, the resulting bitmap file will always be created using the *actual* size of the graphics window, in pixels (not Drawing Units). That means that you can't expect to create a one-inch-square graphics window, and be able to produce a large, high-resolution bitmap file. What you see is what you'll get.

In the default mode, the SaveGfxWindow and SaveGfxArea functions will create Windows bitmap files using a format that *Windows* chooses, based on the monitor's current "color depth" setting. If the computer is currently using 24-bit TrueColor, a bitmap with that format will be created. If your computer is using a different color resolution, a different format bitmap will be created.

You can use the SaveGfxMode function to tell Graphics Tools to save a specific kind of disk file.

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL and you attempt to save a JPEG file, this function will return `ERROR_LIBRARY_NOT_FOUND`.

See Bitmaps and JPEG Files for more information.

Example

```
SaveGfxWindow "C:\MyScreen.BMP"
```

...or...

```
SaveGfxWindow "CLIPBOARD"
```

See Also

SaveGfxArea

ScrollBars

Purpose

This Visual Basic Property determines whether or not the graphics window has scroll bars. (Non-VB programmers should use the `GFX_STYLE_SCROLLBARS_` constants when a graphics window is created.)

Availability

Graphics Tools Pro Only (OCX version only)

Warning

None.

Remarks

This Property can be set to ScrollBarsAuto (the default), ScrollBarsNever, or ScrollBarsAlways. See Graphics Window Styles for more information.

See Also

Using Graphics Tools With Visual Basic

ScrollH and ScrollV

Purpose

These Visual Basic Events are fired when the graphics window is scrolled. (Non-VB programmers see Graphics Window Messages.)

Availability

Graphics Tools Pro Only (OCX version only)

Warning

None.

Parameters

None.

Remarks

These Events are fired when the graphics window is scrolled.

See Also

Using Graphics Tools With Visual Basic

SelectGfxColor

Purpose

Displays a standard Windows "ChooseColor" dialog box.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = SelectGfxColor(lStartColor&, _
                          lStyleOption&)
```

VB method: *GfxWindow**.SelectGfxColor

```
lResult& = SelectGfxColorEx(lWindowNumber&, _
                            lStartColor&, _
                            lStyleOption&)
```

Parameters

lStartColor&

The color that should be selected when the dialog is first displayed. Use GFX_SAME to start with the same color that was selected the last time the dialog was displayed.

lStyleOption&

Use zero (0) for a normal ChooseColor dialog, or one (1) to automatically open the Custom Color portion of the dialog, or two (2) to prevent the Custom Color portion of the dialog from being opened by the user.

Return Value

The value of *lResult&* will be a Windows color value between zero (0) and MAXCOLOR if the user selects the Ok button, or ERROR_USER_CANCEL if the user selects the dialog's Cancel button or closes the dialog without selecting Ok, or another Graphics Tools Error Code if an error is detected.

Remarks

This function can be used to display a standard-format ChooseColor dialog.

If you have used the Graphics Tools CustomColor function to define one or more custom colors, they will be displayed in the custom color portion of the dialog. If the dialog is used to modify the custom color(s), the return value(s) of the CustomColor function will be affected accordingly.

It should be noted that this is a standard Microsoft dialog, and that the custom color portion of the dialog uses a color system that is slightly different from the Graphics Tools HSW color system. The ChooseColor dialog uses both RGB values and a color system called HSL (Hue, Saturation, and Luminance) that is different from HSW. If you want to use the ChooseColor dialog to obtain an HSW color, simply take the function's return value (which is a Windows color value) and use the RGBtoHSW function to extract the H, S, and W values.

The appearance and operation of the ChooseColor dialog are determined by

Windows, and vary somewhat from version to version of Windows. A complete description of its operation is beyond the scope of this document.

Example

```
lResult& = SelectGfxColor(0,0)
```

Sample Program

Selecting Colors

See Also

Windows Colors

SelectGfxFile

Purpose

Displays a standard Windows "Select File" dialog that can be used for the manual selection of various types of graphics files (bitmaps, icons, etc.)

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
sResult$ = SelectGfxFile(sTitle$, _
                         sFileSpec$, _
                         sInitialDir$, _
                         sFilter$, _
                         sDefExtension$, _
                         lFlags)
```

VB method: *GfxWindow**.SelectGfxFile

```
sResult$ = SelectGfxFileEx(lWindowNumber&, _
                           sTitle$, _
                           sFileSpec$, _
                           sInitialDir$, _
                           sFilter$, _
                           sDefExtension$, _
                           lFlags)
```

(See Syntax Options)

Parameters

sTitle$

A string that specifies the title that the dialog box should display in its caption. If an empty string is used for this parameter, the default title is "Open".

sFileSpec$

The initial file specification. The file name (like `MYPIC.BMP`) or file spec (like `*.ICO`) that you specify will appear in the "File Name" field of the dialog box. (It is possible to do so, but you should not normally specify a drive and/or directory with this parameter.) If an empty string is used for this parameter, the File Name field will be blank when the dialog is first displayed.

sInitialDir$

The drive and/or directory that will be displayed in the "Look In" listbox when the dialog box is first displayed. This parameter also affects the initial file-list display. If an empty string is used for this parameter, the dialog box will start in the default directory.

sFilter$

The description(s) and file filter(s), separated with pipe symbols, that the dialog box should display in the "Files Of Type" listbox. See **Remarks** below for more information.

sDefExtension$

The default extension. If the user types a file name without an extension, this string is automatically appended to the file name when the function exits. Also see `OFN_EXTENSIONDIFFERENT` below, for another use of this parameter.

lFlags&

VERY IMPORTANT NOTE: This is an input-output parameter. A bitmasked value that can contain many different options. See **Remarks** below for a complete list.

Return Value

The value of *sResult$* will be the name(s) of the file(s) that are selected by the user. If the user does not select a file, an empty string ("") will be returned. If a Windows error occurs, the string ERROR will be returned. (Your program should check for the string ERROR so that it will not attempt to use a file called ERROR.)

The *lFlags&* parameter, as an input-output parameter, also provides a return value. See **Remarks** below.

Remarks

This function is intended for use with programming languages that do not provide a built-in file selection dialog. If your language (such as Visual Basic) provides such a dialog, you are encouraged to use it instead of this function.

The sFilter$ Parameter

The *sFilter$* parameter can be used to specify one or more *pairs* of strings that control what will be displayed in the "Files Of Type" listbox. Each string pair should represent a "description" and a matching "file spec", separated by the pipe (|) symbol. If you use more that one pair, they should also be separated by pipe symbols. For example, if you use the string...

```
Bitmap Files|*.BMP
```

...the Files Of Type listbox will contain the string "Bitmap Files" and only files that match the filter *.BMP will be displayed. If you use the string...

```
Bitmap Files|*.BMP|Icon Files|*.ICO|Cursor Files|*.CUR
```

...the initial display will be the same as if you had used the shorter string above, but the Files Of Type listbox will allow you to select "Icon Files" or "Cursor Files" and whenever one of those items is selected, the corresponding filter will be used.

Note that the filter spec (in this case *.BMP) itself is not automatically displayed. If you want the listbox to say "Bitmap Files (*.BMP)" you must use a string with duplicate information like this:

```
Bitmap Files (*.BMP)|*.BMP
```

(And if you want the initial filter spec to also be displayed in the File Name field, use the *sFileSpec$* parameter. See below.)

You can also specify multiple filters for a single description by using semicolons. For example, using...

```
Image Files|*.BMP;*.JPG;*.JPEG
```

...would display the files that match *all* of the filters shown.

If you use an empty string for *sFilter$*, files of all types will be shown and the Files Of Type listbox will be empty.

Here is a long string that you can use to allow the selection of most of the common types of image files:

```
Bitmaps|*.BMP|Icons|*.ICO|Cursors|*.CUR|JPEG
Images|*.JPG;*.JPEG|Windows MetaFiles|*.WMF|PNG
Files|*.PNG|TIFF Files|*.TIF|All Files|*.*
```

If you use the same *sFilter$* string when calling the SelectGfxFile function two or more times in a row, the Files Of Type list will be "sticky". For example, if the user selects JPEG Images from the list above, the next time the SelectGfxFile dialog is displayed it will automatically show JPEG Images in the Files Of Type list.

If you want to specify that a specific *sFilter$* item be displayed, add an extra pipe symbol at the end of the string and use an equal sign like this:

```
Bitmap Files|*.BMP|Icon Files|*.ICO|Cursor Files|*.CUR|=3
```

In that example, the third item (Cursor Files) would be automatically selected when the dialog is first displayed. The equal sign and number (and everything that follows) will be removed from the string before it is displayed.

The sFileSpec$ Parameter

It's generally a good idea to use an *sFileSpec$* value that matches the first part of your *sFilter$* parameter. For example, if you use a filter like this...

```
Bitmap Files|*.BMP|Icon Files|*.ICO|Cursor Files|*.CUR
```

...you should use an *sFileSpec$* of `*.BMP`. This will produce an initial file display that matches the initial "Files Of Type" selection.

The lFlags& Parameter

The *lFlags&* value is an input-output parameter.

For input purposes, you can tell the `SelectGfxFile` function how to perform certain operations by using the flag values shown below. You can add any of the flag values together (see **Example** below) to specify multiple options.

To use these `OFN_` values you will need to include the PowerBASIC `COMDLG32.INC` file (or the corresponding file that your language requires) in your program.

Input flag `OFN_ALLOWMULTISELECT`

> Tells the SelectGfxFile function to allow the selection of multiple files. If the user does in fact select more than one file, the *sFileSpec$* parameter will return the path to the current directory, followed by the filenames of the selected files. All of the elements of the *sFileSpec$* string (the directory and all of the file names) will be separated by CHR$(0). Multiple files are selected by holding down a Shift or Ctrl key while clicking on file names.

Input Flag `OFN_CREATEPROMPT`

> This flag has no effect unless the `OFN_FILEMUSTEXIST` flag is also used.

If the user types the name of a file that does not exist, the OFN_CREATEPROMPT option causes the dialog box to prompt the user for permission to create the file. If the user chooses to create the file, the dialog box will close and the *sFileSpec$* parameter will contain the name of the file that was entered by the user. Otherwise, the file-selection dialog box will remain open and allow the user to make another selection. *In any case, the file will not actually be created automatically. Your program must do that.*

Input Flag OFN_FILEMUSTEXIST

If you use this flag, it specifies that the user can only select existing files. If the user types an invalid name and selects the Open button, the SelectGfxFile function will display a warning message and refuse to close. (If this flag is specified, the OFN_PATHMUSTEXIST flag is also used automatically.)

Input Flag OFN_HIDEREADONLY

Hides the Read Only check box that is normally displayed on the dialog box.

Input Flag OFN_NOCHANGEDIR

If the user changes the directory while searching for files, this option restores the current directory to its original value when the Open or Cancel button is selected. It does not, however, keep your program's current directory from being changed *while* files are being selected. This can be important if you are creating a multi-threaded applications, because the current directory of *all threads* will be temporarily changed during the file-selection process. (This is the normal behavior of the Windows select-file dialog. It is not a bug in Graphics Tools.)

Input Flag OFN_NODEREFERENCELINKS

Affects the selection of Windows Shortcut files. It you use this option, and if the user selects a shortcut file, the SelectGfxFile function will return the path and filename of the selected shortcut (.LNK) file. If this option is not used, the function will automatically return the path and filename of the file that is *referenced* by the shortcut

Input Flag OFN_NONETWORKBUTTON

Hides and disables the Network button that is normally displayed on the dialog box.

Input Flag OFN_NOTESTFILECREATE

This description is from the official Microsoft Win32 documentation: "*Specifies that the file is not created before the dialog box is closed. This flag should be specified if the application saves the file on a create-nonmodify network sharepoint. When an application specifies this flag, the library does not check for write protection, a full disk, an open drive door, or network protection. Applications using this flag must perform file operations carefully, because a file cannot be reopened once it is closed.*"

Input Flag `OFN_NOVALIDATE`

Specifies that the SelectGfxFile function should allow invalid characters in the returned filename.

Input Flag `OFN_PATHMUSTEXIST`

Specifies that the user can type only valid (i.e. existing) paths. If this flag is used and the user types an invalid path in the File Name field, the SelectGfxFile function will display a message box.

This flag is used automatically whenever the `OFN_FILEMUSTEXIST` flag is used.

Input Flag `OFN_READONLY`

The use of this flag causes the Read Only check box to be checked when the dialog box is first displayed. Also see Output Flag `OFN_READONLY` below.

The following flags can be *returned* by the SelectGfxFile function. You can test the return value of *lFlags&* for the following values by using the `AND` syntax (see **Example** below).

Output Flag `OFN_EXTENSIONDIFFERENT`

If this flag is set when the SelectGfxFile function returns, it means that the user typed a filename extension that was different from the default extension that you specified with the *sDefExtension$* parameter.

Output Flag `OFN_NOREADONLYRETURN`

If this flag is set when the SelectGfxFile function returns, it means that the selected file does not have the Read Only check box checked, and that it is not in a write-protected directory.

Output Flag `OFN_READONLY`

If this flag is set when the SelectGfxFile function returns, it means that the Read Only check box was checked when the dialog box was closed.

Example

```
'Display a "Select File" dialog that:
'1) has the title "Select a Bitmap File"
'2) starts with nothing in the File Name
'   field,
'3) initially displays the files in the
'   \GfxTools\Images directory,
'4) limits the file display to *.BMP files,
'5) does not have a Read Only button or
'   a Network button,
'6) requires that an existing file be
'   selected by the user,
'7) returns the flag OFN_EXTENSIONDIFFERENT
'   if a file that does not have the
'   default extension BMP is selected, and
'8) automatically resets the default directory
'   if the user changes it while looking for
'   a file.

lFlags& = OFN_FILEMUSTEXIST OR _
          OFN_NOCHANGEDIR OR _
          OFN_HIDEREADONLY OR _
          OFN_NONETWORKBUTTON

sFileSpec$ = ""

sResult$ = SelectGfxFile("Select a Bitmap File:", _
                         sFileSpec$, _
                         "\GfxTools\Images", _
                         "Bitmap Files|*.BMP", _
                         "*.BMP", _
                         lFlags&)

'examine the return values:

If sResult$ = "" Then
   'User selected Cancel
End If

If (lFlags& And OFN_EXTENSIONDIFFERENT) Then
   'User selected a non-BMP file.
   'Note the REQUIRED parentheses, which
   'force a "bitwise" operation.
End If

'display the name of the selected file:
Print sResult$
```

Sample Program

A Simple Image Viewer

See Also

Displaying Images From Files

StretchBitmap

Purpose

Displays a bitmap, and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = StretchBitmap(sBitmap$, _
                         lWidth&, _
                         lHeight&)
```

VB method: *GfxWindow**.StretchBitmap

```
lResult& = StretchBitmapEx(lWindowNumber&, _
                           sBitmap$, _
                           lWidth&, _
                           lHeight&)
```

(See Syntax Options)

Parameters

sBitmap$

Use either...

1) the name of the standard Windows bitmap file that you wish to display, or

2) the word "CLIPBOARD" in uppercase letters to load a file that was previously placed in the Windows clipboard by the SaveGfxArea or SaveGfxWindow function or by an external program or by the Print Screen key, or

3) an empty string, for the default Graphics Tools bitmap, or

4) a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools bitmap or

5) a string with a numeric value between 32,500 and 32,767 for a standard Windows bitmap (see Appendix A), or

6) a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a bitmap that is embedded in a resource file that has been embedded in an EXE or DLL file.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for *both* of these parameters (or simply use the DisplayBitmap function) if you want the bitmap to be displayed in its native size. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For

example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the bitmap is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display a bitmap inside a rectangle of virtually any size. The top-left corner of the rectangle will be located at the LPR. (If you want to display just a portion of a bitmap, see the CropBitmap function.)

Graphics Tools will stretch or compress the image, as necessary, to allow it to be displayed in the rectangle that you specify.

If you display a bitmap in a rectangle that is smaller than its original size, details will be lost. If you display a bitmap in a rectangle that is *significantly* smaller than its original size, the image will usually become distorted.

If you display a bitmap in a rectangle that is larger than its original size, small details will be enlarged but (of course) additional details will not become visible. If you display a bitmap in a rectangle that is significantly larger than its original size, individual pixels will be enlarged to the point where they become visible rectangles of color.

This function can optionally display bitmaps that are "flipped" vertically and/or horizontally. For more information, see GfxOption GFX_IMAGE_FLIP_V and GFX_IMAGE_FLIP_H.

This function can also optionally load bitmaps using many different options which control how the image is added to the graphics window. The default mode is called SRCCOPY and it causes the bitmap image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the bitmap are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds a bitmap to the graphics screen in a way that allows it to be "un-drawn" simply by using the StretchBitmap function a second time, with exactly the same parameters. See GfxOption GFX_IMAGE_DRAW_MODE for more information.

This function can also optionally load bitmaps using three different "special effects" modes, including monochrome (black and white) loading. See GfxOption GFX_IMAGE_LOAD_MODE.

Example

```
'Stretch the default Graphics Tools bitmap to
'fit a wide, short rectangle.
StretchBitmap "", 200, 10
```

See Also

DisplayCursor, StretchCursor, DisplayBitmap, CropBitmap

StretchCursor

Purpose

Displays a cursor, and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = StretchCursor(sCursor$, _
                         lWidth&, _
                         lHeight&)
```

VB method: *GfxWindow*.*StretchCursor

```
lResult& = StretchCursorEx(lWindowNumber&, _
                           sCursor$, _
                           lWidth&, _
                           lHeight&)
```

(See Syntax Options)

Parameters

sCursor$

Use either **1)** the name of a cursor file, or **2)** an empty string, for the default Graphics Tools cursor, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools cursor or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows cursor (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of a cursor that has been embedded in an EXE or DLL file.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for *both* of these parameters (or simply use the DisplayCursor function) if you want the cursor to be displayed in its native size. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the cursor is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display a cursor inside a rectangle of virtually any size, *in a fixed location*, as part of the graphics window. (If you want to change the movable cursor that is displayed by Windows when the mouse cursor is located over the graphics window, see GfxCursor. If you want to display an animated cursor, see the AnimateCursor function.)

The top-left corner of the rectangle will be located at the LPR.

Windows will stretch or compress the image, as necessary, to allow it to be displayed in the rectangle that you specify.

If you display a cursor in a rectangle that is smaller than its original size, details will be lost. If you display a cursor in a rectangle that is *significantly* smaller than its original size, the image will usually become distorted.

If you display a cursor in a rectangle that is larger than its original size, small details will be enlarged but (of course) additional details will not become visible. If you display a cursor in a rectangle that is significantly larger than its original size, individual pixels will be enlarged to the point where they become visible rectangles of color.

This function can optionally load cursors using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE`.

It is common for programmers to want to draw cursors such as the Windows "hourglass" (also known as `OCR_WAIT`), and then to un-draw that cursor later. It is usually much easier to use a bitmap than to use a cursor when un-drawing is required. Please refer to the DisplayBitmap function, and the GfxOption `GFX_IMAGE_DRAW_MODE` function.

Example

```
'Stretch the default Graphics Tools cursor to
'fit a tall, narrow rectangle...
StretchCursor  "", 10, 200
```

See Also

DisplayCursor, AnimateCursor, AnimateIcon, DisplayIcon

StretchIcon

Purpose

Displays an icon, and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = StretchIcon(sIcon$, _
                       lWidth&, _
                       lHeight&)
```

VB method: *GfxWindow**.StretchICon

```
lResult& = StretchIconEx(lWindowNumber&, _
                         sIcon$, _
                         lWidth&, _
                         lHeight&)
```

(See Syntax Options)

Parameters

sIcon$

Use either **1)** the name of an icon file, or **2)** an empty string, for the default Graphics Tools icon, or **3)** a string with a numeric value between one (1) and ninety-nine (99) for a standard Graphics Tools icon or **4)** a string with a numeric value between 32,500 and 32,767 for a standard Windows icon (see Appendix A), or **5)** a string with a numeric value that corresponds to the resource ID number (between 100 and 32,499) of an icon that has been embedded in an EXE or DLL file.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for *both* of these parameters (or simply use the DisplayIcon function) if you want the icon to be displayed in its native size. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the icon is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display an icon inside a rectangle of virtually any size. The top-left corner of the rectangle will be located at the LPR. (If you want to display an animated icon, see the AnimateIcon function.)

Windows will stretch or compress the image, as necessary, to allow it to be displayed

in the rectangle that you specify.

If you display an icon in a rectangle that is smaller than its original size, details will be lost. If you display an icon in a rectangle that is *significantly* smaller than its original size, the image will usually become distorted.

If you display an icon in a rectangle that is larger than its original size, small details will be enlarged but (of course) additional details will not become visible. If you display an icon in a rectangle that is significantly larger than its original size, individual pixels will be enlarged to the point where they become visible rectangles of color.

This function can optionally load icons using three different "special effects" modes. See GfxOption `GFX_IMAGE_LOAD_MODE`.

Example

```
'Stretch the default Graphics Tools icon to
'fit a tall, narrow rectangle...
StretchIcon "", 10, 200
```

See Also

DisplayCursor, AnimateCursor, AnimateIcon, DisplayIcon

StretchImage

Purpose

Displays an image from a disk file, and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Standard and Pro

Warning

This function's ability to display various image formats is dependent on the version of Windows that is installed on the runtime system, and other factors. See Displaying Images From Files for more information. Also see StretchBitmap, StretchIcon, StretchCursor, and StretchJpeg for functions that are *not* system-dependent.

Syntax

```
lResult& = StretchImage(sFileName$, _
                        lWidth&, _
                        lHeight&)
```

VB method: *GfxWindow**.StretchImage

```
lResult& = StretchImageEx(lWindowNumber&, _
                          sFileName$, _
                          lWidth&, _
                          lHeight&)
```

(See Syntax Options)

Parameters

lWidth& and *lHeight&*

The size of the rectangle that the image should fill, in Drawing Units. Use GFX_AUTO for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and GFX_AUTO for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested image is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function displays an image from a disk file, and stretches or compresses it into the size that you specify. Also see DisplayImage.

This function's ability to display various image formats is dependent on the version of Windows that is installed on the runtime system, and other factors. It is usually (but not always) capable of displaying WMF, JPEG, and JPG, among others. See Displaying Images From Files for more information about this.

Graphics Tools Pro licensees can use the DisplayJpeg, StretchJpeg, and CropJpeg functions to display JPEG files regardless of system configuration.

PLEASE NOTE: This function cannot be used to display GIF files even if the runtime

system is configured to do so. The GIF file format is patented by UniSys Corporation, and UniSys does not allow GIF files to be used by any computer program without a license from UniSys. If GIF functionality was included in Graphics Tools, Perfect Sync would be required to obtain a license, and *you* would be required to obtain a license in order to use that functionality, even to simply *display* GIF files. Because of the license costs and legal complexities involved, Graphics Tools locks out the use of GIF files.

Example

```
StretchImage "MyImage.WMF", 100, 200
```

See Also

Displaying Images From Files

StretchJpeg

Purpose

Displays a JPEG image, and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Pro Only

Warning

If the appropriate JPEG Library is not located in the same directory as the Graphics Tools DLL, this function will return `ERROR_LIBRARY_NOT_FOUND`.

Syntax

```
lResult& = StretchJpeg(sFileName$, _
                       lWidth&, _
                       lHeight&)
```

VB method: *`GfxWindow*`*`.StretchJpeg`

```
lResult& = StretchJpegEx(lWindowNumber&, _
                         sFileName$, _
                         lWidth&, _
                         lHeight&)
```

(See Syntax Options)

Parameters

sFileName$

The name of the JPEG file to be displayed.

lWidth& and lHeight&

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for both of these parameters (or simply use the DisplayJpeg function) if you want the cursor to be displayed in its native size. Use `GFX_AUTO` for *one* of these parameters if you want Graphics Tools to calculate the value to maintain the original aspect ratio of the image. For example, using 500 for *lWidth&* and `GFX_AUTO` for *lHeight&* would tell Graphics Tools to calculate the *lHeight&* for the image, based on a width of 500 Drawing Units, so that the image would maintain its aspect ratio.

Return Value

The value of *lResult&* will be `SUCCESS` (zero) if the image is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used to display a JPEG image inside a rectangle of virtually any size. The top-left corner of the rectangle will be located at the LPR.

Windows will stretch or compress the image, as necessary, to allow it to be displayed in the rectangle that you specify.

If you display an image in a rectangle that is smaller than its original size, details will be lost. If you display an image in a rectangle that is *significantly* smaller than its original size, the image will usually become distorted.

If you display an image in a rectangle that is larger than its original size, small details will be enlarged but (of course) additional details will not become visible. If you display an image in a rectangle that is significantly larger than its original size, individual pixels will be enlarged to the point where they become visible rectangles of color.

This function is available only in Graphics Tools Pro.

Example

```
StretchJpeg "MyImage.JPG", 100, 200
```

See Also

Bitmaps and JPEG Files

StretchWindow

Purpose

Displays the contents of a graphics window (i.e. copies an image from one graphics window to another) and optionally stretches or compresses the image to fit the specified rectangle.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = StretchWindow(lSourceWindow&, _
                         lWidth&, _
                         lHeight&)
```

VB method: `GfxWindow*.StretchWindow`

```
lResult& = StretchWindowEx(lWindowNumber&, _
                           lSourceWindow&, _
                           lWidth&, _
                           lHeight&)
```

(See Syntax Options)

Parameters

lSourceWindow&

The graphics window number of the source image, i.e. the graphics window where the image should be copied *from*.

lWidth& and *lHeight&*

The width and height of the image that is to be displayed, in Drawing Units. Use zero (0) for both of these parameters (or simply use the DisplayWindow function) if you want the image to be displayed in its original size.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the image is displayed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See Using Multiple Graphics Windows for background information.

This function can be used to copy the contents of one graphics window into another, inside a rectangle of any size. The top-left corner of the rectangle will be located at the LPR. (If you want to display just a *portion* of another graphics window, see the CropWindow function.)

Graphics Tools will stretch or compress the image, as necessary, to allow it to be displayed in the rectangle that you specify.

If you display an image in a rectangle that is smaller than its original size, details will be lost. If you display an image in a rectangle that is *significantly* smaller than its original size, the image will usually become distorted.

If you display an image in a rectangle that is larger than its original size, small details will be enlarged but (of course) additional details will not become visible. If you display an image in a rectangle that is significantly larger than its original size, individual pixels will be enlarged to the point where they become visible rectangles of color.

This function can optionally display images that are "flipped" vertically and/or horizontally. For more information, see GfxOption `GFX_IMAGE_FLIP_V` and `GFX_IMAGE_FLIP_H`.

This function can also optionally load images using many different options which control how the image is added to the graphics window. The default mode is called `SRCCOPY` and it causes the image to replace the current contents of the specified rectangle. It is also possible to perform "additive" operations where the existing screen and the new image are mixed in different ways. Perhaps the most useful of these techniques is XOR drawing, which adds an image to the graphics screen in a way that allows it to be "un-drawn" simply by using the StretchWindow function a second time, with exactly the same parameters. See GfxOption `GFX_IMAGE_DRAW_MODE` for more information.

Example

```
'Copy the contents of graphics window #2 into
'a rectangle that is 200x100 Drawing Units.
StretchWindow 2, 200, 10
```

See Also

DisplayWindow, CropWindow

SystemColor

Purpose

Returns the current Windows color values for various screen elements.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = SystemColor(lWhich&)
```

Parameters

lWhich&

See **Remarks** below.

Return Value

If a valid *lWhich&* value is used, the value of *lResult&* will be the specified color. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The following standard *lWhich&* values are recognized by Windows.

```
COLOR_SCROLLBAR
COLOR_BACKGROUND
COLOR_ACTIVECAPTION
COLOR_INACTIVECAPTION
COLOR_MENU
COLOR_MSGBOX
COLOR_WINDOW
COLOR_WINDOWFRAME
COLOR_MENUTEXT
COLOR_MSGBOXTEXT
COLOR_WINDOWTEXT
COLOR_CAPTIONTEXT
COLOR_ACTIVEBORDER
COLOR_INACTIVEBORDER
COLOR_APPWORKSPACE
COLOR_HIGHLIGHT
COLOR_HIGHLIGHTTEXT
COLOR_BTNFACE
COLOR_BTNSHADOW
COLOR_GRAYTEXT
COLOR_BTNTEXT
COLOR_INACTIVECAPTIONTEXT
COLOR_BTNHIGHLIGHT
COLOR_3DDKSHADOW
COLOR_3DLIGHT
COLOR_INFOTEXT
COLOR_INFOBK
```

When Windows is first installed, standard colors are used for all of the above. Those colors can, however, vary from version to version of Windows.

The Windows Control Panel's "Display" applet can be used to customize these colors on a system-by-system basis. Other programs such as "desktop customizers" can also affect these colors.

Example

```
'Find out what color Windows is using for pulldown
'and popup menus (usually some shade of gray).
lResult& = SystemColor(COLOR_MENU)
```

See Also

Windows Colors

TempDraw

Purpose

Controls the Graphics Tools "temporary drawing" mode.

Availability

Graphics Tools Standard and Pro

Warning

The TempDraw mode cannot be activated if the graphics window is currently scrolled, i.e. if the graphics window has scroll bars and the top-left corner the graphics window is not currently visible.

Syntax

```
TempDraw lAction&
```

VB method: *GfxWindow**.TempDraw

```
TempDrawEx lWindowNumber&, _
           lAction&
```

Parameters

lAction&

See **Remarks** below for a list of valid values.

Return Value

If you disable the TempDraw mode and then attempt to use it, or if the graphics window is currently scrolled, this function will return ERROR_CANT_BE_DONE. Otherwise, the return value of this function will always be SUCCESS (zero), so it is safe to ignore it.

Remarks

Normally, Graphics Tools uses drawing techniques that are automatically maintained by Windows. For example, if your graphics window is covered up by another application and then revealed, Windows will automatically re-paint the contents of the graphics window. See Refreshing The Display for more information about this process.

If you use the TempDraw mode, Graphics Tools will use drawing techniques that are designed to be temporary. This can be useful when you need to draw things like temporary lines, and then erase them a short time later. (For a completely different "un-drawing" method, also see GfxDrawMode GFX_XOR_DRAW. Also see DrawFocus if you need to draw a temporary rectangle around an area of the graphics window.)

After the TempDraw mode has been activated, anything new that you draw will look perfectly normal on the screen, but it will disappear the next time the graphics window is refreshed. The graphics window (or some portion of it) will be refreshed by Windows if it is **1)** covered up by another window and then uncovered, or **2)** minimized and then restored, or **3)** moved off the screen and then back on, or **4)** resized, or **5)** scrolled or **6)** if the GfxRefresh and/or GfxUpdate functions are used, or **7)** if *any* drawing operation is performed in the permanent (i.e. non-TempDraw) mode. Other things can also cause portions of the graphics window to be automatically refreshed.

Use one of the following values for the *lAction&* parameter...

TEMPDRAW_BEGIN

Activates the TempDraw mode. Note that the TempDraw mode cannot be activated if the graphics window is currently scrolled, i.e. if the graphics window has scroll bars and the top-left corner the graphics window is not currently visible.

TEMPDRAW_END

Turns off the TempDraw mode and returns your program to the normal "permanent" drawing mode.

TEMPDRAW_CLEAR

Refreshes the permanent parts of your graphics window, thereby clearing or "erasing" the temporary parts. (Also see the TempErase function.)

TEMPDRAW_DISABLE

This option can be used to disable the TempDraw mode. If your program does not need the TempDraw mode, disabling it can result in a minor speed improvement when creating Pens, Brushes, and fonts, and when changing things like the font color, background color, and background mode. *Once the TempDraw mode has been disabled, it cannot be used unless the graphics window is re-created with the* InitGfx *function.*

Some Important Points about Temporary Drawing

If you use the SaveGfxWindow or SaveGfxArea function, the temporary parts of your graphics window will *not* be saved in the resulting bitmap file.

The same is true for printing. If you print an image with GfxPrintWindow or GfxPrintArea, only the permanent parts of the image will be printed.

The TempDraw mode and "holes" don't mix well. If you use the DrawHole function (or the Console Tools GfxTextHole function), temporary drawing operations *will* cover up the holes that you have specified. And if you cover up a hole with a temporary drawing element, the hole cannot be uncovered with the GfxRefresh or GfxUpdate function.

Example

```
'DRAW A "PERMANENT" CIRCLE...
DrawFrom 200,200
DrawCircle  100

'DRAW A TEMPORARY LINE...
TempDraw  TEMPDRAW_BEGIN
DrawTo 0,0
TempDraw TEMPDRAW_END

'WAIT 5 SECONDS...
Sleep 5000
```

```
'AND ERASE THE LINE...
TempDraw  TEMPDRAW_CLEAR
```

Sample Program

Using The TempDraw Mode

See Also

The Graphics Window

TempErase

Purpose

If the TempDraw function has been used, TempErase can be used to erase a rectangular section of the temporary image.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = TempErase(lLeft&, _
                     lTop&, _
                     lRight&, _
                     lBottom&)
```

VB method: *GfxWindow**.TempErase

```
lResult& = TempEraseEx(lWindowNumber&, _
                       lLeft&, _
                       lTop&, _
                       lRight&, _
                       lBottom&)
```

(See Syntax Options)

Parameters

lLeft&, lTop&, lRight&, and *lBottom&*

These parameters are used to specify the location and size of the area to be erased, in Drawing Units.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the requested operation is completed without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

The term "temp erase" is not meant to imply "erase something temporarily". It means "erase something that was intended to be temporary".

See the TempDraw function for information about temporary drawing.

Example

```
'DRAW A "PERMANENT" CIRCLE...
DrawFrom 200,200
DrawCircle  100

'DRAW A TEMPORARY LINE...
TempDraw  TEMPDRAW_BEGIN
DrawTo 0,0
TempDraw TEMPDRAW_END

'WAIT 5 SECONDS...
Sleep 5000
```

```
'AND ERASE PART OF THE LINE...
TempErase 100,100,150,150
```

Sample Program

Using The TempDraw Mode

See Also

TempDraw

UnderColor

Purpose

This Visual Basic Property determines the color that is displayed *under* , and in some cases *around*, the graphics window. See **Remarks** below for more information.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

When the OCX version of Graphics Tools is used, a graphics window is actually *two* windows: an ActiveX "container", and the actual graphics window where drawing operations are performed. If you use the GfxWindow `GFX_HIDE` function to make the drawing window invisible, the container will become visible. The UnderColor Property is used to specify the color of the container.

The default UnderColor is gray, to match the default VB Form background color, so the container is not usually visible. If you change the UnderColor property, a one-pixel-wide border *may* become visible on one or more edges of the graphics window. The size and location of the one-pixel edges are determined by the size of the container, the current screen resolution, and whether or not the graphics window has an odd or even number of pixels.

You can increase the size of the UnderColor border by using the UnderShow Property. This will create a colored border around the graphics window.

See Also

Using Graphics Tools With Visual Basic

UnderShow

Purpose

This Visual Basic Property determines how much of the UnderColor is visible around the perimeter of the graphics window.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Remarks

See UnderColor for background information.

This Visual Basic Property determines how much of the UnderColor is visible around the perimeter of the graphics window, in pixels.

Even when this Property is set to zero (the default) Visual Basic may display a one-pixel-wide border of the UnderColor on one or more edges of the graphics window. For this reason, the default UnderColor is gray, to match the default form background color.

See Also

Using Graphics Tools With Visual Basic

UseGfxWindow

Purpose

Tells Graphics Tools which graphics window to use for drawing operations that are not performed with Ex functions.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = UseGfxWindow(lWindowNumber&)
```

VB programs do not use this function.

(See Syntax Options)

Parameters

lWindowNumber&

The window number that will be used for subsequent drawing operations.

Return Value

The value of *lResult&* will be SUCCESS (zero) if *lWindowNumber&* is a valid window number. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

See Using Multiple Graphics Windows for a discussion of this function.

Example

```
UseGfxWindow 3
```

See Also

The Graphics Window

UseGradient

Purpose

Tells Graphics Tools to use a specific gradient when drawing gradient-filled figures. (Can also retrieve the current gradient.)

Availability

Graphics Tools Pro Only

Warning

None.

Syntax

```
lResult& = UseGradient(tGradient)

lResult& = UseGradientEx(lWindowNumber&, _
                         tGradient)
```

(VB programs must always use the Ex function.)

(See Syntax Options)

Parameters

tGradient

A `GT_HSW_GRADIENT` or `GT_RGB_GRADIENT` structure that contains the instructions for drawing a gradient.

Return Value

The value of *lResult&* will be SUCCESS (zero) if the specified gradient is prepared for use without errors. Otherwise, a Graphics Tools Error Code will be returned.

Remarks

This function can be used when you want to 1) create a `GT_HSW_GRADIENT` or `GT_RGB_GRADIENT` structure (User Defined Type), 2) set the values of one or more elements of the structure, and then 3) begin using the structure to draw gradients. This function performs step 3.

See Designing Your Own Gradients for more information about this function.

If you use a `GT_HSW_GRADIENT` or `GT_RGB_GRADIENT` structure that has the `lGradientType` element set to zero (0) or `GFX_QUERY`, then instead of *using* that gradient Graphics Tools will place a copy of the current gradient into the structure. This allows you to retrieve and examine the gradient that Graphics Tools is currently using.

Example

```
Dim tGradient As GT_HSW_GRADIENT

InitGradientHSW tGradient

tGradient.Hue.Rate = 1000

UseGradient tGradient
```

See Also: Gradients

UserDraw

Purpose

Tells Graphics Tools that a Windows API function has been used by your program, and that the value of the LPR is no longer valid.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
UserDraw
```

VB method: *`GfxWindow*`*`.UserDraw`

```
UserDrawEx lWindowNumber&
```

Parameters

None.

Return Value

Unless the specified graphics window does not exist, the return value of this function will always be SUCCESS (zero) so it is safe to ignore it.

Remarks

This function simply sets X and Y components of the LPR to negative one (`-1`), as a signal that the values are no longer valid. We recommend that you use UserDraw just *before* using a Windows API function to perform a drawing operation, as an indication to Graphics Tools that it no longer has control over the LPR.

Example

```
UserDraw
'Perform your API-based drawing operations here...


'...then reset the Graphics Tools LPR tracking system.
GfxLPR 0,0
```

See Also

GfxLPR

WedgeOrder

Purpose

Calculates a "rank" number for a wedge, based on its angles. This value can be used to determine the optimum drawing order for two or more wedges.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = WedgeOrder(dpStartAngle#, _
                      dpEndAngle#)
```

Parameters

dpStartAngle# and *dpEndAngle#*

The start- and end-angles of the Wedge for which a rank is required.

Return Value

This function will always return a value between zero (0) and one thousand (1000).

Remarks

When you are drawing two or more "related" wedges -- such as when you are drawing a wedge chart (a 3D pie chart) -- the *order* in which the wedges are drawn is very important. The 3D effect of the image will be ruined if a wedge is drawn on top of another wedge that is supposed to be "in front". For a realistic 3D effect, wedges must be drawn from "front" to "back".

The WedgeOrder function returns a value between 0 and 1000 than can be used to determine the correct drawing order. While this does not work under all circumstances (see below) it works very well for *most* wedge charts.

Under most circumstances, you should draw wedges according to their WedgeOrder value. For example, you should draw the wedge that has a WedgeOrder value of zero, followed by one, followed by two, and so on.

In reality, unless it has a *huge* number of wedges, your wedge chart will not actually have a wedge with a rank of zero, one, or two. The first wedge might have a rank of 50 or 100, or a much larger number. The important thing is not the actual number, but the number's relationship to the WedgeOrder values of the *other* wedges that need to be drawn.

The following pseudo-code demonstrates one way to use the WedgeOrder function. It assumes that your program maintains an array of wedge angle values in the following array...

```
Dim dpAngle#(9,1) As Double
```

The array has 10 elements (numbered 0-9) for 10 different wedges, and the second dimension of the array is used to store the start- and end-angles for each wedge in the 0 and 1 elements. You might place values in the array like this:

```
'Wedge 0 starts at 10 degrees and ends at 18 degrees
dpAngle#(0,0) = 10
dpAngle#(0,1) = 18

'Wedge 1 starts at 18 degrees and ends at 47 degrees
dpAngle#(1,0) = 18
dpAngle#(1,1) = 47
```

...and so on for the other eight wedges. Then you could use this code to draw the wedges in the correct order.

```
For lWedgeOrder& = 0 To 1000
  For lWedge& = 0 To 9
    lResult& = WedgeOrder(dpAngle#(lWedge,0),
dpAngle#(lWedge&,1)
    If lResult& = lWedgeOrder& Then
      'Draw Wedge number lWedge& here
    End If
  Next
Next
```

As you can see, that code "scans" the possible values in order from 0 to 1000, looking for wedges that match the values. When it finds a match it draws the corresponding wedge, then it checks the next value, until it has checked all 1001 of the possible values.

However, that code would be relatively slow. It is intended to show the theory behind the WedgeOrder function, not to recommend a specific method. Not only does that code scan for 1001 values when a typical wedge chart probably has only 10-20 wedges, it re-calculates the WedgeOrder value for each wedge 1001 times. A somewhat faster strategy would be to calculate the WedgeOrder value for each wedge and store those values in an array, and then scan the array looking for the smallest value, then the next largest, and so on.

The exact technique you use will depend on how your program stores the angle values for the wedges, the number of wedges, and many other factors.

When Using WedgeOrder Won't Work

If a single wedge in a wedge chart is larger than 180 degrees (50% of the chart) it can be difficult (but not impossible) to draw a realistic 3D chart on a 2D computer screen.

For example, imagine a wedge chart that has two wedges. One is very large, starting at 4 o'clock and going all the way around, clockwise, to 2 o'clock. The other wedge fills the small space between 2 and 4 o'clock. If you try to draw that chart you will see the problem. Part of the small wedge needs to be drawn on top of the large wedge -- the part near 2 o'clock -- but part of the large wedge needs to be drawn on top of the small wedge -- the part near 4 o'clock. It isn't possible to draw either wedge first, and end up with a realistic image.

There are several possible solutions to this problem.

1) Ignore it. If you examine the data that you are graphing you may find that no one wedge will ever be larger than 50% of the total. For example, a chart that shows the

gross sales of 10 competing sales regions would be unlikely to be dominated by one wedge.

2) Limit your charts. Make it so that no one wedge *can* be larger than 180 degrees. If that isn't practical you may be able to draw large wedges in smaller pieces of the same color, to "build up" a larger wedge.

3) Rotate the chart. Continuing the 2-4 o'clock example above, rotate the wedges so that the small wedge goes from 5 o'clock to 7 o'clock. Then you can draw the large wedge first, and when you draw the smaller wedge it will cover up the first wedge properly. But because of the way the eye interprets 2D images, this often results in an unrealistic 3D effect. You will usually get better results if you rotate the large wedge so that one of its angles is at 6 o'clock or 12 o'clock.

4) Use a Clip Area. The Pro version's GfxClipArea function can be used to draw *part* of the large wedge, then the small wedge, then the rest of the large wedge. While this usually produces excellent results it is somewhat more difficult to code and it relies on the use of a feature that is not present in the Standard version of Graphics Tools.

Example

See **Remarks** above.

See Also

DrawWedge

WhitenessValue

Purpose

Returns the amount of white that a Windows Color contains.

Availability

Graphics Tools Standard and Pro

Warning

None.

Syntax

```
lResult& = WhitenessValue(lColor&)
```

Parameters

lColor&

A Windows Color between zero (0) and MAXCOLOR.

Return Value

This function will always return a value between zero (0) and 255 which indicates how much white the specified color contains.

Remarks

The HueValue, SaturationValue, and WhitenessValue functions can be used to determine the Hue, Saturation, and Whiteness of a Windows Color.

Example

```
If WhitenessValue(lColor&) = 0 Then
      'The color contains no white.
End If
```

See Also

HSW Colors

WorldAspect

Purpose

This Visual Basic Property returns the current value of the GfxSquareness function.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Return Value

This Property will display (in the VB Properties Window) a fractional value less than one (such as 0.9957) if the graphics window is taller than it is wide, or greater than one (such as 1.0234) if the window is wider than it is tall. *This value does not refer to the "physical" shape of the drawing window, it refers to the "squareness" of the drawing World.*

Remarks

This property is provided as a convenient way for you to monitor the graphics window's squareness value while you are changing the various World properties such as WorldLeft and WorldRight. This property is called WorldAspect instead of Squareness so that it will appear immediately above the other World parameters in the alphabetized Properties Window.

After you have made the graphics window the shape that you need (tall, wide, etc.) you should change the WorldLeft, WorldRight, WorldTop, and WorldBottom properties so that the WorldAspect property is as close as possible to 1.0000. This will create a Drawing World in which circles appear round and squares appear square.

See Also

Drawing Units

WorldBottom, WorldLeft, WorldRight, and WorldTop

Purpose

These Visual Basic Properties are used to set and return the values that specify the coordinates of the edges of the Drawing World for a graphics Window.

Availability

Graphics Tools Standard and Pro (OCX version)

Warning

None.

Return Value

These properties return values that correspond to the parameters of the GfxWorld function.

Remarks

These properties are used to specify the values that define the coordinates of the Left, Right, Top, and Bottom edges of the drawing window.

For more information, see Using Different "Worlds".

See Also

Drawing Units

Appendix A: Windows Bitmaps, Icons, and Cursors

The range of numbers from 32,500 to 32,768 is reserved by Microsoft for Windows bitmaps, icons, and cursors. Not all of those numbers are used, because Windows has not yet defined 268 different standard images.

Some of the images listed may not be present, and additional images may be present, depending on the version of Windows that you are using.

Many of these values are defined in the PowerBASIC `WIN32API.INC` file and the Visual Basic `WIN32API.TXT` file.

Icons

Decimal	Hex	Mnemonic
32512	7F00	IDI_APPLICATION
32513	7F01	IDI_HAND
32513	7F01	IDI_ERROR
32514	7F02	IDI_QUESTION
32515	7F03	IDI_EXCLAMATION
32515	7F04	IDI_WARNING
32516	7F05	IDI_ASTERISK
32516	7F06	IDI_INFORMATION
32517	7F07	IDI_WINLOGO

Cursors

Decimal	Hex	Mnemonic
32512	7F00	IDC_ARROW
32513	7F01	IDC_IBEAM
32514	7F02	IDC_WAIT
32515	7F03	IDC_CROSS
32516	7F04	IDC_UPARROW
32631	7F77	(not documented: "Pen")
32640	7F80	IDC_SIZE
32641	7F81	IDC_ICON
32642	7F82	IDC_SIZENWSE
32643	7F83	IDC_SIZENESW
32644	7F84	IDC_SIZEWE
32645	7F85	IDC_SIZENS
32646	7F86	IDC_SIZEALL
32648	7F88	IDC_NO
32649	7F89	IDC_HAND
32650	7F8A	IDC_APPSTARTING
32651	7F8B	IDC_HELP

Bitmaps

Decimal	Hex	Mnemonic
32731	7FDB	(not documented: Help "?")
32732	7FDC	(not documented: TrueType "TT")
32734	7FDE	OBM_LFARROWI
32735	7FDF	OBM_RGARROWI
32736	7FE0	OBM_DNARROWI
32737	7FE1	OBM_UPARROWI
32738	7FE2	OBM_COMBO
32739	7FE3	OBM_MNARROW
32740	7FE4	OBM_LFARROWD
32741	7FE5	OBM_RGARROWD
32742	7FE6	OBM_DNARROWD
32743	7FE7	OBM_UPARROWD
32744	7FE8	OBM_RESTORED
32745	7FE9	OBM_ZOOMD
32746	7FEA	OBM_REDUCED
32747	7FEB	OBM_RESTORE
32748	7FEC	OBM_ZOOM
32749	7FED	OBM_REDUCE
32750	7FEE	OBM_LFARROW
32751	7FEF	OBM_RGARROW
32752	7FF0	OBM_DNARROW
32753	7FF1	OBM_UPARROW
32754	7FF2	OBM_CLOSE
32755	7FF3	OBM_OLD_RESTORE
32756	7FF4	OBM_OLD_ZOOM
32757	7FF5	OBM_OLD_REDUCE
32758	7FF6	OBM_BTNCORNERS
32759	7FF7	OBM_CHECKBOXES
32760	7FF8	OBM_CHECK
32761	7FF9	OBM_BTSIZE
32762	7FFA	OBM_OLD_LFARROW
32763	7FFB	OBM_OLD_RGARROW
32764	7FFC	OBM_OLD_DNARROW
32765	7FFD	OBM_OLD_UPARROW
32766	7FFE	OBM_SIZE
32767	7FFF	OBM_OLD_CLOSE

Appendix B: Graphics Tools Bitmaps, Icons, and Cursors

The range of numbers from zero (0) to 99 is reserved for Graphics Tools bitmaps, icons, and cursors. Not all of those numbers are used, because Graphics Tools has not yet defined that many standard images.

Most of these images are not really intended to be used in finished programs. They are provided as a convenient way to test various parts of your programs.

All of these bitmaps, icons, and cursors are embedded in the Graphics Tools Runtime Files, so they can be used at any time. The resources are also provided as `.BMP`, `.CUR`, and `.ICO` files so you can use them for other purposes.

Bitmaps

`0` Bitmap Zero (`LOCATOR.BMP`) is the default Graphics Tools bitmap. It is a 32x32 bitmap that consists of a white square outlined by a black rectangle, with two diagonal black lines connecting the opposite corners and a black dot in the center of the square. A small green arrow at the top-left of the bitmap points to the LPR when it is displayed.

`1` Bitmap One (`C_BARS_1.BMP`) is a 28x21 "test pattern" bitmap that resembles standard television "full screen color bars".

`2` Bitmap Two (`C_BARS_2.BMP`) is another 28x21 "color bar" bitmap. it contains the six primary colors (Red, Green, Blue, Yellow, Cyan, and Magenta) in two different brightnesses, plus white and black bars.

`3` Bitmap Three (`C_BARS_3.BMP`) is very similar to Bitmap One, except that it contains extra black and white areas to make it resemble the standard television "split field color bars" test pattern.

Icons

`0` Icon Zero, otherwise known as `IconAll.ICO`, is the default Graphics Tools icon. Like many Windows icons, it contains more than one image. All of the images are in the "16-color" format, so the colors will be consistent on virtually all systems. The 16x16-pixel icon is the number "16", with white numbers on a red background. The 32x32-pixel image is the number "32", with black numbers on a green background. The 48x48-pixel image is the number "48" with white letters on a blue background. This "triple" icon is intended to demonstrate how Windows unilaterally chooses which icon will be displayed when two or more different-sized images are contained in an icon file. If you use the Windows Explorer program to look at the directory that contains the Graphics Tools `.ICO` files, the image that will be displayed for `IconAll.ICO` will depend on your Explorer settings. If you use "View/Small Icons" you will probably see the "16" image. If you use "View/Large Icons" you will probably see the "32" image. Under certain circumstances, particularly when using Windows 98, ME, 2000, and XP, you may see the "48" image. *Windows automatically selects the image that it wants to use for a given purpose*. For example, if you were to use `IconAll.ICO` as the icon that is embedded in an executable program, the "16" image would be used in the program's task bar button and either the "32" or "48" image would be used for the program's desktop icon.

`1, 2, and 3` Icons 1, 2, and 3 (files `Icon16.ICO`, `Icon32.ICO`, and `Icon48.ICO`) are single-image icons that are duplicates of the three images in the `IconAll.ICO` file.

Cursors

1 Cursor Zero (`LargeX.CUR`) is a simple "X" cursor.

2 Cursor One (`Pointer.CUR`) is a larger version of the standard Windows "arrow" cursor.

Appendix C: Graphics Tools Error Codes

Error Codes in Numeric Order *(Also see alphabetical list below.)*

```
0           SUCCESS

999000000   ERROR_FIRST_GFXT_ERROR

999000000   ERROR_DLL_NOT_AUTHORIZED
999000020   ERROR_IMAGE_NOT_FOUND
999000022   ERROR_LIBRARY_NOT_FOUND
999000030   ERROR_BAD_PARAM_VALUE
999000045   ERROR_USER_CANCEL
999000048   ERROR_CANT_BE_DONE
999000081   ERROR_SIZE_LIMIT
999000090   ERROR_NO_WINDOW

999001000   ERROR_FIRST_BAS_ERROR
999001999   ERROR_LAST_BAS_ERROR

999002000   ERROR_FIRST_PRINTER_ERROR
999002999   ERROR_LAST_PRINTER_ERROR

999003000   ERROR_FIRST_ARRAY_ERROR
999003999   ERROR_LAST_ARRAY_ERROR

999004000   ERROR_FIRST_IMAGE_ERROR
999004999   ERROR_LAST_IMAGE_ERROR

999999999   ERROR_UNKNOWN_ERROR

999999999   ERROR_LAST_GFXT_ERROR
```

Error Codes in Alphabetical Order

`ERROR_BAD_PARAM_VALUE` `999000030`

An invalid value was used for a parameter of a Graphics Tools function. For example, a certain function might require a numeric value between 1 and 10 for a certain parameter, and you used 11. You should examine each of the parameters that was passed to the function and confirm that each one contains a valid value.

`ERROR_CANT_BE_DONE` `999000048`

This error indicates that your program attempted to do something that cannot be done, or that is not allowed under the current circumstances. For example, you may have attempted to retrieve a bitmap image from the clipboard when it did not contain a bitmap, or you may have attempted to use the TempDraw mode after you disabled it, or you may have attempted to use the DrawHole function to create a hole with a hole number that is already in use.

Many different functions will return this error code if you attempt to use them while a Clip Area is being defined. Only certain drawing functions can be used to define a Clip Area, and most other drawing functions will return `ERROR_CANT_BE_DONE` if you

attempt to use them after you use GfxClipArea AREA_DEFINE but before the clip area has been "finished".

ERROR_DLL_NOT_AUTHORIZED 999000000

This error code indicates that your program attempted to use a Graphics Tools function before the GraphicsToolsAuthorize function has been used with a valid code.

This error code will also be returned (randomly) if a "dummy code" was used with the GraphicsToolsAuthorize function. Any given Graphics Tools function may or may not work properly if a dummy code has been used, and it may or may not return ERROR_DLL_NOT_AUTHORIZED to indicate that it has failed. See Authorization Codes for more information.

ERROR_FIRST_ARRAY_ERROR 999003000

The range of error codes from ERROR_FIRST_ARRAY_ERROR to ERROR_LAST_ARRAY_ERROR indicate that an array-related error was detected by Graphics Tools. The most common cause of this error is violating a documented rule such as "arrays for the DrawBezier function must have (X*3)+1 elements". Other causes include passing an invalid pointer to a function that is expecting a pointer to an array element.

ERROR_FIRST_BAS_ERROR 999001000

The Graphics Tools Runtime Files were created with the PowerBASIC For Windows compiler, and some portions were created with Microsoft Visual Basic. Under certain circumstances those languages can return BAS errors (i.e. BASIC Error Codes). Graphics Tools adds the value ERROR_FIRST_BAS_ERROR to make it clear what the errors represent. For example, if an unhandled ERROR 53 was detected, Graphics Tools would return ERROR_FIRST_BAS_ERROR+53. This is done to allow your programs to distinguish, for example, between a Windows Error 53 (ERROR_BAD_NETPATH) and BASIC Error 53 (File Not Found).

In many cases, such as File Not Found, these errors should be self-explanatory. If you are not able to determine the meaning of an error code in this range, please contact Perfect Sync Technical Support.

ERROR_FIRST_GFXT_ERROR 999000000

The ERROR_FIRST_GFXT_ERROR and ERROR_LAST_GFXT_ERROR error codes are provided primarily for range-checking. Use these two values to check whether or not a number falls into the range of Graphics Tools Error Codes. Example:

```
lValue& = (some Graphics Tools function)

If lValue& => ERROR_FIRST_GFXT_ERROR And
   lValue& <= ERROR_LAST_GFXT_ERROR Then
        'RETURN VALUE CORRESPONDS TO
        'A GRAPHICS TOOLS ERROR CODE.
End If
```

ERROR_FIRST_IMAGE_ERROR 999004000

The range of error codes from ERROR_FIRST_IMAGE_ERROR to ERROR_LAST_IMAGE_ERROR indicates that an error occurred while Graphics Tools was attempting to display an image such as a JPEG or WMF file. If they do not correspond to an obvious error such as a corrupt image file, please report these errors to Perfect Sync Technical Support.

ERROR_FIRST_PRINTER_ERROR 999002000

The range of error codes from ERROR_FIRST_PRINTER_ERROR to ERROR_LAST_PRINTER_ERROR indicates that an error occurred while Graphics Tools was attempting to print something. See GfxPrintArea and GfxPrintWindow for descriptions of the most common errors. If those descriptions do not apply, please report these Error Codes to support@perfectsync.com .

ERROR_IMAGE_NOT_FOUND 999000020

The requested image 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.

ERROR_LAST_ARRAY_ERROR 999003999

See ERROR_FIRST_ARRAY_ERROR for more information.

ERROR_LAST_BAS_ERROR 999001999

See ERROR_FIRST_BAS_ERROR for more information.

ERROR_LAST_GFXT_ERROR 999999999

See ERROR_FIRST_GFXT_ERROR for more information.

ERROR_LAST_IMAGE_ERROR 999004999

See ERROR_FIRST_IMAGE_ERROR for more information.

ERROR_LAST_PRINTER_ERROR 999002999

See ERROR_FIRST_PRINTER_ERROR for more information.

ERROR_LIBRARY_NOT_FOUND 999000022

A necessary runtime file was not found. Either...

1) Your program attempted to use one of the Graphics Tools Pro JPEG-specific functions, but the Graphics Tools JPEG Library or the Intel JPEG Library could not be located.

2) Your program attempted to use the DisplayImage or StretchImage function on a system where the necessary DLLs do not exist. This should occur only on early versions of Windows 95 and 98, and only when Microsoft Internet Explorer (which contains the necessary files) has *not* been installed.

ERROR_NO_WINDOW 999000090

Your program attempted to use a window number less than zero or greater than the maximum number that your version of Graphics Tools (Standard or Pro) supports, or it attempted to use a graphics window that does not exist, i.e. has not yet been created.

`ERROR_SIZE_LIMIT` `999000081`

This error is generated if your program attempts to use a value that is too large or too small. For example, attempting to use GfxResize to resize a graphics window so that it is 20 pixels wide would generate this error because the minimum allowed size is 32 pixels.

Generally speaking, this error does not indicate a *failure*. For example, the GfxResize function would change the window's size to 32 pixels and return `ERROR_SIZE_LIMIT` to indicate that the limit was reached.

`ERROR_UNKNOWN_ERROR` `999999999`

This error code means that a Graphics Tools function detected that an error has taken place but is unable to determine the cause. This error will also be reported if Graphics Tools uses a Windows API function which fails but does not provide any information about the reason for the failure.

`ERROR_USER_CANCEL` `999000045`

While Graphics Tools was displaying a dialog of some kind (such as the SelectGfxColor dialog) the user selected Cancel instead of making a selection.

Under certain (rare) circumstances Windows will make it appear that the user selected Cancel when in fact it was Windows that canceled the operation for some reason.

`SUCCESS` `0`

No Error. (Same as Microsoft's `ERROR_SUCCESS`, which we feel is oxymoronic.)

Frequently Asked Question:
Whoa! Why are those 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 have 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. Graphics 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.)

Graphics Tools could have easily used the numbers that start with 1,000,000,000 so that they would be easy to read, but we figured that *you* would 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 Graphics Tools Error Codes -- in fact the Error Codes from all Perfect Sync software development products -- fall into that range. The constants `ERROR_FIRST_GFXT_ERROR` and `ERROR_LAST_GFXT_ERROR` correspond to those numbers, and everything else falls in between.

If a Graphics 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 Graphics Tools is simply passing the number along.

Appendix D: Console Tools Plus Graphics

"Console Tools Plus Graphics" is actually the combination of two Perfect Sync products, Graphics Tools and Console Tools.

If you want to use the PowerBASIC PB/CC "Console Compiler" to create programs that display graphics, you will need to use both Console Tools and Graphics Tools. A console window is not like a normal Windows "GUI" window -- it has some very unusual requirements -- and Console Tools contains specialized functions that allow graphics to be displayed in a console window.

If you attempt to use Graphics Tools with PB/CC *without* using Console Tools the graphics window will simply fail to appear.

Programs that are created with the PowerBASIC For Windows compiler, PowerBASIC's PB/DLL compiler, Visual Basic, or any other "GUI" language do *not* need to use Console Tools. It is required only for console applications.

Screen-Refresh Issues

Because console windows (such as those created by the PowerBASIC PB/CC compilers) were not originally designed by Microsoft to provide graphics, Graphics Tools must perform several extra steps in order to draw in a console window. Nearly all of those steps are handled automatically by Graphics Tools, but the process of "refreshing" the graphics window must be handled by your program.

If you don't refresh the screen properly your graphics images will appear to be normal, but if another Windows program temporarily covers up your graphics window, when the other program disappears your graphics window will be empty or partially "corrupted".

The GfxRefresh function is used to periodically refresh the graphics window. In PB/CC programs GfxRefresh is usually executed **1)** several times per second in a "refresh thread", or **2)** by using the Console Tools OnTimer function. The various Graphics Tools sample programs for PB/CC provide examples of using the standard THREAD method. Console Tools *Pro* users can use the OnTimer function instead of a thread. See the Console Tools documentation for more information about OnTimer.

Version Issues

Console Tools versions 1.00 through 2.49 are not compatible with Graphics Tools version 2, they can be used only with Graphics Tools version 1.

To use Graphics Tools version 2 (this version) you must update Console Tools to version 2.50 or above. Console Tools versions 2.50-2.54 can be used, but we strongly recommend the use of version 2.55 or above. No-charge updates are available to all registered Console Tools users from support@perfectsync.com.

Appendix E: Highs Words and Low Words

Microsoft Windows and Graphics Tools often use a programming technique called "High Word and Low Word" to store two different numeric values in a single LONG variable.

It is not necessary for you to understand High Words and Low Words in order to use them, and an explanation of the technical details is beyond the scope of this document. It is sufficient (for the purposes of this document) for you to use the following functions when Graphics Tools returns a High Word / Low Word value.

PowerBASIC

PowerBASIC programmers should use the HIWRD and LOWRD functions that are built into the PowerBASIC languages. If you are using the PowerBASIC For Windows 7.0 compiler (or above) or the PowerBASIC Console Compiler 3.0 (or above) you can use the new HIINT and LOINT functions instead of HIWRD and LOWRD.

It is important to note that because HIWRD and LOWRD always return values between zero (0) and 64k (65,535) if a value is *negative* the HIWORD and LOWRD functions will return *positive* values greater than 32,768. This is a side effect of the way numbers are stored in memory, and your programs may need to include code to compensate. (If you use HIINT and LOINT your program will see the range of numbers from -32k to +32k and no additional code will be necessary.)

Visual Basic

These functions can be used to extract the Low and High Word values from a LONG integer variable. You can use other names for these functions if you like. The names LoWrd and HiWrd are used throughout this document.

```
Public Function LoWrd(ByVal Value As Long) As Integer
    If ((Value And &HFFFF&) > &H7FFF&) Then
        LoWrd = (Value And &HFFFF&) - &H10000
    Else
        LoWrd = Value And &HFFFF&
    End If
End Function

Public Function HiWrd(ByVal Value As Long) As Integer
    HiWrd = (Value And &HFFFF0000) \ &H10000
End Function
```

Other Languages

If your programming language does not provide a built-in Low Word and High Word function, it should be relatively simple for you to translate the Visual Basic code above.

Appendix U: Upgrading From Graphics Tools Version 1.x

This Appendix covers the process of upgrading from Graphics Tools Version 1 to Version 2 in two parts. The first part covers the new features that you will find in Version 2, and the second part covers the things that you *must* do in order for your Version 1 program to work with Version 2.

New Features in Graphics Tools Version 2

Graphics Tools Version 2 contains over 60 new functions, and many Version 1 functions have been enhanced to add new features. This list is intended to give you a *brief* overview of the most important new features that are included in Version 2. For complete information you should read the sections of this document that pertain to each new function or feature.

Unless otherwise noted, these features are available in both the Standard and Pro versions of Graphics Tools Version 2.

"Relaxed" License Agreement Changes in the license agreement allow the use of Graphics Tools Version 2 in freeware, shareware, and no-charge demo software.

New Compatibilities Graphics Tools Version 2 is compatible with all versions of the PowerBASIC for Windows compilers (including PB/DLL), as well as Visual Basic. (PowerBASIC PB/CC programmers should see Console Tools Plus Graphics.)

Multiple Graphics Windows Graphics Tools Standard allows your programs to create as many as four (4) graphics windows at the same time, and Graphics Tools Pro can create up to 256. You can make them all visible, or keep some of them hidden for use as "work windows".

GET/PUT-Style Programming. The new DisplayWindow, StretchWindow, and CropWindow functions allow you to treat *other Graphics Tools graphics windows* as image sources. That means that you can create drawings in one window and then display them in other windows. A single hidden work window can be used to hold dozens of "sprites" that can be displayed in a visible window, on demand. XOR drawing is supported, as well as several other modes.

New Window Styles Graphics Tools Version 2 can create graphics window with many different styles including raised, sunken, bump, and caption borders, and many others.

Sizable and Movable Graphics Windows *GRAPHICS TOOLS PRO ONLY*: Create movable and resizable graphics windows with horizontal and vertical scroll bars, and much more. If desired, you can allow your users to resize and move graphics windows manually, using the mouse. You can even create "stretchable" graphics windows!

Improved Control Over the Graphics Window *GRAPHICS TOOLS PRO ONLY*: The GfxMove, GfxResize, and GfxScroll functions can be used to control the graphics window programmatically.

Focus Rectangles When a graphics window has the Windows focus it can now display a standard Focus Rectangle or a rectangle in the color of your choice. You can also display colored rectangles around graphics windows that *don't* have the focus so that (for example) you could display red borders around inactive windows and a green border around the window where drawing is currently taking place.

User-Defined "Worlds" The new GfxWorld function allows you to define the numbers that represent the left, right, top, and bottom edges of the drawing world. Upside-down worlds are supported, to allow 0,0 to be located at the *bottom* of the graphics window, as well as centered worlds (0,0 in the middle) and anything else that you can imagine.

Drawing With Transparency *GRAPHICS TOOLS PRO ONLY*: With the new OverlayWindow function you can copy an image from one graphics window to another, treating any color that you specify as "transparent".

Gradients *GRAPHICS TOOLS PRO ONLY*: Most Graphics Tools figures (circles, rectangles, polygons, pies, etc.) can now be filled using Gradients instead of solid colors. In addition to the familiar "blue fade", gradients provide a *huge* selection of very impressive color techniques. Predefined gradients can be loaded with the GradientLoad function, or you can create your own with the UseGradient and GradientValue functions. Tell Graphics Tools which figures should be filled with a Gradient instead of a Brush by using the GradientBrush function, save your Gradients for future use with the GradientSave function, and access the colors of your gradients one by one with the GradientColor function.

Three-Dimensional Figures Several different types of "three-dimensional" figures can now be drawn with Graphics Tools...

- The DrawCube function can draw cubes and rectangular boxes of any shape and size. Great for bar charts!
- The DrawWedge function creates a realistic-looking 3D "slice of pie", perfect for creating impressive pie charts!
- DrawCylinder is similar to DrawCube, except that it draws figures that look like 3D cylinders and disks. An attractive alternative for bar charts.
- *GRAPHICS TOOLS PRO ONLY* DrawTube draws a cylinder without an "end cap", resulting in a realistic "tube" effect when an appropriate Gradient is used.

Combine the Pro version's Gradients with the new 3D figures, and you won't believe your eyes!

Exploded Pies and Wedges In addition to displaying Pies and Wedges normally (touching in the center) you can tell Graphics Tools to display them "exploded" by a distance that you define.

Printing of Images *GRAPHICS TOOLS PRO ONLY*: Graphics Tools Pro supports the printing of high-resolution graphics images, including color images. The GfxPrintWindow and GfxPrintArea functions can be used to print all or part of any graphics window's image, and the GfxPrintSetup and GfxPrintPageSetup functions can be used to provide the user with standard printing-oriented dialog boxes. The GfxPrintStatus function can provide information about the most recent print job, GfxPrintParam function can be used to set certain printing parameters programmatically, and GfxPrintDefaults resets the printing functions to use the system default printer.

Display JPEG, WMF, and Other Image Files Using the new DisplayImage and StretchImage functions, Graphics Tools Standard and Pro can display almost any image format that is supported by the version of Windows that is installed on the runtime system. *Not all image formats may be available on all machines.* Systems with very early versions of Windows 95 and Internet Explorer do not support this function. *IMPORTANT NOTE: GIF files are not supported by Graphics Tools even if the runtime machine supports them.*

Display JPEG files *GRAPHICS TOOLS PRO ONLY*: The new DisplayJpeg, StretchJpeg, and CropJpeg functions allow the display of JPEG files on any system.

Save JPEG files *GRAPHICS TOOLS PRO ONLY*: The SaveGfxWindow and SaveGfxArea functions have been enhanced to allow Graphics Tools programs to save JPEG files in various formats and quality levels.

(To be clear... Graphics Tools Standard can display JPEG files, but only if Windows supports JPEG on the runtime machine. Graphics Tools Pro can display and save JPEG files on any Windows system, even if Windows itself does not provide support.)

Graphics Window Messages The list of window messages that are sent to your program has been greatly expanded, allowing much better control of graphics window events such as mouse movement, keystrokes, scrolling, focus events, etc.

Easier Text As with most drawing functions, the "text origin" is normally the top-left corner of the displayed text. To make text easier to display, you can now tell Graphics Tools to use the *bottom*-left corner of the text as the origin, or you can specify several other modes including horizontal and vertical centering. Also, the text-drawing functions can now optionally *move* the origin after text is drawn, making it much easier (for example) to draw two words next to each other using different colors or fonts.

Draw Rotated Ellipses The new DrawEllipseRotated function can draw ellipses that are rotated to any angle. Great for "spirograph"-style effects and many other uses.

Clip Areas: *GRAPHICS TOOLS PRO ONLY*: The flexible GfxClipArea function can be used to define areas of a graphics window where drawing will be allowed or rejected. A clip area can be any size and shape that you want to define -- you can even define multiple clip areas -- and any drawing operations that take place outside the specified area will not be visible. Indispensable for creating a wide variety of otherwise hard-to-draw figures!

Enhanced HSW Color System The Graphics Tools HSW Color System now supports 6-point color calculations, as well as the default 3-point system. For special effects you can also select 12- and 24-point modes.

Use Graphics Tools for Dialog "Skins" You have probably seen programs that have fancy backgrounds behind their buttons and listboxes. Graphics Tools can be used to create a "dynamic" background where your program can draw... anything! You can even draw simulated controls and write code to make them respond to mouse and keyboard events. See Using Graphics For a Window Background.

Drag-Drop Capability *GRAPHICS TOOLS PRO ONLY*: The new GfxDroppedFiles function can be used to obtain the names of disk files that have been dragged and dropped on a graphics window, as well as the exact location where the drop occurred.

Improved TempDraw Mode The new TempErase function complements the existing TempDraw function. It allows you to erase *portions* of figures that have been drawn with the TempDraw mode.

Built-in Dialogs The new SelectGfxColor function can be used to display the standard Windows "Choose A Color" dialog, and the CustomColor function can be used to define up to 16 custom colors that are displayed on the dialog. The new SelectGfxFile function can be used to display a standard Windows "browse for file" dialog, to simplify the process of selecting bitmaps, cursors, icons, gradients, JPEG files, and other graphics-related files.

Auto-Play of Animated Icons and Cursors. *GRAPHICS TOOLS PRO ONLY*: The new AutoPlayCursor, AutoPlayIcon, and AutoPlayControl functions allow your programs to play animations *continuously*, without any extra code to play them in a loop. Several functions that can return width and height values can now return a width and height in a single value.

And Lots Of New Convenience Functions...

- The new GfxFontSize function can be used to change the width and height of a font at the same time (a common operation).
- The GfxSquareness function makes it easier to create graphics window that produce round circles and square squares.
- The WedgeOrder function makes using the DrawWedge function easier.

- The SaveGfxMode function as been added to Graphics Tools to make it easier to specify which file format the SaveGfxWindow and SaveGfxArea functions will use.
- SystemColor lets you ask Windows which colors it is currently using for screen elements like buttons and window backgrounds.
- HighlightColor can be used to calculate a color value that is appropriate for "highlighting" or "shading" another color.
- GfxResponse makes it easier to use Graphics Tools with dialog-based programs.
- The GfxIsOpen function is provided for multi-window programs that need to find out whether or not a given window is currently available for drawing.
- MouseOver provides the ability to determine which graphics window the mouse cursor is currently over.
- New functions called ImageParam, IconParam and CursorParam have been added. They provide the same types of information that the BitmapParam function provides for bitmaps. The PRO version also includes JpegParam.

Many new options have been added to existing functions. A *very* few examples... The DrawFrom function now accepts `GFX_WINDOW_CENTER` and `GFX_WINDOW_EDGE`. The GfxMetrics function can now return information about the scroll position of the graphics window. The Stretch functions (StretchBitmap, etc.) can now automatically maintain the aspect ratio of the original image. And much more!

Many "under the hood" improvements have also been made, resulting in faster and more accurate drawing.

A number of **Sample Images** have been included with Graphics Tools, to make it easier to experiment with various functions. The images are located in the `\GfxTools\Images` directory. The `NASA` subdirectory contains a number of space-related JPEG images.

A much larger selection of **Sample Programs** has been added to Graphics Tools. You'll find them in the `\GfxTools\Samples` directories.

And of course the **Graphics Tools Documentation** has been thoroughly reviewed, updated, and improved. Many new sections have been added, including one called Sample Programs.

ALL IN ALL, GRAPHICS TOOLS VERSION 2 IS THE MOST COMPREHENSIVE UPDATE THAT PERFECT SYNC HAS EVER CREATED FOR ONE OF ITS DEVELOPMENT TOOLS.

Upgrading Your Existing Programs

When we created Graphics Tools Version 2 we tried to make it as easy as possible to upgrade your programs from Version 1. If you check the following items, you should find the upgrade process to be relatively quick and painless. The items are listed in their approximate order of importance. *Many of these items will not apply to most programs, but we strongly suggest that you review the entire list!*

- ALL PROGRAMS: Instead of using `#INCLUDE "GFXTOOLS.INC"`, your programs should now use `"GFXT_STD.INC"` or `"GFXT_PRO.INC"`, depending on the version of Graphics Tools (Standard or Pro) that you have licensed. (If you continue to use `GFXTOOLS.INC` your programs will continue to use Graphics Tools Version 1.)

- ALL PROGRAMS: All Graphics Tools Version 2 programs must use the new GraphicsToolsAuthorize function before they use any other Graphics Tools functions. If you do not use GraphicsToolsAuthorize first, various functions will return the new Error Code `ERROR_DLL_NOT_AUTHORIZED`. For more information about this, see GraphicsToolsAuthorize, Graphics Tools Authorization Codes. and Four Critical Steps For Every Program.

- ALL PROGRAMS: Graphics Tools Version 1 used a runtime file called `GfxTools.DLL`. Version 2 uses separate files for the Standard and Pro versions, called `GfxT_STD.DLL` and `GfxT_PRO.DLL`. When Graphics Tools Version 2 was installed on your hard drive, only one of those files was installed, depending on whether you obtained a Standard or Pro license. If you distribute your Graphics Tools program(s) you will need to distribute the appropriate new file, not `GfxTools.DLL`.

- DEMO PROGRAMS: Because of the licensing changes (see New Features above) it is no longer necessary for you to distribute the `GfxDemo.DLL` file with your not-for-profit programs. You can use the `GfxT_Std.DLL` or `GfxT_Pro.DLL` file for all of your programs.

- CONSOLE TOOLS PLUS GRAPHICS USERS: You must use Console Tools Version 2.55 or above in order to use Graphics Tools Version 2. Contact support@perfectsync.com if you need an upgrade.

- CONSOLE TOOLS PLUS GRAPHICS USERS: The `CT_Gfx.DLL` "linker" file is no longer necessary for Console Tools Plus Graphics programs. The functions that were located in `CT_Gfx.DLL` have been incorporated directly into Graphics Tools, so you no longer need to distribute the linker file.

- ALL PROGRAMS: Because Graphics Tools Version 2 supports multiple graphics windows (see New Features above) the Version 1 InitGfx function has been replaced by the new OpenGfx function. We recommend that if you are upgrading an existing Graphics Tools program to Version 2 you should simply "plug in" the OpenGfx function wherever you used InitGfx before. (But you should be aware that other methods of creating a graphics window are also now available. See The Graphics Window if you're curious.) It is important to note that the OpenGfx function requires parameters called Left, Top, *Width*, and *Height* instead of the Left, Top, *Right*, and *Bottom* parameters that were required by InitGfx.

- The Version 1 SaveGfxScreen function has been renamed to SaveGfxWindow to reflect its ability to save the contents of any Graphics Tools graphics window.

- The Version 1 GfxMetrics function returned many different values that were not really "metrics", .i.e. measurements. For example, it could return a True/False value to indicate whether or not your program had told it to use the GFX_USE_RADIANS option. Those non-measurement values are no longer supported. If your program needs to track the status of a program-specified setting like that, it should include source code to do so.

- The Version 1 GfxTitle function has been replaced with the GfxCaption function. GfxCaption allows your programs to set the window title and, if the graphics window is created with a caption, to change the "state" of the caption background.

- Many Version 1 equate (constant) names have been changed. The most common changes are 1) the addition of underscores and 2) the addition of GFX at the beginning or end of various equates. For example, CLEAR has been renamed to GFX_CLEAR, and SHOW has been changed to GFX_SHOW.

  These changes were made for several reasons, the most important of which was the fact that common words like %SHOW conflict with keywords in various programming languages, including PowerBASIC. It is widely considered to be a bad practice to give two different meanings to the same token. Another reason was that certain Version 1 equates were difficult to read. The equate %ERROR_GT_CANTBEDONE for example, has been changed to %ERROR_CANT_BE_DONE. Other equates have been renamed to make them more generic, so they can be used by more than one function. For example the various BITMAP_ equates now begin with IMAGE_ to indicate that they can be used for several different image types. And finally, some changes were made to conform certain values to other Perfect Sync products. The _GT_ has been dropped from the ERROR_ equates so that all Perfect Sync products can use the same error values.

  So while we realize that it will probably require Graphics Tools Version 1 users to perform a significant number of edits when upgrading their programs to Version 2, we felt that these changes were very important. Here is a list of the changes, in alphabetical order.

VERSION 1	VERSION 2
%ADJUSTRECT	%GFX_ADJUST_RECT
%AUTOPLAY	%GFX_AUTO
%BEGIN_TEMP	%TEMPDRAW_BEGIN
%BLACK_ONLY	%GFX_BLACK_ONLY
%BMP_BITSPERPIXEL	%IMAGE_BITS_PER_PIXEL
%BMP_BYTESPERLINE	%IMAGE_BYTES_PER_LINE
%BMP_HEIGHT	%IMAGE_HEIGHT
%BMP_PLANES	%IMAGE_PLANES
%BMP_WIDTH	%IMAGE_WIDTH
%CHECKBOX	%GFX_CHECKBOX
%CHECKED	%GFX_CHECKED
%CLEAR	%GFX_CLEAR
%CLEAR_TEMP	%TEMPDRAW_CLEAR
%CLOSE_GFX	%GFX_CLOSE
%DASHDOT	%GFX_DASHDOT
%DASHDOTDOT	%GFX_DASHDOTDOT
%DASHED	%GFX_DASHED
%DIAGBACK	%GFX_DIAGBACK
%DIAGBOTH	%GFX_DIAGBOTH
%DIAGFWD	%GFX_DIAGFWD
%DISABLE_TEMP	%TEMPDRAW_DISABLE
%DOTTED	%GFX_DOTTED
%END_TEMP	%TEMPDRAW_END

%ERROR_GT_CANTBEDONE	%ERROR_CANT_BE_DONE
%ERROR_GT_FILENOTFOUND	%ERROR_IMAGE_NOT_FOUND
%ERROR_GT_FIRSTERROR	%ERROR_FIRST_GFXT_ERROR
%ERROR_GT_FIRSTPBERROR	%ERROR_FIRST_BAS_ERROR
%ERROR_GT_INVALIDPARAMETER	%ERROR_BAD_PARAM_VALUE
%ERROR_GT_LASTERROR	%ERROR_LAST_GFXT_ERROR
%ERROR_GT_UNKNOWNERROR	%ERROR_UNKNOWN_ERROR
%FLATBORDER	%GFX_FLAT_BORDER
%FOREGROUND	*no longer supported*
%FREEZE	%GFX_FREEZE
%HATCHED	%GFX_HATCHED
%HIDE	%GFX_HIDE
%HLINE	%GFX_HLINE
%HVLINES	%GFX_HVLINES
%INACTIVE	%GFX_INACTIVE
%INVERSE_COLORS	%GFX_INVERSE_COLORS
%INVERSE_SCREEN	%GFX_INVERSE_SCREEN
%MASK_COLORS	%GFX_MASK_COLORS
%MASK_COLORS_INVERSE	%GFX_MASK_COLORS_INVERSE
%MASK_COLORS_INV_PEN	%GFX_MASK_COLORS_INV_PEN
%MASK_COLORS_INV_SCRN	%GFX_MASK_COLORS_INV_SCRN
%MERGE_COLORS	%GFX_MERGE_COLORS
%MERGE_COLORS_INVERSE	%GFX_MERGE_COLORS_INVERSE
%MERGE_COLORS_INV_PEN	%GFX_MERGE_COLORS_INV_PEN
%MERGE_COLORS_INV_SCRN	%GFX_MERGE_COLORS_INV_SCRN
%MONOBORDER	%GFX_MONO_BORDER
%NO_DRAW	%GFX_NO_DRAW
%NORMAL_DRAW_MODE	%GFX_NORMAL_DRAW_MODE
%OPAQUE_COLORS	%GFX_OPAQUE_COLORS
%PUSHBUTTON	%GFX_PUSHBUTTON
%PUSHED	%GFX_PUSHED
%RADIOBUTTON	%GFX_RADIOBUTTON
%SHOW	%GFX_SHOW
%SOLID	%GFX_SOLID
%TOOLSET_CONSOLE	%GFX_TOOLSET_CONSOLE
%TOOLSET_PRINTER	%GFX_TOOLSET_PRINTER
%TOOLSET_WINDOWS	%GFX_TOOLSET_WINDOWS
%THREESTATE	%GFX_THREESTATE
%UNCHANGED	%GFX_SAME
%UNFREEZE	%GFX_UNFREEZE
%VLINE	%GFX_VLINE
%WHITE_ONLY	%GFX_WHITE_ONLY
%XOR_DRAW	%GFX_XOR_DRAW
%XOR_DRAW_INVERSE	%GFX_XOR_DRAW_INVERSE

- Please also note that the Graphics Tools documentation now uses the standard Microsoft notation and no longer uses the PowerBASIC-specific `%` symbol to signify constants. PowerBASIC programmers must add the `%` symbol to the examples in this document to turn constants like `GFX_SHOW` into PowerBASIC "equates" like `%GFX_SHOW`. (PowerBASIC-specific examples in this document and all PowerBASIC example source code files *do* include the necessary `%`.)

- It is also important to note that a very small number of equates have new *numeric* values. For example, the Version 1 `%CLEAR` value was defined as negative one (-1) and the new `%GFX_CLEAR` value is 99. We strongly recommend the use of named values instead of literal numbers. *If your Graphics Tools version 1 programs use literal numeric values*

instead of named equates, you will need to review use of the following values. If your programs use the named equates you can ignore this list.

```
OLD: %CLEAR                      = -1
NEW: %GFX_CLEAR                  = 99

OLD: %LOC_CLEAR                  = 0
NEW: %LOC_CLEAR                  = 99

OLD: %CLEAR_TEMP                 = 2
NEW: %TEMPDRAW_CLEAR             = 99

OLD: %DISABLE_TEMP               = 99
NEW: %TEMPDRAW_DISABLE           = -1

OLD: %FRAME_BUMP                 = &h3&
NEW: %FRAME_BUMP                 = &h9&

OLD: %UNCHANGED                  = &HFFFFFFFF&
NEW: %GFX_SAME                   = &h7FFFFF0C&

OLD: %AUTOPLAY                   = -1
NEW: %GFX_AUTO                   = &h7FFFFF0A&

OLD: %ERROR_GT_INVALIDPARAMETER = 999000001&
NEW: %ERROR_BAD_PARAM_VALUE      = 999000030&

OLD: %ERROR_GT_CANTBEDONE        = 999000002&
NEW: %ERROR_CANT_BE_DONE         = 999000048&
```

- The MouseOverX and MouseOverY functions no longer return negative one (`-1`) when the mouse is not located over the graphics window. Because of the new "worlds" feature, negative one is a valid drawing location that can be return when the mouse *is* over a window, so those functions now return `GFX_NONE` if the mouse is not currently located over the graphics window.

- If your Graphics Tools Version 1 programs use Pens that are wider than one pixel, you will need to be aware that Pen widths are now "scaled" to compensate for the size of the graphics window. Using a Pen width of "2" will no longer produce a 2-pixel-wide pen. You can either modify your program to use appropriate Pen width values (see PenWidth) or you can disable Pen Scaling by adding this line of code to your program:

  ```
  GfxOption GFX_PEN_SCALE_DISABLE, %TRUE
  ```

  Doing that makes Graphics Tools Version 2 Pens work like Version 1 Pens.

- If your Graphics Tools Version 1 programs use the GfxOption function to change the horizontal and/or vertical scaling of the graphics window, we recommend that you now use the GfxWorld function instead. (You can continue to use the GfxOption method if you prefer, but doing so *disables* the new GfxWorld function.)

- If your Graphics Tools Version 1 programs use `GfxOption GFX_IMAGE_SAVE_MODE` to specify the type of bitmaps that should be saved by the SaveGfxWindow function, you should change your program to use the SaveGfxMode function instead. `GFX_IMAGE_SAVE_MODE` is no longer supported.

- If your version 1 programs use any of the array-based drawing functions -- DrawMultiLine, DrawPolygon, or DrawBezier -- you will need change the way parameters are passed to those functions. Graphics Tools Version 1 accepted an entire array (with empty parentheses) as a parameter, but Version 2 requires that you pass 1) the first element of the array and 2) the number of points in the array. Also, the Version 2 arrays are "backwards" compared to Version 1. If you created an array that was dimensioned as `(15,1)` with Version 1 you must use `(1,15)` to get the same results with Version 2.

- The `GfxOption GFX_FAST_POLYDRAW` mode has been eliminated. All polygons are now automatically drawn using the "fast" mode, so if you use it, you should remove `GFX_FAST_POLYDRAW` from your programs.

- If your programs use the DrawFrame function, please note that the `GfxOption GFX_FRAME_TYPE` and `GFX_FRAME_OPTIONS` settings are no longer supported. Those values can now be specified on a frame-by-frame basis by using the two new parameters of the DrawFrame function.

- If your programs use the `GfxOption GFX_LARGEST_ICON` setting to tell Graphics Tools that you will be using unusually large icons, you should no longer do so. Graphics Tools now automatically detects large icons.

- If your programs use the `DrawTextBox TEXT_CALC_FONTNAME` function to *retrieve* the name of the current font, you should now use the GfxFontName function instead.

- If your programs use the `DrawTextBox TEXT_CALC_SIZE` function to calculate the size of text, please be advised that Graphics Tools Version 1 contained a bug which caused that function to return pixels instead of Drawing Units. That bug has been fixed, and you may need to adjust your code accordingly.

- If your programs use resources (icons, cursors, or bitmaps) that are embedded in a PowerBASIC `#RESOURCE` file, it is no longer necessary for you to use `GfxOption GFX_MODULE_INSTANCE` to give Graphics Tools the module handle. Graphics Tools now automatically finds the module handle of your EXE.

- It is no longer possible to create parent-less graphics windows with Graphics Tools. We have always recommended against their use, but if your program uses a parent-less graphics window, contact Perfect Sync Technical Support and we will help you find an acceptable workaround.

Sample Programs

The \GfxTools\Samples\ directory contains a number of sample programs that demonstrate various Graphics Tools features. Each program demonstrates one basic feature.

The PB subdirectory contains PowerBASIC For Windows and PB/DLL code, the PBCC subdirectory contains PowerBASIC PB/CC (Console Compiler) code for use with Console Tools Plus Graphics, and the VB subdirectory contains Visual Basic code.

Please note that separate versions of the sample Visual Basic programs are provided for the Standard and Pro versions of Graphics Tools. This is required because of the way in which VB links module files. If both _Pro and _Std programs are provided (such as Cubes_Pro.VBP and Cubes_Std.VBP) they will produce identical results. You should attempt to load and run only the programs that are intended for your version of Graphics Tools. If you attempt to load a _Std sample file when the Pro version is installed, or vice versa, a VB error message will be displayed and the project will fail to load.

In all cases -- regardless of the programming language -- if a program's name ends in _ProOnly the program is intended to demonstrate features that exist only in the Pro version of Graphics Tools.

Console Tools Plus Graphics users: If a program's name include _ProOnly it refers to the version of Graphics Tools that must be used. You can use either Console Tools Standard or Pro.

The Sample Programs

The Skeleton Programs
Using Multiple Graphics Windows
Drawing Circles
Gradient Circles
Drawing A Bezier Curve
Drawing X-Sided Figures and Stars
Drawing Cubes and Boxes
Drawing Animated Cursors and Icons
A World With Zero In The Center
A Rectangular Graphics Window
Selecting Colors
Using the TempDraw Mode

A Simple Image Viewer
Resizing a Graphics Window
Printing An Image
Using Clip Areas to Restrict Drawing
Simple Bar Chart
Gradient-Bar Chart
Gradient-Filled Text
Gradient-Filled Cylinders
Gradient-Filled Tubes
Creating a Draggable Graphics Window
Handling Window Messages
Visual Basic "Direct To DLL"
Using Graphics For a Window Background

The Skeleton Programs

The Skeleton programs called...

```
\GfxTools\Samples\PB\Skeleton.BAS
\GfxTools\Samples\PBCC\Skeleton.BAS
\GfxTools\Samples\VB\Skeleton_Pro.VBP
\GfxTools\Samples\VB\Skeleton_Std.VBP
```

...are "bare bones" programs that do very little except create a graphics window. They are intended for use as a starting point for your Graphics Tools projects.

We recommend that you identify the file(s) that you intend to use, and edit them to eliminate all of the TODO items. Then, whenever you want to start a new Graphics Tools project, *copy* the empty skeleton program to provide a framework for the new project.

Keep in mind, however, that if you install a Graphics Tools Update at some point in the future, it is likely that the skeleton programs will be overwritten and your changes will be lost.

Using Multiple Graphics Windows

The sample programs called...

```
\GfxTools\Samples\PB\Skel-4.BAS
\GfxTools\Samples\PBCC\Skel-4.BAS
\GfxTools\Samples\VB\Skel-4_Pro.VBP
\GfxTools\Samples\VB\Skel-4_Std.VBP
```

... demonstrate how to create and draw in four (4) different graphics windows in the same program.

Graphics Tools Standard can create as many as four (4) graphics windows at a time, and Graphics Tools Pro can create up to 256 graphics windows.

Drawing Circles

The sample programs called...

```
\GfxTools\Samples\PB\Circles.BAS
\GfxTools\Samples\PBCC\Circles.BAS
\GfxTools\Samples\VB\Circles_Pro.VBP
\GfxTools\Samples\VB\Circles_Std.VBP
```

... demonstrate how to create and draw circles, but the program can be easily adapted to draw other simple shapes.

Gradient Circles

The sample programs called...

```
\GfxTools\Samples\PB\GradCircles_ProOnly.BAS
\GfxTools\Samples\PBCC\GradCircles_ProOnly.BAS
\GfxTools\Samples\VB\GradCircles_ProOnly.VBP
```

...draw circles using the same Gradient but different gradient "fill types".

Drawing A Bezier Curve

The sample programs called...

```
\GfxTools\Samples\PB\Bezier.BAS
\GfxTools\Samples\PBCC\Bezier.BAS
\GfxTools\Samples\VB\Bezier_Pro.VBP
\GfxTools\Samples\VB\Bezier_Std.VBP
```

...demonstrate how to draw a Bezier Curve. By moving the points you can see the effect that the "control points" have on a Bezier Curve.

Drawing X-Sided Figures and Stars

The sample programs called...

```
\GfxTools\Samples\PB\Xagon.BAS
\GfxTools\Samples\PBCC\Xagon.BAS
\GfxTools\Samples\VB\Xagon_Pro.VBP
\GfxTools\Samples\VB\Xagon_Std.VBP
```

...draw a series of x-sided figures (triangles, squares, pentagons, etc.) and x-sided stars (pentagrams, etc.).

Drawing Cubes and Boxes

The sample programs called...

```
\GfxTools\Samples\PB\Cubes.BAS
\GfxTools\Samples\PBCC\Cubes.BAS
\GfxTools\Samples\VB\Cubes_Pro.VBP
\GfxTools\Samples\VB\Cubes_Std.VBP
```

...are programs that show several different options that can be used for drawing cubes and boxes with the DrawCube function.

Drawing Animated Cursors and Icons

The sample programs called...

```
\GfxTools\Samples\PB\AnimIcon.BAS
\GfxTools\Samples\PBCC\AnimIcon.BAS
\GfxTools\Samples\VB\AnimIcon_Pro.VBP
\GfxTools\Samples\VB\AnimIcon_Std.VBP
```

...show how an animated icon or cursor can be played "on demand".

For information about playing animations automatically, see AutoPlayCursor and AutoPlayIcon.

A World With Zero In The Center

The sample programs called...

```
\GfxTools\Samples\PB\World0.BAS
\GfxTools\Samples\PBCC\World0.BAS
\GfxTools\Samples\VB\World0_Pro.VBP
\GfxTools\Samples\VB\World0_Std.VBP
```

...demonstrate the use of a Drawing World where the 0,0 point is in the center of the graphics window instead of the default top-left corner position.

A Rectangular Graphics Window

The sample programs called...

```
\GfxTools\Samples\PB\World2.BAS
\GfxTools\Samples\VB\World2_Pro.VBP
\GfxTools\Samples\VB\World2_Std.VBP
```

...demonstrate the use of a Drawing World that is much wider than it is tall, but still produces round circles and square squares. The principles that are demonstrated by the World2 programs can be applied to graphics windows of any proportions.

Like the World0 programs, the World2 programs also place the 0,0 point in the center of the graphics window instead of the default top-left corner position.

There is no PBCC version of this sample program because PBCC normally draws in a window that is wider than it is tall.

Selecting Colors

The sample programs called...

```
\GfxTools\Samples\PB\ChooseColor.BAS
\GfxTools\Samples\PBCC\ChooseColor.BAS
\GfxTools\Samples\VB\ChooseColor_Pro.VBP
\GfxTools\Samples\VB\ChooseColor_Std.VBP
```

...show how the SelectGfxColor function can be used to allow the user to select a color.

Using the TempDraw Mode

The sample programs called...

```
\GfxTools\Samples\PB\TempDraw.BAS
\GfxTools\Samples\PBCC\TempDraw.BAS
\GfxTools\Samples\VB\TempDraw_Pro.VBP
\GfxTools\Samples\VB\TempDraw_Std.VBP
```

...show how the TempDraw mode can be used to draw and erase figures on top op permanent figures.

Try clicking on the window caption and dragging the window around the screen. The TempDraw circles will still be visible unless you drag one of them off the screen and back on.

Drawing an Image with Transparent Areas

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\OverlayWindow_ProOnly.BAS
\GfxTools\Samples\PBCC\OverlayWindow_ProOnly.BAS
\GfxTools\Samples\VB\OverlayWindow_ProOnly.VBP
```

...show how to perform drawing operations with transparent areas that are based on a certain color.

A Simple Image Viewer

The ImageView sample programs are called...

```
\GfxTools\Samples\PB\ImageView.BAS
\GfxTools\Samples\PBCC\ImageView.BAS
\GfxTools\Samples\VB\ImageView_Pro.VBP
\GfxTools\Samples\VB\ImageView_Std.VBP
```

ImageView is a simple program that can be used to view images of various types. It also shows the use of the SelectGfxFile function.

Resizing a Graphics Window

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\Resize1_ProOnly.BAS
\GfxTools\Samples\PB\Resize3_ProOnly.BAS
\GfxTools\Samples\PBCC\Resize_ProOnly.BAS
\GfxTools\Samples\VB\Resize1_ProOnly.VBP
\GfxTools\Samples\VB\Resize3_ProOnly.VBP
```

...show various techniques for manually and programmatically resizing a graphics window.

It is important to note that some of these programs can be used to demonstrate two or more techniques, and in some cases two different programs are provided. For example, the `PB\Resize1` program can be compiled four (4) different ways, to produce different types of resizable windows for PowerBASIC programs, and the `PB\Resize3` program shows yet another technique.

Printing An Image

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\Print_ProOnly.BAS
\GfxTools\Samples\PBCC\Print_ProOnly.BAS
\GfxTools\Samples\VB\Print_ProOnly.VBP
```

...show how the various print-related functions are used. See GfxPrintWindow, GfxPrintPageSetup, and GfxPrintSetup.

Simple Bar Chart

The sample programs called...

```
\GfxTools\Samples\PB\Bar.BAS
\GfxTools\Samples\PBCC\Bar.BAS
\GfxTools\Samples\VB\Bar_Pro.VBP
\GfxTools\Samples\VB\Bar_Std.VBP
```

...demonstrate a simple method of creating a bar chart.

Gradient-Bar Chart

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\GradBar_ProOnly.BAS
\GfxTools\Samples\PBCC\GradBar_ProOnly.BAS
\GfxTools\Samples\VB\GradBar_ProOnly.VBP
```

...are exactly the same as the Simple Bar Chart programs, but they have been modified to use a Gradient to fill the bars.

Gradient-Filled Text

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\GradText_ProOnly.BAS
\GfxTools\Samples\PBCC\GradText_ProOnly.BAS
\GfxTools\Samples\VB\GradText_ProOnly.VBP
```

...show several different gradients that can be used to fill text.

Gradient-Filled Cylinders

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\GradCyl*_ProOnly.BAS
\GfxTools\Samples\PBCC\GradCyl*_ProOnly.BAS
\GfxTools\Samples\VB\GradCyl*_ProOnly.VBP
```

...show how gradients can be used to fill cylinders. The stars in the file names represent the numbers 0, 2, and 3, indicating that Gradient Fill Type that is used by the program. For example, `GradCyl`**`2`**`_ProOnly` demonstrates Fill Type 2. (Type 1 produces results that are visually similar to Type 0, so no sample program is provided for Type 1.)

Gradient-Filled Tubes

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\GradTube*_ProOnly.BAS
\GfxTools\Samples\PBCC\GradTube*_ProOnly.BAS
\GfxTools\Samples\VB\GradTube*_ProOnly.VBP
```

...show how gradients can be used to fill tubes. The stars in the file names represent the number 2 or 3, indicating that Gradient Fill Type that is used by the program. For example, `GradTube`**`2`**`_ProOnly` demonstrates Fill Type 2. (Gradient Fill Types 0 and 1 do not produce useful results when you are drawing tubes.)

Using Clip Areas to Restrict Drawing

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\ClipArea_ProOnly.BAS
\GfxTools\Samples\PBCC\ClipArea_ProOnly.BAS
\GfxTools\Samples\VB\ClipArea_ProOnly.VBP
```

...show how you can create and combine Clip Areas to draw figures that are otherwise difficult to draw.

Creating a Draggable Graphics Window

These programs use functions that are available only in the Pro *version of Graphics Tools.*

The sample programs called...

```
\GfxTools\Samples\PB\DragMove_ProOnly.BAS
\GfxTools\Samples\PBCC\DragMove_ProOnly.BAS
\GfxTools\Samples\VB\DragMove_ProOnly.VBP
```

...show how you can create draggable graphics windows. The user can click on the graphics window's caption (the "title bar") and drag the window to a new location.

Handling Window Messages

This sample program applies only to PowerBASIC For Windows, PB/DLL, and other languages which can process Window Messages*. It does not apply to Console Tools Plus Graphics. Visual Basic OCX programmers should use standard Events to handle Window Messages. Visual Basic DLL users should see the* Visual Basic "Direct To DLL" *sample program.*

The sample program called `\GfxTools\Samples\PB\Messages.BAS` demonstrates how to process Graphics Window Messages.

Visual Basic "Direct To DLL"

These sample programs apply only to Visual Basic and other languages which usually rely on OCX controls, but which can also use standard Win32 DLLs.

The `NoOCX*.VBP` projects demonstrate two different "Direct To DLL" techniques that can be used to eliminate the use of the Graphics Tools OCX in Visual Basic programs. Using the Direct To DLL method improves drawing speed and eliminates the need for distributing, installing, and registering the Graphics Tools OCX (and its various support files) when you distribute your VB program. The Graphics Tools DLL is all that is required.

IMPORTANT NOTE: When the rest of this document refers to Visual Basic, it assumes that you are using the OCX version of Graphics Tools. If you use Visual Basic with the Graphics Tools Direct To DLL technique you should (generally speaking) follow the directions that are provided in this document for PowerBASIC programmers. When this document says "Visual Basic programmers must..." or "...cannot..." it is referring to VB programs that use the Graphics Tools OCX, not the DLL.

The NoOCX1 projects...

```
\GfxTools\Samples\VB\NoOCX1_Pro.VBP
\GfxTools\Samples\VB\NoOCX1_Std.VBP
```

...demonstrate a technique which uses a standard Picture control as a "container" for a graphics window. With this technique, the Picture control serves as a convenient placeholder during the form-design process, and certain Picture control properties such as Left, Top, Width, and Height can be used to affect the graphics window. Also, the Picture control's border can be added to the graphics window's border to produce some attractive double-border effects.

The NoOCX2 projects...

```
\GfxTools\Samples\VB\NoOCX2_Pro.VBP
\GfxTools\Samples\VB\NoOCX2_Std.VBP
```

...demonstrate how to create a graphics window directly on the surface of a form. This method is *required* if you want to create a drag-sizable graphics window. If you attempt to do that with the NoOCX1 method you will see that the Picture control is not resized when the user drags the graphics window to a new size. Other than that, the NoOCX1 and 2 methods will produce very similar results.

Perfect Sync recommends the use of the NoOCX1 method if you are *not* creating a drag-sizable graphics window. It provides additional benefits with no performance penalties or other disadvantages.

Using Graphics For a Window Background

This sample program applies to SDK-style programming only.

The sample program `\GfxTools\Samples\PB\DlgBkgd.BAS` demonstrates a technique that allows a Graphics Tools graphics window to be used as the background for a dialog. It would be relatively simple to adapt the code to work with non-dialog windows (such as those created with the CreateWindowEx API function) but this technique cannot currently be used with Visual Basic or Console Tools Plus Graphics.